.UBAfiuba
FACULTAD DE INGENIERÍA

Base de Datos
(TA044) Curso Merlino

Trabajo Práctico
Base de Datos Filmográfica

7 de diciembre de 2024

Agustin	Santiago	Lucca	Lucas	Tomas	Leonardo
Campos	Corrado	Marino	Garcia	Pugliese	Dragone
110525	110453	110592	110099	110140	111136

1. Elección de las Tecnologías

En esta sección del informe vamos a describir los lenguajes de programación, frameworks y bases de datos que hemos elegido para el proyecto. También explicaremos por qué seleccionamos cada uno de ellos, destacando sus ventajas y cómo se adaptan a las necesidades del trabajo.

1.1. Lenguajes

En nuestro trabajo empleamos dos lenguajes de programación, cada uno seleccionado estratégicamente según las necesidades del proyecto. Para la implementación del backend, utilizamos Java, ya que resulta ideal para desarrollar las funciones CRUD y gestionar los datos. Esta elección está estrechamente relacionada con los frameworks que empleamos, los cuales describiremos más adelante. Además, Java nos ofrece un alto nivel de abstracción, lo que facilita la mantenibilidad del código a largo plazo.

Por otro lado, en el desarrollo del frontend optamos por JavaScript. Este lenguaje destaca por permitir un elevado nivel de personalización en la interacción y presentación del sitio web, adaptándose de manera precisa a los requisitos específicos del proyecto. Esto nos brinda la flexibilidad de ajustar y optimizar la interfaz según nuestras necesidades.

1.2. Frameworks

De acuerdo con lo mencionado previamente, en nuestro trabajo empleamos dos frameworks principales. El primero es React, diseñado específicamente para crear interfaces de usuario, lo que facilita el desarrollo de aplicaciones de una sola página. Una de las principales razones por las que optamos por este framework es su capacidad para mejorar la organización y el mantenimiento del código, permitiéndonos dividir la interfaz en componentes más pequeños y reutilizables.

El segundo framework que utilizamos es Node.js, que nos resultó especialmente atractivo debido a su facilidad para desarrollar aplicaciones utilizando JavaScript tanto en el servidor como en el frontend, simplificando así el proceso de desarrollo.

