Pontificia Universidad JAVERIANA Cali | Facultad de Ingeniería y Ciencias

Acta de Correcciones al Proyecto de Grado
Ingeniería de Sistemas y Computación

Fecha: 16/02/2024

Autores:
Paul Harvey Martinez Alcala
Sebastian Mena Ferreira

Nombre del Proyecto de Grado:
Sistema para el seguimiento de trabajos de grado

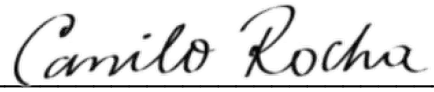
Director:
Dr. Gerardo Mauricio Sarria Montemiranda

Como indica el artículo 2.27 de las Directrices de Trabajo de Grado, he verificado que los estudiantes indicados arriba han implementado todas las correcciones que los Jurados del Proyecto de Grado definieron que se efectuaran, como consta en el Acta de Calificación correspondiente.

Firma de Director(a) del Proyecto de Grado

Nota de Aceptación

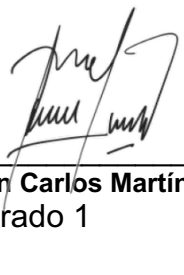
Aprobado por el Comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar el título de Ingeniero de Sistemas y Computación.



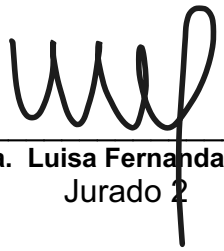
Dr. Hernán Camilo Rocha Niño
Decano de la Facultad de Ingeniería

Dr. Gerardo Mauricio Sarria Montemiranda
Director Carrera Ingeniería Sistemas y Computación.

Dr. Gerardo Mauricio Sarria Montemiranda
Director(a) Trabajo



MSc. Juan Carlos Martínez Arias
Jurado 1



Dra. Luisa Fernanda Rincón Pérez
Jurado 2

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería y Ciencias.
Ingeniería de Sistemas y Computación.
Proyecto de Grado.

Sistema para el seguimiento de trabajos de grado

Sebastián Mena Ferreira
Paul Harvey Martínez Alcalá

Director: Dr. Gerardo Sarria

Enero 2024

Santiago de Cali, Enero 2024.

Señores
Pontificia Universidad Javeriana Cali
Dr. Gerardo Sarria
Director Carrera de Ingeniería de Sistemas y Computación
Cali.

Cordial Saludo.

Por medio de la presente me permito informarle que los estudiantes de Ingeniería de Sistemas y Computación Sebastián Mena Ferreira (cod: 8947234) y Paul Harvey Martínez Alcalá(cod: 8947822) trabajan bajo mi dirección en el proyecto de grado titulado "Sistema para el seguimiento de trabajos de grado".

Atentamente,

Dr. Gerardo Sarria

Santiago de Cali, Enero 2024.

Señores
Pontificia Universidad Javeriana Cali
Dr. Gerardo Sarria
Director Carrera de Ingeniería de Sistemas y Computación
Cali.

Cordial Saludo.

Nos permitimos presentar a su consideración el anteproyecto de grado titulado "Sistema para el seguimiento de trabajos de grado" con el fin de cumplir con los requisitos exigidos por la Universidad para llevar a cabo el proyecto de grado y posteriormente optar al título de Ingeniero de Sistemas y Computación.

Al firmar aquí, damos fe que entendemos y conocemos las directrices para la presentación de trabajos de grado de la Facultad de Ingeniería aprobadas el 26 de Noviembre de 2009, donde se establecen los plazos y normas para el desarrollo del anteproyecto y del trabajo de grado.

Atentamente,

Sebastián Mena F

Sebastián Mena Ferreira
Código: 8947234

Paul Harvey

Paul Harvey Martínez Alcalá
Código: 8947822

Resumen

Este trabajo de grado se enfoca en el desarrollo de una plataforma para el seguimiento administrativo de los trabajos de grado de los estudiantes de la carrera de Ingeniería en Sistemas en la Pontificia Universidad Javeriana Cali. La plataforma busca apoyar los procesos de gestión, comunicación y seguimiento de los proyectos, brindando una herramienta integral que facilite la organización y el seguimiento preciso de los avances. A través de este proyecto, se pretende mejorar la experiencia de los estudiantes, docentes y jurados involucrados. En el trabajo se describirá el proceso de desarrollo de la plataforma, incluyendo el diseño, implementación y evaluación de su funcionamiento, así como los beneficios esperados y las posibles limitaciones que puedan surgir en su implementación.

Palabras clave: Plataforma digital, Seguimiento administrativo, Gestión de proyectos, Organización, Eficiencia, Herramienta integral, Comunicación.

Abstract

This thesis focuses on the development of a platform for the administrative follow-up of the undergraduate projects of the students of the Systems Engineering program at the Pontificia Universidad Javeriana Cali. The platform seeks to support the processes of management, communication and monitoring of projects, providing a comprehensive tool that facilitates the organization and accurate monitoring of progress. Through this project, it is intended to improve the experience of students, teachers and jurors involved. The work will describe the development process of the platform, including the design, implementation and evaluation of its operation, as well as the expected benefits and possible limitations that may arise in its implementation.

Keywords: Digital platform, Administrative follow-up, Project management, Organization, Efficiency, Integral tool, Communication.

Índice general

Capítulo 1

Introducción

En el campo de la educación superior, los trabajos de grado son una parte fundamental para la culminación de los estudios de los estudiantes. En el caso específico de la carrera de Ingeniería en Sistemas, estos proyectos de investigación y de desarrollo representan una oportunidad para aplicar y demostrar los conocimientos adquiridos a lo largo de la formación académica.

Sin embargo, el proceso administrativo relacionado con los trabajos de grado puede resultar complejo y desafiante tanto para los estudiantes como para los docentes y el personal encargado de su seguimiento. La gestión de documentos, la asignación de tutores, el seguimiento de avances y la programación de evaluaciones son solo algunas de las tareas que requieren de una organización eficiente y sistemática.

En este contexto, este trabajo de grado tiene como objetivo principal el desarrollo de una plataforma digital que apoye el seguimiento administrativo de los trabajos de grado de los estudiantes de la carrera de Ingeniería en Sistemas y Computación en la Pontificia Universidad Javeriana Cali. La plataforma proporcionará una herramienta integral que agilice los procesos de gestión, promueva la comunicación fluida entre los involucrados y brinde un seguimiento preciso de los avances de cada proyecto.

En el siguiente documento, se describirá el proceso de contrucción del proyecto que estará enfocado en el desarrollo de la plataforma, incluyendo el diseño, la implementación y la evaluación de su funcionamiento. Asimismo, se presentarán los beneficios esperados y las posibles limitaciones que surgieron en su implementación.

En resumen, este trabajo de grado representa un paso hacia la mejora de los procesos administrativos relacionados con los trabajos de grado en la carrera de Ingeniería de Sistemas y Computación de la Pontificia Universidad Javeriana Cali, aportando una solución tecnológica que promueva la eficiencia y la calidad en la gestión de estos proyectos académicos.

Capítulo 2

Descripción del Problema

2.1. Planteamiento del Problema

El seguimiento de trabajos de grados en la gestión administrativa de la carrera de ingeniería de sistemas y computación en la Pontificia Universidad Javeriana Cali es un proceso que actualmente se realiza de manera manual, además, esta consume mucho tiempo debido a que el seguimiento de trabajos de grados es un proceso que se compone de muchas tareas repetitivas y que se debe realizar de manera manual.

Lo anterior conlleva a que el seguimiento de trabajos de grados sea una labor que provoca que la dirección de carrera y los jurados de tesis no tengan acceso a todo el proceso que han llevado los estudiantes durante el desarrollo de su tesis, por otro lado, esto también genera dificultades en el cumplimiento de la directriz de trabajos de grados y, adicionalmente, que los jurados desmeriten un trabajo por no conocer todo el contexto o esfuerzo que un estudiante ha empleado en su tesis.

Actualmente, la carrera no cuenta con una solución que facilite el seguimiento en la gestión asociada a los trabajos de grado. En este momento los procesos tanto administrativos como de seguimiento con los respectivos asesores son coordinados manualmente por el director de carrera, jurados o los estudiantes que están realizando el trabajo de grado.

Adicionalmente, algunos actores del proceso, específicamente los docentes y jurados, no tienen la noción de trabajo completa pues no existe un repositorio donde se pueda ver el avance constante de cada proyecto aparte del correo institucional que solo tiene acceso el estudiante y el asesor de trabajo.

Por otra parte, como la universidad no cuenta con ningún sistema que lleve a cabo este proceso, no es posible realizar ninguna clase de integración con otros sistemas digitales que posee la universidad, lo que dificulta la comunicación entre los estudiantes y diferentes actores en un proceso de trabajo de grado.

Por lo tanto, lo que se quiere lograr es construir una plataforma digital que, por medio de la tecnología, apoye y automatice las tareas y procesos asociados al seguimiento de los trabajos de grado, con el fin de agilizar dichos procesos entre director de carrera, jurados y estudiantes de la carrera de ingeniería de sistemas y computación.

2.1.1. Formulación

Con respecto a nuestra pregunta central, formulamos la siguiente:

¿Cómo desarrollar un software que apoye el seguimiento en la gestión administrativa de trabajos de grado de la carrera de ingeniería de sistemas y computación en la Universidad Javeriana Cali?

2.1.2. Sistematización

- ¿Cuáles son los requerimientos relevantes de la plataforma digital que se quiere desarrollar?
- ¿Cuál es el diseño de arquitectura e interfaz gráfica que más se adecua a la solución respecto a los requerimientos del software obtenidos?
- ¿Cómo implementar la plataforma digital para automatizar y apoyar la gestión administrativa de trabajos de grado?
- ¿Qué pruebas de software se van a realizar para validar el software?

2.2. Objetivos

2.2.1. Objetivo General

Desarrollar un software para apoyar el seguimiento y la gestión administrativa de trabajos de grado de la carrera de ingeniería de sistemas y computacíon en la Pontificia Universidad Javeriana Cali.

2.2.2. Objetivos Específicos

- Obtener los requerimientos del sistema a desarrollar.
- Diseñar la arquitectura e interfaz gráfica con la que se va a implementar la solución tecnológica.
- Implementar el software de acuerdo a los requisitos y al diseño previamente acordado y establecido.
- Validar el software desarrollado mediante pruebas de aseguramiento de calidad y de usabilidad.

2.3. Justificación

El seguimiento administrativo de los trabajos de grado en ingeniería de sistemas y computación en la Pontificia Universidad Javeriana Cali es crucial para el éxito de los estudiantes. Sin embargo, este proceso manual y repetitivo consume mucho tiempo y dificulta la gestión y el acceso a la información por parte de directores de carrera y jurados. La falta de una solución digital impide una visión completa del progreso de los proyectos. Por lo tanto, se propone desarrollar una plataforma

digital que reduzca intermediarios y mejore la comunicación y el acceso a información en tiempo real. El objetivo es apoyar el seguimiento administrativo de los trabajos de grado, a través del diseño, implementación y validación de un software que cumpla con los requisitos establecidos.

La implementación de esta solución permitirá la automatización de tareas repetitivas y la reducción de intermediarios entre los diferentes actores del proceso, lo que a su vez facilitará la gestión y seguimiento de los proyectos por parte de los jurados, directores de carrera y estudiantes.

Además, la plataforma digital permitirá una mejor integración con otros sistemas de la universidad, lo que facilitará la comunicación y el intercambio de información entre los diferentes actores involucrados en los proyectos.

En conclusión, la implementación de un software de seguimiento en la gestión de trabajos de grado es una herramienta necesaria para mejorar la eficiencia y calidad de los proyectos de investigación de los estudiantes de ingeniería en sistemas de la Pontificia Universidad Javeriana Cali.

2.4. Delimitaciones y Alcances

2.4.1. Alcances

Se desarrollará una solución tecnológica que sea capaz de facilitar la realización de las tareas manuales y repetitivas para los trabajos de grado. Así mismo, se implentará la solución para la carrera de sistemas. De igual manera, la solución se desarrollará conforme a los estándares de las soluciones tecnológicas ya existentes en la universidad para facilitar su escalabilidad e integración a futuro. Por último, se dejará establecido una documentación detallada de la solución.

2.5. Limitaciones

Las limitaciones que se van a tener en cuenta son las siguientes: En primer lugar, la información con la que trabaje el sistema se irá recopilando a partir de su uso. Es decir, para trabajos de grado anteriores esta solución no estará disponible.

De igual manera, los usuarios tendrán que familiarizarse con el sistema por medio del manual de usuario que va a estar disponible para ellos (No habrá jornada de capacitación).

En el mismo sentido, el soporte de la solución final se basará netamente en la documentación entregada. Finalmente, la solución entregada será del tipo producto mínimo viable, teniendo en cuenta las restricciones de tiempo para este trabajo de grado.

Capítulo 3

Desarrollo del Proyecto

3.1. Marco de Referencia

3.1.1. Marco Teórico

Para llevar a cabo el desarrollo de este trabajo de grado es necesario tener presente algunos conceptos claves como Sistemas de Información y es que como lo menciona Alejandro Hernández (2003) todo sistema de información se basa en la utilización de datos como materia prima, los cuales son almacenados, procesados y transformados para generar información como resultado final. Esta información se suministra a los diversos usuarios del sistema, y se establece un proceso de retroalimentación, mediante el cual se evalúa si la información obtenida cumple con las expectativas previstas.[HT03]

Es fundamental tener en cuenta la definición previa en el desarrollo de trabajo de grado debido a que sienta las bases para comprender la naturaleza y el propósito de la plataforma. Al entender que los datos son la materia prima que se transforma en información valiosa para los usuarios, se puede garantizar que la plataforma está diseñada para capturar, procesar y presentar los datos de manera efectiva. Al tener presente esta definición, se asegura que la plataforma esté orientada a brindar una experiencia eficiente y satisfactoria tanto para los estudiantes como para los docentes y personal administrativo involucrado en el seguimiento de los trabajos de grado. Además, permite establecer mecanismos de mejora continua en la plataforma, facilitando la adaptación y optimización de los procesos para lograr un seguimiento administrativo efectivo y exitoso de los proyectos de investigación.

Por otro lado, también surge el término de Ingeniería de Requisitos pues de acuerdo con Jorge Reyes (2020) se resalta la trascendental importancia de la etapa de requisitos en el desarrollo de un producto de software. La etapa de requisitos se posiciona como una fase clave en el proceso de desarrollo de software, ya que aquí se establecen las bases para la planificación del proyecto. En este sentido, al determinar los requisitos, se pueden estimar de manera más precisa los recursos y el tiempo necesario para la implementación del producto de software. Además, la especificación detallada de requisitos sirve para tener un punto de partida al momento de evaluar si los objetivos del proyecto están siendo alcanzados.[RE03]

De acuerdo con lo anterior, se hace de alta importancia contar con una fase de levantamiento de requisitos antes de la etapa de desarrollo pues así se garantiza que el producto final se alinee con lo

establecido en los objetivos, además, al tener bien definido los requisitos desde un principio pueden puede mitigar posibles retrasos durante la fase de desarrollo. Al establecer con claridad y precisión los requisitos desde un inicio, se minimiza la probabilidad de malentendidos y cambios significativos en etapas avanzadas del proyecto. Esto no solo optimiza la utilización de recursos y tiempo, sino que también posibilita un proceso de desarrollo más fluido y eficiente, evitando interrupciones innecesarias y garantizando la consecución exitosa de los objetivos planteados.

Por último, para llevar a cabo la construcción del sistema para el seguimiento de trabajos de grado se optó por emplear como plataforma de desarrollo una aplicación web debido a su accesibilidad, facilidad de uso y amplio respaldo. Este enfoque se adopta con el propósito de brindar a los usuarios una experiencia intuitiva y amigable, permitiendo una interacción fluida con la plataforma sin requerir conocimientos técnicos avanzados. Además, como se menciona en SISDAM (2016) una aplicación web ofrece ventajas en términos de soporte y mantenimiento, simplificando la implementación de actualizaciones y correcciones.[MR16]

3.1.2. Antecedentes

A continuación, se mencionan algunos proyectos relacionados con la plataforma web que se quiere desarrollar en este trabajo de grado.

Web Based Student Information Management System [BGT13]
La evolución constante de la tecnología de la información ha impulsado el desarrollo y la adopción de sistemas de gestión para optimizar diferentes áreas, incluida la administración de estudiantes. El Sistema de Gestión de Información de Estudiantes (SIMS) representa un ejemplo destacado de esta tendencia, al proporcionar una plataforma integral que permite a instituciones educativas y colegios mantener registros precisos y actualizados de sus estudiantes. El SIMS aborda diversos aspectos, desde información personal y académica hasta detalles universitarios, informes y seguimiento de avances en el curso. A través de su interfaz en línea segura integrada en los sitios web de las instituciones, el sistema facilita la comunicación y notificaciones, así como la generación de informes detallados.

Este enfoque exitoso en la gestión de la información estudiantil se convierte en un antecedente valioso para el desarrollo de la propuesta actual en el contexto de la carrera de Ingeniería en Sistemas. Al tomar inspiración de la implementación efectiva del SIMS, se busca diseñar una plataforma que permita una supervisión y administración óptimas de los trabajos de grado. A través de una interfaz intuitiva y segura, el sistema facilitará la gestión de detalles relacionados con los trabajos de grado, el seguimiento de avances, la generación de informes y notificaciones relevantes para el personal académico y los estudiantes involucrados en este proceso.

Design and Implementation of a Novel Student Information Management System[WFQ20]

Este proyecto expone los desafíos presentes en el sistema de gestión de información estudiantil actual, como el prolongado ciclo de desarrollo, el mantenimiento complejo y la mala experiencia del usuario, este trabajo se enfoca en el diseño e implementación de un nuevo sistema de gestión de información estudiantil. Se adopta un enfoque Cliente/Servidor y se integra un terminal móvil utilizando el miniprograma WeChat para el desarrollo, junto con la creación de un cliente basado en Qt Quick.

Esta referencia se convierte en una valiosa guía para el presente trabajo de grado, pues los aprendizajes obtenidos en el diseño e implementación del sistema de gestión de información estudiantil ofrecen una base sólida para abordar eficazmente los retos de la gestión de trabajos de grado. Se espera que esta experiencia aporte un alivio en la intensidad de trabajo del personal de gestión, permita un desarrollo más eficiente, fácil mantenimiento y un ciclo de desarrollo más corto, brindando una solución práctica y orientada a las necesidades específicas de los trabajos de grado de la carrera de Ingeniería en Sistemas.

Sistema de información HERMES[dC22]
El sistema de información Hermes, creado en 2005 y adoptado en 2007 por la Universidad Nacional de Colombia como una herramienta para la recopilación de datos relacionados con las actividades de investigación, ofrece un ejemplo relevante de cómo una plataforma centralizada puede ser implementada para gestionar eficientemente información clave en un entorno académico. En el 2007 se establecieron roles específicos y procedimientos para la entrada de proyectos de investigación en el sistema, demostrando cómo un enfoque estructurado puede facilitar la recopilación sistemática de datos.

Si bien inicialmente se centró en la investigación, fue en 2013 cuando el sistema Hermes experimentó una expansión y consolidación significativa. Este proceso incluyó la incorporación de módulos adicionales relacionados con investigación y laboratorios, con el propósito de extender su alcance y aplicabilidad a nivel nacional y lograr una uniformidad en la automatización de procesos en todas las sedes de la universidad. Esta plataforma resalta la importancia de un enfoque sistémico y la estandarización de procesos en la administración de actividades académicas, sentando las bases para la implementación de sistemas similares destinados a la gestión administrativa de trabajos de grado en el contexto de la carrera de Ingeniería en Sistemas de la Pontificia Universidad Javeriana Cali.

Capítulo 4

Análisis y Consideraciones Iniciales

4.1. Levantamiento de Requerimientos

La obtención de requerimientos para el desarrollo de este trabajo de grado constituye la fase de análisis inicial y esta consta de dos etapas. La primera fue previo al desarrollo y fue en la que se tuvo el primer acercamiento con los usuarios finales, aquí se definieron las necesidades y expectivas de la aplicación. En la etapa 2 se encuentran los requerimientos funcionales y no funcionales de la aplicación los cuales definen el comportamiento esperado del sistema teniendo en cuenta los requerimientos de la etapa 1.

Etapa 1
Requerimientos de Investigación

1. Entrevista con la secretaria de la Facultad de Ingeniería y Ciencias.
   - **Descripción:** Realizar las entrevistas con la secretaria de la Facultad de Ingeniería y Ciencias para comprender y conocer el proceso que se lleva actualmente en la gestión de los trabajos de grados de los estudiante de la carrera de ingeniería de sistemas y computación.
   - **Criterio de éxito:** Se deben analizar los principales procesos que conlleva la gestión de un trabajo de grado en la carrera de ingeniería de sistemas y computación.
2. Entrevista con el director de la carrera ingeniería en sistemas y computacíon.
   - **Descripción:** Realizar las entrevistas con el director de la carrera de ingeniería de sistemas y computación para conocer las expectativas y necesidades de la aplicación final que se quiere desarrollar.
   - **Criterio de éxito:** Se deben analizar las necesidades y expectativas identificadas por parte del director de carrera.
3. Estudio y análisis de la arquitectura de la aplicación.
   - **Descripción:** Realizar una investigación para establecer la arquitectura de la aplicación teniendo en cuenta las necesidades y los estándares actuales del desarrollo de software.
   - **Criterio de éxito:** Se debe definir la arquitectura de la aplicación a desarrollar.
4. Anális de las tecnologías Backend.

- **Descripción:** Realizar una investigación para definir cuales son las tecnologías backend que se adecuan de mejor manera a las necesidades y arquitectura previamente establecidas.
- **Criterio de éxito:** Se deben definir las tecnologías backend que se van a utilizar en el desarrollo final de la aplicación.

5. Anális de las tecnologías Frontend.

   - **Descripción:** Realizar una investigación para definir las tecnologías frontend que se amolden a los requisitos y necesidades de los usuarios finales.
   - **Criterio de éxito:** Se deben definir las tecnologías y librerías Frontend que se van a emplear en el desarrollo final de la aplicación.

Etapa 2

Requerimientos Funcionales

1. Creación de usuarios.

   - **Descripción:** La aplicación debe permitir a los usuarios registrarse el sistema ingresando su nombre, apellido, identificación, correo institucional y contraseña,
   - **Criterio de éxito:** Se valida en el sistema que el usuario ha sido creado.

2. Ingreso al sistema.

   - **Descripción:** La aplicación debe permitir ingresar a los usuarios registrados en el sistema ingresando su correo institucional y contraseña.
   - **Criterio de éxito:** Se valida el ingreso al sistema mostrandole al usuario la página principal de la aplicación.

3. Asignación de roles.

   - **Descripción:** Los usuarios administradores deben poder asignar roles a los usuarios del sistema.
   - **Criterio de éxito:** Se evidencia el cambio de los roles de los usuarios en el sistema.

4. Creación de anteproyectos.

   - **Descripción:** Los usuarios administradores deben poder crear anteproyectos en el sistema ingresando el titulo, número de radicado y al menos un autor.
   - **Criterio de éxito:** Se valida en el sistema la creación de los anteproyectos por parte de los usuarios administradores.

5. Edición de anteproyectos.

- **Descripción:** Los usuarios administradores deben poder modificar los anteproyectos existentes en el sistema.
- **Criterio de éxito:** Se valida en el sistema la modificación de los anteproyectos por parte de los usuarios administradores.

6. Eliminación de anteproyectos.

   - **Descripción:** Los usuarios administradores deben poder eliminar los anteproyectos existentes en el sistema.
   - **Criterio de éxito:** Se valida en el sistema la eliminación de los anteproyectos y en caso de existir el trabajo de grado asociado.

7. Creación de trabajos de grados.

   - **Descripción:** Los usuarios administradores deben poner crear trabajos de grado a partir de un anteproyecto.
   - **Criterio de éxito:** Se valida en el sistema la creación de los anteproyectos por parte de los usuarios administradores.

8. Edición de trabajos de grados.

   - **Descripción:** Los usuarios administradores deben poder modificar los trabajos de grados existentes en el sistema.
   - **Criterio de éxito:** Se valida en el sistema las modificaciones realizadas por los usuarios administradores en los trabajos de grado.

9. Eliminación de trabajos de grados.

   - **Descripción:** Los usuarios administradores deben poder eliminar los trabajos de grados existentes en el sistema.
   - **Criterio de éxito:** Se valida en el sistema la eliminación de los trabajos de grados por parte de los usuarios administradores.

