Desarrollo de API's RESTful multiplataforma para la gestión de datos de clientes empresariales en Pragma S.A.

Santiago Álvarez Pinzón

Semestre de Industria para optar al título de Ingeniero Electrónico

Asesor de la Empresa:

David Alejandro Tolosa Zabala

Líder del equipo de crecimiento

Asesor Interno:
Luis Germán García Morales

Profesor del Departamento de Ingeniería Electrónica y de Telecomunicaciones

Universidad de Antioquia
Facultad de Ingeniería
Ingeniería Electrónica
Medellín
2022

Cita	Álvarez Pinzón [1]
Referencia Estilo IEEE (2020)	[1] Álvarez Pinzón, S. (2022), Desarrollo de API's RESTful multiplataforma para la gestión de datos de clientes empresariales en Pragma S.A. Semestre de industria, ingeniería electrónica, Universidad de Antioquia, Medellín, 2022.

Contenido

Resumen

Pragma es una empresa internacional con más de 700 empleados y presencia en 5 países, dedicada profundamente a la transformación digital de negocios y actividades organizacionales, procesos, competencias y modelos para aprovechar completamente los cambios y oportunidades de una sociedad en constante cambio. Las principales áreas de trabajo de la empresa son el desarrollo de soluciones de software y la provisión de talento en las áreas de tecnología. Actualmente existe una alta demanda de API's que permitan la administración de información de negocios en todos los ámbitos económicos, tal que permitan realizar las tareas necesarias de manera eficiente y confiable. Es así como se propuso la idea de producir dos API's Restful que implementaran las operaciones CRUD (Create, read, update y delete), para la gestión de información de clientes y almacenamiento de imágenes: la primera, aplicación web Java que trabaje sobre 2 bases de datos locales, una relacional provista por MySQL server y otra no relacional en MongoDB Community server; y la segunda, basada en Lambdas de la plataforma Amazon Web Services, las cuales actuarían sobre una base de datos DynamoDB para datos y otra sobre S3 para gestión de imágenes. Durante el desarrollo se afrontaron diversos retos que derivaron en cambios a las ideas iniciales, de manera que el resultado fueron 2 conjuntos de API's, el primero, una API Java MVC conformada por un servidor de gestión y 4 microservicios que implementan las funciones CRUD sobre la base de datos en el servidor MongoDB Community Server y en la nube Amazon Relational Databases (RDS), con acceso de seguridad completamente gestionado, descentralizados, administrables, con capacidad de respuesta amplia, con cobertura de pruebas de código mayor al 80% y completamente documentados. El segundo conjunto consta de 5 lambdas de AWS que implementan las funciones CRUD sobre las bases de datos en la nube DynamoDB y S3, con acceso de seguridad gestionado, ejecutadas de manera completamente remota, económica, de capacidad escalable y monitoreada. Como componente adicional a estas, se creó una API capaz de consumir los microservicios ofrecidos por las anteriormente mencionadas, logrando una integración amplia entre ambos conjuntos y dotándolos de capacidades adicionales con las que no contaban.

1. Introducción

Durante toda la historia humana el constante avance tecnológico afecta la mayoría de los aspectos de la vida diaria de las personas, llegando incluso a definir completamente la estructura de la sociedad de turno. Para ejemplificar lo anterior, "la edad de piedra o también Etapa Lítica es el período de la prehistoria que abarca desde que los seres humanos empezaron a elaborar herramientas de piedra hasta el descubrimiento y uso de metales" [16] o la revolución industrial en el siglo XVIII, la cual produjo una serie de transformaciones económicas, tecnológicas y sociales de carácter trascendental para la humanidad. Ambos eventos, descritos anteriormente, se definen como puntos de inflexión de la historia, comparables únicamente entre sí, debido a la velocidad y la magnitud de los cambios sufridos, los cuales marcarían un antes y un después en aspectos tan simples de la vida como la alimentación de la gente, hasta cuestiones que definirían el futuro del mundo como la escala y capacidad bélica de las naciones. Habiendo resaltado el papel que posee la tecnología sobre las sociedades, es correcto trasladarse a la edad moderna, en la que las tecnologías de comunicación han alcanzado un apogeo tal que fue capaz de acortar las distancias que existieron por miles de años y separaban individuos, países y sociedades enteras, dando paso a nuevas interacciones que facilitaron el estilo de vida de las nuevas generaciones, más concretamente en el ámbito del comercio. Un ejemplo es la relación entre las empresas Blockbuster y Netflix. La primera "fue una franquicia estadounidense de videoclubes, especializada en alquiler de cine y videojuegos a través de tiendas físicas, servicios por correo y video bajo demanda" [17], mientras que la segunda era una empresa de renta de DVD por correo de mucha menor escala. A principios del año 2000, Blockbuster recibió una oferta para comprar Netflix por la suma de 50 millones de dólares, pero el CFO de Blockbuster le pareció más una broma, por lo que ni siquiera consideró la posibilidad de efectuar el negocio, según narran los ejecutivos de Netflix en su momento. 10 años después, Blockbuster se declararía en quiebra ante el auge de las plataformas de streaming, conflictos de derechos de autor y nuevos medios de consumo digital, dejando atrás una historia de como una decisión afectó el futuro del consumo de entretenimiento.

En las últimas décadas, las empresas y establecimientos de comercio han tenido

que afrontar la inevitable necesidad de transformación, con el objetivo de expandirse a la mayor cantidad de usuarios posibles y aumentar su productividad y rentabilidad a futuro. Es así como los comercios tradicionales se trasladaron a plataformas electrónicas aprovechando los avances tecnológicos, sin los cuales, esta transformación habría sido imposible, tales como el desarrollo de nuevos medios de almacenamiento de información, los cuales permitieron la creación de bases de datos con la información de clientes, inventarios, medios, etc. Nuevos lenguajes de programación como Java y Python flexibilizaron la creación de plataformas y servicios digitales, nuevos métodos de transmisión de información como wifi y fibra y la creación de dispositivos capaces de consumirla, como smartphones, weareables e IoT.

Dentro de este proceso de transformación, Pragma se ha hecho un espacio importante y en constante crecimiento, siendo una empresa dedicada a la renovación de los mercados digitales a través de la distribución de talento, siendo su principal tarea el proveer soluciones de software a los problemas que poseen los clientes o Pragma en sí misma, en este caso, la forma en que se administran los clientes y su respectiva información. Hoy en día existen distintos problemas derivados de implementaciones antiguas que hacían uso de API's, frameworks o arquitecturas obsoletas, tal como el uso de Spring a través de archivos xml, la implementación de aplicaciones monolíticas, bases de datos ineficientes o código complejo y difícil de comprender, además de que no se han explorado mayormente las capacidades de AWS para el desarrollo de API's de manera directa. Por lo anterior se propuso, en vez de un proceso de refactorización, la creación de dos conjuntos de API's con la capacidad de administrar la información de clientes con la idea implícita de trabajar con las ultimas herramientas disponibles a nivel de frameworks y del lenguaje propiamente. En total se crearon 3 API's: la primera basada en Java para administrar la información de clientes en bases de datos MySQL e imágenes en MongoDB; la segunda, una en la plataforma AWS que realiza las mismas operaciones sobre bases de datos DynamoDB y S3. La tercera, se desarrolló adicionalmente una API Java que es capaz de consumir a las demás e integrar las ventajas que posee cada una de ellas. Respectivamente se incorporaron, características para aumentar la robustez, la confiabilidad y facilidad tanto de uso como de mantenibilidad, a través de la implementación de frameworks como Lombok para la simplificación

de código, Spring Cloud para descentralización, Swagger para documentación, Spring Security para la seguridad y la última versión disponible de Java 17. En el caso de la API en AWS, se utilizó la última versión del SDK provisto por Amazon para el desarrollo y uso de sus servicios a través de Python.

2. Objetivos

2.1. Objetivo General

Desarrollar dos conjuntos de API's Restful, mediante el uso de los lenguajes de programación Java y Python, gestores de bases de datos y los servicios de la plataforma Amazon Web Services, para exponer los servicios CRUD (creación, lectura, actualización y borrado) para la gestión de dos bases de datos, una para la administración de datos de clientes y otra para el almacenamiento de imágenes.

2.2. Objetivos Específicos

- Definir e implementar el primer conjunto de API's, mediante el lenguaje de programación Java, que expongan las operaciones CRUD para el manejo de dos bases de datos, una relacional (MySQL) para la administración de datos de cliente y una no relacional (MongoDB) para el almacenamiento de imágenes.
- Desarrollar el microservicio para cada operación CRUD y el primer conjunto de API's definidas previamente, mediante el framework Spring Boot (Java), con características de seguridad y descentralización.
- Definir e implementar el segundo conjunto de API's, mediante el lenguaje de programación Python, que expongan las operaciones CRUD para el manejo de dos bases de datos no relacionales, una (DynamoDB) para la administración de datos de cliente y otra (S3) para el almacenamiento de imágenes.
- Desarrollar el microservicio para cada operación CRUD y el segundo conjunto de API's definidas previamente, mediante los servicios de la plataforma Amazon Web Services AWS y el lenguaje de programación Python, con características de seguridad y descentralización.
- Realizar un conjunto de pruebas unitarias, mediante el uso de JUnit y PostMan, para comprobar el correcto funcionamiento de los microservicios y API's creadas.

3. Marco Teórico

Una interfaz de programación de aplicaciones (API), "Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas" [1].

"Los microservicios son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son elementos independientes que funcionan en conjunto para llevar a cabo las mismas tareas" [2]. En este caso particular, los microservicios son aplicaciones web implementadas como aplicaciones Spring Boot, que permiten la integración de funciones y trabajo colaborativo en paralelo a partir de acuerdos, los cuales definen parámetros de entrada y salida para el desarrollo de proyectos en conjunto, siendo estos independientes de IDE, estilos e incluso, de plataforma.

"En la ingeniería de software se denomina aplicación web a aquellas herramientas que los usuarios pueden utilizar accediendo a través de internet o de una intranet mediante un navegador. En otras palabras, es un programa que se codifica en un lenguaje interpretable por los navegadores web en la que se confía la ejecución al navegador" [3].

La arquitectura Model-View-Controller (MVC) separa la estructura de datos, la interfaz de usuario y la lógica de servicio de manera que funcionan de manera conjunta pero no centralizada, permitiendo trabajarlas de manera independiente. En este proyecto particular se trabajará sobre el controlador (REST) [4] y el modelo (Bases de datos MySQL y MongoDB) los cuales se conocen como el back-end.

Se escogió la arquitectura de servicios REST (Representational State Transfer) para la implementación de las API's, los cuales exponen a través de internet una serie de recursos o endpoints a los que se puede acceder a través de peticiones HTTP (GET, PUT, POST, DELETE, etc.), para solicitar datos o generar operaciones sobre los mismos. Cada recurso se identifica por su URL, método HTTP, parámetros, tipo de respuesta y tipo de dato consumido.

Las API's Rest están compuestas por un conjunto de microservicios basados en la

arquitectura Rest, cada uno es una unidad de software que implementa una funcionalidad concreta y se ejecuta de forma independiente, es decir, incluye todo lo necesario para su ejecución y no requiere de ningún software adicional.
Java es un lenguaje de programación cuya sintaxis deriva ampliamente de C y C++ y sigue el paradigma de programación orientada a objetos, caracterizado por ser sencillo, distribuido, seguro e independiente de la arquitectura de la máquina. Uno de los usos más comunes de Java es la creación de aplicaciones web, debido a que puede de ejecutarse en cualquier dispositivo que posea el ambiente de desarrollo Java, su capacidad de comunicarse con un gran número de sistemas, conectividad a bases de datos y amplio soporte. En este proyecto se utilizó Java para la implementación de 2 de las 3 API's creadas, debido a su capacidad y amplio uso en el desarrollo de API's Rest.
Para el desarrollo de servicios web en Java es muy común utilizar el Framework Spring [5], ya que este "ofrece como elemento clave el soporte de infraestructura a nivel de aplicación, brindando un completo modelo tanto para la configuración como para la programación de aplicaciones empresariales desarrolladas bajo Java, sin discriminación en cuanto al despliegue de la plataforma" [14].
De Spring se derivan varios Frameworks muy útiles como lo son Spring JPA, Spring Security, Spring Cloud y Spring Boot, los cuales son implementados dada la necesidad de acceso a datos, securización, descentralización de servicios e implementación de microservicios.
El desarrollo de una de las API's se realizó sobre la plataforma Amazon Web Services la cual "es una colección de servicios de computación en la nube pública (también llamados servicios web) que en conjunto forman una plataforma de computación en la nube, ofrecidas a través de Internet por Amazon.com" [6]. Algunos de los servicios utilizados son Identity and Access Management IAM para autenticación, Virtual Private Cloud VPC para definición de entornos de red, DynamoDB para almacenamiento de datos, Simple Storage Service S3 para almacenamiento de imágenes y Lambda para implementación de lógica. La implementación de esta API se dio con el objetivo de participar del proceso de aprendizaje planteado en Pragma para adquirir capacidades sobre una de las plataformas más potentes y utilizadas actualmente, como es AWS.
Las funciones de la API sobre los servicios AWS son ejecutadas mediante Lambdas,

“AWS Lambda es un servicio de informática sin servidor que ejecuta código en respuesta a eventos y administra automáticamente los recursos informáticos subyacentes” [7]. Lambda puede usarse para ampliar o integrar los servicios de AWS con lógica personalizada o crear servicios back-end con seguridad, rendimiento y escala requerido. “Lambda ejecuta el código en una infraestructura informática de alta disponibilidad y se encarga de la administración integral de los recursos informáticos, incluido el mantenimiento del servidor y del sistema operativo, el aprovisionamiento de capacidad y el escalado automático, la implementación de parches de seguridad y código, así como la monitorización de código y los registros. Lo único que tiene que hacer es proporcionar el código” [7]. Lambda soporta los lenguajes NodeJs, Python, Java, Go y C#.