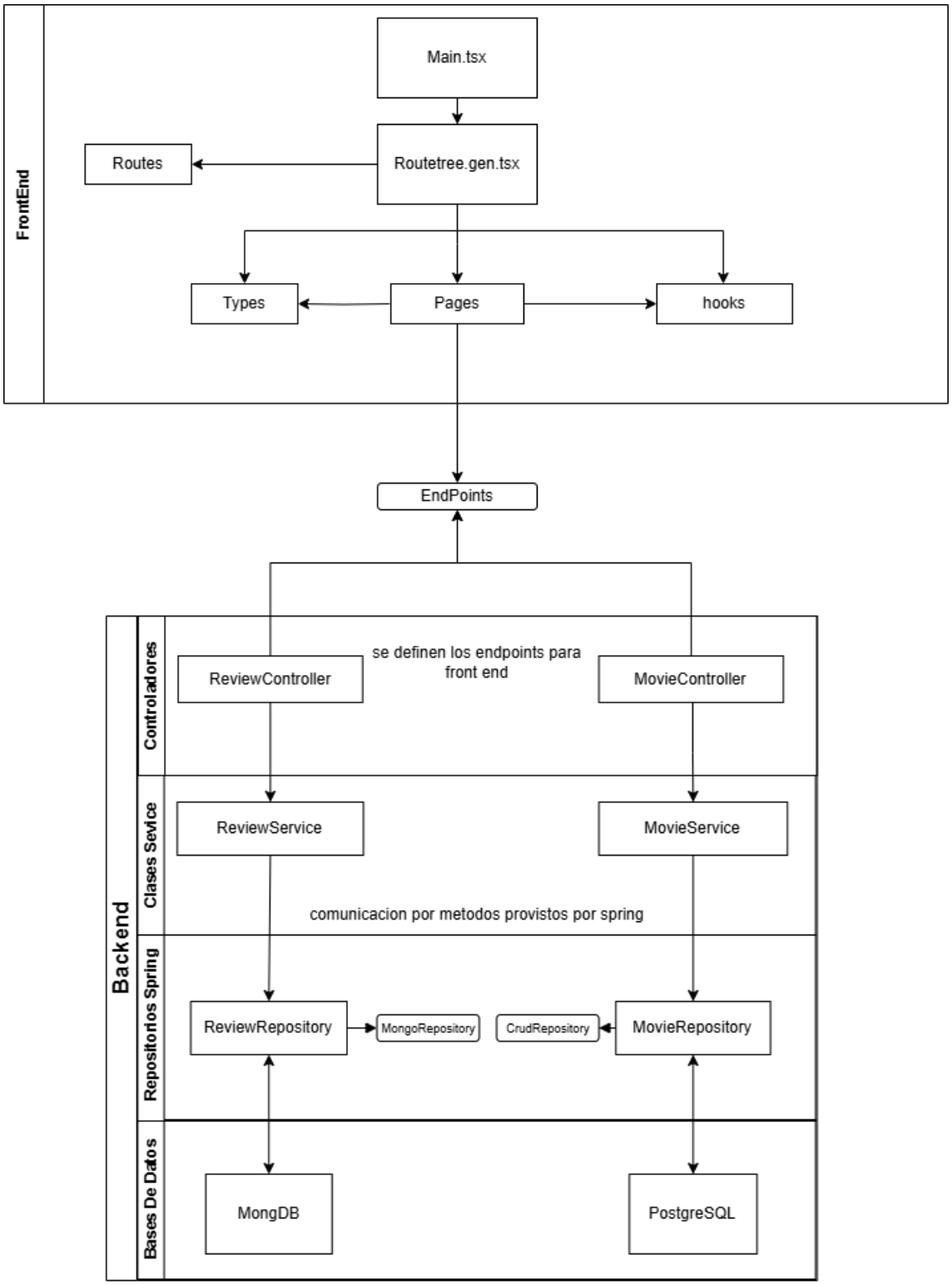
1.3. Bases de datos

Por último, decidimos utilizar dos bases de datos diferentes porque cada una pertenece a un tipo distinto. Para la base de datos SQL, elegimos PostgreSQL debido a que ya contábamos con experiencia previa trabajando con ella, lo que facilitó su manejo. Además, PostgreSQL ofrece importantes ventajas, como la capacidad de soportar grandes volúmenes de trabajo y su compatibilidad con plataformas de alta disponibilidad.

En cuanto a la base de datos NoSQL, seleccionamos MongoDB. Aunque no teníamos experiencia previa con este tipo de bases de datos, optamos por MongoDB debido a su facilidad de uso y a su lenguaje intuitivo. Entre las ventajas de MongoDB destacan su esquema flexible, que permite gestionar datos no estructurados y semiestructurados, y su capacidad para almacenar datos en memoria RAM, lo que mejora la velocidad de acceso y el rendimiento de las consultas.

2. Diagrama de Arquitectura

En esta sección del informe vamos a incluir un diagrama que represente de forma clara cómo está organizada la aplicación. Este mostrará cómo se comunican el frontend, el backend y las dos bases de datos entre sí, para entender mejor su funcionamiento.



3. Configuración y Conexión a Bases de Datos

En esta sección del informe, explicaremos cómo configuramos la aplicación para que se conecte a las 2 bases de datos utilizadas. Además, describiremos las librerías que empleamos para gestionar estas conexiones y cómo nos aseguramos de que funcionen correctamente dentro de la aplicación.

Para simplificar las conexiones e interacción con las bases de datos hacemos uso de la aplicación Spring Framework, agregando la dependencia a esta misma a nuestro archivo "pom.xml". Este marco nos brinda herramientas que simplifican nuestro código y evitan la repetición en el mismo.