10. Subir archivos.

    - **Descripción:** Los usuarios administradores deben poder subir archivos en un anteproyecto o a un trabajo de grado.
    - **Criterio de éxito:** Se registran los archivos subidos en el sistema a un anteproyecto o a un trabajo de grado por parte de los usuarios administradores.

11. Radicar entrega.

    - **Descripción:** Los usuarios administradores deben poder radicar entregas a los evaluadores de los anteproyectos para que estos puedan evaluar el mismo.

- **Criterio de éxito:** Se valida en el sistema la radicacón de entregas para los evaluadores de los anteproyectos.

12. Evaluar entrega.

- **Descripción:** Los usuarios evaluadores deben poder evaluar aquellos anteproyectos en los que son evaluadores y que tienen entregas radicadas, en adición a lo anterior, los evaluadores debe poder agregar un documento a la evaluación de la entrega.
- **Criterio de éxito:** Se valida en el sistema la evaluación de las entregas.

Requerimientos No Funcionales

1. Tiempo de respuesta e índice de error en el módulo de registro.

- **Descripción:** Asegurar un tiempo de respuesta e índice de errores óptimo en el módulo de registro mientras se realizan 5 registros concurrentes.
- **Criterio de éxito:** Se debe validar que el modulo de registro tenga un tiempo de respuesta de como máximo de 5000ms (Milisegundos) y un índice de error de respuesta no mayor al 10 %.

2. Tiempo de respuesta e índice de error en el módulo de inicio de sesión.

- **Descripción:** Asegurar un tiempo de respuesta e índice de errores óptimo en el módulo de inicio de sesión mientras se cargan 5 registros concurrentes.
- **Criterio de éxito:** Se debe validar que el modulo de inicio de sesión tenga un tiempo de respuesta de como máximo de 5000ms (Milisegundos) y un índice de error de respuesta no mayor al 10 %.

3. Tiempo de respuesta e índice de error en el módulo de anteproyeco.

- **Descripción:** Asegurar un tiempo de respuesta e índice de errores óptimo en el módulo de anteproyecto mientras se cargan 5 registros concurrentes.
- **Criterio de éxito:** Se debe validar que el modulo de anteproyecto tenga un tiempo de respuesta de como máximo de 500ms (Milisegundos) y un índice de error de respuesta no mayor al 5 %.

4. Tiempo de respuesta e índice de error en el módulo de proyecto de grado.

- **Descripción:** Asegurar un tiempo de respuesta e índice de errores óptimo en el módulo de proyecto de grado mientras se cargan 5 registros concurrentes.
- **Criterio de éxito:** Se debe validar que el modulo de proyecto de grado tenga un tiempo de respuesta de como máximo de 500ms (Milisegundos) y un índice de error de respuesta no mayor al 5 %.

5. Usabiliad y experiencia de usuario.

- **Descripción:** Los diferentes elementos que componen la interfaz de usuario, los colores y el tipo de letra empleado deben ser consistentes y coherentes a lo largo de todo la aplicación.
- **Criterio de éxito:** La interfaz de usuario de la aplicación es intuitiva, fácil de usar y accesible para los distintos tipos de usuarios de la carrera.

4.2. Arquitectura de la solución

En esta sección, presentaremos diversos diagramas que ofrecen una visualización más clara y detallada de la concepción de nuestra solución y la arquitectura detras de esta. Estos diagramas proporcionarán una perspectiva visual que complementa la descripción textual, ofreciendo una comprensión más completa de la estructura y relaciones entre los componentes de alto nivel de nuestro sistema. De manera natural, los diagramas que veremos a continuación surgen de los requisitos y necesidades levantadas anteriormente.

4.2.1. Diagrama de casos de uso

El diagrama de la figura 4.1 representa un esquema de casos de uso para nuestra aplicación. Se han identificado tres actores, cada uno desempe ñando un rol específico en nuestra solución. La distinción de roles entre estos actores se refleja en sus comportamientos y usos únicos de la aplicación, ya que no todos interactúan con todos los módulos de la solución. Este enfoque permite una comprensión clara de las interacciones entre los actores y el sistema, ofreciendo una visión de alto nivel. Aunque la naturaleza de este tipo de diagramas es simplista y no podemos detallar a fondo todos los componentes de un modulo tan extenso como el de anteproyecto y proyecto, sirve perfectamente para entender a alto nivel nuestro sistema.

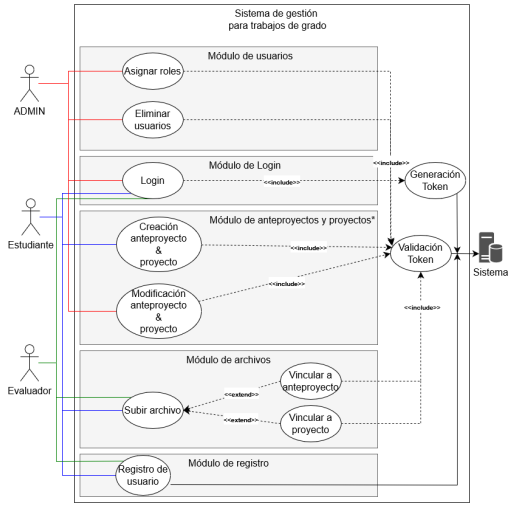


Figura 4.1: Diagrama de casos de uso.

4.2.2. Diagrama de secuencia

El diagrama de secuencia proporciona una visión general de la interacción entre un usuario y nuestro sistema, complementando el diagrama de casos de uso anterior. Sin adentrarse en detalles específicos de cada acción posible, el diagrama destaca las fases clave de la interacción. Inicia con una fase de autenticación o registro, donde el usuario establece su identidad en el sistema. Posteriormente, se observa una fase de solicitud de acciones específicas por parte del usuario. Luego, el sistema lleva a cabo una validación de la solicitud, seguida de la ejecución de la acción solicitada en el sistema de manejo de la información. Finalmente, el sistema proporciona una respuesta al usuario, cerrando el ciclo de interacción de manera clara y estructurada. Este enfoque de alto nivel del diagrama de secuencia permite comprender la dirección general de la interacción usuario-sistema sin entrar en detalles pormenorizados de cada posible acción.

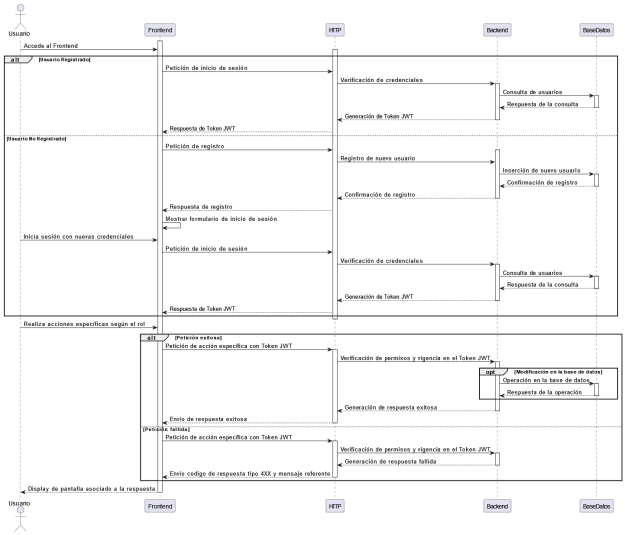


Figura 4.2: Diagrama de casos de secuencia.

4.2.3. Diagrama de arquitectura

El diagrama de la figura 4.3 representa la arquitectura actual de nuestro sistema, conocida como .ªs is". Este esquema proporciona una visión detallada de los componentes que hemos desarrollado e implementado en nuestra solución. Cada elemento del sistema, desde los microservicios hasta la base de datos, se presenta de manera clara en este diagrama, permitiendo una comprensión visual de la arquitectura de nuestra solución y cómo interactúan entre sí.

La figura 4.4 representa la visión de la arquitectura "to be", que refleja el estado deseado implementando las mejoras propuestas en la sección de trabajo futuro. Aunque las limitaciones que se abordan en capitulos mas adelante impidieron alcanzar plenamente esta arquitectura, la solución

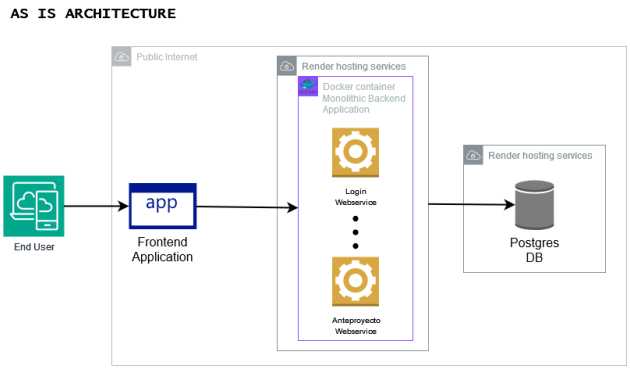


Figura 4.3: Diagrama de arquitectura AS IS.

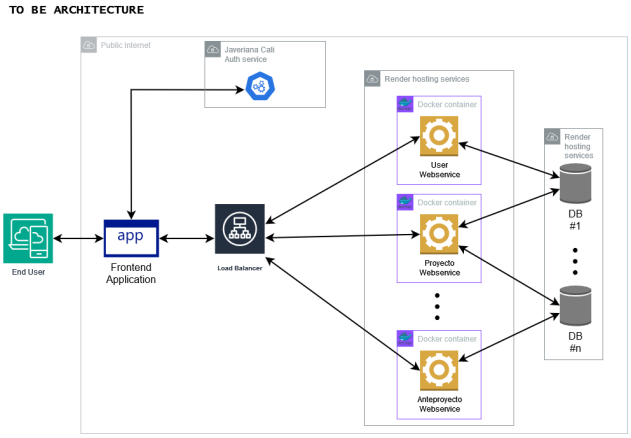


Figura 4.4: Diagrama de arquitectura TO BE.

actual sienta las bases necesarias para su implementación sin mayores inconvenientes. Las mejoras propuestas, como la independencia total de microservicios, despliegue individualizado, una base de datos más distribuida, la integración de un balanceador de cargas, y la interacción con el SSO de la universidad contribuirán significativamente a la evolución hacia esta arquitectura deseada.

Capítulo 5

Referencia de la API

5.1. Introducción a la API

En el contexto del desarrollo de este trabajo de grado, se creó una API [amz23] robusta que desempeña un papel fundamental en el sistema backend de la aplicación en cuestión. Esta API, fue construida utilizando Java [Wha23] con el framework Spring Boot [SWL+23] y respaldada por una base de datos PostgreSQL [Abo23], se erige como un componente esencial en la arquitectura de microservicios implementada en el proyecto.

La API se concibe como el núcleo del backend, encargándose de gestionar y coordinar eficientemente las solicitudes provenientes del frontend. Su función principal consiste en implementar la lógica necesaria para abordar los requerimientos presentados por los usuarios a través de la interfaz de usuario, asegurando una experiencia fluida y satisfactoria. Al interactuar con la base de datos, la API se encarga de fusionar las solicitudes de los usuarios con la información almacenada en el sistema de gestión de datos. Este enfoque garantiza la coherencia y disponibilidad de la información, contribuyendo así a una experiencia de usuario cohesiva y eficaz.

Esta API no solo actúa como un puente eficiente entre el frontend y la capa de datos, sino que internamente también incorpora principios de diseño como la cohesión y la separación de responsabilidades. La aplicación de estos principios se traduce en un código más limpio y mantenible, permitiendo adaptaciones ágiles a medida que los requisitos del usuario evolucionan. La elección de PostgreSQL como base de datos subyacente se basa en su capacidad para manejar grandes volúmenes de datos de manera eficiente y su robustez en términos de integridad y confiabilidad. De igual manera, la elección de Spring Boot como framework de trabajo se da en aras de construir una API robusta desde el principio, sin perder de vista la escalabilidad a futuro.

5.2. Propósito del API

El propósito fundamental de esta API radica en facilitar la interacción entre el frontend y la capa de datos, sirviendo como intermediario para la implementación eficiente de la lógica de negocio. Al gestionar las solicitudes del usuario, la API realiza una serie de operaciones críticas, desde la validación de datos, validación de sesión, transformación de datos, hasta la ejecución de sentencias específicas en la base de datos. Además, busca optimizar el rendimiento general del sistema al adoptar una arquitectura de microservicios, permitiendo una escalabilidad más eficaz y una gestión

modular de las funcionalidades.

En resumen, la API constituye el componente central que permite la comunicación entre el frontend y la base de datos, desempeñando un papel crucial en la realización de la visión de la aplicación propuesta en este trabajo de grado. Su diseño cuidadoso y su implementación en Java con Spring Boot se alinean con las mejores prácticas de desarrollo backend y las arquitecturas implementadas, asegurando no solo la funcionalidad óptima sino también la mantenibilidad y escalabilidad a largo plazo del sistema, que son unos de los objetivos planteados para este trabajo.

5.3. Referencia

En la documentación disponible en linea a través de Swagger [Wha] se presenta una descripción detallada de los endpoints que la API ofrece para su consumo por parte del frontend. Cada uno de estos endpoints desempeña un papel específico en la interacción entre las capas frontend y backend de la aplicación. Estas documentaciones indican (cuando apliquen) los **métodos** disponibles para cada ruta, los **parámetros de ruta**, los **parámetros de consulta**, los **códigos de respuesta** y los mensajes asociados a estas respuestas, tanto exitosas como erroneas. De igua manera, también están indicadas las **estructuras** o **schemas** necesarios para las peticiones, los **cuerpos** para los métodos POST o para los **cuerpos** de respuestas exitosas de los métodos GET. Así mismo, también está indicado si las rutas están o no protegidas y por qué esquema de seguridad, por ahora solo está implementado **bearer token**.

Esta documentación detallada de la API ofrece además del método de comunicación de sistemas de la solución, una manera en que los stakeholders pueden probar la solución de una manera sencilla. El recurso en linea en Swagger está disponible en: `https://app.swaggerhub.com/apis-docs/PAULMARTINEZ_1/API-TESIS-SPEC/1.3.0`.

Capítulo 6

Detalles de la base de datos

6.1. Selección de la base de datos

La elección meticulosa de PostgreSQL [Abo23] como sistema de gestión de bases de datos relacional [Har16] orientado a objetos para el desarrollo del trabajo de grado se basa en una evaluación profunda y consideración estratégica de diversos elementos cruciales, abarcando desde la experiencia acumulada por el equipo hasta aspectos prácticos y presupuestarios.

En primer lugar, la selección de PostgreSQL [Abo23] se sustenta en la experiencia del equipo en el manejo de bases de datos relacionales [Har16] que se adhieren al estándar SQL. Este conocimiento del modelo relacional y del lenguaje SQL surge de la formación académica y se ha fortalecido a lo largo de varias materias fundamentales de la carrera de ingeniería de sistemas y computación, a su vez como en el trabajo práctico. La familiaridad con estos conceptos no solo otorga al equipo una ventaja para el manejo de esta tecnología, simplificando las tareas de diseño, implementación y mantenimiento de la base de datos, sino que también estimula la coherencia y la eficiencia operativa en el desarrollo de la aplicación.

Un factor determinante en la elección de PostgreSQL [Abo23] es su licencia open source [Fug03]. Este enfoque no solo está en sintonía con los principios de accesibilidad y transparencia, sino que también se adapta perfectamente a las restricciones presupuestarias establecidas en el anteproyecto para este trabajo de grado. La libertad y flexibilidad proporcionadas por una base de datos de código abierto [Fug03] permiten ajustes personalizados y adaptaciones específicas según las necesidades evolutivas del proyecto, ofreciendo una plataforma sólida y personalizable.

Asimismo, la decisión de optar por PostgreSQL [Abo23] se ve respaldada por el soporte nativo de esta base de datos en la plataforma de hosting [Pol13] seleccionada. La integración fluida entre la base de datos y el entorno de alojamiento no solo simplifica la implementación y la gestión, sino que también garantiza una operación sin contratiempos y sobretodo también proporciona una sinergia efectiva entre la base de datos y la aplicacón backend también alojada en el mismo servico de hosting [Pol13]. Esto resulta beneficioso para poder lograr el cometido de entregar nuestro producto mínimo viable.

En resumen, PostgreSQL [Abo23] se establece como la tecnología fundamental para el manejo de la información del trabajo de grado, aprovechando no solo la experiencia técnica sino también el conocimiento adquirido en diversas materias de la carrera. Este enfoque estratégico garantiza un

entorno robusto y eficiente para respaldar el desarrollo y la implementación exitosa de la aplicación propuesta en el trabajo de grado.

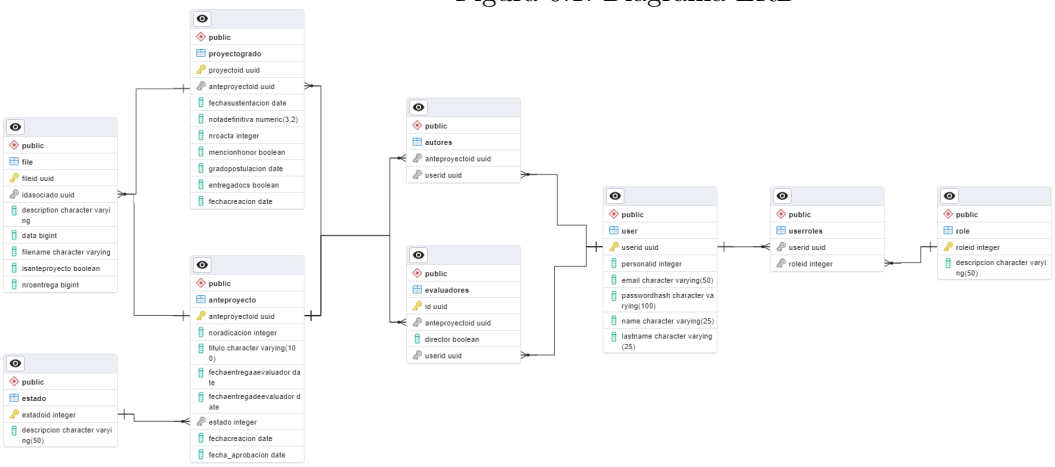
6.2. Diseño de la base de datos

Antes de adentrarnos en cualquier implementación en la base de datos o codificación en la plataforma, reconocimos la importancia de establecer una base sólida a través del diseño. El primer paso estratégico que emprendimos fue la creación y conceptualización detallada de la base de datos. Este proceso no solo sienta las bases para la eficacia y eficiencia futuras, sino que también sirve como guía esencial para el desarrollo y la interacción de datos.

Para plasmar esta visión y estructurar de manera coherente la base de datos, optamos por la creación de un Diagrama de Entidad-Relación (ERD) [LC09]. Este diagrama representa visualmente las entidades clave dentro del sistema, sus atributos y las relaciones entre ellas. La utilización del ERD se destaca como un enfoque fundamental en la planificación y conceptualización de la arquitectura de la base de datos.

El ERD proporciona una representación gráfica que va más allá de la mera estructura tabular de la base de datos. Permite visualizar y comprender las conexiones entre las diversas entidades, identificar dependencias y establecer claramente las relaciones que sustentan la lógica del sistema. Este enfoque proactivo de diseño nos ofrece una visión integral antes de pasar a las fases de implementación.

Figura 6.1: Diagrama ERD



En la Figura 5.1 que corresponde al diagrama ERD [LC09], podemos observar las nueve entidades que son conceptualizadas inicialmente para la solución propuesta. A su vez, podemos observar mediante las lineas de relación las relaciones que mantienen estas entidades entre sí. Estas entidades son: ***User***, ***role***, ***userrole***, ***autores***, ***evaluadores***, ***anteproyecto***, ***proyectograd***, ***file***, y ***estado***. Aunque no todas las entidades se encuentran en el API 5.3, internamente se hace uso de estas para poder construir las relaciones, esto sucede especialmente para las entidades que funcionan como tablas intermedias [Say20].

6.3. Configuración de la base de datos

A pesar de que nuestras tablas están directamente alineadas con las entidades de nuestro backend, lo que permite a JPA (Java Persistence API) [G+12] generar automáticamente las tablas con los requisitos necesarios, optamos por realizar una configuración manual inicial de la base de datos y cada una de las tablas. Este enfoque nos brinda un mayor control sobre la arquitectura de la base de datos y asegura que cumpla con precisión con los objetivos específicos de nuestro diseño. Así como también nos aseguró en instancias iniciales que nuestras entidades declaradas tanto en el backend como en la base de datos estaban totalmente alineadas, en columnas, tipo de datos, relaciones de llaves, consistencia de operaciones, entre otros aspectos.

A continuación, presentamos el script de creación de la entidad Anteproyecto a modo de ejemplo para mostrar como funciona esta inicialización de las entidades en nuestro sistema de manejo de la información y cómo esto respalda nuestro diseño presentado en la Figura 5.1. Para el resto de entidades como lo son: autores, estado, evaluadores, proyectograd, user, userroles, y role las particularidades de cada una de estas inicializaciones pueden ser encontradas en los atributos correspondientes para cada entidad como se ve consignado en la Figura 5.1.

```
CREATE TABLE IF NOT EXISTS public.anteproyecto
(
    anteproyectoid uuid NOT NULL,
    noradicacion integer NOT NULL,
    titulo character varying(100) COLLATE pg_catalog."default" NOT NULL,
    fechaentregaaevaluador date,
    fechaentregadeevaluador date,
    estado integer NOT NULL,
    fechacreacion date,
    fecha_aprobacion date,
    CONSTRAINT anteproyecto_pkey PRIMARY KEY (anteproyectoid)
);
```

Figura 6.2: Script de creación de tabla/entidad Anteproyecto

6.3.1. Detalles de conexión

Una vez que la base de datos está lista, obtenemos los detalles necesarios para establecer una conexión con el sistema. En nuestro caso, utilizamos la aplicación backend para esta conexión. Por lo general, configuramos manualmente estos datos, pero al alojar nuestro sistema de bases de datos en un servicio de hosting externo, estos valores nos son proporcionados automáticamente, ya que el servicio se encarga de la configuración inicial del sistema.

Detalles de conexión	
Parámetro	**valor**
Hostname	dpg-cka93uaa8h2s73bln1e0-a
Port	5432
Database	`root_m6e9`
Username	admin
Password	**********

Figura 6.3: Detalles necesarios para la conexión con la base de datos.

Aunque solo utilizamos estos valores para conectarnos a través de la aplicación backend, nos permiten establecer virtualmente una conexión desde cualquier medio compatible con el sistema de bases de datos Postgres. Por ejemplo, desde un sistema de gestión de bases de datos como PgAdmin [Rob11] o desde un IDE como IntelliJ.

Una consideración fundamental es la opción que nos brinda el servicio de hosting para conectar nuestra aplicación a la base de datos, ya sea a través de su red interna o mediante la red externa, conforme a la práctica convencional. En este caso, hemos decidido emplear la conexión a través de la red externa, una elección que, como es natural, conlleva sus propias ventajas y desventajas.

Inicialmente, al optar por esta forma de conexión, tenemos la capacidad de simular un entorno más próximo a lo que sería la aplicación desplegada en un entorno de producción. Esto implica la posibilidad de que la aplicación y la base de datos residan en servidores diferentes o incluso en ubicaciones geográficas distintas. Además, esta elección nos proporciona la flexibilidad de desvincularnos en cierta medida del hosting actual, permitiéndonos no depender exclusivamente de su infraestructura interna. Esta flexibilidad es especialmente valiosa en caso de que deseemos migrar tanto el backend como la base de datos a otro servicio similar en el futuro.

A pesar de reconocer que esta forma de conexión puede aumentar la latencia al realizar consultas a la base de datos, consideramos que las ventajas que aporta son más significativas para nuestro proyecto, especialmente en el contexto de despliegue a producción y escalado de la aplicación.

6.3.2. Configuraciones necesarias para Spring Boot

Con los detalles proporcionados anteriormente en la Figura 6.3, estamos listos para configurar los parámetros esenciales en nuestra aplicación Spring Boot para establecer la conexión con la base de datos PostgreSQL. En particular, necesitaremos especificar los valores para *'**spring.datasource.url**'*, *'**spring.datasource.username**'*, y *'**spring.datasource.password**'*.