Para la gestión de código en el proyecto se implementó un sistema de control de versiones, “también conocido como "control de código fuente", es la práctica de rastrear y gestionar los cambios en el código de software” [17]. Esto con el objetivo de administrar cambios que fueran surgiendo durante el desarrollo y de garantizar estados funcionales del código. Adicionalmente, permite el desarrollo simultaneo de características a través de trabajo distribuido y el acceso universal al respaldo de los datos.

4. Metodología

Este proyecto consistió en el desarrollo de dos API's para la administración de información e imágenes de clientes de Pragma, la primera basada en Java con el framework Spring Boot, la segunda, sobre Python y la plataforma AWS. Adicionalmente se desarrolló una API cliente para el consumo e integración de las demás. La metodología que se implementó en este proceso consistió en una serie de pasos, análisis y desarrollos individuales de cada característica propuesta sobre los microservicios que componen cada una de las API's. Cada decisión sobre la forma en que se implementarían estas características fue tomada basada en razones puntuales.

4.1. Entorno de desarrollo para las API's Java

Para la creación y desarrollo de los microservicios que componen las API's, se utilizó el entorno de desarrollo Intellij IDEA de JetBrains. Este IDE destaca particularmente por la gran cantidad de herramientas que provee a la hora de desarrollar, tales como autocompletación de código, integración completa con el sistema de control de versiones Git, debugger, integración de frameworks, asistentes de configuración y herramientas de visualización. Existen diversas formas de programar en Java, completamente desde cero o implementando frameworks que agregan configuraciones iniciales y funciones que ya han sido implementadas. En este caso particular, se trabajó sobre el Spring, un framework open source para el desarrollo de aplicaciones en Java caracterizado por su amplio uso, flexibilidad, facilidad, eficiencia, seguridad y amplio soporte.

Las aplicaciones web en Java basadas en Spring deben ser ejecutadas en un servidor para poder funcionar, por lo que inicialmente se consideró utilizar un servidor con Tomcat para ejecutar el proyecto. Spring en su forma más simple requiere una gran cantidad de configuración manual a partir de archivos XML, que escalan la complejidad del proyecto en un ámbito que no es relevante. Debido a esto, se optó por utilizar el framework Spring Boot para le ejecución del proyecto, el cual fue desarrollado para evitar la molestia de crear servidores que ejecuten aplicaciones Java basadas en Spring. Adicionalmente, provee gran facilidad de control de Beans y componentes, de manera que el proyecto se ejecuta como una

aplicación Spring Boot en un servidor Tomcat embebido y sus componentes se instancian a partir de anotaciones.

Después de que se definió que el proyecto sería una aplicación web Spring Boot basada en Spring Web MVC, se utiliza la herramienta Spring Initializr, embebido en Intellij Idea para crear los distintos microservicios. La herramienta de inicialización permitió la facilidad de escoger características del proyecto como el tipo, lenguaje y versión, empaquetamiento y la elección de los frameworks con los que se trabajó.

El primer conjunto de API's es el que se ejecuta sobre el lenguaje Java, para esto se decidió utilizar la versión 17 de Java, siendo la última versión estable, posee las ultimas características y ventajas desarrolladas. Se optó por crear el proyecto con Maven como herramienta de gestión y construcción de proyectos en vez de Gradle, dado que en su momento se contaba con más experiencia con el primero.

Dado que el proyecto es de tipo Maven, las dependencias de los microservicios se manejan a través de un archivo XML llamado pom, el cual contiene la información de paquetes, las dependencias y la versiones de estas. Igualmente, Maven es el encargado de indexar los componentes del proyecto y realizar la construcción y despliegue de cada microservicio.

Una vez inicializados los microservicios se creó una distribución de carpetas básica, cada uno se ejecuta de manera independiente con un modelo de carpetas en común, siendo este manejado como se presenta en la figura 1. La estructura de carpetas se explica a continuación:

- Carpeta del proyecto: Contiene todas las carpetas y los archivos que contienen el código a ejecutar, además de los archivos de configuración y los generados durante la ejecución.
- src: Es la carpeta "Source" donde se encuentran todos los archivos con el código a ejecutar.
- Main: Contiene los archivos necesarios para la ejecución del microservicio.
- Test: Contiene los archivos de pruebas del microservicio.
- Java: Contiene los archivos de código Java que ejecutara el microservicio.
- Resources: Contiene los archivos de configuración del microservicio.

- ApplicationConfig: Contiene las clases que servirán como configuración de los frameworks implementados y la clase principal del microservicio.
- Business: Contiene las interfaces y clases que ejecutan la lógica de negocio de las operaciones del microservicio.
- Domain: Contiene los Data Transfer Objects (Dto).
- Exception: Contiene las clases de excepción que son generadas de manera controlada.
- Model: Contiene las entidades para acceso a las tablas de la base de datos.
- Repository: Contiene las clases e interfaces que implementan la lógica de acceso a base de datos.
- Rest: Contiene las clases que definen los endpoints del microservicio.
- Útil: Contiene utilidades varias a utilizar en las demás clases como validadores y constantes.

Para el caso del microservicio cliente se incluye además una carpeta provider con los archivos de configuración de Feign y la clase proveedor de servicios AWS.

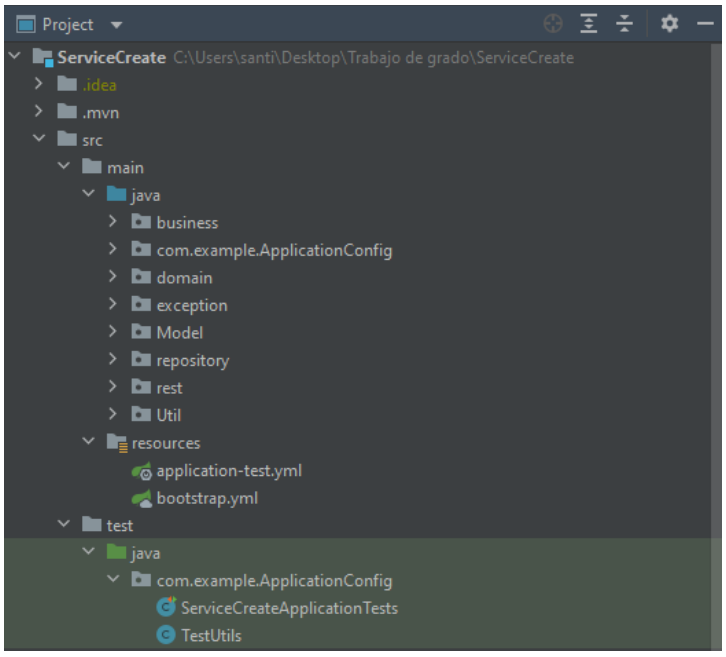


Figura 1. Ejemplo de distribución de carpetas en el microservicio de consultas.

Después de crear e inicializar los microservicios, se implementó Git como sistema de control de versiones debido a su facilidad de uso y experiencia previa trabajando con él. La integración de Git con el IDE y la facilidad que provee la plataforma de GitHub para alojar proyectos de manera remota agrega un respaldo constante para los avances del proyecto en caso de inconvenientes con el PC o problemas introducidos por errores de código derivados del desarrollo.

4.2. Primer conjunto de API's

Antes de crear el primer microservicio, se escogió el patrón de arquitectura MVC debido varias características que incluye como la distribución de funciones y estructuras de código según su función. La estructuración generalizada y la reutilización de código permiten aumentar la velocidad en el desarrollo del proyecto y simplifican el mantenimiento de este. Adicionalmente, la arquitectura permite el trabajo colaborativo y el enfoque en características particulares, como por ejemplo la vista y la administración de datos.

De esta manera, se desarrolló el proyecto bajo el patrón de arquitectura MVC, basándose casi exclusivamente en el modelo y el controlador de este. Adicionalmente, se creó una vista básica capaz de consumir la información provista a través del framework Swagger [19].

Ya teniendo el IDE, el framework base y la arquitectura, se decidió el uso de Spring Web MVC [20], subproyecto del framework base Spring. Este framework facilita la creación de proyectos basado en un DispatchServlet que redirige las peticiones hacia el manejador, generalmente un controlador que en este caso es de tipo Rest, como se observa en la figura 2.

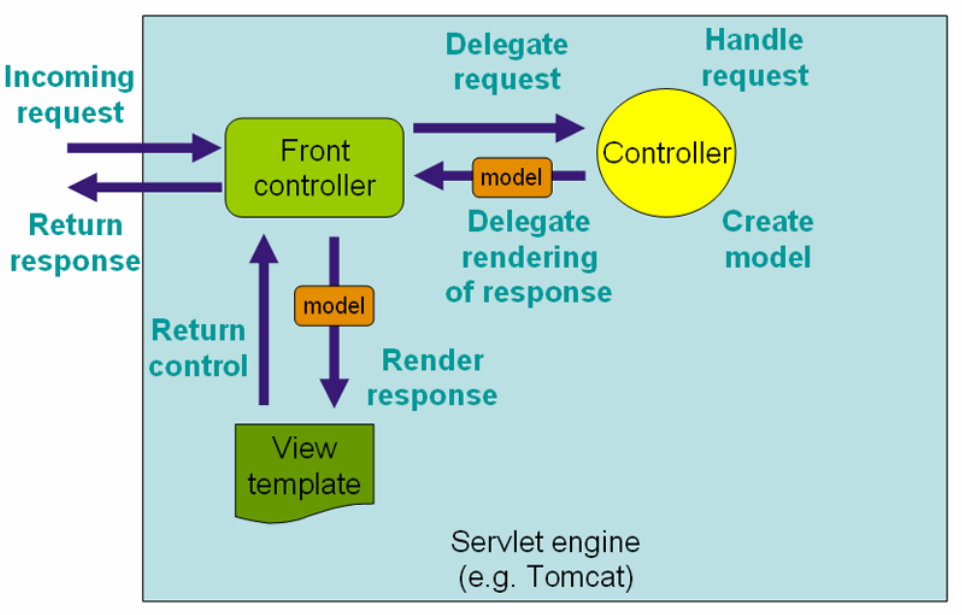


Figura 2. Flujo de trabajo de procesamiento de peticiones en Spring MVC [11].

Como se mencionó en la propuesta de este proyecto, inicialmente se decidió la implementación de 2 bases de datos separadas, una relacional SQL que se ejecutaría en el servidor MySQL Server [22] y se supervisaría a través de la herramienta MySQL Workbench y una no relacional que se ejecutaría en MongoDB Community [23] Server y se supervisaría a través de MongoDB Compass.

La base de datos MySQL contiene 2 tablas, una correspondiente a la información de los clientes, sobre la cual se hicieron la mayoría de los desarrollos, y una tabla de pedidos, la cual se implementó para añadir la funcionalidad de consultar información a través de 2 tablas relacionadas a través del campo de identificación del cliente.

La tabla de clientes contiene los siguientes campos:

- Nombres
- Apellidos
- Tipo de identificación.
- Número de identificación, que se usa como llave primaria.
- Edad
- Ciudad de nacimiento
- Foto (Archivo base64)

Como se puede ver en las figuras 3, 4 y 5, se crearon bases de datos de pruebas para el desarrollo del proyecto.

idCliente	nombres	apellidos	docType	edad	ciudad
0	santiago	alvarez	cc	24	med
1	santiago	alvarez	cc	24	med
2	veronica	alvarez	cc	27	med
19	santiago	alvarez	cc	24	med
20	david	pinzon	ti	2	med
22	santiago	alvarez	cc	24	med
27	santiago	alvarez	cc	24	med
28	santiago	alvarez	cc	24	med
29	santiago	alvarez	cc	24	med
55	daniela	alvarez	cc	24	med
100	santiago	alvarez	cc	24	med
123	sebastian	ortega	ti	40	med
124	sebastian	ortega	ti	40	med
125	santiago	alvarez	cc	24	med
128	santiago	alvarez	cc	24	med
129	santiago	alvarez	cc	24	med
465	david	sasf	ti	23	med
666	samuel	alvarez	ti	24	med

Figura 3. Ejemplo de contenido de la tabla de clientes.

La tabla de pedidos consta de los siguientes campos.

- Id del pedido.
- Id del cliente.
- Valor total.
- Fecha de creación.

id	id_cliente	valor	fecha_creacion
2	12	1	2021-12-11
3	1	1	2021-12-11
4	1	1	2021-12-11
5	1	1	2021-12-11
NULL	NULL	NULL	NULL

Figura 4. Ejemplo de contenido de la tabla de pedidos.

La tabla en la base de datos en MongoDB contiene objetos cuya información corresponde a objetos foto de los clientes asociados mediante el campo clientId y contiene los siguientes campos:

- Id del objeto en la tabla, como llave primaria.
- clientId, el número de identificación del cliente asociado a la foto, el cual se define como un índice en la base de datos.
- Image, correspondiente al binario de la foto.
- Class, clase (foto) del objeto.

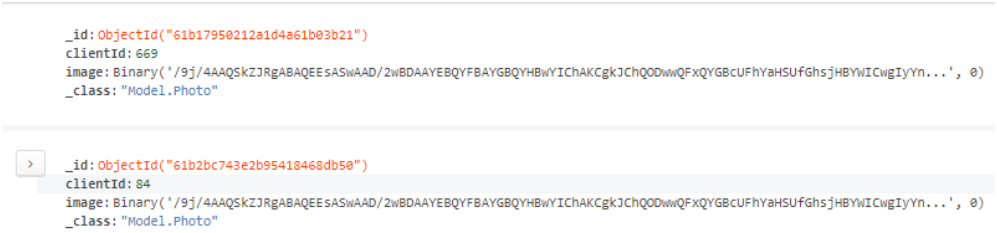


Figura 5. Ejemplo del contenido de la tabla de fotos.