Para la coneccion, el proceso se realiza con 2 repositorios (interfaces públicas de java) una que extienden a un repositorio de Mongodb "ReviewRepository" (guarda las reseñas de varias películas) y el otro a un "CrudRepossitory" para sql que llamamos "MovieRepository"(tiene datos de la película), ambos Repositorios a los que extienden nuestras interfaces son brindadas mediante la aplicación Spring frame, simplificando de manera significativa la estructura del código. Luego los datos necesarios para conectarse a las bases de datos se deben especificar en el archivo .env (en el formato explicado en el "README.md" que se encuentra en nuestro repositorio) para que la conexión a ambas se realice correctamente.

En las clases llamadas del tipo Service de spring framework ambas interfaces definidas anteriormente son utilizadas ("MovieRepository" y "ReviewRepository"). Estas clases Service son las encargadas de interactuar de manera directa con ambas bases de datos para guardar nuevos registros, actualizar registros anteriores, eliminarlos o recuperar información de los mismos, todo esto haciendo uso de métodos brindados por los repositorios de Spring.

Además para las validaciones de los datos y simplificación del código hacemos uso de "Jakarta Validation". Este marco funciona mediante la definición de entidades(en nuestro caso : "Actor", "Movie"y "Review") que son objetos con campos a los cuales se les define limitaciones que debe cumplir al momento de validar, y al hacerlo, la validación se hace de forma automática y sin ocupar líneas de código de más.

4. Descripción de Funcionalidades CRUD

En esta sección del informe vamos a explicar paso a paso cómo implementamos cada operación CRUD en las dos bases de datos que usamos. Además, destacaremos las principales diferencias entre cómo se manejan los datos en una base de datos relacional y en una NoSQL.

4.1. Crear

```
@PostMapping("/create")
    public ResponseEntity<Movie> createMovie(@RequestBody @Valid MovieRequest
    movieRequest){
        Movie movie = movieService.createMovie(movieRequest);
        return new ResponseEntity<>(movie, HttpStatus.CREATED);
    }
```

En el primer fragmento de código, la función createMovie se encarga de crear una nueva película. Al ser un endpoint definido con @PostMapping(/create"), se indica que esta función se ejecutará cuando se realice una solicitud HTTP POST a la ruta /create. El parámetro @RequestBody se utiliza para recibir los datos de la solicitud en formato JSON y convertirlos en un objeto MovieRequest. Además, el uso de @Valid asegura que los datos del MovieRequest sean validados antes de ser procesados. Posteriormente, se llama al servicio movieService.createMovie(movieRequest) para crear una nueva película, y el resultado, que es un objeto Movie, se retorna dentro de un ResponseEntity. Finalmente, se establece el código de estado HTTP HttpStatus.CREATED para indicar que el recurso ha sido creado correctamente.

```
@PostMapping("/create")
    public ResponseEntity<Review> createReview(@RequestBody @Valid ReviewRequest
    reviewRequest) {
        Review review = reviewService.createReview(reviewRequest);
        return new ResponseEntity<>(review, HttpStatus.CREATED);
    }
```

El segundo fragmento de código sigue una estructura similar, pero en lugar de crear una película, se encarga de crear una nueva reseña de una película. Al igual que en el primer caso, se usa

@PostMapping(/create") para definir la ruta del endpoint y @RequestBody para recibir los datos de la reseña. Los datos son convertidos en un objeto ReviewRequest, que luego es procesado por el servicio reviewService.createReview(reviewRequest) para crear la reseña correspondiente. El resultado, un objeto Review, es retornado con un código de estado HTTP HttpStatus.CREATED.