Para la *'**spring.datasource.url**'*, que es esencial para la conexión JDBC [SS20], podemos desglosarla en sus componentes principales. Para nuestra aplicación, la URL se configura como:

- spring.datasource.url = `jdbc:postgresql://{hostname}.ohio-postgres.render.com/{database}`
- spring.datasource.username = **{Username}**
- spring.datasource.password = **{Password}**

Aquí, **jdbc:postgresql:**// indica que estamos utilizando el controlador JDBC [SS20] para PostgreSQL. La parte siguiente, ***.ohio-postgres.render.com***, representa la dirección del servidor de la base de datos. Finalmente, **{database}** se refiere al nombre de la base de datos específica a la que nos conectamos, en este caso: `root_m6e9`.

En conjunto, con la configuración adecuada de estos parámetros, nuestra aplicación Spring Boot estará preparada para interactuar de manera efectiva con la base de datos PostgreSQL. Todos estos detalles se encuentran resumidos en la Figura 6.3 para una referencia clara y concisa.

Capítulo 7

Desarrollo backend con Spring Boot

7.1. Arquitectura por microservicios

En la fase de desarrollo del componente backend de nuestro proyecto, optamos por utilizar la arquitectura de microservicios [LZJ+21] debido a sus numerosas ventajas y las buenas prácticas que conlleva para el despliegue de aplicaciones. Esta elección se alinea con la tendencia actual en el desarrollo de software y ofrece beneficios significativos en términos de escalabilidad, mantenimiento y flexibilidad.

La arquitectura de microservicios lleva a la máxima expresión el principio de **responsabilidad única** [Mar14]. Esta arquitectura implica descomponer una aplicación monolítica [GZ20] en servicios más pequeños e independientes, cada uno de los cuales representa una funcionalidad específica de la aplicación. Estos servicios, conocidos como microservicios, son unidades autónomas que pueden ser desarrolladas, desplegadas y escaladas de forma independiente.

En la implementación de la arquitectura de microservicios para nuestro proyecto, adoptamos un enfoque donde cada ”entidad” se representa como un microservicio independiente. Cada microservicio tiene su propio conjunto de código, lógica de negocio, capa de conexión y demás implementaciones específicas. Este enfoque de codificación por entidades proporciona una separación clara de responsabilidades y facilita la comprensión y el mantenimiento del sistema.

La elección de esta arquitectura se fundamentó en su notable capacidad de **tolerancia a fallos**. La disposición independiente de cada funcionalidad, donde cada microservicio se encarga de un componente específico dentro de la solución más amplia, juega un papel crucial. Esta estructura garantiza que, en caso de un fallo en algún componente de la solución, la aplicación continúe operando de manera adecuada, excepto por el componente afectado. Este enfoque no solo fortalece la resiliencia de la aplicación, sino que también minimiza el impacto de posibles errores, permitiendo que el resto de la solución siga funcionando de manera ininterrumpida.

La elección de esta arquitectura también se sustenta en su capacidad para ofrecer **escalabilidad selectiva** de sus componentes, una ventaja que consideramos esencial al tomar la decisión de implementarla, especialmente porque si se adopta la aplicación como solución al problema planteado otras personas también estarán involucradas en el ciclo de vida de la aplicación. En situaciones futuras, por ejemplo, podría surgir la necesidad de introducir nuevas funcionalidades en la entidad de Usuarios y en el microservicio asociado, incluso abarcando comportamientos no inicialmente

modelados. La flexibilidad inherente a esta arquitectura permite a los stakeholders realizar implementaciones adicionales de manera más eficiente. En este escenario, pueden realizar modificaciones específicamente en este componente sin afectar otras partes del sistema, reduciendo la posibilidad de errores y facilitando la escalabilidad a medida que evolucionan los requisitos.

En este mismo sentido, también encontramos la capacidad para **facilitar el mantenimiento** de la aplicación. Al descomponer la solución en microservicios independientes, cada uno gestionando una funcionalidad específica, se simplifica significativamente el proceso de mantenimiento. Actualizaciones, correcciones de errores o mejoras pueden implementarse de manera aislada en un microservicio sin afectar el funcionamiento de otros componentes. Esta modularidad incrementa la agilidad del equipo de desarrollo y reduce los riesgos asociados con cambios en el sistema, ya que las modificaciones se pueden realizar de manera focalizada y controlada.

En última instancia, otro aspecto que influyó en la elección de la arquitectura de microservicios es la flexibilidad que proporciona en términos de **tecnologías diversas**. Aunque, en nuestra solución, todos los microservicios fueron implementados utilizando las mismas tecnologías, es esencial destacar que cada microservicio tiene la capacidad de ser desarrollado con tecnologías específicas que se ajusten de manera óptima a sus requerimientos particulares. Para ilustrar este punto, consideremos la posibilidad de que, en el futuro, se requiera agregar una nueva entidad y, por consiguiente, un nuevo microservicio a la aplicación. Supongamos que los stakeholders encargados de desarrollar esta nueva funcionalidad, solo tienen conocimiento del lenguaje Python [Pyt21]. En este escenario, podrían desplegar un microservicio con FastAPI [Lat23] que se integre perfectamente con los servicios ya implementados en Spring Boot [SWL+23], sin afectar la funcionalidad de otros componentes. Esta capacidad de mezclar tecnologías de manera coherente destaca la versatilidad natural de la arquitectura de microservicios, y brinda a la aplicación la capacidad para adaptarse sin restricciones a las necesidades cambiantes del proyecto.

Sin embargo, al desplegar la aplicación, reconocemos que actualmente no estamos adhiriéndonos completamente al paradigma de microservicios. En lugar de desplegar cada microservicio de manera independiente, todos los microservicios se despliegan como una sola aplicación. Esta decisión fue tomada inicialmente debido a limitaciones en la plataforma de hosting y su estructura de precios, que no permitía despliegues independientes en la versión gratuita.

A pesar de esta limitación, hemos organizado la aplicación de manera que en el futuro sea posible desplegar cada microservicio de forma independiente. Esta estructura se ha diseñado cuidadosamente para facilitar la transición hacia un modelo de despliegue más alineado con la filosofía de microservicios.

7.2. Arquitectura hexagonal

Decidimos usar de manera conjunta con la arquitectura de microservicos, la arquitectura hexagonal [GG21] como arquitectura interna. Aunque la arquitectura de microservicios teóricamente brinda la libertad de que cada microservicio aplique su propia arquitectura interna, en nuestro caso, decidimos implementar uniformemente la arquitectura hexagonal en todos los microservicios. La elección de esta arquitectura específica se fundamenta en su capacidad para descomponer la solución en varias capas, proporciona un marco sólido y modular para el desarrollo.

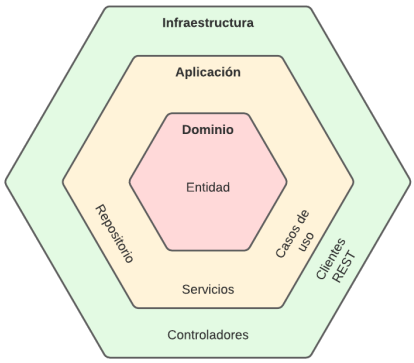


Figura 7.1: Modelo de la Arquitectura Hexagonal

Decidimos utilizar la arquitectura hexagonal [GG21] de manera conjunta porque esta arquitectura se caracteriza por su enfoque en la separación de preocupaciones y la facilidad de mantenimiento. En esta estructura, la solución se divide en tres capas principales: **Dominio**, **Aplicación**, y **Infraestructura** como podemos observar en la Figura 7.1.

Las dependencias de las capas se dan desde las más externas a las más internas, es decir, la capa de Dominio no depende de nada más que de ella misma, por ende no puede invocar componentes presentes en capas más externas. Por su parte, la capa de Infraestructura puede hacer invocaciones de componentes tanto en su capa como en la capa de Aplicación o en la de Dominio, como por ejemplo es el caso de los servicios, que hacen invocaciones de repositorios pues se encuentran en el mismo nivel de capa.

Para entender mejor el funcionamiento de las capas, entremos a definirlas. La **capa de dominio** juega un papel fundamental al albergar las entidades, que representan los conceptos fundamentales y los objetos clave del dominio de la aplicación. Estas entidades encapsulan tanto la información como el comportamiento asociado. Las entidades actúan como el núcleo conceptual y lógico de la aplicación, distinguiéndose por su independencia de los detalles de implementación, como la persistencia en una base de datos específica. En esta capa definimos las estructuras de datos que representan

los conceptos esenciales del dominio, estas estructuras van muy de la mano con lo que declaramos anteriormente en nuestro diagrama ERD. Para nuestra aplicación estas entidades son: **User**, **Role**, **Proyecto**, **Login**, **File**, **Evaluador**, y **Anteproyecto**.

Por su parte, la **capa de aplicación** desempeña un papel central al alojar tanto la lógica de aplicación como los adaptadores internos. En esta capa, se encuentran los servicios, que contienen la **lógica de negocio** específica de la aplicación, y los repositorios, que actúan como **adaptadores internos** encargados de gestionar la interacción entre la lógica de dominio y los detalles de persistencia en la base de datos.

Los servicios en la capa de aplicación de nuestra solución son los responsables de implementar la lógica de negocio esencial. Aquí se encuentran los componentes que orquestan la interacción entre las entidades del dominio y los adaptadores internos como los repositorios. Esta separación de responsabilidades permite mantener la capa de dominio libre de detalles de implementación y centrada en la esencia del negocio.

Por otro lado, los repositorios en la capa de aplicación actúan como adaptadores internos, proporcionando una interfaz para la persistencia y recuperación de datos en la base de datos. Estos repositorios conectan la lógica de dominio con los detalles específicos de la infraestructura de almacenamiento, facilitando la independencia y flexibilidad de la aplicación frente a cambios en la tecnología de bases de datos.

Para las entidades ya mencionada anteriormente en la capa de dominio y para componentes en la capa de aplicación como: **Registration** se encuentran disponibles tanto sus componentes de servicio como sus componentes de repositorio.

En el mismo sentido, la **capa de infraestructura** juega un papel vital al alojar los adaptadores externos. en esta capa encontramos nuestros controladores, que son los esponsables de exponer los métodos y servicios disponibles para los usuarios finales, como la aplicación frontend.

Estos controladores actúan como puertos externos que facilitan la interacción entre el sistema y el mundo exterior, proporcionando una interfaz para que los usuarios finales accedan a las funcionalidades ofrecidas por la aplicación. Estos adaptadores externos traducen estas solicitudes y las transmiten a la capa de aplicación, permitiendo que la lógica de negocio y los servicios internos respondan de manera adecuada. Para todos los componentes anteriormente mencionados también encontramos nuestro componente de controlador.

Teniendo claro el funcionamiento y cómo se implementó esta arquitectura en nuestra aplicación ya podemos dejar claro cómo es el flujo de información en el backend. Tomando como referencia la Figura 7.1 vemos que el flujo es desde las capas más externas hacia las capas más internas.

7.2.1. Inyección de dependencias

Una de las ventajas principales que nos ofrece implementar la arquitectura hexagonal como arquitectura interna de nuestros microservicios en adición a spring Boot es que tenemos acceso a la inyección de dependencias [Pra09]. En lugar de que un componente cree o gestione sus dependencias directamente, estas se le suministran desde el exterior. Este enfoque ofrece ventajas significativas en términos de modularidad, mantenibilidad y pruebas unitarias.

Spring Boot nos proporciona de manera nativa el anotador "@Autowired" que nos permite hacer este tipo de inyección que es muy útil a la hora de manejar nuestra aplicación. Este concepto lo tuvimos en cuenta desde un inicio, pues si se adopta nuestra aplicación como solución al problema inicial las personas que reciban este aplicativo van a poder hacer uso de múltiples tecnologías sin preocuparse por la implementación interna.

Un ejemplo práctico de la utilidad de la inyección de dependencias en la arquitectura hexagonal es el cambio de la base de datos subyacente. Ahora, la aplicación está configurada para utilizar PostgreSQL, y los repositorios interactúan con esta base de datos. Gracias a la inyección de dependencias, la transición hacia una base de datos distinta del mismo paradigma o incluso NoSQL se vuelve más sencilla.

En la capa de infraestructura, donde residen los adaptadores internos, como los repositorios que se comunican con la base de datos, la inyección de dependencias permite suministrar dinámicamente una implementación específica del repositorio durante el tiempo de ejecución. Al cambiar la configuración de la inyección de dependencias, es posible reemplazar la implementación de los repositorios que interactúan con PostgreSQL por a otras bases de datos SQL o incluso a una base de datos NoSQL.

Este cambio no afectaría la lógica de dominio o la capa de aplicación, ya que estas capas se mantienen independientes de los detalles de implementación de la infraestructura. La inyección de dependencias permite una fácil adaptación a cambios en los componentes externos, facilitando la evolución de la aplicación a medida que se modifican los requisitos o se exploran nuevas tecnoloías de bases de datos. En consecuencia, la arquitectura hexagonal, combinada con la inyección de dependencias, proporciona una flexibilidad valiosa para la aplicación de cara al futuro.

7.3. Configuración inicial en Spring Boot

En la fase inicial de nuestro proyecto, optamos por utilizar la conocida herramienta Spring Initializer [sta] para establecer las configuraciones iniciales y la estructura básica de nuestra aplicación. Spring Initializer simplifica enormemente el proceso de inicio al proporcionar una interfaz web intuitiva para personalizar las configuraciones del proyecto.

En esta herramienta, pudimos especificar el tipo de proyecto, el lenguaje de programación, la versión de Spring Boot que queríamos utilizar. Además, tuvimos la flexibilidad de agregar dependencias específicas para satisfacer los requisitos de nuestra aplicación.

En cuanto a la versión de Java, tomamos la decisión de utilizar Java 17 [SD21] como la base para nuestro proyecto. Esta elección se da pués es una versión razonablemente nueva de Java pero que nos ofrece una estabilidad mayor que la última versión disponible. Sin dejar de lado que la versión 17 nos ofrece funcionalidades relativamente nuevas.

Para administrar las dependencias de nuestro proyecto, elegimos Maven [Var19], una herramienta de gestión de proyectos. Maven simplifica la gestión de las bibliotecas y dependencias del proyecto, lo que facilitó la incorporación de nuevas funcionalidades y bibliotecas a medida que avanzó el desarrollo. La estructura de nuestro archivo **pom.xml** refleja la lista de dependencias y configuraciones esenciales. En la siguiente sección ahondaremos más en estas dependencias y su aporte al proyecto. De manera general, escogimos Maven con respecto a Gradle [BM11] por preferencias personales, y porque ya estabamos familiarizados con la manera de trabajar con Maven.

Todas las configuraciones esenciales para nuestro proyecto se encuentran detalladas en el archivo pom.xml. La figura 7.2 presenta la información clave de la configuración inicial, incluyendo la versión de Spring Boot, el grupo, el artefacto, el nombre, la descripción, la versión de Java y otros detalles relevantes.

Spring Initializr	
Configuración	**valor**
Project	Maven
Versión de Spring Boot	3.0.6
Group	tesis.backend
Artifact	backend
Name	backend
Package name	tesis.backend.backend
Description	Spring Boot tesis Backend
Packaging	jar
Java version	17

Figura 7.2: Configuraciones iniciales en Spring Initializr.

7.3.1. Dependencias

En esta sección, como habíamos hablado anteriormente mostraremos cuales son las dependencias que configuran nuestro proyecto. Cada una de estas dependencias desempeña un papel fundamental

en la construcción y funcionamiento de nuestra aplicación. A continuación, presentaremos un análisis detallado de cada dependencia especificada en el archivo pom.xml, destacando su funcionalidad y relevancia dentro del contexto de nuestro proyecto. Este enfoque nos permitirá comprender de manera integral cómo estas dependencias contribuyen a la construcción y el desempeño efectivo de nuestra aplicación.

Grupo: org.springframework.boot
Artefacto: spring-boot-starter-data-jpa
Funcionalidad: Esta dependencia pertenece al conjunto de starters de Spring Boot y está diseñada para simplificar el desarrollo de aplicaciones que utilizan JPA (Java Persistence API) para interactuar con bases de datos. Proporciona configuraciones predeterminadas, bibliotecas y dependencias necesarias para la capa de acceso a datos, facilitando las operaciones de persistencia y manipulación de entidades. Como está expuesto en la sección 7.2.1, si se quisiera utilizar la inyeccion de dependencias para la base de datos esta sería una de las dos configuraciones a cambiar. el cambio debe darse a partir de la nueva base de datos, las referencias están disponibles en la siguiente dirección.

Grupo: org.springframework.boot
Artefacto: spring-boot-starter-security
Funcionalidad: Esta dependencia también forma parte de los starters de Spring Boot y está destinada a proporcionar funcionalidades de seguridad para la aplicación. Al incluir esta dependencia, se activa la seguridad por defecto, y se pueden personalizar aspectos como la configuración de usuarios, roles, y políticas de seguridad. En el contexto particular de nuestro proyecto, esta dependencia nos permitió establecer un marco de seguridad para nuestro proyecto de manera rápida, realizar las autenticaciones de usuarios de manera sencilla y eficaz, cómo también integrar con otras dependencias para realizar filtros adicionales y reforzar el esquema de seguridad de la aplicación.

Grupo: org.springframework.security
Artefacto: spring-security-crypto
Funcionalidad: Esta dependencia pertenece al proyecto Spring Security y proporciona utilidades criptográficas comunes utilizadas en aplicaciones seguras. Particularmente, La inclusión de esta dependencia es crucial para la seguridad de las contraseñas de los usuarios en nuestra aplicación. Al utilizar spring-security-crypto, aprovechamos el *PasswordEncoder* de Spring Security, una herramienta fundamental para el manejo seguro de contraseñas. Esta funcionalidad se aplica en el proceso de inicio de sesión, registro de usuarios y almacenamiento seguro de credenciales en la base de datos, garantizando prácticas seguras y evitando el almacenamiento directo de contraseñas en texto plano.

Grupo: org.springframework.boot
Artefacto: spring-boot-starter-web
Funcionalidad: Esta dependencia es fundamental para el desarrollo de aplicaciones web utilizando Spring Boot. Nos facilitó la implementación de controladores, y el manejo de solicitudes y respuestas

HTTP. Además, nos permitió la construcción de servicios web RESTful mediante la anotación de controladores con @RestController. Esta fue vital para exponer los endpoints HTTP, y para gestionar las interacciones con el cliente.

Grupo: org.springframework.boot
Artefacto: spring-boot-devtools
Funcionalidad: Esta dependencia proporciona herramientas de desarrollo para mejorar la productividad durante la fase de desarrollo de la aplicación Spring Boot, por lo tanto no es indispensable para el entregable final ni el aplicativo que ya está desplegado. Sin embargo, contribuyó a una experiencia de desarrollo más ágil y eficiente, permitiéndonos realizar cambios rápidos en el código y viendo el resultado de estos cambios de manera casi instantanea. Es valioso para las personas que luego serán encargadas del mantenimiento de esta aplicación.

Grupo: org.postgresql
Artefacto: postgresql
Funcionalidad: Esta dependencia facilita la integración de PostgreSQL, proporciona las bibliotecas y configuraciones necesarias para la conexión y manipulación de datos en una base de datos PostgreSQL, como la que usamos en nosotros. De manera similar a la dependencia '*spring-boot-starter-data-jpa*', esta es la otra configuración a cambiar si se desea implementar lo expuesto en la sección 7.2.1. Además de cambiar el tipo de Spring Data Manager, también debemos cambiar esta dependencia que maneja la conexión con una base de datos en particular, con estos dos cambios podriamos realizar lo expuesto en la Sección 7.2.1 sin alterar ninguna otra capa en nuestra arquitectura.

Grupo: org.projectlombok
Artefacto: lombok
Funcionalidad: Esta dependencia adiciona una biblioteca que simplifica y agiliza el desarrollo en Java al eliminar la necesidad de escribir código boilerplate repetitivo, como getters, setters, constructores y otros métodos. En el contexto particular de nuestro proyecto, utilizamos anotaciones como @Data, @Getter, @Setter, entre otras, para generar automáticamente métodos y constructores, simplificando así la legibilidad, la escritura, y el mantenimiento del código fuente.

Grupo: io.jsonwebtoken
Artefactos: jjwt-api, jjwt-impl, jjwt-jackson
Funcionalidad: Estas dependencias trabajan en conjunto para gestionar la autenticación por medio de tokens JWT. También nos permitió extender los filtros de seguridad de la aplicación que están de manera predeterminada y añadir un filtro adicional basado en un token JWT. Esta funcionalidad adicional se tratará en un capítulo posterior.

Grupo: org.springframework.boot
Artefacto: spring-boot-starter-test
Funcionalidad: Esta dependencia facilita la escritura y ejecución de pruebas unitarias y de inte-

gración en proyectos Spring Boot. Incluye bibliotecas y configuraciones comunes para pruebas, como JUnit, Spring Test, y otras utilidades para simplificar la escritura y ejecución de pruebas.

Grupo: org.springframework.security
Artefacto: spring-security-test
Funcionalidad: La inclusión de esta dependencia es fundamental para realizar pruebas de seguridad de manera efectiva en nuestro proyecto. Al utilizar spring-security-test, obtenemos acceso a herramientas que facilitan la configuración de entornos de prueba seguros, la simulación de autenticación de usuarios y la realización de pruebas que implican aspectos de seguridad.

7.4. Desarrollo backend

7.4.1. Estructura del proyecto backend

La estructura de nuestro proyecto refleja la escogencia de las arquitectura de microservicios y de la arquitectura hexagonal, proporcionando una organización modular y escalable como discutimos en las secciones anteriores.
En la capa más externa, cada microservicio representa una entidad independiente con su propia lógica de negocio, capas de conexión y demás componentes esenciales. Esta separación facilita el desarrollo, mantenimiento y escalabilidad individual de cada entidad tal y como se expuso en la sección 7.1, esta estructura la podemos ver en la figura 7.3, y está estructurado de esta manera para que si en un futuro se pretende desplegar la solución adheridos completamente a la arquitectura de microservicios, la refactorización del codigo sea lo más sencillo posible.

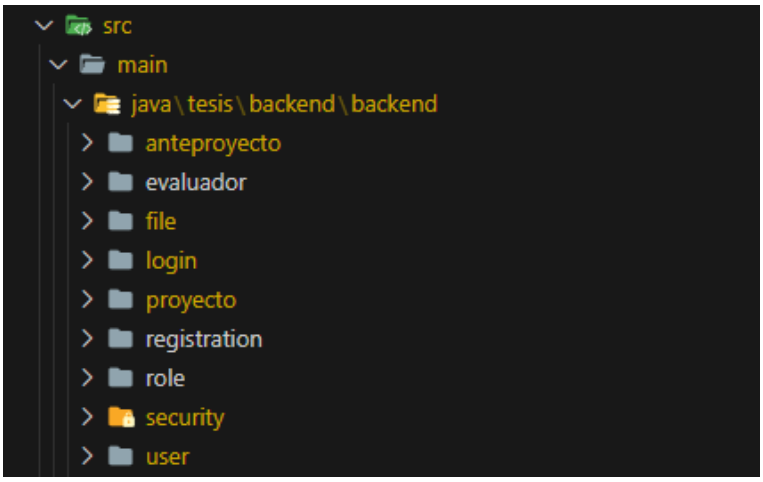


Figura 7.3: Estructura externa del proyecto.

Internamente, cada microservicio expuestos en la anterior figura sigue los principios de la arquitectura hexagonal. Cada entidad cuenta con sus propias definiciones de entidades, controladores, servicios y repositorios, encapsulando la lógica de negocio, la interacción con la base de datos, y

la exposición de los endpoints de manera independiente. Esta estructura hexagonal permite una clara separación de preocupaciones, facilitando el mantenimiento y la evolución de cada entidad de forma aislada. La estructura completa la podemos observar en la Figura 7.4, que aunque esta se muestra particularmente para la entidad User, aplica de la misma manera para el resto de entidades presentes en la figura 7.4.

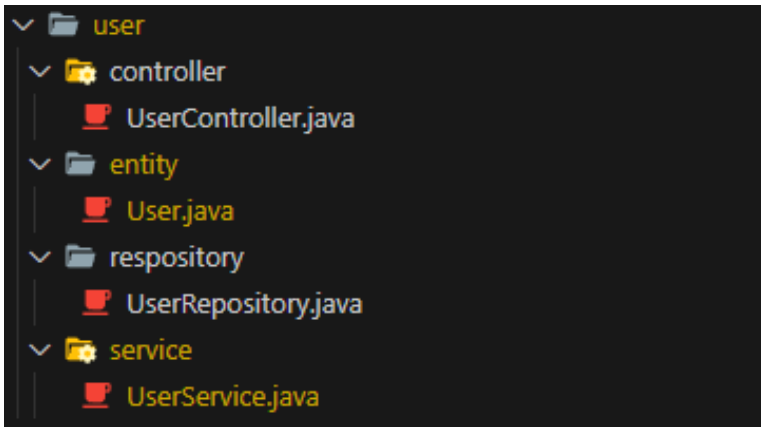


Figura 7.4: Estructura interna de las entidades del proyecto.