El campo de clientId se definió como índice de manera que se pueda hacer consultas de objetos a partir de este; adicionalmente garantiza que no exista más de una foto asociada al número de identificación de un cliente.

El proceso de creación de las tablas de las bases de datos se pudo hacer de 2 formas, a través de una interfaz gráfica respectiva (MySQL Workbench o MongoDB Compass), los cuales presentan todas las opciones aplicables de manera sencilla e intuitiva. La segunda opción, es implementar las entidades o modelos en Java y configurar Hibernate, de manera que al ejecutar cualquier microservicio este se encargue de crear las bases de datos y las tablas a partir de las propiedades definidas en las clases.

Durante el desarrollo del proyecto se consideró migrar las bases de datos a la nube, de manera que se evite perdidas de información. Por esto, se implementó el servicio RDS (Relational Database Service) de AWS, que se encuentra en la capa gratuita, para generar una copia de los datos existentes en la base de datos local de clientes. El único cambio que se realizó fue la url y las credenciales de la base de datos sobre la que se consultaba en los microservicios, todo a partir de los archivos de configuración que se encuentran en el repositorio del servidor de configuración. Igualmente se consideró el mismo cambio para la base de datos MongoDB, pero el servicio tanto en AWS como en la plataforma MongoDB Atlas conlleva un costo que no se había contemplado antes. Dado lo anterior se optó por usar archivos de copia de seguridad para la base de datos no relacional.

Una vez creadas las bases de datos y los microservicios, se definieron una serie de características comunes en estos como clases, métodos, patrones de diseño, etc.

Dado que la función principal de los microservicios que componen el conjunto de API's es implementar las funciones CRUD, inicialmente se implementaron los modelos o entidades. Estos objetos son similares a una clase común de java, poseen propiedades, constructores y métodos de acceso como cualquier otra, la diferencia es que su función es casi exclusivamente para representar y definir las propiedades de una tabla en la base de datos, de manera que a partir de instancias de estas se pueda consultar o crear registros en las bases de datos. Ahora bien, a diferencia de las demás clases de java, las entidades requieren una serie de anotaciones que indican a JPA la forma en que debe ser administrada la clase, estas son:

- @Entity, le indica a JPA que la clase es una entidad u objeto de persistencia, además que le indican al motor de procesamiento de anotaciones de Spring el tipo de componente de la clase.
- @Table, permite definir el nombre de la tabla en la base de datos a la que está asociada la clase.

Adicionalmente se le debe indicar al motor de Hibernate a que campos están asociados las propiedades de la clase y marcar cuál de estas es la llave primaria, para esto se utiliza las anotaciones @Column y @Id. La primera se utiliza sobre cada propiedad y se incluye el nombre del campo en la tabla, la segunda es única y debe existir en todas las entidades.

Como se mencionó antes, se tienen 2 bases de datos, en la base de datos de clientes se tienen 2 tablas, una de pedidos y una de clientes, en la base de datos de imágenes se tiene una colección con los objetos de foto, por lo que se implementan 3 entidades, cliente, pedido y photo.

Para el caso de la entidad de cliente, dado que se utiliza el id del cliente como llave primaria es obligatorio que esta siempre este incluida en las peticiones de creación o modificación de registros. Por otro lado, las entidades de photo y pedido se crearon con llaves primarias autogeneradas, tal que no es necesario que se incluyan en las peticiones, ya que Hibernate se encarga de generar un id a partir de un método seleccionado.

Antes de empezar a desarrollar consultas o lógica de los microservicios, se definió el protocolo de peticiones y respuestas que utilizaron los

microservicios. Como se mencionó en la propuesta, se utiliza Rest junto el protocolo de transferencia de hipertexto Http, de manera que el tipo y el contenido de la petición define el comportamiento de la aplicación y su debida respuesta.

Se utilizan los 4 tipos de petición más comunes:

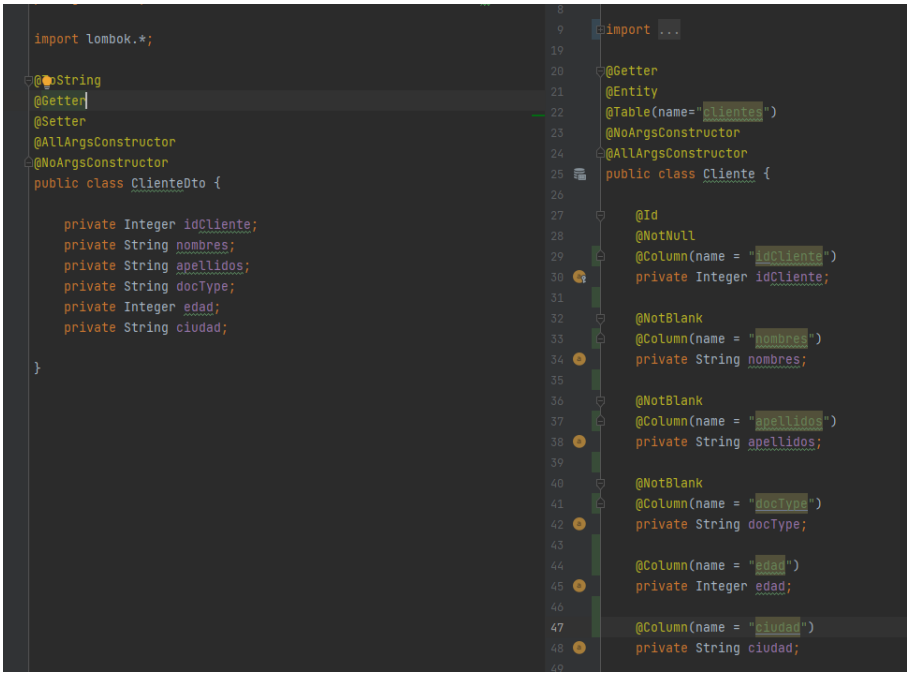
- GET, para realizar consultas de datos.
- POST, para la creación de registros en las bases de datos.
- PUT, para la actualización de registros.
- DELETE, para borrar registros.

Para cada tipo de petición se generó un endpoint que posteriormente es asociado a un método en la lógica de negocio, de manera que pueda generar una respuesta a partir de objetos de transferencia de datos (Dto). Al realizar una operación a partir de una petición se debe retornar una respuesta que de información acerca de lo que ha ocurrido del lado del servidor. Cada respuesta debe informar de manera clara si la petición fue exitosa o si por el contrario ocurrió un error y su debida causa. El protocolo http ofrece respuestas a partir de objetos de tipo httpEntity, los cuales incluyen información como headers, estado de la petición, mensaje y cuerpo. Con el fin de aumentar la flexibilidad a la hora de manipular las respuestas que provee el servidor, se implementó una clase de tipo Dto llamado ResponseDto, la cual cuenta con propiedades similares a httpEntity, un código entero del estatus de la petición, un código de respuesta, un mensaje y un valor genérico de datos. Este último es un objeto genérico de clase T, de manera que permite incluir en la respuesta a la petición un objeto de cualquier tipo, por ejemplo, un objeto de tipo cliente cuando se realice una consulta.

Para poder recibir o responder apropiadamente las peticiones, se definen Dto's diferentes al ResponseDto, para poder transferir información de la operación realizada. A pesar de que se puede realizar la transferencia con las entidades, se considera una mala práctica dado que se expone las estructuras internas de datos de la aplicación.

Dado lo anterior se crean 2 Dto's, el de respuesta y el de cliente, siendo el segundo muy similar al modelo, con la diferencia de que no cuenta con las

anotaciones que describen el acceso a datos, tal como se observa en la figura 6.



```java
import lombok.*;

@ToString
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class ClienteDto {

    private Integer idCliente;
    private String nombres;
    private String apellidos;
    private String docType;
    private Integer edad;
    private String ciudad;

}
```

```java
import ...

@Getter
@Entity
@Table(name="clientes")
@NoArgsConstructor
@AllArgsConstructor
public class Cliente {

    @Id
    @NotNull
    @Column(name = "idCliente")
    private Integer idCliente;

    @NotBlank
    @Column(name = "nombres")
    private String nombres;

    @NotBlank
    @Column(name = "apellidos")
    private String apellidos;

    @NotBlank
    @Column(name = "docType")
    private String docType;

    @Column(name = "edad")
    private Integer edad;

    @Column(name = "ciudad")
    private String ciudad;
```

Figura 6. Dto cliente (Izquierda) y modelo Cliente (Derecha).

Adicionalmente el uso de Dto's agregar una capa de abstracción a la aplicación, separando los objetos de la capa de presentación de la capa de servicio, de manera que se puedan realizar cambios de manera independiente en cualquiera de las dos, sin afectar el funcionamiento de la aplicación.

Las variables o campos pertenecientes a una clase (Entidad o Dto) son fundamentales a la hora de realizar implementaciones en java, por lo que su correcta administración se vuelve una materia de gran importancia. Se tuvo como convención utilizar variables anotadas con el modificador private, el cual es el más restrictivo en temas de acceso, de forma que la consulta y modificación de estos valores sea única y exclusivamente a través de métodos propios de la clase. Estos métodos son llamados getter y setter, la implementación de estos garantiza que durante la ejecución siempre se es consciente desde que instancias se accede a determinado valor, tal que no ocurran inconvenientes a la hora de utilizarlo. Para la implementación de estos métodos se genera un método público que

actúa sobre los campos de la instancia, siendo necesarios uno para cada uno tanto para lectura como para escritura. Dado lo repetitivo y extenso de este trabajo, se optó por utilizar anotaciones del framework Lombok [21] con las respectivas anotaciones @Getter y @Setter para generarlos en cada clase.

En el caso de los constructores de igual manera se generan a través del framework, ya que tiene la capacidad de generar constructores apropiados según la necesidad, tales como @NoArgsConstructor, @RequiredArgsConstructor o @AllArgsConstructor.

Con el fin de verificar que el cuerpo de una petición cumpla con los estándares predefinidos y contenga los datos necesarios para ejecutarse correctamente en el servicio, se implementa una clase validation utils. Esta clase posee un método el cual, a través de un objeto de tipo Validator, verifica que los objetos de la petición sean válidos y en caso contrario, genera una excepción con una respuesta apropiada. Las reglas de validación son definidas en el Dto para cada campo mediante anotaciones, por ejemplo, al campo id se le agrega @Notnull, para que no pueda ser nulo.

Al recibir una petición, el cuerpo de esta puede contener un Dto el cual no puede ser usado para realizar operaciones sobre bases de datos, de la misma manera que al consultarla no se puede enviar una entidad al cliente de la petición. Para solucionar este problema, se implementó una capa de abstracción usando los Dto's para la transferencia de datos únicamente y las entidades para acceso a datos. Como interfaz entre estos, se implementaron métodos que puedan realizar la conversión de uno a otro. Para esta tarea se contó con las siguientes opciones:

- Implementar una clase de conversión, la cual de manera manual creara una instancia del objeto salida y a partir de operaciones get y set asignara los valores obtenidos a partir del objeto entrada.
- Implementar una interfaz MapStruct, la cual a partir de la anotación @Mapper y una pareja de entidad y Dto, implementa los métodos de conversión bidireccionales entre estos.

Se opto por la segunda, ya que el proceso de creación de los métodos del

primer caso resulta engorrosos y lentos, además de que extienden innecesariamente el código. Adicionalmente, aumenta mucho la complejidad del mantenimiento, debido a que realizar una creación, modificación o borrado de un campo por cualquiera de los dos lados, genera inevitablemente la necesidad de modificar el convertidor de manera manual con el fin de evitar errores o excepciones.

A partir de esto, se generan 4 métodos que permiten realizar la conversión entre objetos de tipo cliente y clienteDto de manera individual o a través de una lista de estos.

Una vez creados los endpoints, el siguiente paso es implementar la lógica correspondiente a la operación de este. Para esto, se requieren los métodos de acceso a datos. Dado que se tienen dos bases de datos de tipo totalmente distinto, se hizo necesaria la implementación de consultas de forma aparte, implementando los frameworks de Spring Data para la base de datos no relacional MongoDB y Spring JPA para la base de datos relacional MySQL así:

- Para realizar consultas a la tabla clientes sobre la entidad cliente y pedidos, se realizó la creación de interfaces que extienden a JpaRepository, el cual permite implementar los métodos de acceso a la base de datos. Además, se incluye la anotación @Repository, de manera que esta se convierta en un objeto de acceso a datos (Dao) e indicar a Spring que este se usara para el acceso a datos. Dado que es una base de datos SQL, se puede realizar consultas a las tablas a partir del repositorio de 3 formas distintas:
    - Criteria JPA, implementadas programáticamente.
    - Consultas por nombre, implementadas por Spring a partir del nombre del método.
    - Consultas manuales, definidas a través de la anotación @Query, que pueden ser nativas de SQL o en JPQL.

  Se opto por utilizar principalmente consultas por nombre, debido a que no se requiere implementar cada método manualmente, lo que evita fallos que pueden surgir en el mismo. Adicionalmente se incluyen una consulta en JPQL para aumentar la flexibilidad de los microservicios. Se descarto la implementación de consultas en

Criteria debido a que la ventaja que ofrecen es para la realización de consultas variables, las cuales no son usadas en este caso.