4.2. Leer

```
@GetMapping("/all")
    public ResponseEntity<List> getAllMovies() throws Exception {
        List<Movie> movies = movieService.getMovies();
        return new ResponseEntity<>(movies, HttpStatus.OK);
    }
```

La primera función, getAllMovies(), está implementada para manejar solicitudes HTTP de tipo GET en la ruta /all". Cuando se realiza una solicitud a esta ruta, la función se ejecuta y llama al servicio movieService.getMovies() para obtener una lista de todas las películas almacenadas en el sistema. Esta lista de películas se guarda en una variable de tipo List[Movie]. Luego, la función crea y devuelve una respuesta HTTP utilizando ResponseEntity. En esta respuesta se incluye la lista de películas junto con el código de estado HttpStatus.OK, lo que indica que la solicitud se ha procesado correctamente y se ha devuelto la información solicitada.

```
@GetMapping("/all")
    public ResponseEntity<List> getAllReviews() throws Exception {
        List<Review> reviews = reviewService.getAllReviews();
        return new ResponseEntity<>(reviews, HttpStatus.OK);
    }
```

La segunda función, getAllReviews(), tiene una implementación similar pero se enfoca en las reseñas. También maneja solicitudes GET a la ruta /all". En esta función, se llama al servicio reviewService.getAllReviews() para obtener una lista de todas las reseñas disponibles. Esta lista se almacena en una variable de tipo List[Review]. Al igual que en la primera función, se devuelve una respuesta HTTP utilizando ResponseEntity, donde se incluye la lista de reseñas y el código de estado HttpStatus.OK, indicando que la solicitud fue exitosa y la información solicitada se ha entregado correctamente.

4.3. Actualizar

```
@PatchMapping("/updateRating/{movieId}")
    public ResponseEntity<Movie> updateRating(@PathVariable Long movieId,
    @RequestBody @Valid UpdateRatingRequest updateRatingRequest) throws Exception {
        Movie movie = movieService.updateRating(movieId, updateRatingRequest);
        return new ResponseEntity<>(movie, HttpStatus.OK);
    }
```

En el primer caso, la función updateRating es un controlador que maneja solicitudes HTTP de tipo PATCH para actualizar la calificación de una película. La ruta está definida como /update-Rating/movieId, donde movieId es una variable de ruta que se utiliza para identificar la película cuyo rating será actualizado. Al recibir una solicitud, el valor de movieId se extrae de la URL mediante la anotación @PathVariable, mientras que los nuevos datos para la calificación son recibidos en el cuerpo de la solicitud y se validan mediante la anotación @RequestBody @Valid. Esta validación garantiza que los datos de la calificación cumplan con las reglas establecidas en el objeto UpdateRatingRequest. La función luego llama al servicio movieService.updateRating para procesar la actualización de la calificación de la película con la información proporcionada. Una vez completada la actualización, el controlador devuelve una respuesta HTTP con el objeto Movie actualizado, junto con un código de estado HttpStatus.OK que indica que la solicitud se procesó correctamente.

```
@PatchMapping("/updateReview/{id}")
    public ResponseEntity<Review> updateReview(@PathVariable String id,
    @RequestBody @Valid ReviewRequest reviewRequest) throws Exception {
```

```
        Review review = reviewService.updateReview(id, reviewRequest);
        return new ResponseEntity<>(review, HttpStatus.OK);
    }
```

En el segundo caso, la función updateReview es similar, pero su propósito es actualizar una revisión de una película. La ruta está definida como /updateReview/id, donde id representa el identificador único de la revisión que se desea actualizar. Al igual que en el primer caso, el identificador de la revisión se extrae de la URL utilizando @PathVariable. Los datos de la revisión, que están en el cuerpo de la solicitud, se reciben y validan mediante @RequestBody @Valid, garantizando que la información cumpla con los criterios especificados en el objeto ReviewRequest. Luego, el controlador llama al servicio reviewService.updateReview, el cual se encarga de actualizar la revisión en la base de datos. Tras realizar la actualización, el controlador devuelve el objeto Review actualizado como parte de la respuesta, junto con el código de estado HttpStatus.OK, que indica que la operación fue exitosa.