En las secciones que siguen, profundizaremos en cada uno de los microservicios, que de ahora en adelante denominaremos 'entidades', utilizando ambas terminologías de manera intercambiable. Analizaremos en detalle las características más destacadas de cada capa por entidad, desentrañando la esencia de su funcionamiento. Este análisis nos proporcionará una comprensión más profunda de cómo operan individualmente estas entidades y, a su vez, cómo se entrelazan entre sí para conformar la solución final del sistema.

7.4.2. Role

La entidad "Role" no posee nada en la capa de controlador ni en la capa de servicio debido a su naturaleza no transaccional. En otras palabras, los roles están predefinidos y no experimentarán cambios en su definición durante la ejecución del sistema. La decisión de separar el rol como una entidad independiente se basa en la anticipación de una posible escalada en esta entidad. Al adoptar esta estrategia, hemos facilitado la expansión y modificación de roles sin afectar a las entidades que hacen referencia a la entidad "Role". Inicialmente, conceptualizamos tres roles principales: **ESTUDIANTE**, **EVALUADOR** y un superrol administrativo **ADMIN**, que gestiona una amplia gama de funcionalidades en el sistema, similar al proceso actual.

Estas entidades cumplen un papel crucial al asignarse como propiedades a otras entidades, otorgándoles significado adicional. Es importante destacar que no modificamos directamente las entidades "Role" en si, sino que las asignamos a las entidades de usuario. Aunque la entidad "Role" carece de un controlador independiente y no tiene lógica en la capa de servicios, sí cuenta con una capa de

repositorio.

La gestión de roles se lleva a cabo directamente desde la entidad de usuario, eliminando la necesidad de un controlador independiente para los roles. En este contexto, el administrador es responsable de asignar roles a usuarios específicos, determinando así el papel que desempeñarán en el sistema. Este enfoque simplificado se alinea con la estructura y lógica de nuestra arquitectura, proporcionando una solución eficiente para la gestión de roles en el contexto de la entidad de usuario.

Entidad La definición de la entidad en nuestro sistema la podemos ver en la Figura 7.5, esta sigue las mejores prácticas para la persistencia de datos en una base de datos relacional. Utilizamos anotaciones como @Entity y @Table para establecer la relación entre la clase Java y la tabla correspondiente en la base de datos. Además, incorporamos @NoArgsConstructor para asegurarnos de tener un constructor sin argumentos, necesario para el funcionamiento adecuado de JPA (Java Persistence API).

Un aspecto importante es que implementamos la interfaz Serializable en la clase. Esto permite que los objetos de esta entidad puedan ser serializados y deserializados, lo cual es fundamental cuando se trata de traer esta entidad como propiedad en otras clases, como es el caso del comportamiento con la entidad Usuarios.

```
@Entity
@Table(name = "role", schema = "public")
@NoArgsConstructor
public class Role implements Serializable {
}
```

Figura 7.5: Definición de la entidad Role

En cuanto a los atributos de la entidad, contamos con un identificador marcado con los anotadores @Id, @GeneratedValue(strategy = GenerationType.IDENTITY) y @Column(). El anotador @Id indica que este atributo representa la clave primaria de la entidad, mientras que @GeneratedValue especifica que el valor del identificador se generará automáticamente y @Column() establece el nombre de la columna correspondiente en la base de datos. La generación del identificador en el lado del backend, utilizando GenerationType.IDENTITY, se realiza para aliviar la carga en la base de datos y garantizar la consistencia en la asignación de identificadores.

Además, tenemos otro atributo para la descripción, que simplemente utiliza el anotador @Column(name = "descripcion") para mapear este atributo con la columna correspondiente en la tabla de la base de datos. En el mismo sentido, proporcionamos solamente los Getters para estos atributos

pués como ya habíamos explicado esta es una entidad no transaccional. Toda la definición anterior la podemos ver en la figura 7.6

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "roleid")
    private Integer roleId;

    @Column(name = "descripcion")
    private String descripcion;

    public Role(Integer roleId, String descripcion) {
        this.roleId = roleId;
        this.descripcion = descripcion;
    }

    public int getRoleId() {}
    public String getDescripcion() {}
```

Figura 7.6: Atributos y métodos de la entidad Role

Repositorio Así como lo dijimos anteriormente, el repositorio no declara métodos específicos. Sin embargo, extendemos la interfaz JpaRepository, para aprovechar los métodos predefinidos que esta clase proporciona. La interfaz JpaRepository incluye métodos estándar para operaciones CRUD y consultas básicas en la base de datos, que luego utilizamos estos métodos en otros servicios cuando necesitamos realizar operaciones sobre el repositorio de roles. La definición completa de nuestro repositorio para esta entidad la podemos ver en la Figura 7.7. Cuando extendemos la interfaz JpaRepository, le pasamos el tipo que en este caso es la Entidad Rolez el tipo de dato del identificador para esta entidad, que como vemos en el atributo roleId, es Integer.

```
    @Repository
    public interface RoleRepository extends JpaRepository<Role, Integer> {
    }
```

Figura 7.7: Atributos y métodos de la entidad Role

7.4.3. User

En el contexto de nuestra aplicación, la entidad User es esencial, ya que actúa como un punto central para diversas asociaciones con otras entidades. Por ejemplo, se establecen relaciones entre usuarios y roles, lo que determina los privilegios y accesos que tiene cada usuario dentro del sistema. De igual manera, tambien asociamos esta entidad con otras entidades como Anteproyecto y ProyectoGrado para modelar el comportamiento de ser un usuario asociado con alguna de estas dos entidades en cualquier capacidad, tanto como estudiante o como evaluador. Además, la entidad User sirve como base para la implementación de la seguridad a través de Spring Security, proporcionando un marco para la autenticación y autorización. A continuación, exploraremos en detalle la implementación y estructura de la entidad User, desglosando cada capa para comprender su funcionamiento.

Entidad La definición de la entidad en nuestro sistema la podemos ver en la Figura 7.8. En esta definición utilizamos de manera similar a la definición de la entidad Role, los anotadores @Entity, @Table especificando la tabla correspondiente para esta entidad, y @NoArgsConstructor, estos anotadores cumplen el mismo proposito de establecer una entidad, una relacion con la base de datos, y proporcionar el constructor necesario para JPA. Adicionalmente, también utilizamos el anotador @Data que nos permite reducir la verbosidad de la definición, incluyendo de forma automática los métodos Constructores, Setters, y Getters.

De igual manera, para la firma de nuestra clase implementamos la interfaz UserDetails pues en el contexto de seguridad de nuestra aplicación es la que nos va a permitir gestionar la autorización y autenticación de los usuarios en nuestro sistema sobrescribiendo algunos metodos de esta interfaz.

```
@Entity
@Table(name = "user", schema = "public")
@Data
@NoArgsConstructor
public class User implements UserDetails {
}
```

Figura 7.8: Definición de la entidad User

En cuanto a los atributos de nuestra entidad, todos son muy similares entre ellos pero explicaremos los que presenten alguna particularidad. Para todos ellos utilizamos el anotador @Column() para asociar el atributo con la columna en la tabla especificada en el anotador @Table(). De manera similar a la entidad Role, tenemos un atributo userId que esta anotado con @Id lo que denota que va a ser nuestro campo identificador, también utiliza el anotador @GeneratedValue() con una estrategia de generación UUID para que la aplicación genere un universal unique identifier [LMS05]

que se le asignará al campo.

En el mismo sentido, el atributo username esta anotado con @JsonProperty('email') esto se debe a que en etapas iniciales del desarrollo no se consideraba el manejo de un usuario por su email sino con un nombre de usuario. En las iteraciones del desarrollo nos dimos cuenta como equipo que era mejor acotar a 'email'. Sin embargo, la lógica ya estaba implementada y este anotador nos permitió que como precisamente solo era un cambio de nomenclatura, hacia el exterior es decir los requests, se dieran con email y la lógica siguiera estando codificada con username.

Finalmente, el atributo roles es de los más importantes pués es el que le otorga a un usuario los roles, valga la redundancia, y por ende las interacciones que este pueda tener con el sistema. Este está anotado con @ManyToMany(fetch = FetchType.EAGER) lo que indica una relación mucho a muchos entre la entidad User y la entidad Role. Así mismo, nos indica que cuando se trae un User de la base de datos también se debe traer de manera ansiosa las entidades Rol que este posee. Luego, con el anotador @JoinTable indicamos la manera en que vamos a unir estas dos entidades, especificando el nombre de la tabla que contiene la relación entre las entidades, que en este caso y como definimos en el diseño 6.2 va a ser 'userroles', y cuales van a ser las columnas para hacer la unión, que son las para esta instancia las columnas asociadas a los identificadores.

Todo lo anterior lo podemos ver en las Figuras 7.9 y 7.10, correspondiente a todos los atributos de nuestra entidad que presentan alguna particularidad, el resto de atributos como lo son **personalId**, **password**, **name**, y **lastname** solo están anotados con @Column() como lo explicamos inicialmente.

```
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "userid", updatable = false)
    private UUID userId;

    @Column(name = "email")
    @JsonProperty("email")
    private String username;
```

Figura 7.9: Definición de los atributos de la entidad User 1

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
        name = "userroles",
        joinColumns = @JoinColumn(name = "userid"),
        inverseJoinColumns = @JoinColumn(name = "roleid")
)
private Set<Role> roles;
```

Figura 7.10: Definición de los atributos de la entidad User 2

En cuanto a los métodos de esta clase, como explicamos inicialmente al usar el anotador @Data, nos obviamos declarar constructores, Getters, y setters. Sin embargo, si tenemos que implementar los métodos de la clase UserDetails. La implementación de estos métodos, en particular de getAuthorities() es la que permite que a la hora de hacer nuestros requests podamos saber en el sistema si ese usuario tiene o no autorización para acceder a el recurso solicitado. La Figura 7.11, muestra la implementación de este método.

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    // GET the roles
    Set<Role> roles = getRoles();
    // Create a new collection to hold the GrantedAuthority objects
    Set<GrantedAuthority> authorities = new HashSet<>();

    for(Role role: roles) {
        authorities.add(new SimpleGrantedAuthority(role.getDescripcion()));
    }
    return authorities;
}
```

Figura 7.11: Implementación método getAuthorities()

Controlador La clase controlador de la entidad User está marcada con los anotadores @RestController y @RequestMapping(). El primero se encarga de encarga de decir que esta clase es un controlador de Spring, lo que nos premite manejar las solicitudes HTTP y devolver los resultados directamente en el cuerpo de la respuesta. El segundo anotador, nos permite establecer la ruta base para todas las solicitudes manejadas por este controlador, que en este caso será '/api/v1/user'. La figura 7.12 muestra la definición de esta clase.

Dentro de la clase UserController, se han implementado varios métodos, cada uno de los cuales representa un endpoint en nuestra aplicación getAllUsers(), deleteById(), y addRole(). Estos méto-

```java
@RestController
@RequestMapping("/api/v1/user")
public class UserController {
}
```

Figura 7.12: Definición del controlador para la entidad User

dos estan anotado con los anotadores @GetMapping() y @PreAuthorize(), el primer anotador sirve para establecer el verbo HTTP o método del endpoint a exponer y adicionalmente si se va a extender la dirección de la ruta. El segundo anotador por su parte sirve para configurar las autorizaciones de manera más estricta a nivel de métodos.

De manera similar los métodos deleteById(), y addRole() reciben en su petición Path parameters, esto lo logramos especificando en la firma de los métodos variables con el anotador @Pathvariable().

Cada uno de estos métodos llama a su homónimo en la clase UserService, por medio del atributo userService. Este atributo esta demarcado con el anotador @Autowired que nos permite realizar una inyección de dependencias de manera automatica y en tiempo de ejecución. Todo lo discutido anteriormente lo podemos ver en la Figura 7.13

```java
@Autowired
private UserService userService;

@GetMapping("/all")
@PreAuthorize("hasAuthority('ADMIN')")
public List<User> getAllUsers() {...

@DeleteMapping("/{id}")
@PreAuthorize("hasAuthority('ADMIN')")
public ResponseEntity<String> deleteById(@PathVariable("id") UUID id) {...

@PutMapping("/role/{id}/{role}")
@PreAuthorize("hasAuthority('ADMIN')")
public String addRole(@PathVariable("id") UUID id, @PathVariable("role") Integer role) {...
```

Figura 7.13: Definición de los endpoints para la entidad User

Servicio En la clase de servicio correspondiente a la entidad User, se implementan los mismos métodos que se encuentran en la clase de controlador. Sin embargo, la distinción clave radica en que estos métodos encapsulan la lógica de negocio asociada a las operaciones sobre los usuarios. La

clase de servicio está anotada con @Service, indicando que esta clase es un componente de servicio de Spring y facilitando la inyección de dependencias.

Para llevar a cabo las operaciones de persistencia en la base de datos, la clase de servicio utiliza la anotación @Autowired para inyectar los repositorios necesarios. En este caso, se incluyen los repositorios tanto para la entidad User como para la entidad Role. La inyección de dependencias garantiza que los servicios tengan acceso a las instancias necesarias para interactuar con la capa de persistencia y realizar operaciones CRUD.

La Figura 7.14, nos muestra las firmas y el esquema general de nuestra clase de servicios para esta entidad.

```
    @Service
    public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private RoleRepository roleRepository;

    public List<User> getAllUsers() {...
    public ResponseEntity<String> deletebyId(UUID id) {...
    public void addRole(UUID id, Integer roleId) {...
}
```

Figura 7.14: Definición de los servicios para la entidad User

Repositorio La clase repositorio correspondiente a la entidad User sigue una estructura similar a la clase de repositorio de la entidad Role. Sin embargo, en este caso, la clase repositorio de User incluye métodos específicos que se utilizan en la capa de servicio para implementar la lógica de negocio asociada a las operaciones sobre usuarios.

Estos métodos especéficos están anotados con @Query(), indicando que se utiliza una consulta JPQL (Java Persistence Query Language) [JJ18] personalizada para realizar operaciones más complejas que van más allá de las operaciones CRUD estándar proporcionadas por Spring Data JPA. Dentro de cada anotación @Query, se proporciona una sentencia JPQL que define la lógica específica requerida para cada operación. Esta estrategia permite una mayor flexibilidad al definir consultas personalizadas adaptadas a los requisitos de la aplicación. A nivel general, podemos ver esta definición en la Figura 7.15.

```
@Query("SELECT u FROM User u WHERE u.personalId = ?1")
Optional<User> findByPersonalId(int personalId);

@Query("SELECT u FROM User u WHERE u.username = ?1")
Optional<User> findByUsername(String username);

@Query("SELECT u.userId FROM User u WHERE u.username = ?1")
Optional<UUID> findIdByEmail(String email);
```

Figura 7.15: Métodos del repositorio para la entidad User

7.4.4. Evaluadores

Cuando examinamos la manera en que modelamos las relaciones con los usuarios en nuestro sistema, ya sea como autores o evaluadores de anteproyectos o proyectos, y observamos las similitudes evidentes entre las tablas de evaluadores y autores, surge naturalmente la pregunta de por qué Evaluadores es tratada como una entidad separada, mientras que Autores no sigue la misma estructura, a pesar de que ambos representan usuarios y el comportamiento que modelan se establece a través de relaciones presentes en ambas tablas.

La razón principal detrás de esta distinción radica en la concepción inicial del sistema, explicada previamente en la definición de la entidad Role 7.4.2. En nuestra base de datos, inicialmente contamos con tres roles predefinidos, entre los cuales se encuentra EVALUADOR. Este rol, según la información proporcionada por nuestros stakeholders, abarca lo que conceptualmente podrÃamos llamar 'evaluador1', 'evaluador2' y también 'director'. Al consolidar estos 'subroles' en uno solo, nos enfrentamos al desafío de diferenciar, en una relación especÃfica de la tabla 'evaluadores', si un usuario referenciado cumplía el papel de evaluador o director. Esta distinción es crucial para presentar la información de manera precisa en el frontend.

Inicialmente, esperábamos poder agregar un campo al atributo que incluyera la relación muchos a muchos en la entidad de anteproyecto, donde es fundamental conocer los evaluadores y autores de manera similar. Sin embargo, nos topamos con una restricción de Spring Boot: no podemos agregar campos a una relación muchos a muchos. La solución más adecuada que encontramos fue separar a Evaluadores en una entidad propia, pués agregar un atributo 'esDirector' a la entidad usuario rompe el proposito de separación de responsabilidades que quisimos implementar desde el principio. En lugar de utilizar un solo anotador @ManyToMany, empleamos varios anotadores @OneToMany. Esto nos permitió, en la entidad intermedia Evaluadores, agregar los campos necesarios que proporcionan la connotación de Director, y mantener la relación mucho a muchos. Debido a estas razones, Evaluadores se convirtió en su propia entidad.

Entidad En cuanto a la definición de la entidad Evaluadores, esta sigue la misma estructura que hemos aplicado a las demás entidades en nuestro sistema. La firma de la clase incluye los anotadores comunes @Entity, @Table(), @Data, y @NoArgsConstructor. Estos anotadores establecen la entidad en la base de datos, especifican el nombre de la tabla y el esquema correspondiente, generan automáticamente métodos estándar, y así también generan el constructor sina rgumentos necesario para JPA.

En lo que respecta a los atributos, mantenemos la práctica de incluir un identificador automático (id). Además, introducimos atributos de tipo UUID para representar las relaciones entre los anteproyectos y los usuarios. Estos atributos indican que un usuario es evaluador para un anteproyecto específico. De manera principal, también incluimos el atributo 'director' de tipo booleano que surgió como resultado de la necesidad expuesta anteriormente y dió paso a la refactorización de la solución inicial propuesta para este apartado. Con respecto a los métodos para esta clase, como ya lo hemos hablado en repetidas ocasiones, al usar el anotador @Data no tenemos la necesidad de declarar ningún método que no sea estándar de manera explícita. Todo esto lo podemos ver en la Figura 7.16.

```
@Entity
@Table(name = "evaluadores", schema = "public")
@Data
@NoArgsConstructor
public class Evaluador {
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(name = "anteproyectoid")
    private UUID anteproyectoId;

    @Column(name = "userid")
    private UUID userId;

    @Column(name = "director")
    private Boolean director;
}
```

Figura 7.16: Definición de la entidad Evaluador

Repositorio La capa de repositorio para la entidad Evaluadores es casi idéntica a la de la entidad Rol. En esta clase de repositorio, también extendemos la interfaz JpaRepository, pasándole el tipo de nuestra entidad (Evaluador) y el tipo de dato de su identificador, en este caso UUID. Aunque

en esta clase no declaramos explícitamente ningún método, heredamos los métodos proporcionados por la interfaz JpaRepository, lo que nos permite realizar operaciones comunes en la base de datos, como buscar, guardar y eliminar registros. Esta estructura mantiene la coherencia en el diseño de nuestras capas de repositorio. Esta definicón la podemos ver en la Figura 7.17.

```
@Repository
public interface EvaluadorRepository extends JpaRepository<Evaluador, UUID>{
}
```

Figura 7.17: Definición del repositorio de la entidad Evaluador

7.4.5. Anteproyecto

La entidad Anteproyecto es fundamental para nuestra aplicación, siendo más extensa y compleja debido a que desde esta entidad se derivan numerosos comportamientos que buscamos modelar. Esta entidad sirve como pilar central, y a partir de ella se desprenden relaciones clave con otras entidades, como Evaluadores y Autores, que han sido explicadas anteriormente. Aunque la tabla 'anteproyecto' en la base de datos puede no reflejar todos los atributos adicionales que hemos incluido en nuestra entidad, como Evaluadores y Autores, estos elementos son esenciales para representar de manera completa las relaciones y la complejidad de nuestro sistema.

A pesar de que Anteproyecto es la entidad principal, en la capa de dominio también tenemos una clase similar que nos permite especificar la estructura de entrada en la capa de infraestructura mediante nuestros controladores. Luego, a través de la capa de aplicación y los servicios, inicializamos la entidad principal. Esta separacion entre la representación interna y la interfaz de entrada nos permite manejar de manera más efectiva la interacción con la aplicación y garantizar la coherencia en la manipulación de los datos. Estos aspectos se explorarán con mayor profundidad cuando entremos a detallar las capas de esta entidad.

Entidad En la capa de entidad para la entidad Anteproyecto, hemos adoptado una estrategia consistente con las demás entidades para mantener la coherencia en nuestra aplicación. La clase principal que representa la entidad Anteproyecto está anotada con los siguientes anotadores: @Entity, @Table, @Data y @NoArgsConstructor. Estos anotadores cumplen las mismas funcionalidades antes descritas en otras entidades. La firma de nuestra clase la podemos ver en la Figura 7.18.

En cuanto a los atributos de la entidad Anteproyecto son anteproyectoId, noRadicacion, titulo, fechaEntregaAEvaluador, fechaEntregaDeEvaluador, fechaCreacion, fechaAprobacion y estado, cada uno de ellos anotado con @Column para especificar la columna correspondiente en la tabla 'anteproyecto'. Además, tenemos dos atributos adicionales, autores y evaluadores. El atributo autores, de tipo List<User>, representa la relación muchos a muchos con la tabla 'autores'. Para modelar

```
@Entity
@Table(name = "anteproyecto", schema = "public")
@Data
@NoArgsConstructor
public class Anteproyecto {
    ...
}
```

Figura 7.18: Firma de la entidad Anteproyecto

esta relación, utilizamos los anotadores @ManyToMany() y @JoinTable(). Esto nos permite especificar la relación entre Anteproyecto y User, también especificamos que este atributo va a cumplir la función de una tabla intermedia sin traer la relación en sí, sino los usuarios a los que hace referencia. La definición de este atributo la podemos ver en la Figura 7.19.

```
@OneToMany(fetch = FetchType.EAGER)
@JoinTable(name = "autores",
            joinColumns = @JoinColumn(name = "anteproyectoid"),
            inverseJoinColumns = @JoinColumn(name = "userid")
        )
private Set<User> autores;
```

Figura 7.19: Definición atributo autores.

El atributo evaluadores, por otro lado, es de tipo List<Evaluador>, una entidad definida previamente. Para modelar la relación de la tabla 'evaluadores', utilizamos el anotador @OneToMany(). Esta configuración refleja el comportamiento deseado que describimos a profundidad en la sección 7.4.4, asegurando que la relación con Evaluadores se maneje adecuadamente en términos de persistencia y eliminación de datos relacionados. Los parámetros incluidos en el anotador nos aseguran que a la hora de eliminar un evaluador de un anteproyecto, esta eliminación se lleve a acabo también en la entidad Evaluador, pero que no traspase más allá a la entidad usuario, pués al eliminar un evaluador de un anteproyecto en particular si no se especifica de esta manera puede haber una eliminación en cascada. La definición de este atributo también la podemos ver en la Figura 7.20.

A pesar de haber utilizado el anotador @Data, hemos especificado un método constructor en la clase Anteproyecto que recibe dos parámetros, nroRadicacion y titulo. Este constructor, además de ser utilizado para la creación de instancias, internamente establece valores predeterminados para ciertos atributos de la clase. La razón detrás de esta elección radica en la estructura de entrada que

```
@OneToMany(mappedBy = "anteproyectoId",  cascade = CascadeType.REMOVE, orphanRemoval = true)
private Set<Evaluador> evaluadores;
```

Figura 7.20: Definición atributo autores.

inicialmente recibimos a través de nuestros controladores. Al recibir únicamente nroRadicacion y titulo, utilizamos este constructor específico en la clase de servicio para aplicar la lógica necesaria y configurar correctamente la entidad principal Anteproyecto, garantizando así que todos los atributos necesarios estén debidamente inicializados y listos para ser persistidos en la base de datos. Este constructor lo podemos ver en la Figura 7.21.