- Para las consultas en la base de datos sobre la entidad Photo, se define una interfaz que extiende a MongoRepository junto a la anotación @Repository, de manera similar al repositorio de clientes. Para realizar consultas sobre la tabla se tienen los siguientes métodos:
    - Consultas por nombre, implementadas por Spring.
    - Criteria, para expresar las consultas que se realizan a través de un cliente Mongo.

  En este caso se opta por utilizar las consultas por nombre, ya que al igual que las consultas de clientes, Spring implementa los métodos a partir de los parámetros obtenidos a partir del nombre del método, adicionalmente Criteria agregaría un grado de complejidad innecesario para este proyecto.

Los resultados de ambos tipos de consultas pueden retornar entidades directamente u objetos de clase Optional, que permiten manejar los resultados en caso de que sean o no nulos. Se optó por usar objetos Optional para facilitar el lanzamiento de excepciones controladas, las cuales permitieron simplificar el flujo del código y generalizar los métodos de respuesta. De esta manera se pudo definir el comportamiento de la aplicación en caso de que alguna consulta produzca resultados nulos, por ejemplo, si al realizar la consulta de un cliente a través de su id, este no es encontrado, se utiliza métodos OrElse con el fin de interrumpir el flujo y generar una respuesta acorde.

Una vez creados los métodos de acceso a datos, se deben instanciar los repositorios en la clase bussines para realizar las operaciones CRUD según el microservicio, el endpoint y el respectivo método.

Dado que se lanzan excepciones en los métodos de lógica de negocio, se deben crear métodos para manejarlas. A diferencia de los errores, las excepciones pueden ser controladas, tal que, si se genera alguna durante la ejecución, no se detenga la operación y el programa pueda responder adecuadamente. Para esto se dispuso una lógica común en los microservicios, con el fin de asegurar que se pueda generar mensajes,

códigos y objetos que indiquen tanto al desarrollador como a un usuario, la información necesaria para entender que está ocurriendo y una idea de cómo evitar que se reproduzca la misma excepción. Para controlar el flujo de la aplicación se realizaron 2 tareas, primero se creó una clase llamada ServiceException que extiende de la clase RuntimeException, la cual corresponde a errores generados por el programador, por ejemplo, divisiones por cero, índices fuera de rango o como en este caso, una excepción personalizada. La clase excepción fue creada con el objetivo de describir completamente errores o comportamientos no deseados que se produzcan, de manera que se pueda responder de manera particular a cada uno.

La segunda tarea, es el manejo de las excepciones generadas, para esto se utiliza la cláusula try/catch, cuyo funcionamiento es sencillo, ya que permite decidir que secciones del código manejar a través del try, ya que pueden generar una excepción. La cláusula catch permite ejecutar una sección de código que en caso de que se genere la excepción cuyo tipo sea igual al argumento la cláusula. El argumento de esta sección de código es la excepción generada, la cual es utilizada para generar el objeto respuesta.

Para todas las lógicas de negocio implementadas se utilizó la misma estructura, un try que contenga el código a ejecutar, una cláusula catch para interceptar las excepciones de tipo ServiceException y otra de tipo Exception, la cual es la clase padre de todas las demás excepciones. La cláusula catch generalizada, se implementó con el objetivo de que la aplicación responda a las excepciones cuyo origen sea totalmente desconocido. Un ejemplo de esta estructura de código es la figura 7.

```
try {
    Cliente client = clientRepository.findById(clientId).orElseThrow(() -> new ServiceException(HttpStatus.NOT_FOUND.value(),
            ServiceConstants.SA001, ServiceConstants.SA001M));
    response = new ResponseDto<>(HttpStatus.OK.value(), ServiceConstants.SA000, ServiceConstants.SA000M, clienteMapper.clienteToClienteDto(client));
}catch (ServiceException e) {
    LOGGER.error("Error in findClientById", e);
    response = new ResponseDto<>(e.getStatus(), e.getCode(), e.getMessage());
}catch (Exception e){
    LOGGER.error("Error in findClientById", e);
    response = new ResponseDto<>(HttpStatus.INTERNAL_SERVER_ERROR.value(), ServiceConstants.SA100, ServiceConstants.SA100M);
}
```

Figura 7. Ejemplo de la implementación de cláusulas try/catch.

Una vez implementada la lógica del endpoint, se debe generar el objeto respuesta, el cual incluye códigos y mensajes de respuesta. Para esto se

incluyó una lista personalizada de respuestas que dan información acerca de la ejecución, por ejemplo, un error de consulta en la base de datos o advertir que el cliente no existe. Este proceso simplifica la forma de hacer debug al código y facilita el entendimiento de las respuestas en caso de que se quisiera desarrollar una vista que consuma los microservicios implementados.

Los mensajes y códigos de respuesta fueron implementados como campos estáticos de la clase ServiceConstants en la carpeta útil, de manera que pueden ser consultados sin necesidad de crear una instancia de la clase.

Como característica adicional, se implementaron loggers en la lógica de negocio de cada microservicio, los cuales simplifican la impresión de información relevante durante la ejecución. En caso de presentarse excepciones, se puede mostrar la información de esta al desarrollador o a un tercero. La implementación de loggers puede realizarse de manera manual, creando un objeto Logger o a través de la anotación @slf4j para generarlo automáticamente. En este caso se realizó de manera manual, aunque no represente una ventaja significativa.

Una vez creados los endpoints y la lógica, se implementó la configuración de seguridad de los microservicios, para esto se utilizó el framework Spring Security para restringir el acceso tanto a recursos como métodos de manera particular. Para configurar el framework, se crea una clase de configuración anotada con @EnableWebSecurity, tal que sobrepase los métodos por defecto para crear 2 usuarios de prueba en memoria "user1" y "admin". Cada uno de estos tendrá acceso a distintos recursos según el microservicio.

Finalmente, con el fin de documentar cada microservicio se empleó el framework Swagger, el cual permite crear una interfaz gráfica que actúa como cliente de este e incluye la documentación de cada endpoint y modelo. Para configurar Swagger se creó una clase de configuración anotada con @EnableSwagger2 y @configuration, de manera que puedan generar un bean para determinar el comportamiento de la interfaz. Para describir el comportamiento de cada endpoint generado, se incluyeron las siguientes anotaciones:

- @Operation para definir el comportamiento o función del endpoint.
- @ApiResponses para anotar las distintas respuestas que se pueden generar.
- @Parameter para definir cada parámetro que requiere el endpoint.

Adicionalmente se incluyó una descripción de cada uno de los métodos, clases e interfaces implementados en el código.

El acceso a la interfaz se realiza a través la Url y puerto del microservicio junto con el path "/swagger-ui.html#/client-rest".

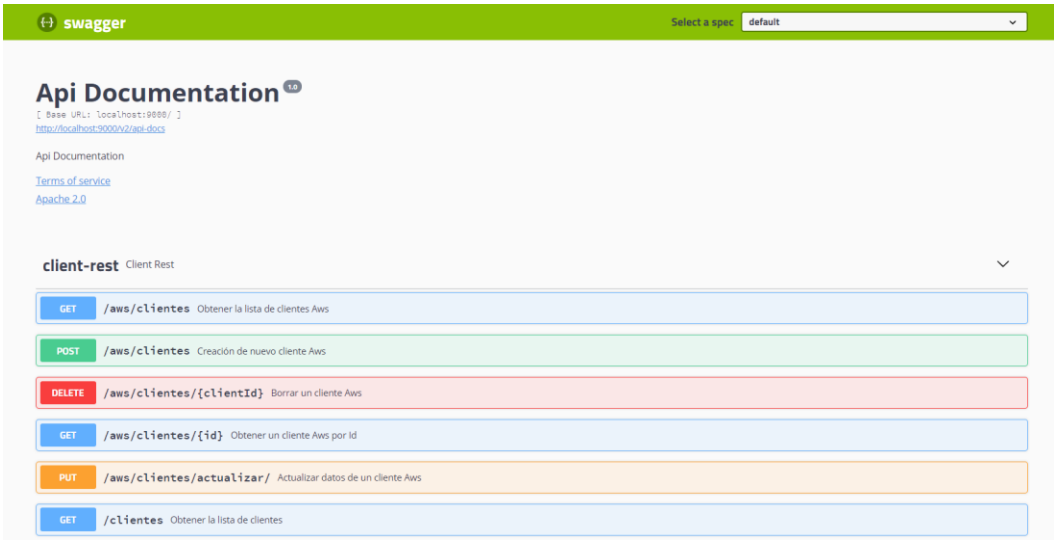


Figura 8. Ejemplo de la interfaz de documentación de la API cliente.

4.3. Microservicios para el CRUD y el primer conjunto de APIs

4.3.1. Servidor de configuración

El servidor de configuración, como se mencionó brevemente en la sección 4.2, tiene una única tarea, proveer de los archivos de configuración a cada uno de los microservicios que se ejecuten en el host. Para esto requiere una configuración particular y totalmente autónoma, además de que posee una distribución de carpetas distinta y archivos a la mencionada inicialmente, debido a que por sí solo no posee muchas de las características de los otros microservicios, de esta manera:

- Carece de acceso a base de datos y por tanto de entidades y repositorios.
- No hace transferencia de objetos, por lo que no usa ningún Dto.
- Dado que no ejecuta ninguna lógica diseñada propiamente, carece de business y de excepciones.
- No recibe peticiones personalizadas, por lo que no hace uso de clases controlador Rest.

El servidor de configuración se basa casi exclusivamente en el framework Spring Cloud Config Server por lo que para crearlo solo se requiere crear una aplicación Spring Boot que cuente con la dependencia y cuya clase principal cuente con la anotación @EnableConfigServer, de manera que el entorno de Spring entienda que la aplicación es un servidor de configuración y pueda instanciarlo de manera correcta. Adicionalmente se requiere que en el archivo de extensión "yml" o "properties" se defina la ubicación del repositorio en GitHub donde se encuentran los archivos de configuración con el nombre correspondiente a cada microservicio:

- service-create
- service-delete
- service-get
- service-modify

Por último, el puerto sobre el cual se ejecuta el servidor es el 8888, debido a que usualmente ese es el que se implementa para configuración.

Como se puede observar en la figura 3, el microservicio realiza un fetch a partir de su nombre para consultar su información de configuración al servidor en la dirección localhost:8888, el cual le responde el archivo correspondiente, el perfil y la ubicación de este.

4.3.2. Servidor Eureka

"Eureka es un servicio REST, utilizándose principalmente en la nube de AWS, a la cual está estrechamente ligado. Eureka se comporta como servidor, cuyo objetivo es registrar y localizar microservicios existentes, informar de su localización, su estado y datos relevantes de cada uno de ellos." [18]. De esta manera y al igual que el servidor de

configuración, el servidor Eureka cuenta con una tarea única y exclusivamente, por lo que no tiene la misma distribución de carpetas y archivos.

Este servidor se basa casi exclusivamente en el framework Spring Cloud Starter Netflix Eureka Server, por lo que solo se requiere una aplicación Spring Boot cuya clase principal contenga la anotación @EnableEurekaServer, lo cual la habilita como un servidor Eureka. Adicionalmente se debe configurar la aplicación a partir del archivo de extensión "yml" o "properties" así:

- Nombre eureka_server.
- El puerto, por defecto 8761.
- El registro con eureka debe ser falso debido a que no puede registrarse a sí mismo.
- La url de servicio /eureka.
- El margen porcentual de renovación en 0.85 por defecto, el cual permite calcular la cantidad de pulsos de vida se espera para cada microservicio.

El servidor eureka cuenta con una interfaz de supervisión que provee información acerca de los servicios que están ejecutándose en el momento, a esta se puede acceder en el enlace http://localhost:8761/.

4.3.3. Microservicio de consultas

Llamado Service-Get en el proyecto, es como indica su nombre, el servicio encargado de realizar las consultas sobre las bases de datos, para lo cual se dispone una serie de endpoints para peticiones de tipo Get, los cuales requieren distintos parámetros que serán utilizados como argumento de los métodos de lógica de negocio, los cuales los utilizarán para realizar las consultas y generar una respuesta o un comportamiento apropiado según el resultado de esta.

las funciones para las cuales se generaron los endpoints junto con sus respectivos métodos son los siguientes:

- Consultar la lista completa de clientes, no posee parámetros.
- Consultar un cliente a partir de su número de identificación, su único parámetro es la variable Path Id.
- Consultar la lista de clientes mayores a cierta edad, su único

parámetro es la variable Path age.

- Consultar una foto a partir de su Id en la base de datos, su único parámetro es la variable Path mongold.
- Consultar una foto a partir de la identificación del cliente al que está asociada, su único parámetro es la variable Path clientId.

Este servicio puede generar respuestas a partir del objeto de respuesta ResponseDto, con códigos de error propios y los siguientes mensajes:

- Error inesperado durante el proceso
- Consulta correcta
- No hay clientes para mostrar
- Solicitud incorrecta. Por favor valide los datos enviados.

4.3.4. Microservicio de creación

Llamado en el proyecto como Service-Create, que como su nombre indica, permite la creación de registros en las tablas de las bases de datos, por lo cual se dispone de una serie de endpoints para peticiones Post, los cuales deben llevar como único parámetro el cuerpo del recurso que será generado.

Las funciones de los endpoint generados y sus respectivos métodos son:

- Creación de un nuevo cliente, requiere como parámetro un cuerpo de tipo ClienteDto.
- Creación de una nueva foto, requiere como parámetros el clientId o id del cliente asociado y el archivo con la foto.

Este servicio puede generar respuestas a partir del objeto de respuesta ResponseDto, con códigos de error propios y los siguientes mensajes:

- Cliente guardado exitosamente.
- El cliente ya existe.
- Solicitud incorrecta. Por favor valide los datos enviados.
- Error inesperado durante el proceso.