4.4. Eliminar

```
@DeleteMapping("/delete/{movieId}")
    public ResponseEntity<Void> deleteMovie(@PathVariable Long movieId) throws
    Exception {
        movieService.deleteMovie(movieId);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
```

La función deleteMovie está implementada en el controlador mediante la anotación @DeleteMapping, que indica que se trata de una operación HTTP DELETE. Esta función recibe como parámetro un movieId de tipo Long, que se obtiene de la URL de la solicitud mediante la anotación @PathVariable. El valor de movieId se pasa al servicio movieService, que se encarga de eliminar la película correspondiente en la base de datos. Una vez completada la eliminación, la función retorna una respuesta de tipo ResponseEntity con el estado HTTP NO CONTENT , lo que indica que la operación se completó exitosamente y no hay contenido adicional que devolver.

```
@DeleteMapping("/delete/{id}")
    public ResponseEntity<Void> deleteReview(@PathVariable String id) throws
    Exception {
        reviewService.deleteReview(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
```

De manera similar, la función deleteReview también está implementada con la anotación @DeleteMapping, lo que indica que maneja una solicitud HTTP DELETE. En este caso, la función recibe un parámetro id de tipo String, que se obtiene de la URL a través de la anotación @PathVariable. Este identificador se pasa al servicio reviewService, que se encarga de eliminar la reseña asociada en la base de datos. Al igual que en la función anterior, después de completar la eliminación, la función devuelve una respuesta de tipo ResponseEntity con el estado HTTP NO CONTENT, indicando que la operación fue exitosa sin necesidad de devolver contenido adicional.

4.5. Manejo de datos

En cuanto a las diferencias clave entre el manejo de datos relacionales (como en el caso de una base de datos SQL) y NoSQL (como en MongoDB), es importante destacar lo siguiente:

4.5.1. Estructura de los Datos:

SQL: Los datos se organizan en tablas con un esquema fijo. Cada tabla tiene columnas definidas y las filas deben cumplir con estas estructuras. Esto requiere que los datos sean consistentes y estructurados.

NoSQL: Los datos no requieren un esquema fijo. Esto permite una mayor flexibilidad al manejar datos no estructurados o semiestructurados. En MongoDB, por ejemplo, los datos se almacenan en documentos JSON, lo que permite diferentes formas de almacenar la información.

4.5.2. Relaciones entre los Datos:

SQL: En bases de datos relacionales, las relaciones entre las tablas se manejan mediante claves primarias y foráneas, lo que permite establecer vínculos entre los diferentes tipos de datos. Esto es ideal para aplicaciones que requieren transacciones y un fuerte control sobre la integridad de los datos.

NoSQL: En bases de datos como MongoDB, no se utilizan claves foráneas de la misma manera. Los documentos pueden almacenar otros documentos dentro de ellos, lo que facilita consultas más simples pero a menudo menos estructuradas. Las relaciones en bases de datos NoSQL pueden ser más flexibles y no tan estrictas como en las bases de datos relacionales.

4.5.3. Escalabilidad:

SQL: Las bases de datos SQL son generalmente verticales, lo que significa que se escalaban añadiendo más recursos a un solo servidor. Esto puede ser limitado en cuanto a capacidad.

NoSQL: Las bases de datos NoSQL, como MongoDB, son más fáciles de escalar horizontalmente, distribuyendo los datos en múltiples servidores, lo que las hace más adecuadas para manejar grandes volúmenes de datos y consultas distribuidas.

4.5.4. Consultas:

SQL: Las consultas en bases de datos SQL se realizan mediante el lenguaje SQL (Structured Query Language), que permite realizar consultas complejas y combinaciones de tablas para obtener información detallada.

NoSQL: MongoDB utiliza un lenguaje de consulta basado en documentos que es más intuitivo y flexible, aunque no tiene la misma capacidad para realizar consultas complejas con múltiples relaciones entre datos como en SQL.