```
public Anteproyecto(Integer nroRadicacion, String titulo) {
    this.noRadicacion = nroRadicacion;
    this.titulo = titulo;
    this.estado = 3;

    LocalDate today = LocalDate.now();
    this.fechaCreacion = Date.valueOf(today);
    this.autores = new HashSet<User>();
    this.evaluadores = new HashSet<Evaluador>();
}
```

Figura 7.21: Constructor de la entidad Anteproyecto.

La otra clase relevante en la capa de entidad es AnteproyectoInput, diseñada para estructurar la entrada al controlador cuando creamos una entidad Anteproyecto. Esta separación en dos entidades fue necesaria debido a la complejidad de crear un anteproyecto con autores y evaluadores asociados desde el principio. Dado que la entidad principal, Anteproyecto, tiene atributos autores y evaluadores de tipo List<User>y List<Evaluador>, respectivamente, se requería pasar toda la estructura de estas entidades al controlador, lo cual no resultaba eficiente.

La clase AnteproyectoInput, por otro lado, tiene los anotadores @Data, @NoArgsConstructor y @AllArgsConstructor. No tiene el anotador @Entity, ya que no representa una entidad de Spring, sino una estructura de datos de entrada. Internamente, esta clase incluye los atributos nroRadicacion y titulo, ambos necesarios para el constructor de la entidad principal Anteproyecto. Tambien tiene el atributo autores, de tipo List<String>, que modela una lista de usuarios donde le pasamos el email de cada usuario. Además, cuenta con el atributo evaluadores, de tipo List<EvaluadoresInfo>, donde EvaluadoresInfo es una clase interna compuesta por un string Email y un boolean isDirector, permitiéndonos inicializar las relaciones de evaluadores de manera más efectiva. Toda la estructura de esta clase la podemos ver en la Figura 7.22.

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AnteproyectoInput {
    private int nroRadicacion;
    private String titulo;
    private List<String> autores;
    private List<EvaluadorInfo> evaluadores;

    @Data
    @NoArgsConstructor
    @AllArgsConstructor
    public static class EvaluadorInfo {
        private String email;
        private boolean director;
    }
}
```

Figura 7.22: Clase que defina la estructura de entrada para Anteproyecto.

Controlador En cuanto a la capa de controlador, mantenemos la consistencia con la estructura previamente establecida. Utilizamos los anotadores @RestController y @RequestMapping(), que nos permiten definir esta clase como un controlador REST de Spring Boot y establecer la ruta base para los endpoints que se generarán. Estos anotadores son cruciales para exponer los servicios a través de la API REST, proporcionando una interfaz clara para interactuar con la lógica de negocio de nuestra aplicación que es la capa de servicio. En el caso específico de esta capa, configuramos los endpoints relacionados con las operaciones CRUD de la entidad, y endpoints para operaciones más especificas. También declaramos con el anotador @Autowired nuestra capa de servicio que le permite a todos nuestros endpoints acceder a la lógica de negocio detras de cada uno de estos llamados. Esta estructura general la podemos ver en la Figura 7.23.

```java
@RestController
@RequestMapping("/api/v1/anteproyecto")
public class AnteproyectoController {
    @Autowired
    private AnteproyectoService anteproyectoService;
    ...
}
```

Figura 7.23: Firma de la clase controlador para la entidad anteproyecto.

En relación con los métodos de la capa de controlador, se puede explorar a fondo su funcionalidad y los detalles de sus valores de entrada y salida consultando la referencia de la API. Esta información se encuentra disponible en los anexos adjuntos o puede accederse a través de la dirección Swagger proporcionada En la Sección 5.3. A nivel general, todos los métodos cuentan con el anotador de mapeo correspondinete, que especifica el verbo HTTP utilizado y, si es necesario, la extensión de la ruta base para la petición. Además, se emplea el anotador @PreAuthorize para establecer autorizaciones más estrictas a nivel de método en comparación con la autorización aplicada a nivel general. Ningún método realiza alguna lógica y solo llaman a sus métodos homónimos en la clase de servicios de la capa de aplicación correspondiente para esta entidad.

Sin embargo, si es relevante destacar el método de adición de un nuevo anteproyecto. La definición detallada de este método se encuentra en la Figura 7.24. Observamos que el método recibe un cuerpo JSON, indicado con el anotador @RequestBody(), de tipo EvaluadorInput, que es la clase de entrada discutida anteriormente. Mapear los datos de entrada con la clase AnteproyectoInput y luego en la capa de aplicación hacer las transformaciones necesarias resulta en un proceso más lógico y natural, pues en una petición de creacón de un anteproyecto solo mandamos unas listas de 'emails' en vez de mandar una lista de una estructura compleja como lo son User y Evaluadores.

```
@PostMapping()
@PreAuthorize("hasAuthority('ADMIN')")
public ResponseEntity<?> addAnteproyecto(@RequestBody() AnteproyectoInput anteproyectoInput) {
    return anteproyectoService.addAnteproyecto(anteproyectoInput);
}
```

Figura 7.24: Metodo POST para la entidad anteproyecto.

Servicio En la capa de servicio para la entidad Anteproyecto, declaramos la clase con el anotador @Service, siguiendo la misma estrategia utilizada para otras entidades. Como se muestra en la Figura 7.25, importamos el repositorio específico para la entidad Anteproyecto (anteproyectoRepository). Además, traemos los repositorios para userRepository y evaluadorRepository, lo que nos permite realizar varias comprobaciones y operaciones en la lógica del servicio. La inclusión de estos repositorios adicionales amplía las capacidades de la capa de servicio al proporcionar acceso a información relacionada con usuarios y evaluadores, necesaria para realizar validaciones y lógica de negocio en la manipulación de anteproyectos.

Los métodos para esta capa reflejan las operaciones expuestas en el controlador. En líneas generales, cada uno de estos métodos implementa la lógica de negocio asociada, lleva a cabo validaciones, gestión de errores, y otras tareas necesarias. Posteriormente, se conectan con el repositorio correspondiente para dar persistencia a estas operaciones en la base de datos. Aunque la lógica interna de cada método es extensa y específica, la Figura 7.26 proporciona una visión general de la firma de cada uno, lo que nos permite entender la funcionalidad sin adentrarnos en la implementación

```
@Service
public class AnteproyectoService {
    @Autowired
    private AnteproyectoRepository anteproyectoRepository;
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private EvaluadorRepository evaluadorRepository;
    ...
}
```

Figura 7.25: Firma de la capa de servicios para la entidad Anteproyecto.

específica. Este enfoque facilita la comprensión del comportamiento general de la capa de servicios y su relación con las operaciones de la capa de controlador.

```
public List<Anteproyecto> getAllAnteproyectos() {...}
public ResponseEntity<?> addAnteproyecto(AnteproyectoInput anteproyectoInput) {...}
public ResponseEntity<String> deleteAnteproyecto(UUID id) {...}
public ResponseEntity<String> addAutorToAnteproyecto(UUID idAutor, UUID idAnteproyecto) {...}
public ResponseEntity<String> deleteAutor(UUID idAutor, UUID idanteproyecto) {...}
public ResponseEntity<String> addEvaluadorToAnteproyecto(UUID idEvaluador,
                            UUID idAnteproyecto,     Boolean isDirector) {...}
...
```

Figura 7.26: Firma de los métodos de la capa de servicio para la entidad Anteproyecto.

Repositorio El repositorio asociado a la entidad Anteproyecto sigue la estructura habitual que hemos empleado para otras entidades. Utiliza el anotador @Repository y extiende la interfaz JpaRepository proporcionada por Spring Data JPA, pasando el tipo de entidad (Anteproyecto) y el tipo de su identificador (UUID). Aunque la clase de repositorio es relativamente sencilla, se destaca por la presencia de dos métodos anotados con @Query. Estos métodos permiten realizar consultas más específicas, adaptándose a las necesidades particulares de la aplicacipn y proporcionando flexibilidad en la recuperación de datos desde la base de datos. Estos elementos, junto con los métodos heredados de la interfaz JpaRepository, conforman la interfaz entre la capa de servicios y la capa de persistencia para la entidad Anteproyecto. Las sentencias JPQL las podemos ver en la figura 7.27.

7.4.6. ProyectoGrado

La entidad ProyectoGrado también es una entidad fundamental en nuestro sistema, sirviendo como la culminación del proceso de trabajo de grado que se inicia con la creación de un anteproyecto.

```
@Repository
public interface AnteproyectoRepository extends JpaRepository<Anteproyecto, UUID> {
    @Query("Select a FROM Anteproyecto a where a.noRadicacion = ?1")
    Optional<Anteproyecto> findByNoRadicacion(Integer noRadicacion);

    @Query("SELECT a from Anteproyecto a JOIN a.autores u WHERE u.userId = ?1")
    Optional<Anteproyecto> findByAutorId(UUID idAutor);
}
```

Figura 7.27: Clase Repository para la entidad anteproyecto.

Al concebir esta entidad, nos enfrentamos al desafío de decidir qué atributos incluir directamente y cuáles obtener a través de la relación con la entidad Anteproyecto. Consideramos que algunos campos, como la lista de autores y el director del proyecto, eran redundante almacenarlos directamente en ProyectoGrado, ya que estos datos pueden recuperarse mediante la asociación con el anteproyecto correspondiente.

Esta elección de diseño también refleja la naturaleza dinámica de un proyecto de grado. Al crear un ProyectoGrado, muchos de sus atributos no pueden ser inicializados de inmediato, ya que estos detalles solo se conocen y se establecen a medida que el proyecto avanza. Por ejemplo, la fecha de sustentación se determina a lo largo del tiempo, a medida que se programan y completan hitos relevantes. Esto implica que los métodos HTTP PUT, que permiten actualizar y modificar información, predominan en la API de ProyectoGrado.

En la referencia de la API, podemos observar que los endpoints para la entidad ProyectoGrado están centrados en la actualización de datos. Estos métodos PUT facilitan la modificación de atributos clave a medida que se avanzan en las diferentes etapas del proceso de trabajo de grado. Desde la fecha de sustentación hasta el estado actual del proyecto, estos endpoints ofrecen la flexibilidad necesaria para reflejar la evolución de un proyecto de grado. La estructura de esta entidad se adapta a la realidad de que los detalles específicos de un proyecto de grado se definen y ajustan a lo largo del tiempo, reflejando así su carácter dinámico en nuestro sistema.

Entidad La entidad ProyectoGrado se caracteriza por su simplicidad. En línea con la consistencia de diseño que hemos mantenido en nuestro sistema, inicializamos la entidad ProyectoGrado con los anotadores estándar, incluyendo @Entity, @Table, @Data, y @NoArgsConstructor. Estos anotadores aseguran que la entidad cumpla con los requisitos de persistencia de Spring Boot y proporciona los métodos estándar necesarios para operar con la entidad de manera eficiente.

Los atributos de la clase ProyectoGrado son fundamentales para representar la información relevante asociada a un proyecto de grado. El campo proyectoId, anotado con @Id y @GeneratedValue, fun-

ciona como el identificador único del proyecto, mientras que anteproyectoId establece la relación con el anteproyecto del cual se origina este proyecto de grado. Otros atributos como fechaSustentacion, fechaCreacion, notaDefinitiva, nroActa, mencionHonor, gradoPostulacion, y entregaDocs proporcionan información clave sobre el estado y los resultados del proyecto.

Para mantener una inicialización coherente y eficiente de la entidad ProyectoGrado, hemos declarado un constructor que acepta solo el anteproyectoId, ya que este es el único atributo necesario para iniciar la entidad. Con el anteproyectoId como hablamos anteriormente, implícitamente tenemos acceso a la información de autores, director y el propio proyectoId. Esta estructura de inicialización se alinea con la lógica del sistema, donde la creación de un ProyectoGrado está vinculada directamente a un anteproyecto existente. La firma general de nuestra clase junto a su constructor principal la podemos observar en la Figura 7.28.

```
@Entity
@Table(name = "anteproyecto", schema = "public")
@Data
@NoArgsConstructor
public class Proyecto {
    ...
    public Proyecto(UUID anteproyectoId) {
        this.anteproyectoId = anteproyectoId;
    }
}
```

Figura 7.28: Firma de la entidad ProyectoGrado

Controlador El controlador asociado a la entidad ProyectoGrado implementa métodos HTTP estándar como GET y POST, pero destaca especialmente por la prominencia de los métodos PUT, que son cruciales para actualizar y avanzar en el estado de los proyectos de grado. Este enfoque se alinea con la naturaleza dinámica de la entidad, donde muchos de sus atributos se actualizan a medida que progresa el proceso de evaluación y sustentación como explicamos anteriormente. En coherencia con los otros controladores del sistema, utilizamos anotadores como @RestController y @RequestMapping para establecer la naturaleza REST del controlador y especificar la ruta base de los endpoints.

Siguiendo la lógica consistente del sistema, implementamos anotadores como @GetMapping, @PostMapping y @PutMapping para definir los verbos HTTP asociados con cada método. Además, empleamos @PreAuthorize para reforzar las autorizaciones a nivel de método, asegurando que solo usuarios con los roles adecuados puedan acceder y modificar la información de los proyectos de grado.En cuanto a la estructura interna de los métodos del controlador, seguimos la cohesión y el acoplamiento adecuados al invocar directamente a los métodos homónimos en la capa de servicio.

Esta práctica mantiene la separación de responsabilidades que venimos profesando.

La estructura general del controlador para esta entidad, obviando las firmas de los metodos, la podemos ver en la Figura 7.29. También podemos observar la ruta base para nuestra entidad.

```
@RestController
@RequestMapping("/api/v1/proyecto")
public class ProyectoController {
    @Autowired
    private ProyectoService proyectoService;
    ...
}
```

Figura 7.29: Firma de la entidad ProyectoGrado

Servicio En la capa de servicio asociada a la entidad ProyectoGrado, implementamos la lógica de negocio esencial para el manejo de proyectos de grado en el sistema. Siguiendo la práctica común en todas las entidades, la capa de servicio se encarga de centralizar la lógica de negocio y manipulación de datos necesaria para los métodos expuestos en los controladores. Para garantizar la coherencia y consistencia en las operaciones, traemos tanto el repositorio propio de la entidad ProyectoGrado como el repositorio de Anteproyecto.

La Figura 7.30 proporciona un esbozo general de las firmas de los metodos y de la estructura de la clase de servicio.

```
@Service
public class ProyectoService {
    @Autowired
    private ProyectoRepository proyectoRepository;
    @Autowired
    private AnteproyectoRepository anteproyectoRepository;
    public ResponseEntity<String> addProyecto(Proyecto proyecto) {...}
    public List<Proyecto> getProyectos() {...}
    public ResponseEntity<String> setFechaSustentacion(UUID id, Date date) {...}
    ...
}
```

Figura 7.30: Firma de la capa de servicio para la entidad ProyectoGrado

Repositorio La capa de repositorio asociada a la entidad ProyectoGrado se mantiene sencilla y coherente con la práctica seguida para otras entidades en el sistema. Al igual que en casos anteriores,

no declaramos ningún método específico en esta clase de repositorio. En su lugar, aprovechamos las operaciones CRUD (Create, Read, Update, Delete) proporcionadas por JpaRepository de Spring Data JPA. Estas operaciones estándar nos permiten realizar las operaciones básicas de persistencia en la base de datos de manera eficiente, siguiendo las convenciones de JPA. Este enfoque simplificado contribuye a una gestión más clara y mantenible de la capa de persistencia para la entidad ProyectoGrado. La definición completa de la clase para esta capa la podemos ver en la Figura 7.31.

```
@Repository
public interface ProyectoRepository extends JpaRepository<Proyecto, UUID> {
}
```

Figura 7.31: Definición clase reposritoio de ProyectoGrado.

7.4.7. File

La entidad File en nuestro sistema desempeña un papel crucial al gestionar todas las entregas de documentos asociadas tanto a los anteproyectos como a los proyectos. Su importancia radica en que las entregas de documentos representan una parte fundamental del flujo de trabajo del sistema. Cada entrega realizada a lo largo del proceso, ya sea para anteproyectos o proyectos de grado, se registra y gestiona a través de esta entidad. Los archivos adjuntos en estas entregas, se almacenan y vinculan a las respectivas instancias de anteproyecto o proyecto. Esta entidad permite una organización eficiente de la información asociada a las entregas, facilitando el seguimiento y la revisión de los documentos subidos por los usuarios a lo largo del ciclo de desarrollo de los proyectos académicos.

Además, esta entidad se beneficia de funciones de utilidad que facilitan la compresión y descompresión de archivos. Esta práctica no solo optimiza el manejo de los documentos, sino que también mejora la eficiencia del almacenamiento en la base de datos. En la siguiente sección, exploraremos en detalle cómo estas funciones contribuyen a una gestión más eficiente de los archivos en el sistema.

Entidad La capa de entidad para la entidad File en nuestro sistema sigue la misma estructura que hemos empleado en otras entidades, utilizando anotadores como @Entity, @Table, @Data y @NoArgsConstructor. Sin embargo, en cuanto a los atributos, encontramos particularidades notables. El atributo idAsociado es de tipo UUID, similar al tipo de identificador utilizado en las entidades Anteproyecto y ProyectoGrado. Este campo juega un papel crucial al establecer la asociación entre la entrega de documentos y la entidad respectiva. Además, contamos con el atributo isAnteproyecto, el cual nos permite determinar si la entrega está asociada a un anteproyecto o a un proyecto de grado.

Un atributo significativo que merece mención especial es el marcado con la anotación @Lob, 'data'.

Este atributo está anotado con @Lob, que hace referencia a "large object". En este contexto, este atributo de tipo byte[] almacena los datos de la entrega en sí misma. La elección de utilizar un array de bytes para representar los datos permite manejar eficientemente la información contenida en los archivos adjuntos. Todos los atributos mencionados anteriormente los podemos observar en la Figura 7.32.

```
@Column(name = "idasociado")
private UUID idAsociado;

@Column(name = "isanteproyecto")
private Boolean isAnteproyecto;

@Lob
@Column(name = "data")
private byte[] data;
```

Figura 7.32: Atributos con particularidades para la entidad File.

Controlador La capa de controlador para la entidad File sigue la estructura coherente que hemos adoptado en otras entidades. En esta clase de controlador, llamada FileController, nos conectamos con los servicios pertinentes, en este caso, fileService y jwtService. El servicio fileService se encarga de gestionar toda la lógica de negocio asociada a las operaciones con archivos. Por otro lado, utilizamos el servicio jwtService para validar y gestionar la información contenida en el token JWT proporcionado en los encabezados de la solicitud.

Para esta entidad se declaran pocos métodos, con un enfoque en las operaciones clave relacionadas con la gestión de archivos. Entre estos métodos se encuentran uploadFile, que se encarga de cargar un archivo en la base de datos, downloadFile, que permite descargar un archivo almacenado, y getFiles, que recupera la lista de archivos disponibles. Con estos métodos mencionados anteriormente somos capaces de modelar el comportamiento buscado para nuestra aplicación y en particular para esta entidad.

Particularmente el método correspondiente al endpoint para realizar una entrega de documentos presewnta un comportamiento un poco distinto. Este método hace uso del servicio JWT. En este método, extraemos el ID del usuario que realiza la solicitud a través del token JWT presente en los encabezados. Posteriormente, invocamos el servicio JWT para verificar la validez de la asociación entre este usuario y la entidad (anteproyecto o proyecto) a la cual se está realizando la entrega. Esta validación es crucial para garantizar que solo los usuarios autorizados y asociados correctamente pueden realizar entregas. Si la validación es exitosa, llamamos al servicio fileService para llevar a cabo la operación de entrega. Aunque en este método de la capa de controlador se implementa una pequeña lógica, es importante destacar que esta no constituye lógica de negocio; se mantiene sepa-

rada de la implementación principal y se centra en aspectos de validación y seguridad. La definición de este método controlador la podemos apreciar en la Figura 7.33.

```
    @Autowired
    private FileService fileService;
    @Autowired
    private JwtService jwtService;

    @PostMapping("/upload")
    @PreAuthorize("hasAuthority('ESTUDIANTE')")
    public ResponseEntity<String> uploadFile(
            @RequestHeader(name = "Authorization") String Authorization,
            @RequestParam("file") MultipartFile file,
            ...
        ) throws IOException {
        String jwt = Authorization.substring(7);
        UUID id = UUID.fromString(jwtService.extractUserId(jwt));
        if (!fileService.isAssociated(id, idAsociado)) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(
            "El usuario que intenta enviar una entrega no estÃ¡ asociado al anteproyecto.");
        }
        return fileService.uploadFile(...);
    }
```

Figura 7.33: Método de hacer una entrega en la entidad File.

Servicio En la capa de servicios para la entidad File, seguimos una estructura similar a la empleada en otras entidades. La clase de servicio, denominada 'FileService', se organiza de manera coherente con las prácticas previamente establecidas. La figura 7.34 proporciona una visión general de las firmas de los métodos en esta clase y de la lógica que estos implementan internamente. No consideramos que es necesario detallar los anotadores utilizados, dado que ya han sido explicados en entidades anteriores. Aquí se implementa la lógica de las operaciones esenciales, como la carga, descarga y obtención de archivos, garantizando un manejo eficiente y seguro de las entregas de documentos en el sistema.

En esta capa de aplicación, introducimos una particularidad relevante: además de la clase de servicio 'FileService', incorporamos una clase utilitaria 'FileUtils' dedicada a facilitar el manejo adecuado de los archivos. Esta utilidad desempeña un papel crucial al proporcionar métodos para comprimir y descomprimir archivos, mejorando la eficiencia de su almacenamiento en la base de datos y minimizando el impacto en la aplicación backend. Los procesos de compresión y descompresión son esenciales en operaciones como la entrega de documentos, donde se busca optimizar la manipulación y el rendimiento del sistema. Estos métodos de compresión y descompresión son utilizados estratégicamente en la clase de servicio 'FileService', específicamente en los métodos asociados con

```
@Service
public class FileService {
    @Autowired
    private FileRepository fileRepository;
    @Autowired
    private AnteproyectoService anteproyectoService;

    public ResponseEntity<String> uploadFile(
    MultipartFile file, String description, UUID idAsociado,
    Boolean isAnteproyecto, Integer nroEntrega) throws IOException {...}
    public List<File> getAllFiles() {...}
    public File getFile(UUID id) {...}
    public byte[] downloadFile(File file) {...}
    public Boolean isAssociated(UUID idUser, UUID idAnteproyecto) {...}
}
```

Figura 7.34: Firmas en la clase de servicio para la entidad File.

la realización de entregas y la descarga de archivos, garantizando una gestión eficaz y segura de los documentos en el sistema.

Repositorio La capa de repositorio para la entidad 'File' mantiene su simplicidad como pra muchas otras entidades. Sin emabrgo, en ella encontramos una sentencia particular que puede ser visualizada en la figura 7.35. Esta sentencia es de especial importancia debido a la manera en que modelamos el atributo de entregas. Este atributo lo modelamos desde la perspectiva de la entidad 'File' en lugar de las entidad 'Anteproyecto' o 'Proyecto'. El propósito principal de esta sentencia es poder recuperar la última entrega que se haya hecho asociada a un anteproyecto o poryectoparticular, para así de ser necesario poder restringir la posibilidad de que un usuario realice una entrega una vez que ha excedido el número máximo permitido de entregas. De esta manera, aseguramos un control adecuado sobre las operaciones de entrega, garantizando el cumplimiento de las restricciones establecidas en nuestro sistema.

```
@Repository
public interface FileRepository extends JpaRepository<File, UUID> {
    @Query("SELECT MAX(f.nroEntrega) FROM File f WHERE f.idAsociado = ?1")
    Integer getCurrentNroEntrega(UUID idAsociado);
}
```

Figura 7.35: Clase repository para la entidad File.

7.4.8. Login

La entidad 'Login' presenta una particularidad significativa en nuestra arquitectura, ya que a diferencia de otras entidades, no cuenta con una capa de repositorio. Esta ausencia se debe a que la naturaleza de esta entidad no requiere persistencia en la base de datos. La entidad 'Login' se utiliza exclusivamente en tiempo de ejecución y no almacena información de manera permanente. En consecuencia, la clase que representa la entidad en la capa de dominio no se anota como una entidad de Spring, sino que se orienta más hacia la estructuración de los datos necesarios para el flujo inicial de la aplicación. Las demás capas de la arquitectura, como controlador, servicio, y repositorio de servicio, están presentes y contribuyen a la funcionalidad del sistema.

Adicionalmente, es importante destacar que la entidad 'Login' desempeña un papel crucial en la generación del token JWT. Este token es fundamental para la autenticación y autorización de los usuarios en todas las peticiones del sistema como ya lo hemos visto. La entidad 'Login' se encarga de solicitar a otra entidad que maneja todo lo relacionado a JWT un nuevo token y proporcionar este token al usuario final, que luego se utiliza para validar la identidad de los usuarios en cada interacción con la aplicación. La ausencia de persistencia en la base de datos para esta entidad se alinea con su función específica de facilitar el proceso de autenticación y seguridad durante la ejecución del sistema.