4.3.5. Microservicio de modificación

Llamado en el proyecto Service-Modify, permite la modificación de un registro existente en las tablas de las bases de datos, dispone de una serie de endpoints que para peticiones Put, de manera muy similar al servicio de creación, solo requieren como parámetro el recurso a

modificar.

Las funciones de los endpoint generados y sus respectivos métodos son:

- Modificación de un cliente, requiere como parámetro un cuerpo de tipo ClienteDto.
- Modificación foto, requiere como parámetros el clientId o id del cliente asociado y el archivo con la foto.

Dada la similitud con el servicio de creación, se optó por implementar los mismos mensajes de respuesta.

4.3.6. Microservicio de borrado

En el proyecto es llamado Service-Delete, permite el borrado de un registro en las tablas de las bases de datos, atendiendo únicamente a peticiones Delete.

Las funciones de los endpoint generados y sus respectivos métodos son:

- Borrar un cliente a través de su identificación, recibe como único parámetro el clientId del cliente asociado.
- Borrar una foto a través de la identificación del cliente, recibe como único parámetro el clientId del cliente asociado.

Este servicio puede generar respuestas a partir del objeto de respuesta ResponseDto, con códigos de error propios y los siguientes mensajes:

- Cliente borrado.
- No se encuentra información para borrar.
- Error inesperado durante el proceso.

4.4. Entorno de desarrollo para las API's en Python en AWS.

Antes de empezar el desarrollo, se creó una cuenta de AWS, la cual posee acceso a servicios de la capa gratuita durante 1 año y en caso de exceder el cupo de esta, se hacen cobros mensuales.

Para el desarrollo de las API's se utilizó la consola de AWS, una herramienta web que permite la configuración e implementación de todos los servicios y herramientas utilizados, a través de interfaces graficas sencillas e intuitivas.

Para empezar el desarrollo se configuró el API Gateway. Esta herramienta es utilizada de manera similar al controlador Rest de la API Java, de manera que este redirija las peticiones que reciba a una determinada función

Lambda, tal que esta genere una respuesta a la petición original. Con este propósito se creó una API, la cual posee los recursos base "/", "/bucket", "/create" y "/update", los cuales reciben peticiones de tipo Get, Post, Put y Delete. Para las peticiones que realizan consultas sobre variables de path, se utilizó una plantilla de mapeo provista por AWS, que se encarga de mapear el evento que será enviado a la respectiva Lambda junto con todos los parámetros de la petición. Posteriormente se definió el control de acceso de peticiones a la API mediante claves llamadas API Key. Estas claves se generan en el API Gateway y deben ser incluidas en los headers de las peticiones, de lo contrario, estas son rechazadas.

De manera similar a las API's Java, se hizo uso de 2 bases de datos distintas, aunque en este caso ambas fueron implementadas con herramientas de AWS. Se utilizo DynamoDB para implementar la base de datos con la información de los clientes, creando una tabla cuyo acceso es restringido a la cuenta del administrador y a otros servicios implementados en esta. Los campos de la tabla son los mismos que se implementaron en la tabla clientes de la base de datos en MySQL, incluida la llave primaria Id del cliente.

Para el almacenamiento de imágenes asociadas a un cliente se implementó una base de datos no relacional en S3, de manera similar a la base de datos en MongoDB, con la diferencia de que estas ya no poseen una llave o un índice aparte. El acceso a los archivos se da a través del nombre de este, el cual es en este caso el Id del cliente. Debido a la incapacidad del API Gateway de manejar peticiones con archivos multi parte, solo se implementan las lambdas con los métodos de Get y Delete, los cuales retornan la información de la imagen como una cadena de caracteres en base 64.

4.5. Segundo conjunto de API's

Antes de crear las Lambdas que contienen la lógica para realizar las operaciones CRUD sobre las bases de datos creadas, se implementa un entorno de red virtual a través de la herramienta VPC. En este caso la red posee acceso a internet y contiene las Lambdas utilizadas, de manera que se puede controlar el acceso a estas a partir de grupos y políticas de

seguridad propias de la red.

Se configuró el servicio IAM para controlar los accesos que poseen las Lambdas implementadas a otros recursos de AWS a través de roles. Se creo un rol que permite el acceso y modificación de recursos en S3 a las Lambdas que requerían de estas funciones, mientras que para Lambdas que actuaron sobre DynamoDB se definió un rol con acceso completo a esta.

AWS Lambda es el recurso principal que se implementó para las funciones de la API, creando una para cada método de cada recurso el cual atendiera a cada tipo de petición. Cada una de estas fue implementada en Python 3.9 y el SDK de AWS Boto3 junto a una serie de librerías que permiten el procesamiento de los datos.

Como se mencionó en la sección 4.4, las peticiones a las API's son manejadas a través del API Gateway, el cual posee parámetros que deben existir según el tipo de petición que se haga, los cuales son objetos en formato Json que son procesados en la Lambda.

Una vez realizada una petición, se genera un evento, que es el objeto que contiene la información de esta. En el caso de una consulta, el evento contiene el id, mientras que, para una operación de creación o actualización, el evento contiene el objeto que se guardara en la base de datos.

La lógica general de todas las operaciones consiste en realizar una conexión a la base de datos correspondiente y a través de esta realizar la operación correspondiente al endpoint. En algunas operaciones (Sobre S3), se pueden generar excepciones, por lo que se utilizaron clausulas try/catch para definir el comportamiento y la respuesta de la Lambda. En este caso, las únicas excepciones que pueden generarse son debido a errores en consultas, generalmente al no encontrar datos sobre los cuales realizar una determinada operación en la base de datos.

Se definió un objeto de respuesta con una estructura exactamente igual a la que se implementó en Java, tal que permita el consumo de estas a partir del microservicio Cliente sin presentar conflictos. El objeto consta de un código Http, un código, un mensaje y el contenido de respuesta.

Para poder enviar peticiones al API Gateway, se debe desplegar una

etapa que posee una url publica que se usa como base para los recursos generados.

4.6. Microservicio para exponer las CRUD y el segundo conjunto de API's

4.6.1. Lambda para consultar la lista de usuarios

Para realizar la consulta de la lista de usuarios que se encuentran en la base de datos de clientes, se crea una función Lambda sin argumentos la cual responde a peticiones GET. Esta función solo tiene 2 posibles respuestas:

- Consulta correcta.
- No hay clientes para mostrar, en caso de que no haya clientes en la tabla.

4.6.2. Lambda para consulta de un usuario por Id.

Para consultar un usuario puntual a través de su Id, se utiliza el mismo recurso base que para consultar la lista de usuarios, pero en este caso se debe incluir el Id en el path como parámetro.

Esta función solo tiene 2 posibles respuestas:

- Consulta correcta
- No hay clientes para mostrar, en caso en que no se encuentre el cliente asociado al Id.

4.6.3. Lambda para creación de clientes

Esta Lambda permite crear un cliente a través de una petición POST, la cual debe incluir en su cuerpo la información de clientes en formato Json. Esta función solo tiene 3 posibles respuestas:

- Cliente guardado exitosamente.
- El cliente ya existe, en caso de que ya haya un cliente con ese Id en la tabla.
- Solicitud incorrecta. Por favor valide los datos enviados, en caso de que el cuerpo de la petición sea invalido o no exista.

4.6.4. Lambda para la actualización de clientes

Esta lambda permite actualizar la información de un cliente a través de una petición PUT. Al igual que la petición de creación, se debe incluir la información del cliente en el cuerpo de la petición. Esta función solo

tiene 2 posibles respuestas:

- Cliente guardado exitosamente.
- Solicitud incorrecta. Por favor valide los datos enviados, en caso de que el cuerpo de la petición sea invalido o no exista.

4.6.5. Lambda para el borrado de un cliente por Id

Esta lambda permite el borrado de un cliente a través del Id que debe ser incluido en el path como parámetro de la petición DELETE. Esta función solo tiene 2 posibles respuestas:

- Cliente borrado exitosamente.
- No se encuentra información para borrar, en caso de que no se encuentre un cliente asociado al Id en la tabla.

4.6.6. Lambda para la consulta de la foto por Id de un cliente

Esta Lambda permite consultar la imagen de un cliente a partir del Id asociado a este, el cual se debe incluir en el path de la petición GET. Esta función solo tiene 2 posibles respuestas:

- Consulta correcta.
- No hay clientes para mostrar, en caso de que no se encuentre una foto asociada al Id en la tabla.

4.6.7. Lambda para el borrado de la foto por Id de un cliente

Esta lambda permite borrar la foto asociada a un cliente a partir del Id de este, el cual debe ser incluido en el path de la petición DELETE. Esta función solo tiene 2 posibles respuestas:

- Cliente borrado.
- No se encuentra cliente para borrar, en caso de que no se encuentre una foto asociada al Id en la tabla.

4.7. API Cliente

Como se ha mencionado con la metodología, se decidió implementar una API adicional que permita consumir a las demás, de manera que se unifico el acceso a ellas sin perder la robustez derivada de la división de los microservicios. En este caso, se utiliza el mismo ambiente de desarrollo mencionado en la sección 4.1 (Excepto las bases de datos) y algunos elementos de la sección 4.2, como lo son los Dto, los tipos de petición, los objetos de respuesta y la configuración de seguridad.

La API consta de un único microservicio, el cual no posee acceso a ninguna base de datos. En su lugar, posee una serie de clientes Feign (Para consumir Microservicios) y métodos Rest (Para consumir API AWS). Esta API replica los endpoints que se encuentran en los demás API's para realizar las operaciones a partir el consumo de estos.

A diferencia de los microservicios de la primera API, este no requiere de configuración a partir de un servidor, por lo que se generó un archivo de configuración de extensión "properties" local, el cual define el tipo de conexión con eureka, el puerto y nombre del microservicio.

Con el fin de implementar el framework Feign y consumir los microservicios de la API java, se debe generar una configuración común para todos los clientes. Por esto, se implementó una clase de configuración, la cual contiene un bean que retorna la configuración estándar de Feign y un bean para interceptar las credenciales de la petición original.

Dado que Feign utiliza el balanceador de carga Ribbon, este se debe configurar través de una clase de configuración, la cual posee un bean que provee valores por defecto. Cuando se realizan peticiones a través de clientes Feign, se utiliza el mismo objeto de respuesta para evitar inconvenientes de compatibilidad.

Una vez se tiene la clase de configuración, el balanceador de carga y el servidor eureka, se realiza la creación de las clases encargadas de realizar las peticiones. Cada microservicio (Get, Create, Modify, Delete) requiere una interfaz para poder consumirlos. Cada una de estas interfaces contiene los mismos endpoints que debe consumir, además de la anotación @FeignClient, la cual define el path o url a donde se realizan las

peticiones y el nombre de la clase de configuración. En la figura 9, se encuentra un ejemplo de una interfaz cliente Feign.

```java
@FeignClient(name = "service-get", configuration = FeignConfiguration.class)
public interface FeignServiceGet {

    @GetMapping(value="/clientes",produces= MediaType.APPLICATION_JSON_VALUE)
    ResponseDto<List<ClienteDto>> recuperarClientes();

    @GetMapping(value="clientes/{id}",produces=MediaType.APPLICATION_JSON_VALUE)
    ResponseDto<ClienteDto> getClientId(@PathVariable("id") int id);

    @GetMapping(value="/clientes/mayores/{age}",produces= MediaType.APPLICATION_JSON_VALUE)
    ResponseDto<List<ClienteDto>> recuperarClientesMayores(@PathVariable("age") int edad);

    @GetMapping("/photos/{Mongoid}")//imagen by MongoId
    ResponseDto<String> getPhoto(@PathVariable String Mongoid) ;

    @GetMapping("/photo/{ClientId}")
    ResponseDto<String> getPhotoid(@PathVariable int ClientId);

}
```

Figura 9. Ejemplo de interfaz cliente Feign del microservicio Get.

En este proyecto se optó por utilizar el nombre del microservicio a consumir en vez de la url directamente, ya que se tiene configurado el servidor Eureka, el cual puede resolverla a partir del nombre. Esto es útil en caso de que se ejecuten los microservicios en máquinas distintas, pero en el mismo entorno de red, lo que evita la necesidad de poner las url manualmente.
Para realizar el consumo de la API AWS, se creó una clase AwsProvider, la cual implementa los métodos para realizar peticiones a los endpoint públicos de la API. Se crearon las peticiones a través de un objeto RestTemplate a las url's dispuestas por el API Gateway, el cual incluye entre sus headers, la API key. Las respuestas se evalúan, se mapean a un objeto respuesta y finalmente se verifica el contenido, a partir del cual se genera un nuevo objeto respuesta para la petición original sobre el microservicio cliente.

5. Resultados y análisis

Dado que la naturaleza del proyecto implica casi única y exclusivamente generar respuesta a peticiones de realizar acciones del lado del servidor, se presentan dos formas de mostrar los resultados obtenidos: las pruebas unitarias y pruebas funcionales. Ambas pruebas proveen información relevante, las unitarias muestran características y utilidad del código, mientras que las funcionales muestran la forma en que se comporta la aplicación en circunstancias reales.

Las pruebas unitarias permiten predecir el comportamiento de la API ante determinadas entradas. Estas fueron desarrolladas en Java a través de la API Junit5, la cual permite realizar peticiones y evaluar el resultado esperado, para garantizar el uso y comportamiento del código.

Cada microservicio Java requiere de una clase pruebas anotada con @SpringBootTest, la cual contiene un objeto MockMvc para realizar las peticiones a los endpoint e instancias de los repositorios. A diferencia de las instancias de repositorios normales, en pruebas, son anotadas con @MockBean. Esta anotación permite interceptar el comportamiento de los repositorios, tal que, cada que un determinado método sea llamado, se pueda definir el comportamiento programáticamente, por ejemplo, el resultado de una consulta a una base de datos sin necesidad realizarla.