5. Comparación entre Bases de Datos Relacionales y NoSQL

En este apartado analizaremos los puntos positivos y negativos que encontramos al trabajar con cada tipo de base de datos según el contexto de nuestra aplicación. También reflexionaremos sobre en qué casos sería mejor usar una base de datos relacional y en cuáles sería más conveniente optar por una NoSQL.

5.1. Ventajas y Desventajas

Al trabajar con las dos bases de datos en el contexto de la aplicación, se observaron tanto ventajas como desventajas que dependen del tipo de base de datos utilizado y de las funciones que implementamos. En el caso de PostgreSQL, como base de datos SQL, la principal ventaja fue su familiaridad debido al conocimiento previo que teníamos, lo que facilitó su integración y manejo. Además, PostgreSQL demostró ser una opción robusta, capaz de soportar grandes volúmenes de datos y consultas complejas, lo que resulta ventajoso en aplicaciones que requieren alta disponibilidad y fiabilidad. Sin embargo, una desventaja que surgió fue que, al ser una base de datos con esquema rígido, resultó más difícil manejar datos no estructurados o semiestructurados, lo que obligó a realizar un esfuerzo adicional en la estructura y diseño de la base de datos.

Por otro lado, al trabajar con MongoDB, una base de datos NoSQL, la principal ventaja fue su flexibilidad. Al no requerir un esquema fijo, permitió manejar de forma más sencilla y eficiente

datos no estructurados y semiestructurados, lo que se adaptó mejor a las necesidades de ciertos módulos de la aplicación. Además, MongoDB almacena los datos en memoria RAM, lo que mejora la velocidad de acceso y el rendimiento de las consultas. A pesar de estas ventajas, la falta de un conocimiento previo de MongoDB hizo que al principio su implementación fuera más desafiante, aunque la curva de aprendizaje fue rápida debido a su lenguaje de consultas intuitivo. En resumen, cada tipo de base de datos aportó beneficios según el contexto de uso, pero también presentó desafíos específicos que requirieron atención durante su implementación.

5.2. Reflexiones

En el contexto de la implementación de funciones para gestionar bases de datos, es importante considerar las situaciones en las que sería más conveniente optar por una base de datos relacional, como PostgreSQL, frente a una base de datos NoSQL, como MongoDB, y viceversa. Las bases de datos relacionales son ideales cuando los datos tienen una estructura bien definida, como tablas con relaciones entre sí, y es necesario realizar operaciones complejas como uniones y transacciones. Esto es especialmente útil cuando se requiere garantizar la integridad de los datos y realizar consultas detalladas que involucren múltiples tablas o relaciones, como en sistemas de gestión de inventarios, aplicaciones bancarias o de administración de recursos humanos. La capacidad de PostgreSQL para manejar grandes volúmenes de datos y su compatibilidad con plataformas de alta disponibilidad lo hacen adecuado para escenarios que exigen alta fiabilidad y rendimiento.

Por otro lado, las bases de datos NoSQL como MongoDB son más adecuadas en situaciones donde los datos son menos estructurados o evolucionan rápidamente, como en aplicaciones web, plataformas sociales o sistemas de análisis de grandes volúmenes de datos no estructurados. MongoDB permite trabajar con datos en formatos flexibles, lo que facilita la adaptación a cambios en el modelo de datos sin necesidad de una estructura fija. Además, su capacidad de almacenar datos en memoria RAM mejora el rendimiento de las consultas y permite un acceso más rápido a la información, lo que resulta beneficioso para aplicaciones que necesitan escalabilidad y velocidad, pero no requieren relaciones complejas entre los datos. En resumen, la elección entre una base de datos relacional y NoSQL dependerá de la naturaleza de los datos, la necesidad de flexibilidad, el tipo de consultas y el rendimiento requerido por la aplicación.