Entidad En la capa de dominio, hemos diseñado una implementación simple para la entidad 'Login' como podemos ver en la Figura 7.36. Esta clase básica estructura dos atributos fundamentales: 'email' y 'password'. Estos campos son esenciales para permitir un inicio de sesión convencional en el sistema. La entidad 'Login' en esta capa actúa como una estructura de datos para representar las credenciales de usuario necesarias para la autenticación. Con esta representación básica, la capa de dominio establece la estructura fundamental para el proceso de inicio de sesión en el sistema.

```
@Data
public class Login {
    @JsonProperty("email")
    private String username;
    private String password;

    public Login(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Figura 7.36: Declaración de entidad Login.

Controlador En cuanto al controlador asociado a la entidad 'Login' es notablemente sencillo,

ya que expone un solo endpoint mediante el método POST denominado 'loginUser', que representa la funcionalidad de inicio de sesión. La declaración de esta clase controlador se encuentra reflejada en la Figura 7.37. El método 'loginUser' recibe un cuerpo estructurado acorde a la entidad 'Login', y su única tarea es llamar al método correspondiente en la capa de servicio, donde se encuentra implementada la lógica necesaria para la autenticación. Este diseño minimalista cumple con el propósito de gestionar el flujo de inicio de sesión de manera eficiente y de acuerdo a los requisitos que hemos planteado inicialmente.

```
@RestController
@RequestMapping("/api/v1/login")
public class LoginController {
    @Autowired
    private LoginService loginService;

    @PostMapping()
    public ResponseEntity<?> loginUser(@RequestBody Login login) {
        return loginService.login(login);
    }
}
```

Figura 7.37: Controlador de la entidad Login.

Servicio La clase 'LoginService' está anotada con '@Service', indicando que se trata de una clase de servicio de Spring. Además, se observa la anotación '@RequiredArgsConstructor', que automáticamente genera un constructor que inicializa los campos marcados como 'final' en la clase.

Dentro de esta clase, se hace uso de la inyección de dependencias con la anotación '@Autowired' como ya hemos visto anteriormente, conectando con el repositorio de usuarios ('UserRepository'). También se emplea 'AuthenticationManager' y 'JwtService', instancias necesarias para llevar a cabo la autenticación y la generación de tokens JWT.

El método 'login' de esta clase juega un papel crucial en el proceso de autenticación. Primero, utiliza 'AuthenticationManager' para autenticar las credenciales proporcionadas mediante el token de nombre de usuario y contraseña. Luego, se recupera el usuario correspondiente desde el repositorio de usuarios ('UserRepository'). Si el usuario no tiene roles asignados, se devuelve una respuesta de error, para poder asegurarnos que un usaurio que no tenga un rol definido no pueda acceder a la aplicación y primero el adminstrador del sistema le asigne un rol. En caso contrario, se genera un token JWT utilizando 'JwtService' y se construye una respuesta exitosa ('RegistrationResponse') que contiene el token. El grueso de esta clase, que es el método login() lo podemos ver en la Figura 7.38, el resto de la clase es una clase de servicio estándar como la hemos manejado para el resto de entidades.

```
public ResponseEntity<?> login(Login login) {
    authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                    login.getUsername(),
                    login.getPassword()
            )
    );
    User user = userRepository.findByUsername(login.getUsername()).orElseThrow();
    if (user.getRoles().isEmpty()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(
        "The user has no roles, contact your ADMIN.");
    }
    String jwtToken = jwtService.generateToken(user);
    RegistrationResponse response = new RegistrationResponse(jwtToken);

    return ResponseEntity.status(HttpStatus.OK).body(response);
}
```

Figura 7.38: Método login.

7.4.9. Registration

La entidad correspondiente al web service para el registro de usuarios en la aplicación presenta una estructura similar a la entidad 'Login', ya que prescinde de la capa de repositorio y de entidad. En este contexto, la funcionalidad principal se centra en el registro de nuevos usuarios en la aplicación, gestionando todas las operaciones internas asociadas a este proceso. La capa de controlador expone los endpoints necesarios para registrar usuarios, mientras que la capa de servicio implementa la lógica de negocio correspondiente. La ausencia de la capa de repositorio se debe a que, igual que en la entidad 'Login', no se requiere persistencia directa de los datos de registro en la base de datos, ya que este servicio se centra principalmente en la autenticación y autorización de usuarios en tiempo de ejecución.

Controlador La clase RegistrationController corresponde a la capa de controlador de la aplicación y se encarga de gestionar los endpoints relacionados con el registro de usuarios. Este controlador está anotado con @RestController e especifica la ruta base de los endpoints mediante @RequestMapping(/api/v1/registration"). La anotación @RequiredArgsConstructor genera automáticamente un constructor que inyecta las dependencias necesarias, en este caso, el RegistrationService.

En el método registerUser, anotado con @PostMapping(), se espera recibir un objeto JSON que representa un usuario a través de la anotación @RequestBody. Este método delega la lógica de

registro al servicio correspondiente, registrationService.register(user), y retorna la respuesta proporcionada por el servicio.

En resumen, esta clase de controlador, que podemos ver en la Figura 7.39, actúa como punto de entrada para las solicitudes relacionadas con el registro de usuarios, transmitiendo la responsabilidad a la capa de servicio correspondiente para la ejecución de la lógica de negocio asociada.

```
@RestController
@RequestMapping("/api/v1/registration")
@RequiredArgsConstructor
public class RegistrationController {
    @Autowired
    RegistrationService registrationService;

    @PostMapping()
    public ResponseEntity<?> registerUser(@RequestBody User user) {
        return registrationService.register(user);
    }
}
```

Figura 7.39: Controlador para Registration.

Servicio La clase RegistrationService representa la capa de servicio encargada de la lógica de negocio asociada al registro de usuarios en la aplicación. Está anotada con @Service e incluye la anotación @RequiredArgsConstructor para generar un constructor que inyecta las dependencias necesarias, en este caso, el UserRepository y el PasswordEncoder.

El método principal de esta clase es register, el cual recibe un objeto User como parámetro. El primer paso es verificar si el documento de identidad proporcionado ya está asociado a un usuario existente mediante la llamada al método isAlreadyRegistered. En caso afirmativo, se devuelve una respuesta de error con el mensaje correspondiente.

A continuación, se verifica la disponibilidad del nombre de usuario mediante el método isUsernameAvailable. Si el email no está disponible, se retorna una respuesta de error. Cabe recordar que en cuanto a la lógica la nomenclatura es de 'username', pero para el usuario final es 'email'.

Luego, se utiliza el PasswordEncoder que lo provee la dependencia *spring-security-crypto* 7.3.1, para codificar la contraseña del usuario antes de almacenarla en la base de datos. Este proceso mejora la seguridad almacenando contraseñas de forma encriptada. Posteriormente, se guarda el usuario en el repositorio mediante la llamada a userRepository.save(user).

En resumen, esta clase de servicio encapsula la lógica necesaria para realizar el registro de usuarios,

abstrayendo los detalles de implementación y delegando las operaciones de persistencia al repositorio correspondiente. La Figura 7.40 referencia el método en cuestión y ejemplifica toda la explicación anterior.

```
    public ResponseEntity<?> register(User user) {
        if(!isAlreadyRegistered(user.getPersonalId())) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(
            "El documento de identidad ya estÃ¡ asociado a un usuario.");
        }

        if(!isUsernameAvailable(user.getUsername())) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(
            "El nombre de usuario no estÃ¡ disponible.");
        }

        // With encoder bean
        user.setPassword(passwordEncoder.encode(user.getPassword()));

        userRepository.save(user);

        return ResponseEntity.status(HttpStatus.OK).body("Usuario creado satisfactoriamente");
    }
```

Figura 7.40: Método register.

Capítulo 8

Características de seguridad del backend

8.1. Security Filter

La seguridad en Spring Boot se logra a través de la integración de la dependencia Spring Security 7.3.1, que proporciona un conjunto robusto de herramientas para gestionar la autenticación y autorización en aplicaciones Java. Al incluir esta dependencia en nuestro proyecto, obtenemos automáticamente una estructura de seguridad sólida que se encarga de manejar muchos aspectos de la protección de nuestra aplicacion. En el corazón de Spring Security se encuentra la clase **SecurityFilterChain**, que actúa como un filtro principal encargado de gestionar el flujo de seguridad. Esta clase permite la adición de varios filtros, cada uno con lógica específica, para personalizar el comportamiento de seguridad.

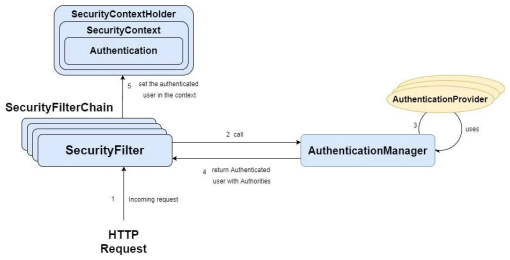


Figura 8.1: Esquema general de seguridad de la aplicación. [Dis23]

El comportamiento general de nuestra aplicación se esquematiza de manera efectiva en la Figura 8.1, que describe a grandes rasgos los seis pasos que ocurren durante el manejo de peticiones. En primer lugar, la recepción de una solicitud marca el inicio del proceso. Esta solicitud luego pasa a través de la SecurityFilterChain, que actúa como una línea de defensa principal. Internamente, esta cadena puede configurarse con varios filtros; sin embargo, para nuestra aplicación, se ha configurado un filtro personalizado destinado a la validación de tokens JWT, pues es la manera en que decidimos manejar las sesiones sin tener que guardar información adicional en el servidor.

En segundo lugar, la SecurityFilterChain llama al Authentication Manager. Posteriormente, este Authentication Manager utiliza un Authentication Provider para verificar las credenciales del usuario. En cuarto lugar, si la autenticación es exitosa, el Authentication Manager devuelve un objeto Authentication al SecurityFilterChain. Luego, se actualiza el contexto de seguridad (Security Context). Finalmente, el usuario puede, o no, acceder al recurso solicitado, dependiendo de la validez de su autenticación, autorización, y como veremos más adelante, su token.

8.2. Token JWT

El token JWT (JSON Web Token) [JBS15] es un estándar abierto (RFC 7519) que se utiliza para transmitir informaci'ón de forma segura entre partes como un objeto JSON. Este token consta de tres secciones: la cabecera (header), el cuerpo (payload), y la firma (signature).

La cabecera especifica el tipo de token y el algoritmo de cifrado utilizado, HS256 en concreto para nuestra aplicación, mientras que el cuerpo contiene los llamados "claims", que son como los atributos de nuestra entidad. Para nuestra aplicación nuestras claims son: **sub** que se refiere al subject que hace la petición, **id** que es el uuid del usuario haciendo la petición, **roles** que es la lista de roles o authorities que tiene el usuario haciendo la petición, **iat** que es cuándo se emitió este token, y finalmente **exp** que nos dice hasta cuando tiene validez el token (actualmente el token dura 2 horas desde su emisi'on).

La tercera parte del token JWT, la firma, desempeña un papel crucial en la seguridad del sistema. Esta firma es un hash generado a partir de la combinación del header, el payload y un secreto que reside exclusivamente en el lado del servidor. Aquí radica la magia de la seguridad: el secreto actúa como una llave privada que solo el servidor conoce, y garantiza la integridad y autenticidad del token. Si algún actor malintencionado intentara manipular el token para vulnerar nuestras peticiones, necesitaría conocer este secreto. Durante la verificación del token, si el hash generado no coincide con el resultado de aplicar el secreto sobre el header y el payload, la autenticación no se realizará con éxito, proporcionando una capa adicional de seguridad.

En cuanto a la generación de nuestra palabra secreta, utilizamos una herramienta online para generar llaves encriptadas, y esta la generamos con un nivel de seguridad de 256 bits.

Particularmente, en nuestra implementación, utilizamos el esquema de autenticación Bearer con tokens JWT. Esto significa que para autenticar las solicitudes, incluimos el token JWT en el encabezado Authorization de la solicitud precedido por la palabra "Bearer". Este método simple de agregar "Bearer" al principio del token en el encabezado garantiza que el servidor pueda identificar y validar el token correctamente.

La elección de utilizar tokens JWT en nuestra aplicación se fundamenta en la necesidad de adherirnos a las prácticas actuales y mantener nuestra aplicación alineada con las tendencias de la industria. En el panorama actual de desarrollo de aplicaciones web, el uso de tokens JWT se ha convertido en una práctica estándar para la implementación de sistemas de autenticación y autorización. Al optar por esta solución, estamos siguiendo unas buenas pácticas que mejoran nuestra aplicación tanto en su eficiencia como en su seguridad.

Además, la elección de tokens JWT está muy alineada con nuestra arquitectura de microservicios. Dado que los microservicios son componentes independientes y desacoplados que pueden imple-

mentarse y actualizarse de manera independiente, es esencial adoptar estándares que faciliten la interoperabilidad entre ellos. JWT, al ser un estándar abierto basado en JSON, proporciona una forma uniforme de transmitir información de autenticación y autorización entre los diferentes microservicios, independientemente de la tecnología utilizada para su implementación. Esta característica asegura que la adopción de nuevos microservicios, incluso aquellos implementados con tecnologías distintas, no afecte la consistencia y la eficiencia de nuestro sistema, contribuyendo así a la flexibilidad y escalabilidad de nuestra aplicación.

8.2.1. Implementación en el proyecto

Para implementar todo lo discutido anteriormente en nuestra aplicación, en primer lugar, hemos desarrollado una clase de servicio dedicada que se encarga de diversas tareas, como la emisión de un JWT, el análisis de su contenido, la verificación de su integridad, y la extracción de claims específicos, entre otras tareas. Esta clase centraliza todas las operaciones relacionadas con JWT, facilitando el mantenimiento y la coherencia de la lógica de seguridad.

En términos de implementación de seguridad, hemos creado una clase especializada denominada 'JwtAuthenticationFilter'. Esta clase extiende la clase 'OncePerRequestFilter', lo que implica que este filtro se ejecuta una vez por cada solicitud. Este filtro desempeña un papel fundamental al interceptar todas las peticiones entrantes y aplicar la lógica de autenticación basada en JWT. A continuación veremos como funciona esta clase.

Como se ilustra en la Figura 8.2, el proceso de autenticación en nuestra aplicación comienza extrayendo el token de la solicitud entrante. Este token se encuentra en el encabezado 'Authorization' y se extrae para su posterior procesamiento. Una verificación inicial se realiza para asegurarnos de que el token esté presente y cumpla con el formato esperado. En particular, se verifica si el encabezado está vacío y también que este contiene el prefijo 'Bearer'.

```
final String authHeader = request.getHeader("Authorization");
if (authHeader == null || !authHeader.startsWith("Bearer ")) {
    filterChain.doFilter(request, response);
    return;
}
```

Figura 8.2: Fase inicial en el filtro de seguridad JWT.

Una vez confirmada la presencia y el formato adecuado del token en el encabezado de autorización, procedemos a realizar el parsing del string para obtener únicamente el token como tal. Utilizamos la clase de servicio previamente mencionada, 'JwtService', para extraer la información esencial del token. En particular, nos centramos en la obtención del usuario al que hace referencia

el token, y este paso se representa de manera clara en la Figura 8.3 del proceso de autenticación.

```
jwt = authHeader.substring(7);
username = jwtService.extractUsername(jwt);
```

Figura 8.3: Parsing del token.

Después, verificamos que se haya obtenido exitosamente un usuario en la etapa de parsing del token y que actualmente no haya ningún usuario autenticado en el contexto de nuestra aplicación. En caso de que el usuario haya sido correctamente extraído y no exista ya una autenticación en curso, procedemos con la siguiente fase del proceso de autenticación.

Posteriormente, como vemos en la Figura 8.4, cargamos los detalles del usuario con userDetailsService a partir del nombre de usuario. Luego, verificamos la validez del token JWT y su correspondencia con el usuario mediante jwtService. Si la validación es exitosa, creamos un objeto UsernamePasswordAuthenticationToken con la información del usuario y sus roles. Lo crucial es que, al extender este token, agregamos detalles de la solicitud y, finalmente, actualizamos el contexto de seguridad con SecurityContextHolder. Esta actualización del contexto es fundamental, ya que establece la información de autenticación en toda la aplicación, y como este filtro lo configuramos para que tuviera una ejecución prioritaria, es decir, que fuera el primer filtro en toda la cadena, los filtros posteriores ya van a tener el SecurityContextHolder actualizado con la información del usuario haciendo las peticiones.

```
UserDetails user = this.userDetailsService.loadUserByUsername(username);
if (jwtService.isTokenvalid(jwt, user)) {
    UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken();
    authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
    SecurityContextHolder.getContext().setAuthentication(authToken);
}
```

Figura 8.4: Validación final en el jwt.

En términos generales, la validación del token JWT y la implementación de la seguridad en nuestra aplicación se llevan a cabo mediante el código que hemos explorado. Sin embargo, para una comprensión más detallada y un análisis profundo de la configuración de seguridad, se recomienda revisitar los archivos ubicados en la carpeta security/config. En esta sección específica del código fuente, se encontrarán detalles adicionales sobre la configuración específica de la seguridad. Explorar estos archivos proporcionará una visión más completa de cómo se ha estructurado y configurado la seguridad en nuestra aplicación.

Capítulo 9

Despliegue

9.1. Dokerization

Como mencionamos previamente, optamos por realizar el despliegue tanto de la aplicación backend como de la base de datos en el mismo servicio de hosting. En el caso específico de la aplicación backend, nos encontramos con una limitación, ya que el servicio de hosting que seleccionamos no admitía nativamente aplicaciones Java. Sin embargo, nos ofreció la flexibilidad de realizar un build and deploy mediante un Dockerfile.

Docker [RBA17] es una plataforma de software que simplifica el despliegue de aplicaciones al encapsularlas junto con sus dependencias en entornos aislados llamados contenedores. Utilizando un Dockerfile, pudimos describir y empaquetar nuestra aplicación Java, junto con sus dependencias, en un contenedor independiente, lo que facilitó su despliegue en el servicio de hosting seleccionado. Sin embargo, tuvimos que adicionar a la raíz de nuestro proyecto un archivo llamado 'Dockerfile' el cual es una convención para esta tecnología.

En el primer bloque, se utiliza la imagen base *maven:3.8.5-openjdk-17* para construir la aplicación Java, esto alineado con la versión seleccionada. Se copian todos los archivos del directorio de trabajo actual al contenedor, y luego se ejecuta el comando *mvn clean package -DskipTests* para compilar y empaquetar la aplicación utilizando Maven, excluyendo la ejecución de pruebas.

En el segundo bloque, se utiliza la imagen base *openjdk:17.0.1-jdk-slim* para crear un contenedor más liviano. Desde el primer bloque, se copia el archivo JAR resultante que es el ejecutable de nuestra aplicación, renombrándolo como backend.jar. Se expone el puerto 8080 para permitir la comunicación con la aplicación, y finalmente damos los comandos a ejecutar dentro de nuestro contenedor, comandos similares a si estuviermos ejecutando nuestra aplicación desde la consola, *java -jar backend.jar*. Este Dockerfile garantiza que la aplicación backend esté contenida en un entorno reproducible y aislado, lo que facilita su despliegue en cualquier entorno compatible con Docker sin depender de la configuración específica del host, la definición completa la podemos ver en la Figura 9.1.

```
FROM maven:3.8.5-openjdk-17 AS build
COPY . .
RUN mvn clean package -DskipTests