Para los microservicios que requerían el envío de objetos, se crearon métodos y constantes en una clase TestUtils, los cuales son mapeados a objetos Json para ser enviados en las peticiones.

Las pruebas funcionales que como su nombre lo indica, son las que se realizan con las API's ejecutándose en circunstancias reales, de manera que den información acerca del comportamiento que tendrán al ser desplegadas. Estas pruebas se pueden realizar a través de peticiones HTTP, las cuales pueden ser generadas a través de un software de terceros como Postman (Figura 10), tal como se propuso inicialmente, o a través de la interfaz gráfica creada específicamente para cada microservicio, incluido el de cliente. Esta interfaz, como se mencionó anteriormente en la sección 4.2, está debidamente documentada con respecto a sus métodos, modelos, entradas y respuestas. Cabe recalcar que las pruebas funcionales de las API's Java de consulta y borrado se presentan a través de la interfaz gráfica, mientras que las de creación y modificación, así como las de la API AWS se presentan a través de Postman.

Además, las API's en AWS no cuentan la posibilidad de implementar pruebas unitarias y tampoco poseen interfaz gráfica.

Se generaron 2 colecciones llamadas Services y AWS Provider, las cuales incluyen todas las peticiones que se realizan a los microservicios de la API java y las Lambdas de AWS.

Para simplificar el acceso a las peticiones de manera local y a través del microservicio cliente de la API Java o a través de la web y a través del cliente, se definieron 4 ambientes de pruebas, los cuales poseen valores diferentes para las constantes de las url, de manera que se pueda cambiar los endpoints a los que se realizan las peticiones sin tener que generar otras o cambiar las existentes.

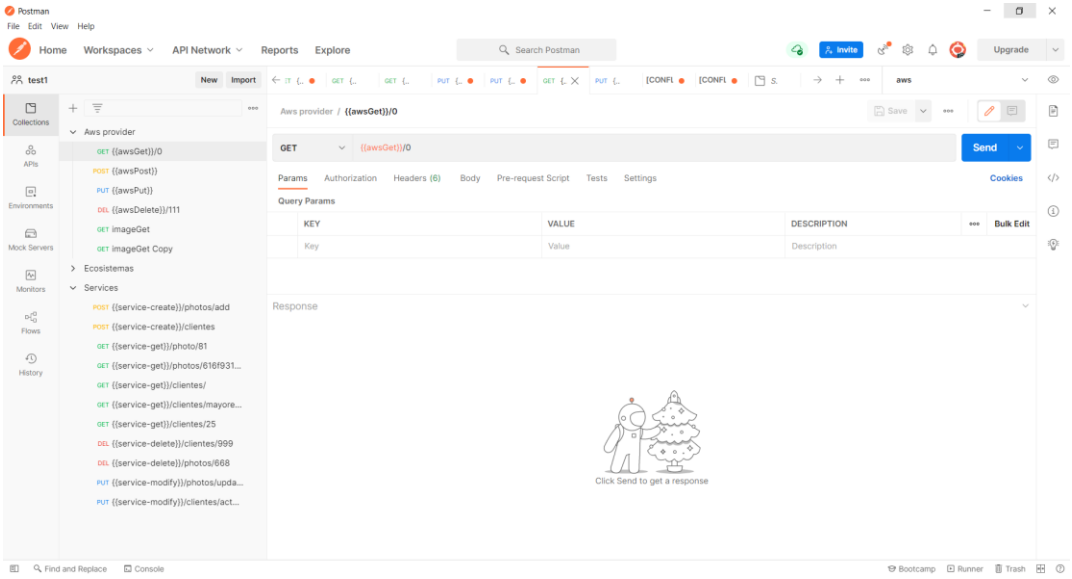


Figura 10. Interfaz de Postman con las peticiones a las API's.

Como se mencionó antes, en la configuración de cada microservicio se definieron credenciales que son necesarias para cada el consumo de estos, por los cuales son incluidos en las peticiones de Postman, junto con sus respectivos parámetros y cuerpo.

5.1. Pruebas funcionales API Java

Se desarrollaron 5 endpoints para consultas, 2 creación, 2 para modificación y 2 para borrado de registros, los cuales son ilustrados en las figuras 11, 12, 13 y 14 respectivamente.

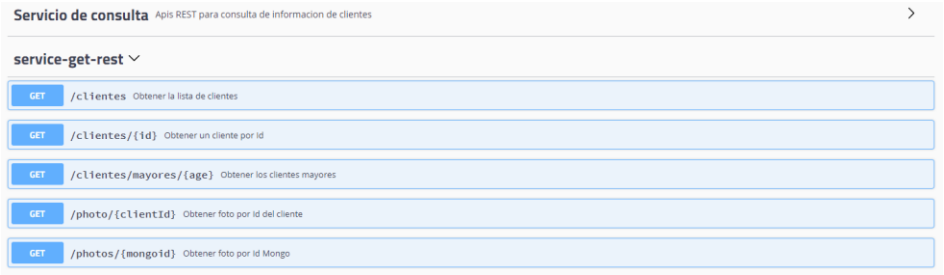


Figura 11. Endpoints del microservicio de consulta.

Figura 12. Endpoints del microservicio de creación.

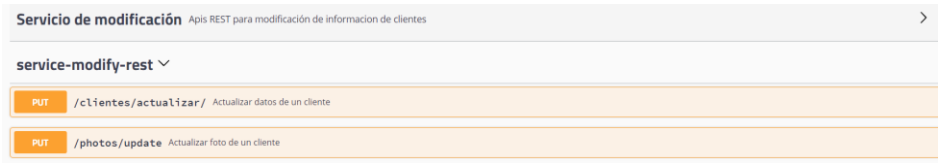


Figura 13. Endpoints del microservicio de modificación.

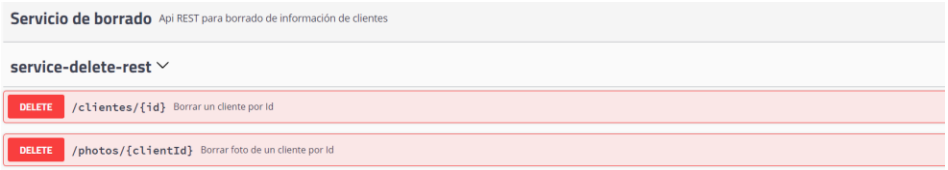
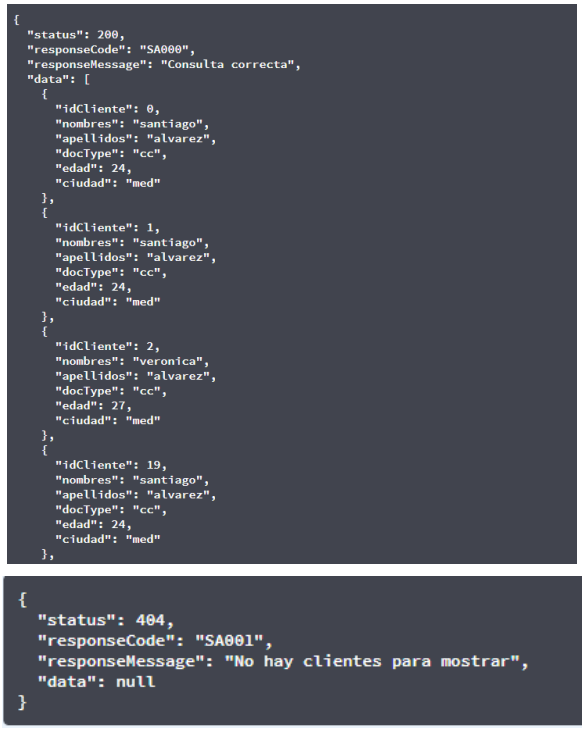


Figura 14. Endpoints del microservicio de borrado.

5.1.1. Obtener lista de todos los clientes

Al realizar peticiones al endpoint de obtener la lista de todos los clientes se obtuvieron las respuestas de la figura 15.



```
{
  "status": 200,
  "responseCode": "SA000",
  "responseMessage": "Consulta correcta",
  "data": [
    {
      "idCliente": 0,
      "nombres": "santiago",
      "apellidos": "alvarez",
      "docType": "cc",
      "edad": 24,
      "ciudad": "med"
    },
    {
      "idCliente": 1,
      "nombres": "santiago",
      "apellidos": "alvarez",
      "docType": "cc",
      "edad": 24,
      "ciudad": "med"
    },
    {
      "idCliente": 2,
      "nombres": "veronica",
      "apellidos": "alvarez",
      "docType": "cc",
      "edad": 27,
      "ciudad": "med"
    },
    {
      "idCliente": 19,
      "nombres": "santiago",
      "apellidos": "alvarez",
      "docType": "cc",
      "edad": 24,
      "ciudad": "med"
    },
```

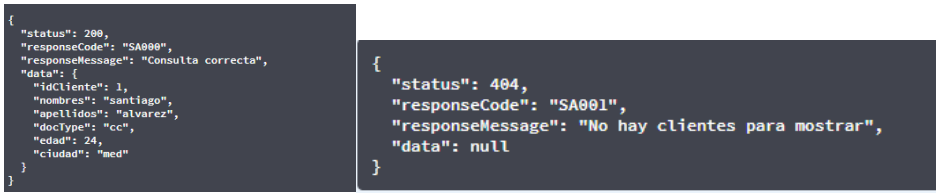
```
{
  "status": 404,
  "responseCode": "SA001",
  "responseMessage": "No hay clientes para mostrar",
  "data": null
}
```


Figura 15. Sección de la respuesta a la petición correcta de obtener lista de clientes(arriba) e incorrecta(abajo).

La petición puede generar dos respuestas, cuando encuentran clientes es correcta, de lo contrario es incorrecta.

5.1.2. Obtener un cliente por Id

Al realizar peticiones al endpoint obtener un cliente por id se obtuvieron las respuestas de la figura 16.



```
{
  "status": 200,
  "responseCode": "SA000",
  "responseMessage": "Consulta correcta",
  "data": {
    "idCliente": 1,
    "nombres": "santiago",
    "apellidos": "alvarez",
    "docType": "cc",
    "edad": 24,
    "ciudad": "med"
  }
}
```

```
{
  "status": 404,
  "responseCode": "SA001",
  "responseMessage": "No hay clientes para mostrar",
  "data": null
}
```


Figura 16. Respuesta a la petición de obtener un cliente por Id correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando encuentra el cliente es correcta, de lo contrario es incorrecta.

5.1.3. Obtener los clientes mayores

Al realizar peticiones al endpoint obtener los clientes mayores se obtuvieron las respuestas de la figura 17.

Figura 17. Respuesta a la petición de clientes mayores que una edad, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando encuentran clientes mayores a la edad de consulta es correcta, de lo contrario es incorrecta.

5.1.4. Obtener foto por Id del cliente

Al realizar peticiones al endpoint obtener foto por Id del cliente se obtuvieron las respuestas de la figura 18.

Figura 18. Respuesta a la petición obtener foto por Id del cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se encuentra la imagen asociada al cliente en la colección, es correcta, de lo contrario es incorrecta.

5.1.5. Obtener foto por Id Mongo

Al realizar peticiones al endpoint obtener foto por Id Mongo se obtuvieron las respuestas de la figura 19.

Figura 19. Respuesta a la petición obtener foto por Id Mongo, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se encuentra la imagen en la colección, es correcta, de lo contrario es incorrecta.

5.1.6. Creación de nuevo cliente

Al realizar peticiones al endpoint de creación de nuevo cliente se obtuvieron las respuestas de la figura 20.


```
{
  "status": 201,
  "responseCode": "SA002",
  "responseMessage": "Cliente guardado exitosamente",
  "data": null
}
```

```
{
  "status": 400,
  "responseCode": "SA003",
  "responseMessage": "El cliente ya existe",
  "data": null
}
```

```
{
    "status": 400,
    "responseCode": "SA004",
    "responseMessage": "Solicitud incorrecta. Por favor valide los datos enviados.",
    "data": null
}
```


Figura 20. Respuesta a la petición crear nuevo cliente, correcta (Izquierda), incorrecta (Derecha) e invalida (Abajo).

La petición puede generar tres respuestas, al crear un nuevo cliente es correcta, al tratar de crear un cliente con el mismo número de identificación de otro que ya esté en la tabla es incorrecta. Finalmente, la petición es invalida si se envía sin la información de cliente necesaria.

5.1.7. Creación de imagen de un cliente

Al realizar peticiones al endpoint creación de imagen de un cliente se obtuvieron las respuestas de la figura 21.


```
{
    "status": 201,
    "responseCode": "SA002",
    "responseMessage": "Cliente guardado exitosamente",
    "data": "MongoId de la imagen: 61cbf7bd66cf870138ab7bfa"
}
```

```
{
    "status": 400,
    "responseCode": "SA003",
    "responseMessage": "El cliente ya existe",
    "data": null
}
```

```
{
    "status": 400,
    "responseCode": "SA004",
    "responseMessage": "Solicitud incorrecta. Por favor valide los datos enviados.",
    "data": null
}
```


Figura 21. Respuesta a la petición crear nueva imagen de un cliente, correcta (Izquierda), incorrecta (Derecha) e invalida (Abajo) en Postman.

La petición puede generar tres respuestas, al crear una nueva imagen es correcta, al tratar de crear una imagen con el mismo número de identificación de un cliente que ya esté en la colección, es incorrecta.

Finalmente, al no enviar una imagen en la solicitud, es invalida.

5.1.8. Actualizar datos de un cliente

Al realizar peticiones al endpoint de actualizar datos de un cliente se obtuvieron las respuestas de la figura 22.