6. Dificultades y Aprendizajes:

En esta sección explicaremos los desafíos más importantes que surgieron mientras trabajábamos en el proyecto y cómo los enfrentamos para resolverlos. Además, compartiremos nuestras conclusiones finales y las lecciones que aprendimos, tanto de manera individual como en equipo, durante el desarrollo del trabajo.

6.1. Desafios enfrentados

Durante el desarrollo del proyecto, uno de los principales desafíos fue establecer una conexión adecuada entre la base de datos y el código. Para lograrlo, fue necesario investigar y comprender los detalles de la configuración y las credenciales necesarias para cada tipo de base de datos, tanto SQL como NoSQL. Esto implicó aprender a usar bibliotecas y herramientas específicas que facilitaban la comunicación entre la base de datos y el código del backend, asegurando que la información pudiera ser consultada y almacenada correctamente.

Otro reto importante fue la integración del backend con el frontend. Dado que ambos trabajan en diferentes capas de la aplicación, fue necesario implementar una interfaz de comunicación clara y eficiente. Esto se resolvió mediante el uso de API RESTful, que permitió al frontend hacer solicitudes HTTP al backend para acceder a los datos almacenados en las bases de datos. La gestión de estas solicitudes y las respuestas adecuadas entre ambas partes fue una tarea compleja que requirió atención a los detalles en la implementación de rutas y controladores.

Además, reunir los conocimientos necesarios para trabajar con ambas bases de datos fue otro desafío. Aunque parte del equipo ya tenía experiencia previa con PostgreSQL, fue necesario actualizar y profundizar en el manejo de esta base de datos, mientras que, por otro lado, MongoDB representó un área de aprendizaje nueva. Para superar esta barrera, se dedicaron tiempo y esfuerzos a investigar y comprender las características y ventajas de cada base de datos. Esto permitió al equipo utilizar de manera efectiva las herramientas y las funcionalidades de ambas, aprovechando las ventajas de cada una según el tipo de datos y las necesidades específicas del proyecto.

6.2. Conclusiones

A lo largo del desarrollo de este proyecto, se han implementado diversas funciones y estrategias para abordar los requerimientos de la consigna. Se trabajó con dos tipos diferentes de bases de datos, SQL y NoSQL, eligiendo PostgreSQL y MongoDB, respectivamente. La elección de PostgreSQL se basó en el conocimiento previo que teníamos de esta base de datos, lo que facilitó su implementación. A su vez, se valoraron sus ventajas, como la capacidad de manejar grandes cargas de trabajo y su compatibilidad con plataformas de alta disponibilidad. Por otro lado, MongoDB fue seleccionado por su facilidad de uso, especialmente debido a su lenguaje intuitivo, y por sus características, como la flexibilidad en el esquema y su eficiencia en el manejo de datos en memoria RAM, lo que permite consultas rápidas y de alto rendimiento.

En cuanto al trabajo en equipo, aprendimos la importancia de la comunicación y la colaboración, ya que al ser dos tipos diferentes de bases de datos, cada uno presentaba desafíos únicos que requerían un enfoque específico. Además, fue fundamental el conocimiento previo y el aprendizaje de nuevas tecnologías para poder integrar y aprovechar de la mejor manera las ventajas de cada base de datos. Esta experiencia nos permitió fortalecer nuestra capacidad para tomar decisiones basadas en las necesidades del proyecto y en las características de las herramientas disponibles.

A nivel personal, aprendí la relevancia de la flexibilidad al enfrentar problemas complejos, así como la importancia de contar con una base sólida de conocimientos previos que facilite la implementación de soluciones. También fue un ejercicio valioso para mejorar nuestra capacidad de adaptación a nuevas tecnologías y enfoques, lo cual es esencial en el desarrollo de proyectos de software. En resumen, el proyecto no solo nos permitió aplicar conocimientos previos, sino también aprender y adaptarnos a nuevas herramientas y métodos que enriquecerán nuestra práctica profesional.