FROM openjdk:17.0.1-jdk-slim
COPY --from=build /target/backend-0.0.1-SNAPSHOT.jar backend.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "backend.jar"]
```

Figura 9.1: DockerFile para la aplicación backend.

9.2. Despliegue en Render

9.2.1. Aplicación de spring Boot

Una vez finalizado nuestro Dockerfile, el siguiente paso para desplegar la aplicación backend fue conectar nuestro repositorio de código, que en este caso utilizamos GitHub, con el servicio de hosting seleccionado, posteriormente también indicamos qué rama queríamos que actuara como la fuente para el despliegue automático, en nuestro caso, seleccionamos la rama 'main', y ya teniamos todo configurado. Al seguir estos sencillos pasos, la plataforma de hosting detectó automáticamente nuestro Dockerfile y ejecutó las configuraciones que le proporcionamos en este archivo.

Esta forma de despliegue no solo simplificó el proceso de despliegue, sino que también nos brindó una ventaja adicional en términos de Integración Continua/Despliegue Continuo (CI/CD) [Pul13]. Al realizar un push a nuestra rama target ('main'), la plataforma de hosting realiza un despliegue automático. Aunque esta implementación CI/CD es bastante básica y no incluye ejecución de pruebas u otros stages, representa un beneficio adicional. Este enfoque automatizado agiliza el ciclo de desarrollo y garantiza que la última versión del código esté siempre disponible en el entorno de producción, proporcionando una eficiencia valiosa para nuestro flujo de trabajo.

Como mencionamos en secciones anteriores, nuestro servicio de hosting presenta algunas limitaciones que afectan la implementación completa de la arquitectura de microservicios; no pudimos desplegar cada microservicio de manera individual, y por ende no nos pudimos adherir de manera completa a esta arquitectura. Además, debido al plan gratuito, la aplicación desplegada entra en un estado de inactividad después de 15 minutos sin recibir peticiones, lo que resulta en un apagado automático. Aunque la aplicación se reactiva cuando recibe una solicitud en este estado 'apagado', el proceso no es instantáneo y puede afectar el rendimiento inicial. Sin embargo, estas desventajas de la plataforma de hosting no representan una limitación para nuestra solución ya que el haberla desplegado es un plus con respecto a lo planteado inicialmente y para mostrar nuestro pmv es una buena ventaja independientemente de estos factores.

9.2.2. Base de datos

Para la base de datos, procedimos a instanciar una entidad en nuestra plataforma de hosting. De manera inmediata, obtuvimos acceso a las credenciales que detallamos en la sección 6.3, permitiéndonos establecer una conexión con la base de datos, como habíamos especificado anteriormente. Siguiendo la decisión de instanciar manualmente todas las entidades, nos conectamos a esta nueva base de datos mediante las credenciales mencionadas utilizando una interfaz de manejo de bases de datos, en este caso, pgAdmin4. A través de esta interfaz, llevamos a cabo la creación de las tablas siguiendo las especificaciones detalladas en la misma sección 6.3. Este enfoque como ya fué mencionado en su momento nos brindó un control total sobre la configuración y estructura de la base de datos, permitiéndonos adaptarla a las necesidades específicas de nuestra aplicación, y conforme iterabamos en el desarrollo, asi como también asegurando una gestión precisa de los datos desde el inicio.

Capítulo 10

FrontEnd

En el ámbito del desarrollo de aplicaciones web, el Frontend juega un papel fundamental al proporcionar la interfaz de usuario con la cual los distintos usuarios interactúan desde un navegador web. Este capítulo tiene como objetivo exponer al lector cuál fue el proceso que se llevó a cabo para la construcción del Frontend de este trabajo de grado, con el fin de servir como documentación y también para que se hagan evidentes las distintas decisiones que se tomaron para diseñar y desarrollar el mismo.

El término "FrontEnd" hace referencia a la parte visible y accesible de una aplicación, es aquella con la cual los usuarios interactúan directamente. Es la cara pública de la aplicación, donde los elementos visuales, la disposición de la información y las interacciones son cuidadosamente diseñados para ofrecer una experiencia de usuario fluida y eficiente [fro].

En el contexto de este trabajo de grado, el frontend de la aplicación no solo actúa como una interfaz amigable, sino que también despliega un rol crucial en la gestión y presentación de los trabajos de grado en la aplicación. También se explorarán la arquitectura subyacente, las decisiones de diseño, y el proceso de desarrollo que se llevó a cabo en el mismo, permitiendo a directores, evaluadores y profesores navegar y gestionar de manera efectiva los trabajos de grados.

A lo largo de las siguientes secciones, se mostrará la estructura de las carpetas y archivos, la organización de componentes, y las tecnologías clave empleadas en la construcción del frontend. Además, se explorarán distintos conceptos de tecnología web que fueron esenciales para la contribución de una arquitectura integral en la interfaz de usuario.

Al comprender estos elementos, no solo se obtendrá una visión clara de cómo se ha diseñado y desarrollado el frontend de este trabajo de grado, sino que también se apreciará la importancia de cada decisión tomada en pro de la experiencia del usuario y de cumplir con los objetivos específicos de este trabajo de grado.

10.1. Diseño

El proceso de diseño del FrontEnd de esta aplicación se ha guiado por la premisa de ofrecer una experiencia visual coherente e intuitiva, al tiempo que refleja la identidad institucional de la

universidad. Para lograr lo anterior, primero se adoptoó una paleta de colores basada en los colores institucionales que se muestran en la Figura 10.1, que no solo resaltan la estética de la aplicación, sino que también establecen una conexión con la identidad visual de la universidad.

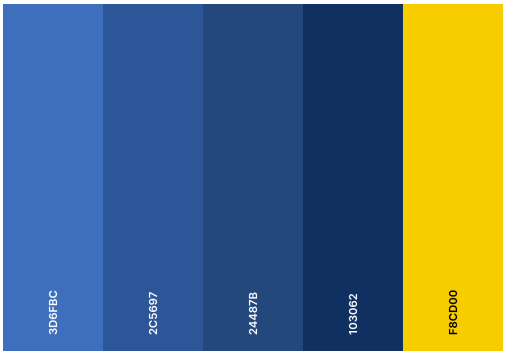


Figura 10.1: Colores Institucionales en Hexadecimal

Por otro lado, para el diseño de la interfaz de usuario de la aplicación, se tomó como referencia las interfaces de los servicios web existentes de la universidad tales como: OneGate para el Login y la página principal, la página de la universidad para los colores institucionales, y el servicio de práctica estudiantil para la barra de navegación y los demás componentes de la interfaz. Al alinear el diseño con los de estos servicios, se busca crear una experiencia uniforme para los usuarios que ya están familiarizados con los servicios en línea de la institución. Esto no solo facilita la adopción de esta aplicación, sino que también proporciona una sensación de continuidad a los usuarios.

Para la construcción de los diferentes elementos de la interfaz de usuario, se optó por utilizar una librería de componentes, el cual se detalla en la siguiente sección. Esta librería simplifica el proceso de desarrollo al proporcionar componentes prediseñados que son fáciles de integrar y personalizar según los requisitos específicos de este trabajo de grado. Esta elección no solo agiliza el desarrollo, sino que también garantiza una coherencia visual en todos los elementos de la interfaz.

A lo largo de este capítulo, se mostrará cómo la elección de la paleta de colores, la inspiración en interfaces de usuario existentes y la implementación de componentes se combinan para dar forma al diseño visual y funcional del FrontEnd de este trabajo de grado. Cada decisión de diseño ha sido deliberada, con el objetivo de mejorar la usabilidad, la estética y la accesibilidad general de la interfaz de usuario.

10.2. Tecnologías Utilizadas

El proceso de desarrollo del FrontEnd de este trabajo de grado ha estado impulsado por la adopción de tecnologías modernas y eficientes. Cada elección juega un rol crucial para garantizar un desarrollo robusto, escalable y orientado a ofrecer una experiencia de usuario optima. A continuación se muestra en detalle cada dependencia utilizada:

Dependencia: TypeScript
Versión: 5.2.2
Descripción: Para el desarrollo web de este trabajo de grado se optó por TypeScript como lenguaje de programación principal en el desarrollo del frontend. TypeScript, al ser una extensión tipada de JavaScript, proporciona beneficios significativos en términos de detección de errores temprana, facilitando el mantenimiento del código y mejorando la claridad en el desarrollo.

Dependencia: React
Versión: 18.2.0
Descripción: La elección de React es fundamental en el stack de tecnologías para el desarrollo del frontend ya que react simplifica el desarrollo mediante componentes reutilizables, facilitando la gestión del estado y mejorando la eficiencia del código. Su enfoque en el renderizado virtual optimiza el rendimiento, brindando una experiencia de usuario ágil y receptiva.

Dependencia: Next JS
Versión: 13.4.19
Descripción: Next.js es un framework de React que agrega potentes capacidades de enrutamiento y renderizado del lado del servidor. Esta combinación permite la creación eficiente de páginas web, facilitando tanto la generación de páginas estáticas como la implementación de rutas dinámicas. La integración de Next.js en este trabajo de grado no sólo simplifica el proceso de desarrollo, sino que también mejora significativamente el rendimiento, asegurando una experiencia de usuario rápida y eficiente.

Dependencia: Tailwind CSS
Versión: 3.3.3
Descripción: Tailwind CSS es un framework de estilos utilitarios que simplifica la creación y personalización de estilos a través de clases predefinidas. Este enfoque basado en clases agiliza el proceso de desarrollo y brinda una apariencia visual coherente en la aplicación. La integración de Tailwind CSS no solo facilita la creación de una interfaz estéticamente atractiva, sino que también mejora la mantenibilidad del código y la eficiencia del desarrollo.

Dependencia: React Table
Versión: 8.10.3

Descripción: En cuanto a la presentación de los datos tabulares de los anteproyectos y los trabajos de grado, se utilizó React-Table. Esta biblioteca proporciona una solución flexible y extensible para la creación de tablas interactivas, permitiendo agregar filtros, paginación, ordenamiento y otras funcionalidades a las tablas de manera sencilla, lo que mejora la experiencia del usuario al permitir una navegación y visualización eficiente de la información.

Dependencia: React Hook Form
Versión: 7.47.0
Descripción: React-hook-form es una biblioteca que simplifica y optimiza la gestión del estado de los formularios en aplicaciones React. Al incorporar esta herramienta, se hace más fácil la validación, manipulación y envío de datos ingresados por el usuario de manera eficaz. Con esta biblioteca se proporciona una implementación eficiente y confiable de los formularios de la aplicación.

Dependencia: Headless UI
Versión: 1.7.17
Descripción: Headless UI es una biblioteca que permite agregar animaciones y efectos a los componentes de React, lo que permite brindar una experiencia de usuario más fluida y cómoda.

Dependencia: Axios
Versión: 1.6.0
Descripción: Axios es una biblioteca que simplifica las operaciones de red al proporcionar una interfaz amigable para realizar solicitudes HTTP. Esta integración permite interactuar de manera eficiente con la API del backend, facilitando la obtención, envío y procesamiento de los datos de manera asíncrona, con el fin de ofrecer un tiempo de respuesta rápido y sin interrupciones.

10.3. Estructura de carpetas