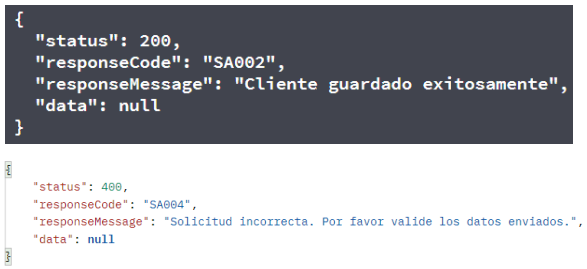


Figura 22. Respuesta a la petición modifica datos de un cliente, correcta (arriba) e invalida (abajo).

La petición puede generar dos respuestas, correcta al actualizar un cliente e invalida al enviar una petición sin la información de cliente necesaria.

5.1.9. Actualizar foto de un cliente

Al realizar peticiones al endpoint de actualizar foto de un cliente se obtuvieron las respuestas de la figura 23.

Figura 23. Respuesta a la petición actualizar la imagen de un cliente, correcta (Izquierda) e invalida (Derecha) en Postman.

La petición solo puede producir dos respuestas, correcta al actualizar la información de un cliente e invalida cuando los datos enviados son inválidos o inexistentes.

5.1.10. Borrar un cliente por id

Al realizar peticiones al endpoint de borrar un cliente por id se obtuvieron las respuestas de la figura 24.

Figura 24. Respuesta a la petición borrar un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar 2 respuestas, al borrar un cliente es correcta, si no se encuentra un cliente para borrar es incorrecta.

5.1.11. Borrar foto de un cliente por Id

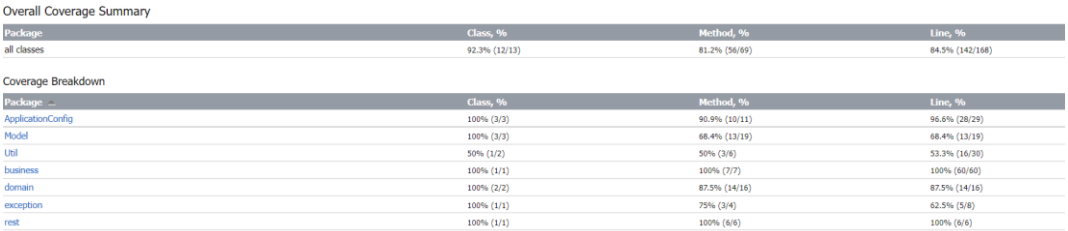
Al realizar peticiones al endpoint de borrar foto de un cliente por Id se obtuvieron las respuestas de la figura 25.

Figura 25. Respuesta a la petición borrar la foto de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar 2 respuestas, cuando se borra la imagen es correcta, cuando no se encuentra una foto asociada al Id del cliente es incorrecta.

5.2. Pruebas unitarias API Java

Al realizar las pruebas unitarias de cada microservicio se genera un informe web que describe la cobertura de código que estas ofrecen. El resumen de la cobertura por paquete de los microservicios de consulta, creación, modificación y borrado se encuentra en las figuras 26, 27, 28 y 29



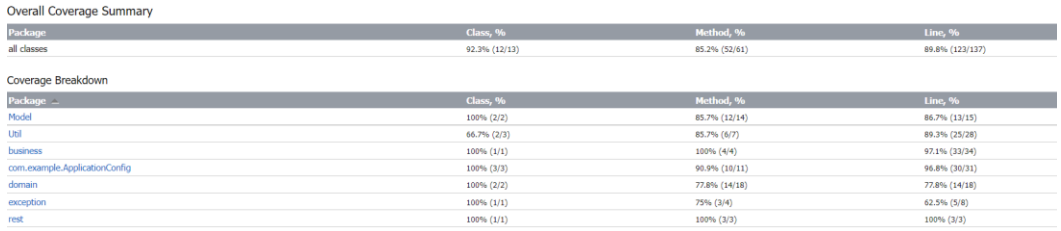
Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	92.3% (12/13)	81.2% (56/69)	84.5% (142/168)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
ApplicationConfig	100% (3/3)	90.9% (10/11)	96.6% (28/29)
Model	100% (3/3)	68.4% (13/19)	68.4% (13/19)
Util	50% (1/2)	50% (3/6)	53.3% (16/30)
business	100% (1/1)	100% (7/7)	100% (60/60)
domain	100% (2/2)	87.5% (14/16)	87.5% (14/16)
exception	100% (1/1)	75% (3/4)	62.5% (5/8)
rest	100% (1/1)	100% (6/6)	100% (6/6)

Figura 26. Resumen de cobertura de pruebas unitarias del servicio de consultas.



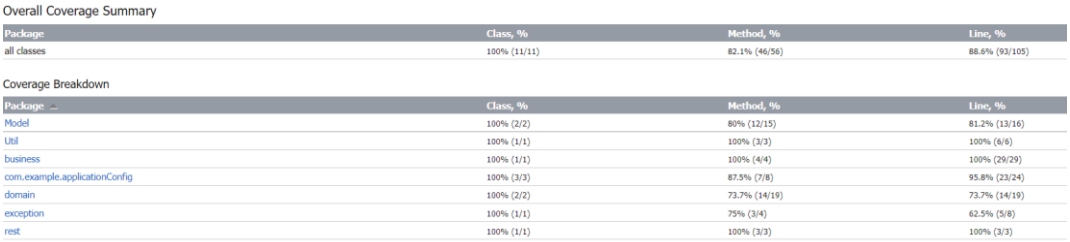
Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	92.3% (12/13)	85.2% (52/61)	89.8% (123/137)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
Model	100% (2/2)	85.7% (12/14)	86.7% (13/15)
Util	66.7% (2/3)	85.7% (6/7)	89.3% (25/28)
business	100% (1/1)	100% (4/4)	97.1% (33/34)
com.example.ApplicationConfig	100% (3/3)	90.9% (10/11)	96.8% (30/31)
domain	100% (2/2)	77.8% (14/18)	77.8% (14/18)
exception	100% (1/1)	75% (3/4)	62.5% (5/8)
rest	100% (1/1)	100% (3/3)	100% (3/3)

Figura 27. Resumen de cobertura de pruebas unitarias del servicio de creación.



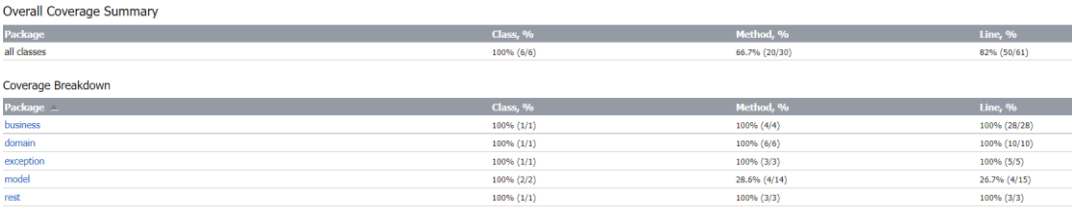
Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (11/11)	82.1% (46/56)	88.6% (93/105)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
Model	100% (2/2)	80% (12/15)	81.2% (13/16)
Util	100% (1/1)	100% (3/3)	100% (6/6)
business	100% (1/1)	100% (4/4)	100% (29/29)
com.example.applicationConfig	100% (3/3)	87.5% (7/8)	95.8% (23/24)
domain	100% (2/2)	73.7% (14/19)	73.7% (14/19)
exception	100% (1/1)	75% (3/4)	62.5% (5/8)
rest	100% (1/1)	100% (3/3)	100% (3/3)

Figura 28. Resumen de cobertura de pruebas unitarias del servicio de modificación.



Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (6/6)	66.7% (20/30)	82% (50/61)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
business	100% (1/1)	100% (4/4)	100% (28/28)
domain	100% (1/1)	100% (6/6)	100% (10/10)
exception	100% (1/1)	100% (3/3)	100% (5/5)
model	100% (2/2)	28.6% (4/14)	26.7% (4/15)
rest	100% (1/1)	100% (3/3)	100% (3/3)

Figura 29. Resumen de cobertura de pruebas unitarias del servicio de borrado.

5.3. Pruebas funcionales API AWS

A diferencia de las peticiones a la API Java, aquellas realizadas a la AWS no requieren autenticación mediante usuario y clave, sino que se realiza a través de una llave de seguridad que se envía en los headers de la solicitud junto con el cuerpo y los parámetros correspondientes.

5.3.1. Consulta lista de clientes completa

Figura 30. Respuesta a la petición consultar los datos de un cliente, correcta (derecha) e incorrecta (izquierda).

La petición puede generar dos respuestas, cuando encuentran clientes es correcta, de lo contrario es incorrecta.

5.3.2. Consulta de un cliente por Id

Al realizar peticiones al endpoint de consulta de un cliente por Id se obtuvieron las respuestas de la figura 31.

Figura 31. Respuesta a la petición consultar los datos de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando encuentra el cliente es correcta, de lo contrario es incorrecta.

5.3.3. Creación de un cliente

Al realizar peticiones al endpoint de creación de un cliente se obtuvieron las respuestas de la figura 32.

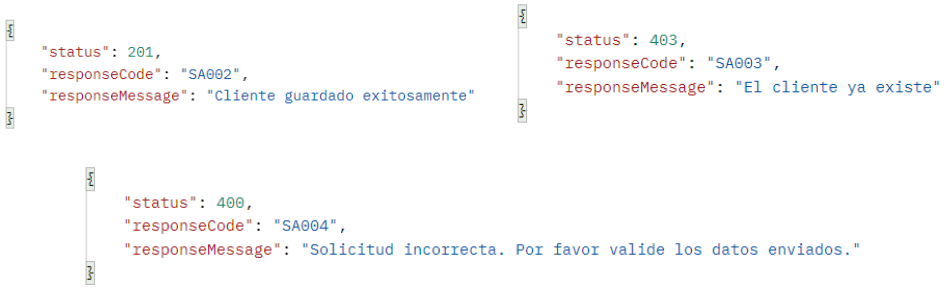


Figura 32. Respuesta a la petición de crear un cliente, correcta (Izquierda), incorrecta (Derecha) e invalida (abajo).

La petición puede generar tres respuestas, cuando se crea un cliente nuevo es correcta, si ya existe en la base de datos es incorrecta. Si el cuerpo de la petición es erróneo o no existe, la petición es invalida.

5.3.4. Actualización de un cliente

Al realizar peticiones al endpoint de actualización de un cliente se obtuvieron las respuestas de la figura 33.

Figura 33. Respuesta a la petición actualizar los datos de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se actualiza un cliente es correcta, si el cuerpo de la petición es erróneo o no existe, la petición es invalida.

5.3.5. Borrado de un cliente por Id

Al realizar peticiones al endpoint de borrado de un cliente por Id se obtuvieron las respuestas de la figura 34.

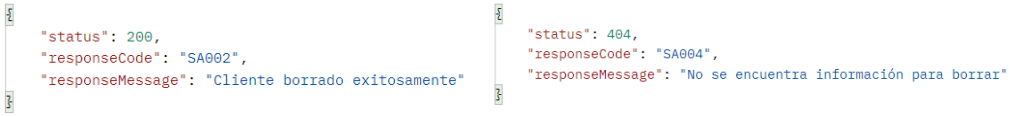


Figura 34. Respuesta a la petición borrar los datos de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se borra un cliente de la tabla es correcta, si no lo encuentra es incorrecta.

5.3.6. Consulta de imagen de un cliente por Id

Al realizar peticiones al endpoint de consulta de imagen de un cliente por Id se obtuvieron las respuestas de la figura 35.

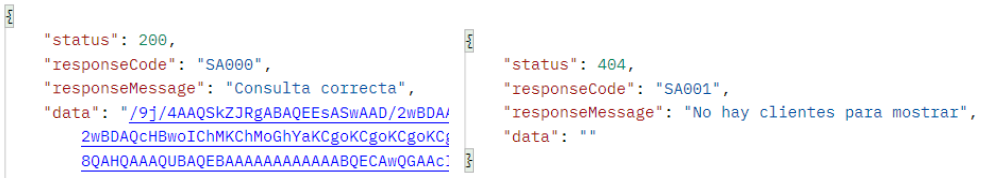


Figura 35. Sección de la respuesta a la petición consultar la imagen de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se encuentra la imagen del cliente en el bucket es correcta, de lo contrario es incorrecta.

5.3.7. Borrado de imagen de un cliente por Id

Al realizar peticiones al endpoint de consulta de un cliente por Id se obtuvieron las respuestas de la figura 36.

Figura 36. Sección de la respuesta a la petición consultar la imagen de un cliente, correcta (Izquierda) e incorrecta (Derecha).

La petición puede generar dos respuestas, cuando se borra la imagen del cliente en el bucket es correcta, si no la encuentra es incorrecta.

5.4. API cliente

Dado que el servicio de cliente se encarga únicamente de consumir a los demás microservicios, no se implementaron pruebas unitarias ya que serían redundantes, aunque se incluyó la interfaz gráfica en Swagger para su consumo.

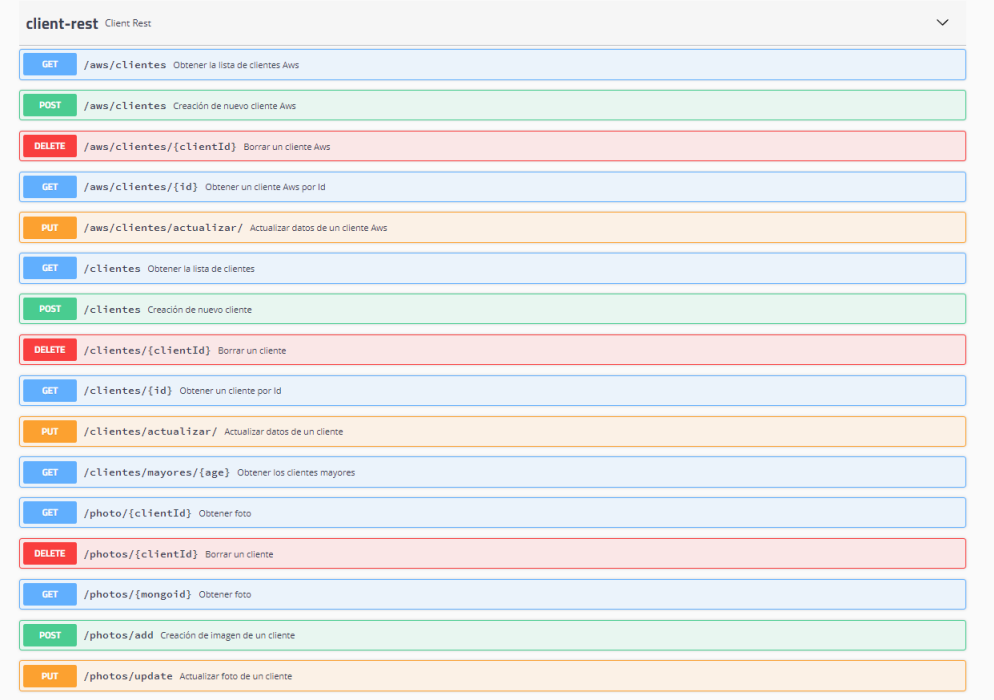


Figura 37. Endpoints del microservicio de cliente.

Dada la naturaleza de este microservicio, los endpoints, sus argumentos y respuestas son exactamente iguales, tanto para los servicios de la API Java como la de AWS.

Todas y cada una de las respuestas obtenidas son utilizadas durante la ejecución normal de las API's, de tal manera que se garantiza que pueden responder ante las distintas peticiones que se puedan realizar. Cada respuesta incluye los mensajes y códigos de función junto a los datos de respuesta. Además, permite analizar la correspondencia en la información obtenida con la esperada. Para verificar las respuestas se modificaron las peticiones de distintas formas, por ejemplo, enviando un cuerpo y/o parámetros incorrectos, tal que se cubran posibles errores que pueda tener un cliente al consumirlas. Es el amplio rango de

respuestas posibles el que permite aumentar la robustez individual de cada microservicio y de la API como tal.

La seguridad fue descrita individualmente con en la sección 4.2, su respuesta es genérica y se ejecuta a nivel del framework Spring Boot, por lo que es común en el caso de la API Java y se encuentra en la figura 37.


```
{
    "timestamp": "2022-01-04T02:33:52.289+00:00",
    "status": 401,
    "error": "Unauthorized",
    "path": "/clientes/actualizar/"
}
```


Figura 38. Repuesta a petición sin credenciales.

En el caso de la API en AWS, se tiene una respuesta igual para toda solicitud que no cuente con la llave de acceso, la cual se encuentra en la figura 38:


```
{
    "message": "Forbidden"
}
```


Figura 39. Repuesta a petición sin llave de API.

Para descentralizar las API's se utilizó el API Gateway para invocar lambdas independientes en el caso de la AWS, para la API Java se implementaron servidores de configuración y de registro para los microservicios que la componen, tal que para ambos casos se logra tener implementaciones distribuidas a través de un punto común, dotándolas de independencia y resistencia a fallos individuales.

Para el caso de las pruebas unitarias implementadas en la API Java, se obtuvo una cobertura de código por línea de más del 80%, lo cual da evidencia de que la mayor parte del código producido se implementa durante la ejecución normal de esta; en términos de la lógica de negocio de los microservicios, se tuvo que la cobertura era del 100% para cada uno, siendo esta la parte esencial para integrar los demás componentes.

6. Conclusiones

Se desarrolló un conjunto de API's mediante el lenguaje de programación Java que exponen las operaciones CRUD para el manejo de dos bases de datos. La primera, una base de datos relacional MySQL ejecutada en el servicio RDS de AWS, que contiene la información de clientes. La segunda, una base de datos no relacional MongoDB sobre MongoDB Community Server. Durante el desarrollo se encontraron inconvenientes que derivaron en mejoras no contempladas en la implementación original. Una de estas mejoras fue la implementación de la base de datos MySQL en un servicio de la nube, tal que fuera independiente de la volatilidad que representa mantenerla en un equipo local. Otro ejemplo de esto fue la creación de ambientes o perfiles de desarrollo, los cuales permitieron reducir el tiempo de lanzamiento de los microservicios al hacer pruebas en el IDE.

Se desarrolló un microservicio para cada operación CRUD a partir del framework Spring Boot que integran el primer conjunto de API's, el resultado de estas es evidenciado en la sección 5.1. Las pruebas funcionales, al menos en su caso ideal, muestran claramente que los microservicios son capaces de ejecutar operaciones de consulta, creación, actualización y borrado sobre las bases de datos mencionadas anteriormente. Como componente adicional y con el objetivo de aumentar la robustez del sistema, considerando que no siempre se tienen usuarios completamente informados, se implementaron medidas de contingencia. Estas medidas consisten en considerar los casos en los que se puedan presentar errores al consumir los endpoint y posteriormente implementar contramedidas a estos. La muestra de este desarrollo se da también en pruebas funcionales, por esto, se presentaron igualmente las posibles respuestas que ofrecen los endpoint al ser consumidos de maneras erróneas, lo cual no detiene la ejecución ni genera inconvenientes para posteriores operaciones.

La descentralización originalmente planteada se ejecutó a través de un servidor de configuración y un servidor Eureka. El primero, provee a los microservicios de configuraciones almacenadas en la nube, haciéndolos más flexibles y sencillos de mantener. El segundo, provee la capacidad de consumir microservicios que se encuentren en el mismo ambiente de red sin conocer necesariamente su url, lo cual permite realizar cambios a la estructura de la API sin requerir mayor cambio. Como

se tenía previsto, se implementaron reglas de seguridad para cada operación de los distintos microservicios, que como se evidencio en el final de la sección 5, permitió restringir y administrarlos de manera segura y eficiente.

Se definió el segundo conjunto de API's a partir del lenguaje de programación Python, que expongan las operaciones CRUD para el manejo de dos bases de datos no relacionales. La primera, fue implementada en DynamoDB y contiene la información de clientes, mientras que la segunda fue implementada en S3 y almacenas imágenes asociadas a clientes. Como se tenía previsto, se implementaron ambas bases de datos y se insertaron datos de prueba. Se aprovechó que la aplicación del proyecto es limitada en escala, lo que implica que el uso de estos servicios aún se encuentra en la capa gratuita de AWS.

Se desarrollaron los microservicios asociados a las operaciones CRUD para la base de datos de clientes en DynamoDB y los correspondientes a la consulta y borrado de imágenes en S3. Para estos desarrollos, se implementó el SDK provisto por AWS para Python, teniendo consideraciones similares a las que se tuvieron en el primer conjunto de API's. Cada uno de estos microservicios no solo tienen la capacidad de realizar su respectiva operación, cuentan además con la capacidad de responder en casos en que se tengan inconvenientes, como se evidencia en la sección 5.3. Es esta capacidad la que aumenta la robustez del sistema, que de por si es administrado por AWS y no requiere de un servidor, haciéndolo resistente a fallos de hardware y resistente a ataques informáticos.

Las operaciones que no pudieron implementarse fueron las de creación y actualización de imágenes en S3, debido a la falta de soporte del servicio API Gateway de archivos multi parte. Esta limitación era desconocida originalmente y no se encontraron alternativas que mantuvieran la estructura de las API's.

Se implementaron las pruebas unitarias y funcionales correspondientes, las cuales denotan el comportamiento de los microservicios implementados ante los casos de uso. Estas proporcionaron una gran cantidad de información, tal que, a partir de las respuestas obtenidas para cada endpoint fue posible afirmar que las API's desarrolladas cumplen los resultados esperados. Como se mencionó en

conclusiones anteriores, las pruebas funcionales fueron fundamentales para mostrar las API's son capaces de realizar las operaciones CRUD sobre las tablas, buckets y colecciones implementadas. Por otro lado, las pruebas unitarias describen la calidad del código implementado, en términos de eficiencia y utilidad.

En términos de comparación entre las API's Java y AWS, se tienen ventajas y desventajas en ambos lados. La primera de estas la que más potente en términos de versatilidad y personalización, ya que como se mostró en este proyecto, cuenta con una enorme cantidad de opciones, configuraciones, frameworks, API's y métodos. Es esto lo que permite personalizar ampliamente el comportamiento final de la API, en términos la escalabilidad, flexibilidad, robustez y seguridad. A pesar de lo anterior, la API AWS posee una ventaja significativa en el apartado de facilidad de implementación y uso. Esto es debido a que Amazon provee una gran cantidad de herramientas que facilitan el trabajo del desarrollador, de manera que se creó con una rapidez mucho mayor en comparación. Adicionalmente, al desarrollar directamente en la plataforma AWS, el proyecto se encuentra completamente integrado con los servicios de bases de datos, notificación, monitoreo, etc. Tal que, al hacer uso de estos es más sencillo y eficiente. En términos de despliegue, ambas pueden ser implementadas en servicios de AWS, directamente en Lambdas en el caso de la API AWS y a través de una instancia Elastic Cloud Computing (EC2) en el caso de la Java.

Como valor agregado se desarrolló la API cliente, la cual permitió integrar las API's a través de una sola, combinando las capacidades de personalización de seguridad, acceso y distribución a la facilidad de manejo e implementación, tal que se obtiene como resultado final un servicio mucho más completo, accesible, desplegable en plataformas cloud, robusto y escalable.

7. Referencias Bibliográficas

[1] Y. Fernández, "API: qué es y para qué sirve", Xataka.com, 23-ago-2019. [En línea]. Disponible en: https://www.xataka.com/basics/api-que-sirve. [Accedido: 26-ene-2022].

[2] "¿Qué son y para qué sirven los microservicios?", Redhat.com. [En línea]. Disponible en: https://www.redhat.com/es/topics/microservices. [Accedido: 26-ene-2022].

[3] Wikipedia contributors, "Aplicación web", Wikipedia, The Free Encyclopedia. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Aplicaci%C3%B3n_web&oldid=139278615.

[4] "Transferencia de Estado Representacional", Wikipedia.org. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representaciona. [Accedido: 26-ene-2022].

[5] "Learn", Spring.io. [En línea]. Disponible en: https://spring.io/learn. [Accedido: 26-ene-2022].

[6] "Amazon Web Services About Us", Amazon.com, 01-sep-2011. [En línea]. Disponible en: https://Amazon.com. [Accedido: 26-ene-2022].

[7] "AWS Lambda – Características del producto", Amazon.com. [En línea]. Disponible en: https://aws.amazon.com/es/lambda/features/. [Accedido: 26-ene-2022].

[8] S. Clifford, "Other retailers find ex-Blockbuster stores just right", The New York times, The New York Times, 09-abr-2011.

[9] M. Zetlin, "Blockbuster could have bought Netflix for $50 million, but the CEO thought it was a joke", Inc, 20-sep-2019. [En línea]. Disponible en: https://www.inc.com/minda-zetlin/netflix-blockbuster-meeting-marc-randolph-reed-hastings-john-antioco.html. [Accedido: 26-ene-2022].

[10] "Qué es MVC", Desarrolloweb.com, 02-ene-2014. [En línea]. Disponible en: https://desarrolloweb.com/articulos/que-es-mvc.html. [Accedido: 26-ene-2022].

[11] "Web MVC framework", Spring.io. [En línea]. Disponible en: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html. [Accedido: 26-ene-2022].

[12] G. Blokdyk, IBM docs: Complete self-assessment guide. North Charleston, SC: Createspace Independent Publishing Platform, 2018.

[13] "Conozca más sobre la tecnología Java", Java.com. [En línea]. Disponible en: https://www.java.com/es/about/. [Accedido: 26-ene-2022].

[14] "Qué es Spring Framework y por qué usarlo", OpenWebinars.net, 05-jun-2018. [En línea]. Disponible en: https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/. [Accedido: 26-ene-2022].

[15] Wikipedia contributors, "Edad de Piedra", Wikipedia, The Free Encyclopedia. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Edad_de_Piedra&oldid=140014982.

[16] Wikipedia contributors, "Blockbuster (empresa)", Wikipedia, The Free Encyclopedia. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Blockbuster_(empresa)&oldid=141212229.

[17] Atlassian, "Qué es el control de versiones", Atlassian. [En línea]. Disponible en: https://www.atlassian.com/es/git/tutorials/what-is-version-control. [Accedido: 26-ene-2022].

[18] D. Fernandez, "Arquitecturas basadas en microservicios: Spring Cloud Netflix Eureka", BI Geek Blog, 04-dic-2017. [En línea]. Disponible en: https://blog.bi-geek.com/arquitecturas-spring-cloud-netflix-eureka/. [Accedido: 26-ene-2022].

[19] "API Documentation", API Documentation Made Easy - Get Started | Swagger, 2017. .

[20] "Project lombok", Projectlombok.org. [En línea]. Disponible en: https://projectlombok.org/. [Accedido: 26-ene-2022].

[21] "MySQL :: MySQL documentation", Mysql.com. [En línea]. Disponible en: https://dev.mysql.com/doc/. [Accedido: 26-ene-2022].

[22] "MongoDB: the application data platform", MongoDB. [En línea]. Disponible en: https://www.mongodb.com/es. [Accedido: 26-ene-2022].