```
/tesis-web-app-front
├── /api
│   ├── index.ts
│   └── tesisAPI.ts ..... Conexión con la API del BackEnd
├── /app ..... Enrutador de la Aplicación
│   ├── /(app)
│   │   ├── /anteproyecto
│   │   ├── /trabajo-de-grado
│   │   ├── layout.tsx
│   │   └── page.tsx
│   ├── /(auth)
│   │   ├── /login
│   │   ├── /register
│   │   └── layout.tsx
│   ├── globals.css
│   └── layout.tsx
├── /components
├── /context
├── /data-temp
├── /hooks
├── /interfaces
├── /utils
├── .env
├── tailwind.config.ts
└── ...
```

La estructura de carpetas mostrada previamente detalla la disposición empleada de los diferentes archivos que componen el FrontEnd de este trabajo de grado. Esta sección tiene como objetivo explicar la función de cada directorio con el fin de conocer el impacto que tiene cada una sobre la aplicación web final.

10.3.1. Convención de Archivos

Next JS utiliza un enrutador basado en un sistema de archivos donde cada carpeta que se encuentre en el directorio `/app` se utiliza para definir una ruta [App23]. Para que la ruta sea pública cada carpeta debe contener el archivo `page.js`. En este archivo se define el HTML y las funcionalidades que se desea para esa ruta. Si una carpeta no contiene el archivo `page.js` esta ruta no será expuesta publicamente, por lo que se podría utilizar para almacenar archivos estáticos como scripts, imágenes, estilos CSS, etc.

En la Figura 10.2 se evidencia de una manera más clara cómo funciona la creación de rutas de

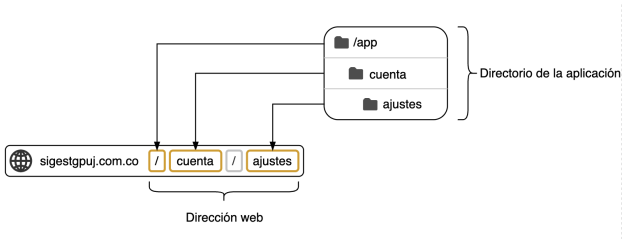


Figura 10.2: Sistema de archivos para generar rutas de Next JS

acuerdo a las carpetas que se encuentren en el directorio `/app`. Cabe recordar que para que una ruta sea pública esta debe contener el archivo `page.js`.

En el sistema de archivos de **Next JS** existen diferentes características que se pueden utilizar de acuerdo al proyecto que se esté realizando. Por ejemplo, si una carpeta que se encuentra en el directorio `/app` tiene el nombre entre paréntesis, este se convierte en un "grupo de rutas" y evita que la carpeta se incluya como un segmento de ruta en la aplicación [Rou23]. Esta característica permite organizar las rutas de la aplicación en grupos lógicos sin afectar la estructura de rutas.

En el desarrollo FrontEnd de este trabajo de grado se aprovechó la característica de los grupos de rutas para dividir las rutas de la aplicación en dos secciones. Como se puede observar en la estructura de carpetas al inicio de esta sección, es fácil notar que hay dos grupos de rutas:

- `(app)` : Este grupo de rutas alberga las rutas principales de la aplicación las cuales sólo pueden ser accedidas por un usuario si este ha iniciado sesi'on previamente.
- `(auth)` : En este grupo de rutas se definen las rutas del sistema de autenticación de la aplicación, donde los usuarios pueden iniciar sesión, crear una cuenta o recuperar su contraseña.

Cabe aclarar que hay rutas dentro de `(app)` que no están disponibles para todos los tipos de usuarios, sin embargo, esta validación se realiza con funciones auxiliares y no en el sistema de archivos.

10.4. Desarrollo

En la fase inicial del desarrollo del frontend se inició con la creación de la página principal, que sirve como el punto de entrada inicial para los usuarios luego de iniciar sesión. Esta página presenta una interfaz intuitiva y fácil de navegar, diseñada para proporcionar a los usuarios acceso rápido a las funciones clave de la aplicación.

10.4.1. Página Principal

La página principal fue estructurada de manera clara y concisa, priorizando la usabilidad y la accesibilidad. Se implementó enlaces directos a las secciones de "Anteproyectos" y "Trabajos de

Grado", lo que brinda a los usuarios la posibilidad de acceder rápidamente a los anteproyectos o a los trabajos de grados registrados en el sistema.

Además, se incorporó la opción de "usuarios" que es exclusiva de los usuarios con rol **ADMIN** la cual permite gestionar los perfiles de todos los usuarios del sistema, pudiendo modificar roles, eliminar usuarios o editar información.

En la figura 10.4 se muestra la página principal de la aplicación, tendiendo en cuenta que esa es la vista de un usuario con rol de **ADMIN**.

Figura 10.3: Página Principal de la aplicación. Vista de un usuario **ADMIN**

10.4.2. Barra de Navegación

La barra de navegación es un componente que se muestra en todas las pantallas de la aplicación, permietiendo al usuario navegar fácilmente entre secciones, además tambien tiene el submenú "cuenta" el cual agrupa las diferentes acciones que puede realizar el usuario actual, cómo ver su perfíl o cerrar sesión.

Figura 10.4: Barra de Navegación

El item "usuarios" es exclusivo de los usuarios con rol **ADMIN** del mismo modo como sucede

en las opciones de la página principal.

El desarrollo de la página principal y de la barra de navegación sentaron la base del desarrollo FrontEnd de la aplicación, ya que en estas se definieron los colores y la arquitectura de la interfaz con la cual se basó el desarrollo de las otras páginas.

10.4.3. Página de Anteproyectos

La página de anteproyectos, que se accede con la url `/anteproyectos` se encuentra la tabla que muestra la información de todos los anteproyecto registrados en la base de datos de la aplicación. La tabla consta de 5 columnas **AUTOR/AUTORES**, **TÍTULO**, **FECHA CREACIÓN**, **FECHA APROBADO** y **ESTADO** que identifica a cada anteproyecto. La columna **ESTADO** indica la fase en la que se encuentra un determinado anteproyecto, el cual puede ser el siguiente:

- APROBADO : El anteproyecto ha sido revisado y aprobado por el evaluador asignado.
- PENDIENTE : El anteproyecto está a la espera de ser revisado por el evaluador, o tiene correcciones pendientes.
- NO APROBADO : Luego de tres entregas, el anteproyecto no cumplió con lo mínimo exigido por el evaluador, o no fue corregido en el tiempo estipulado.

La tabla fue desarrollada utilizando la librería **React Table**, la cual facilitó agregar funcionalidades a la tabla cómo filtrar, ordenar, buscar y mostrar los anteproyectos de acuerdo a las necesidades del usuario. A continuación se detallan las funcionalidades con las que cuenta la tabla:

- Barra de búsqueda: Esta barra es una entrada de texto que se encuentra en la parte superior izquierda de la tabla y permite buscar un anteproyecto por su título.
- Filtro por Estado: Esta funcionalidad, que se encuentra al lado de la barra de búsqueda, permite filtrar los anteproyectos por su estado. Se puede seleccionar más de un filtro.
- Ordenamiento: Permite ordenar los anteproyectos de forma ascendente o descendente de acuerdo a la columna a la que se le aplique. Estas pueden ser la columna **TÍTULO**, **FECHA CREADO** o **FECHA APROBADO**. En el caso de la columna **TÍTULO** el ordenamiento será alfabético. El ordenamiento se muestra con un ícono al lado del nombre de las columnas mencionadas anteriormente y tiene tres estados:

  1. Este es el estado por defecto, es un ícono de dos flechas y refleja que no se está aplicando ningún ordenamineto a la tabla.
  2. El segundo estado muestra un ícono de una flecha apuntando hacia abajo indicando que se esta ordenando la tabla descendentemente de acuerdo a la columna aplicada.
  3. El tercer estado muestra una flecha hacia arriba indicando que se está ordenando ascendentemente.

El ordenamiento se puede modificar haciendo click sobre el ícono de las flechas al lado del nombre de la columna seleccionada.

- Paginación : La paginación es un componente que permite navegar a través de las diferentes páginas de la tabla. Cada página por defecto muestra hasta 5 anteproyectos. Este valor puede ser modificado en la definición del estado incial de la tabla que se encuentra en el archivo `/app/(app)/anteproyecto/page.tsx`

En la Figura 10.5 se muestra la página de anteproyectos con las funcionalidades mencionadas anteriormente. Cabe recordar que esta página solo es visible para usuarios con el rol de **ADMIN** o **EVALUADOR**.

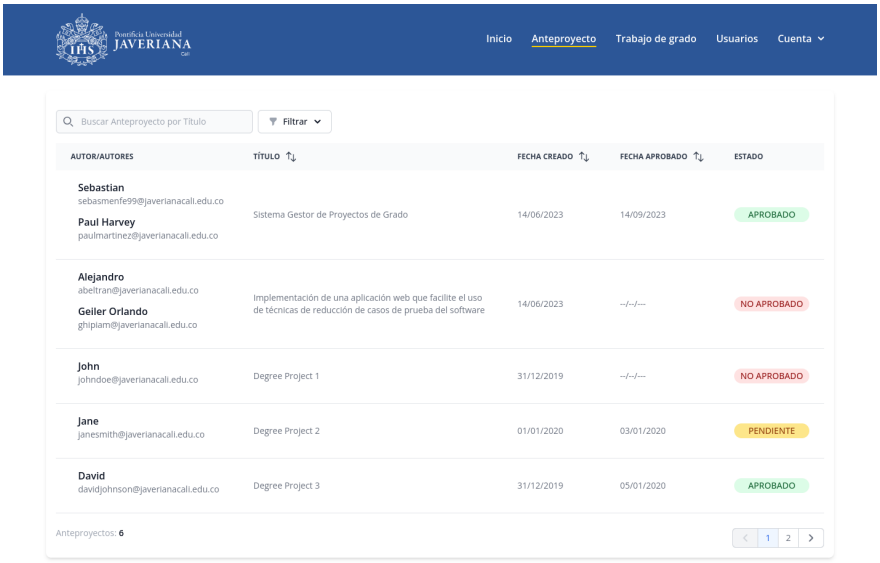


Figura 10.5: Página de Anteproyectos

Cuando se selecciona un anteproyecto se despliega un cuadro flotante con toda la información de ese anteproyecto. Si la acción la realiza un usuario **ADMIN** este tendrá habilitado un botón que lo redireccionará a otra página para editar la información de ese anteproyecto, caso contrario, si la acción la realiza un usuario **EVALUADOR** este tendrá habilitado el botón para realizar la evaluación de ese anteproyecto. En la siguiente figura se muestra la información de un anteproyecto seleccionado.

10.4.4. Página de Trabajos de Grado

De manera similar al desarrollo abordado en la sección 10.4.3 y aprovechando la ventaja de la reutilización de componentes, se utilizó la misma estructura para desarrollar la página de trabajos de grados que utiliza el mismo componente de tablas pero con unos ajustes diferentes.

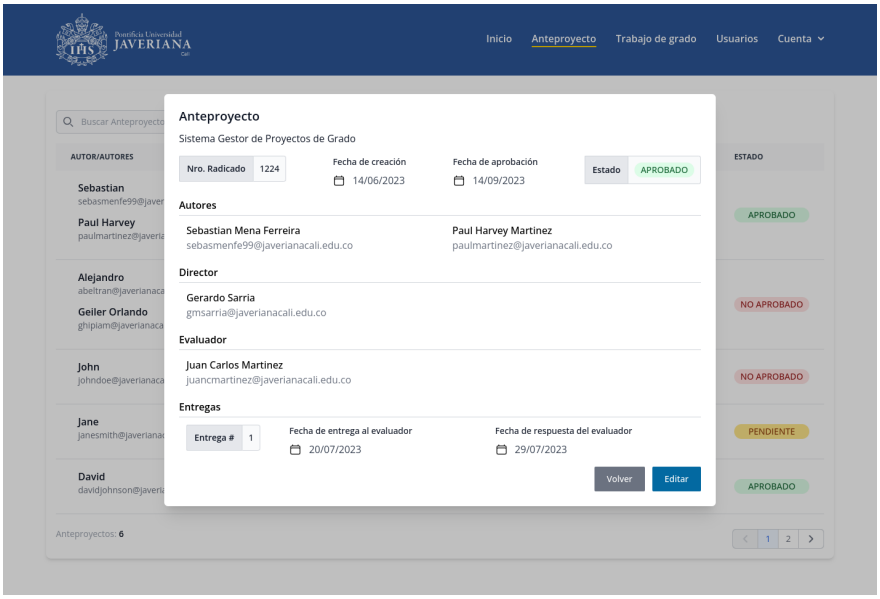


Figura 10.6: Información de un Anteproyecto Seleccionado

Esta página tiene como fin mostrar todos los trabajos de grados registrados previamente en el sistema y cuenta con las mismas funcionalidades de búsqueda y filtración de la tabla de **Anteproyectos**.

El acceso a esta página es por medio de la ruta `/trabajos-de-grado` y se muestra solo a usuarios que cuenten con el rol de **ADMIN** o **EVALUADOR**.

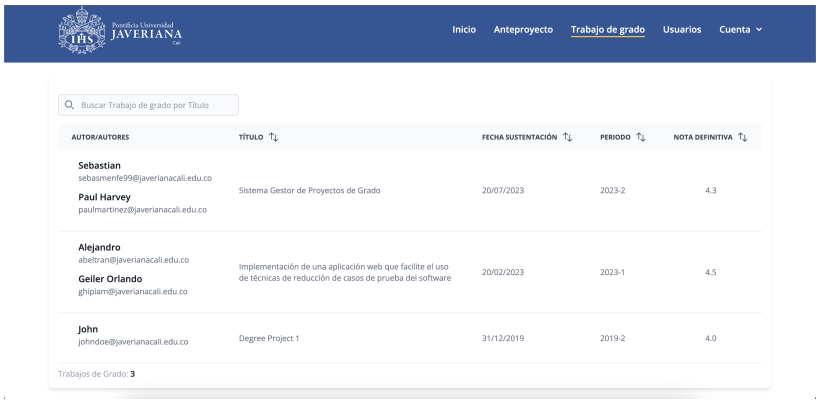


Figura 10.7: Página de Trabajos de Grado

10.4.5. Inicio de Sesión

En la página de inicio de sesión los usuarios pueden acceder a la aplicación ingresando sus credenciales. En caso de olvidar la contraseña se debe contactar con el administrador para poder efectuar el cambio de contraseña. Esta página sólo accesible si no hay una sesión activa.

Figura 10.8: Página de Inicio de Sesión

10.4.6. Registro de Usuario

En la página de registro los usuarios pueden crear una cuenta en el sistema. Al momento de crear la cuenta el rol por defecto asignado es de **ESTUDIANTE**, si se requiere un rol diferente se debe solicitar el cambio al administrador. De manera similar a la página de **Inicio de Sesión** esta sólo es accesible si no hay una sesión activa.

Figura 10.9: Página de Registro de Usuario

Capítulo 11

Pruebas

11.1. Backend

En la fase de pruebas para el backend, se llevaron a cabo evaluaciones destinadas a garantizar tanto la funcionalidad como el rendimiento óptimo de la aplicación. Este proceso abarcó pruebas funcionales, que se centraron en verificar que cada módulo y microservicio operara según lo previsto, y pruebas de rendimiento para evaluar el comportamiento del sistema bajo cargas diversas. Dentro de las pruebas funcionales, se diseñaron escenarios específicos que reflejan diversos comportamientos y situaciones que podrían surgir en la aplicación. Este enfoque no solo abarca pruebas unitarias, sino que, en varios escenarios, también se considera la interacción entre múltiples módulos, lo que podría considerarse como pruebas de integración. Estos procesos de evaluación no solo aseguran la correcta funcionalidad del backend, sino que también proporcionan una visión integral la robustez y confiabilidad de los módulos particulares sino también del sistema en su conjunto.

Funcionales

En el primer escenario funcional, se probó la creación, asignación de roles y eliminación de un usuario. El proceso incluyó la creación de un nuevo usuario, el login del administrador, la asignación de un rol al usuario, la verificación de que el usuario pudiera iniciar sesión con el nuevo rol, la revisión del administrador, y finalmente, la eliminación del usuario. Todos los pasos se ejecutaron sin problemas como podemos observar en la figura 11.1, confirmando la correcta ejecución de cada endpoint y a su vez que la interacción entre los módulos de Registro, Login y Usuario se da de manera adecuada. Este escenario validó eficazmente la funcionalidad de creación y gestión de usuarios, así como la asignación de roles en el sistema.

En el segundo escenario funcional, se evaluó la interacción entre los módulos de Login, Anteproyecto y Usuario. La prueba consistió en el login del administrador, la creación de un anteproyecto, la modificación de este anteproyecto mediante diferentes solicitudes de actualización, la adición de otro autor, la eliminación de evaluadores y, finalmente, la eliminación del anteproyecto en si. Los resultados fueron exitosos, y el tiempo medio de respuesta para las 10 peticiones se mantuvo por debajo de los 700 ms. Este escenario validó eficientemente la funcionalidad de creación, modificación y eliminación de anteproyectos, demostrando una interacción efectiva entre los módulos involucrados. Los resultados de esta prueba están en la Figura 11.2.

En el tercer escenario funcional, se evaluó la interacción entre los módulos de Login, Usuario,

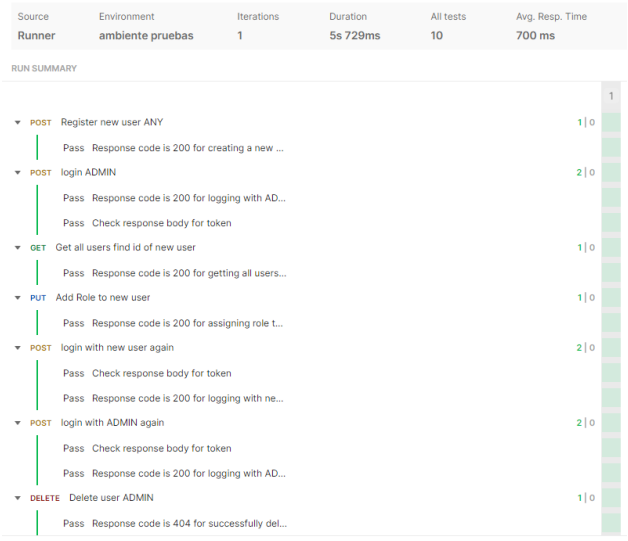


Figura 11.1: Primera prueba funcional.

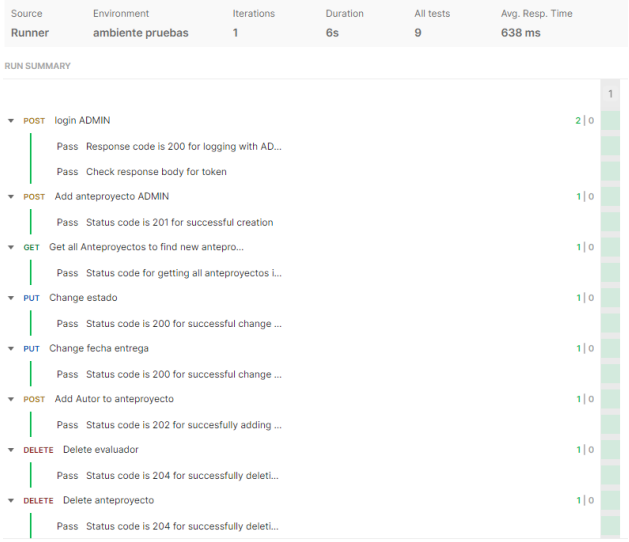


Figura 11.2: Segunda prueba funcional.

Anteproyecto y ProyectoGrado. La prueba consistió en el login del administrador, la creación de un anteproyecto del cual se obtuvo el ID, la creación de un proyecto, la modificación de campos de este proyecto, y finalmente, la eliminación del anteproyecto asociado, lo que debería haber provocado la eliminación en cascada del proyecto correspondiente. Los resultados de la prueba fueron exitosos, que podemos ver en la figura 11.3, validando tanto la ejecución de cada endpoint particular como la integración efectiva de los módulos involucrados. Esto destaca la consistencia y funcionalidad adecuada de la creación y modificación de proyectos en el sistema, confirmando que las interacciones entre los módulos se desarrollan según lo esperado.

En la última prueba funcional, se evaluaron los módulos de Archivo, Login, Anteproyecto, Proyecto y Usuario. El escenario abarcó desde el login de un usuario asociado a un anteproyecto o proyecto en particular, subir un archivo a este trabajo, hasta la eliminación de este trabajo asociado. Los resultados obtenidos fueron exitosos, validando la ejecución adecuada de cada endpoint específico y la integración eficiente entre los módulos. La prueba demuestra que tanto las operaciones individuales de gestión de archivos como la interacción entre los distintos componentes del sistema se desarrollan de manera coherente y satisfactoria. Los resultados antes mencionados los podemos ver en la Figura 11.4.

Performance

En el escenario de prueba de rendimiento, se simularon condiciones de carga para el módulo de

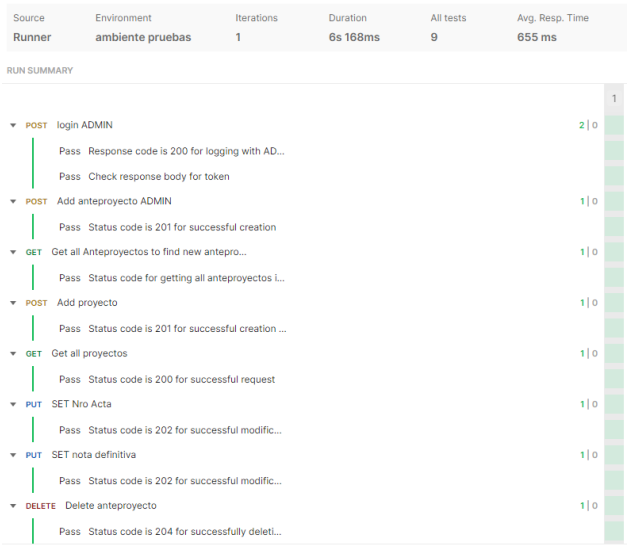


Figura 11.3: Tercera prueba funcional.

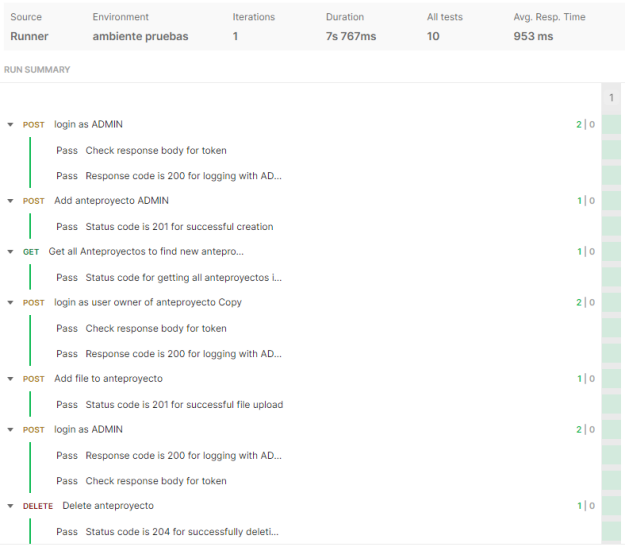


Figura 11.4: Cuarta prueba funcional.

registro con 5 usuarios concurrentes. Se monitorearon los milisegundos de respuesta, y aunque se realizaron más de 110 solicitudes desde usuarios diferentes, se observó un índice de error cercano al 10 %. Aunque esta tasa de error no es alarmante, identificamos oportunidades para mejorar la confiabilidad de este endpoint. Es importante señalar que sometimos el sistema a un escenario más exigente de lo que se espera en condiciones reales. Simulamos una carga fija de un usuario durante 30 segundos, seguida de un aumento gradual a 5 usuarios durante 1 minuto y 20 segundos, manteniendo esta carga durante 10 segundos. Consideramos el resultado de la prueba exitoso pués la carga fué mayor a lo que esperamos que reciba nuestra aplicación. El resultado se puede ver a mas detalle en la Figura 11.5.

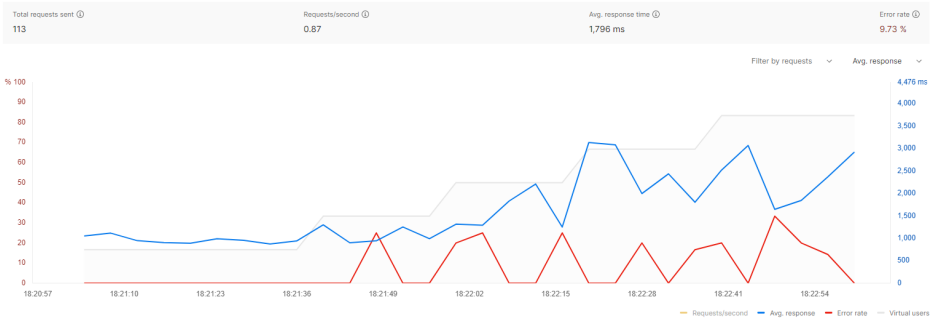


Figura 11.5: Primera prueba de performance.

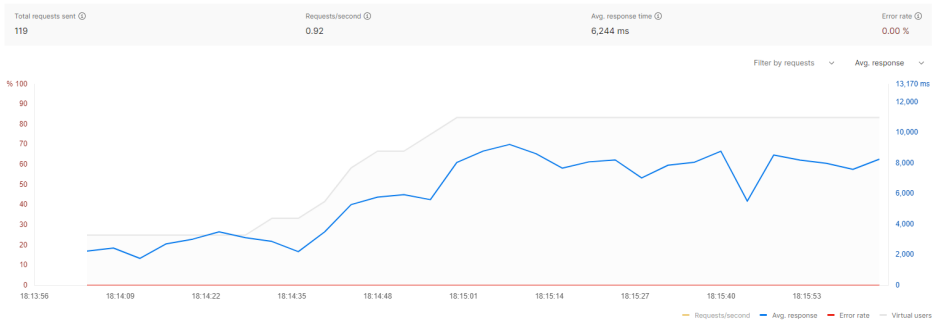


Figura 11.6: Segunda prueba de performance.

En el escenario de prueba de rendimiento para el módulo de inicio de sesión, sometimos el sistema a condiciones simuladas de carga con 5 usuarios concurrentes. Se monitorizaron los milisegundos de respuesta, y los resultados fueron altamente satisfactorios, con un 0 % de tasa de error en las más de 119 solicitudes realizadas por 5 usuarios diferentes en el lapso de dos minutos. La modalidad de la prueba consistió en simular una carga fija de 1 usuario durante 30 segundos, seguida de un aumento constante de la carga a 5 usuarios durante los próximos 30 segundos, manteniendo este nivel durante 1 minuto. Aunque consideramos la prueba exitosa y confirmamos la alta confiabilidad de este módulo, observamos que existe margen para mejorar el tiempo de respuesta. El esquema completo de la prueba lo podemos ver en la Figura 11.6.

En el escenario 3, que aborda el módulo de anteproyecto, ejecutamos las mismas pruebas que en el escenario funcional número 2 en 10 iteraciones. Aunque se presentó un único fallo en el endpoint de agregar autor a un anteproyecto, consideramos las pruebas exitosas, ya que se enviaron 90 solicitudes con un tiempo de respuesta promedio inferior a 500 ms. Profundizando en el fallo, identificamos que se debió a un problema en Postman y no en la aplicación, ya que las credenciales se ingresaron correctamente.

De manera similar, en el escenario 4, centrado en el módulo de proyecto, realizamos pruebas análogas a las del módulo de anteproyecto en 10 iteraciones. Obtuvimos resultados muy similares, enviando 90 peticiones con un tiempo de respuesta promedio inferior a 500 ms y sin experimentar fallos en ninguna petición. Tanto el módulo de anteproyecto como el de proyecto demostraron ser robustos y eficientes en condiciones de estrés, reafirmando la calidad y confiabilidad de sus endpoints. Los resultado de ambas pruebas los podemos ver en las Figuras 11.11 y 11.12.

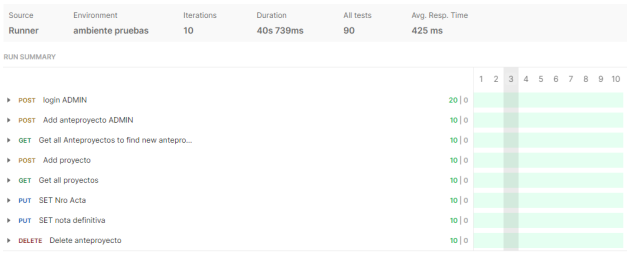


Figura 11.7: Tercera prueba de performance.

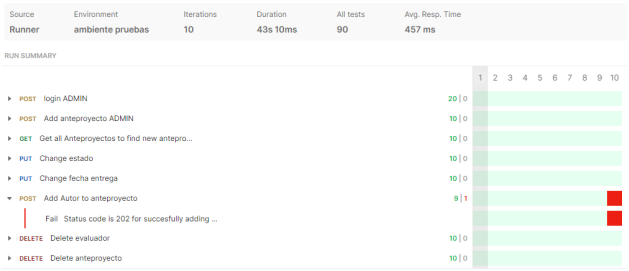


Figura 11.8: Cuarta prueba de performance.

11.2. Frontend

Para verificar la usabilidad de la aplicación, se emplearón pruebas end-to-end utilizando Cypress, una herramienta de automatización de pruebas moderna. Cypress permite simular interacciones de usuario reales y verificar el comportamiento de la aplicación en un entorno controlado.

- **Objetivos de las Pruebas:** Las pruebas end-to-end tienen como objetivo principal evaluar la usabilidad de la aplicación desde la perspectiva del usuario final. Se verificó la navegación entre las diferentes secciones de la aplicación, la interacción con los elementos de la interfaz de usuario y la respuesta del sistema a las acciones del usuario.
- **Aspectos a Evaluar:** Durante las pruebas, se prestó especial atención a diversos aspectos de la usabilidad, como la claridad y consistencia de la interfaz de usuario, la facilidad de navegación, la respuesta del sistema a las acciones del usuario, y la accesibilidad para usuarios con diferentes niveles de habilidad.

11.2.1. Plan de pruebas

1. **Creación de usuario exitoso:** Para probar la creación de usuario en la aplicación se partió desde la pantalla de registro ingresando los siguientes datos:
   - **Nombre:** Nombres usuario
   - **Apellidos:** Apellidos usuario
   - **Identificación:** 123456
   - **Correo:** usuario1@mail.com
   - **Contraseña:** usuario1
   - **Confirmación contraseña:** usuario1

   La prueba se marca como exitosa cuando despues de hacer click en el boton de registro este redirecciona a la página principal del usuario.

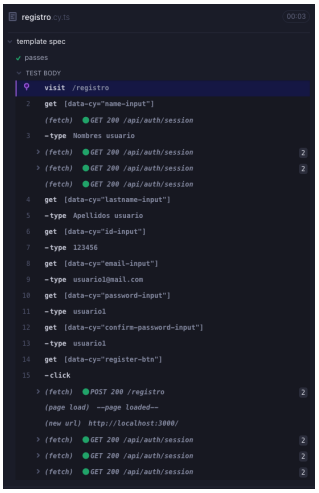


Figura 11.9: Inicio de la prueba de creación de usuario ejecutada por Cypress.

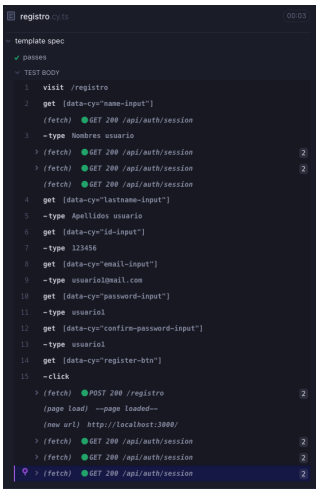


Figura 11.10: Fin de la prueba de creación de usuario ejecutada por Cypress.

2. **Inicio de sesión exitoso** Para probar el inicio de sesión de la aplicación se utilizó el usuario administrador con credenciales `correo: admin@admin.com` y `contraseña: 123123`. Para comprobar el éxito de la prueba esta tiene que comenzar en la página de login de aplicación y terminar en la página principal de la aplicación, demostrando la autenticación exitosa del usuario.

3. **Creación de anteproyecto:** Para probar la creación de anteproyectos se debe ingresar con un usuario administrador. Para este caso, se utilizó el mismo usuario empleado en la prueba de inicio de sesión el cual es administrador, luego de iniciar sesión este debe ir a la página de anteproyectos, dar click en "Crear Anteproyecto" e ingresar los siguientes datos en los campos correspondientes:

   - **Nro Radicado:** 123456
   - **Título:** Anteproyecto de prueba 1
   - **Autores:** usuario1@mail.com
   - **Directores:** director1@mail.com
   - **Evaluadores:** evaluador1@mail.com

   Para comprobar que la prueba fue exitosa luego de dar click en el botón de "crear", el usuario debe ser redireccionado a la página del anteproyecto creado.

4. **Edición de anteproyecto:** De manera ańaloga a la prueba de creación de anteproyectos, se debe ingresar con un usuario administrador, luego de iniciar sesión este debe ir a la página de anteproyectos, buscar el anteproyecto "Anteproyecto de prueba 1", dar click en él, darle al

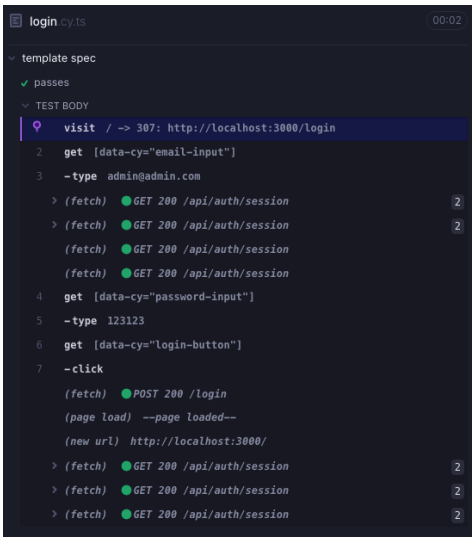


Figura 11.11: Inicio de la prueba de incicio de sesión ejecutada por Cypress.

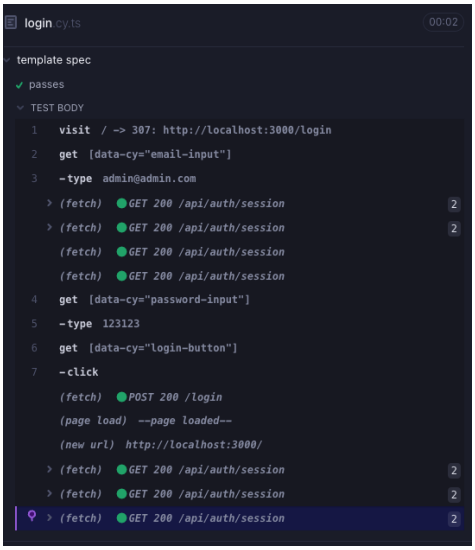


Figura 11.12: Fin de la prueba de incicio de sesión ejecutada por Cypress.

botón “Ver” y cuando se encuentre en la página del anteproyecto debe dar click en “Editar”, finalmente cuando se encuentre en página de edición debe cambiar los siguientes campos:

- **Título:** Anteproyecto de prueba modificado 1
- **Autores:** Agregar el nuevo autor “usuario2@mail.com”

Para comprobar que la prueba fue exitosa, luego de dar click en el botón de “Guardar”, el usuario debe ser redireccionado a la página del anteproyecto editado.

5. **Eliminación de anteproyecto:** Al igual que las dos pruebas anteriores, se debe ingresar con un usuario administrador, luego de iniciar sesión este debe ir a la página de anteproyectos, buscar el anteproyecto “Anteproyecto de prueba 1”, dar click en él, darle al botón “Ver” y cuando se encuentre en la página del anteproyecto debe dar click en “Editar”, finalmente cuando se encuentre en página de edición debe dar click en el botón “ Borrar”, si la prueba fue exitosa el usuario será redirigido a la página de los anteproyectos confirmando así que el anteproyecto fue borrado:

Capítulo 12

Conclusiones

A nuestra consideración este proyecto ha alcanzado con éxito los objetivos planteados inicialmente, logrando la implementación cohesiva y coherente de los tres componentes fundamentales: la aplicación backend desarrollada en Java con Spring Boot, el sistema de gestión de la información con PostgreSQL, y la capa de frontend. Estos componentes se han dise ñado y desarrollado de manera que no solo cumplen con las funcionalidades requeridas, sino que también exhiben una escalabilidad inherente. Esta flexibilidad permite adaptarse a los cambios en las necesidades de los usuarios y en los requisitos de la solución, asegurando que la aplicación pueda evolucionar sin problemas a medida que se presenten nuevos desafíos o se requieran mejoras. La coherencia en el diseño y la interacción entre estos elementos proporciona una base sólida para el mantenimiento futuro y posibilita la continuidad del desarrollo en respuesta a las dinámicas demandas del entorno. En conjunto, el proyecto no solo ha logrado satisfacer las metas establecidas, sino que también ha sentado bases sólidas para un sistema robusto y adaptable a medida que evoluciona en consonancia con las necesidades cambiantes tanto de los usuarios como de la solución misma.

En el Cuadro 12.1 se encuentran todas las referencias clave para acceder tanto a la documentación prometida, que se presentará en forma de video para el manual de usuario, como a la documentación técnica de la API a través de Swagger, como se discutió en la sección de referencia de la API. Además, esta tabla proporciona las direcciones a los repositorios de código, lo que permite un fácil acceso y desarrollo futuro del código fuente si fuera necesario.

Direcciones de entregables	
Manual de usuario	Link[MFMA24]
Documentación de la api	Link[MAMF23]
Código fuente del Backend	Link[MA23]
Código fuente del Frontend	Link [MF23]

Figura 12.1: Direcciones importantes.

12.1. Conclusiones para el Backend

En lo que respecta al desarrollo específico de la aplicación backend, la documentación de la API se ha estructurado de manera completa, siguiendo las prácticas actuales de la industria. La implementación de la documentación con Swagger ha desempeñado un papel fundamental, ya que

no solo proporciona una representación clara y detallada de los puntos finales de la API, sino que también ofrece la flexibilidad de ser probada en diferentes entornos. Durante las etapas iniciales de implementación, la API puede ser probada en un servidor de mockup ofrecido por Swagger, lo que facilita la validación temprana de su funcionamiento sin depender de datos reales. Posteriormente, también puede ser probada en el servidor de producción con información real, la misma documentación se convierte en una herramienta valiosa para desarrolladores y usuarios, garantizando una comprensión completa de las capacidades y el uso de la API en un entorno de producción. Este enfoque integral asegura que la documentación no solo cumple con los estándares actuales de la industria, sino que también contribuye a la eficiencia y confiabilidad de la aplicación backend.

En este mismo sentido, la solución implementada en el backend ha seguido nuestra visión inicial, adoptando prácticas que están alineadas con los estándares de la industria actual. Aunque se trata de una aplicación de menor escala en términos de servicios ofrecidos, hemos aplicado conceptos y tecnologías de vanguardia. Este enfoque, aunque no se estableció explícitamente en los objetivos iniciales, era una de las metas fundamentales que buscábamos alcanzar: desarrollar un backend que, a pesar de su orientación hacia una aplicación de menor envergadura, destacara por su robustez en rendimiento, tecnologías utilizadas y arquitectura general. Este logro contribuye significativamente al mantenimiento y eficiencia de la aplicación, sentando las bases para futuras expansiones o actualizaciones sin comprometer la integridad y eficacia del backend.

Considerando una posible mejora al desarrollo expuesto anteriormente, una opción valiosa sería la implementación de un balanceador de cargas al desplegar la aplicación. Esto no solo optimizaría el rendimiento general de la aplicación backend, sino que también mejoraría la distribución eficiente de las peticiones entre los microservicios. Además, existe la oportunidad de explorar soluciones más avanzadas ofrecidas por plataformas como Amazon Web Services (AWS) en términos de escalamiento. Tanto el escalamiento vertical, que implica agregar recursos a la máquina existente, como el escalamiento horizontal, que implica replicar instancias de la aplicación (conocido como "workers"), podrían ser estrategias valiosas, si la solución así lo requiriera, para garantizar un rendimiento óptimo incluso en situaciones de mayor demanda. Estas opciones ofrecen una mayor flexibilidad y permiten ajustar dinámicamente la capacidad de la aplicación según las necesidades cambiantes del entorno.

Por otro lado, El servicio de hosting que seleccionamos proporcionó las condiciones necesarias para llevar a cabo pruebas completas de nuestra solución en un entorno semi-productivo. Este entorno resultó ser más que adecuado para nuestras necesidades, superando incluso las expectativas establecidas inicialmente en nuestros alcances. La capacidad de realizar pruebas en un ambiente que simula condiciones cercanas a las de producción fue esencial para validar el rendimiento y la fiabilidad de la aplicación en situaciones similares a las del mundo real. Esta flexibilidad proporcionada por el hosting no solo facilitó el desarrollo y la prueba de nuestra solución, sino que también demostró ser crucial para evaluar el comportamiento del sistema en diversas condiciones antes de su implementación definitiva.

A pesar de las ventajas proporcionadas por el servicio de hosting utilizado para pruebas semi-productivas, es crucial reconocer una oportunidad de mejora al considerar el despliegue completo de la solución en un ambiente totalmente productivo. En este sentido, es necesario evaluar servicios de hosting que ofrezcan la capacidad de desplegar cada microservicio de manera independiente, para acoplarnos completamente a la arquitectura por microservicios como hemos discrutido anteriormente. Aunque esta característica es común y no representa una dificultad significativa para encontrarla, nuestra restricción presupuestaria para este proyecto limitó la adopción de esta opción desde las etapas iniciales. La capacidad de desplegar microservicios de manera independiente proporcionaría una mayor flexibilidad operativa, facilitando la gestión, escalabilidad y mantenimiento eficiente de cada componente de la aplicación en un entorno productivo. Esta consideración se presenta como una oportunidad valiosa para futuras iteraciones o implementaciones que busquen optimizar aún más la arquitectura y el rendimiento de la solución.

De manera similar, La estructura actual del proyecto se ve directamente influenciada por la restricción presupuestaria mencionada anteriormente. Aunque se ha implementado un "scaffold" basado en microservicios, lo cual representa una organización no convencional, es una estructura que satisface las necesidades de nuestro poryecto y cumple con un rol particular dentro de nuestro proyecto. Esta estructura se adaptó para cumplir con las limitaciones económicas presentes en esta fase del proyecto. Sin embargo, es importante destacar que esta configuración no es inmutable, y de hecho está diseñada de manera que, en iteraciones futuras donde las restricciones económicas sean menos apremiantes, pueda ser modificada de manera sencilla. En estas fases posteriores del proyecto, existe la flexibilidad para ajustar la estructura y alinearse completamente con el paradigma convencional de microservicios, como se discutió en secciones anteriores. Este enfoque permite que la arquitectura del proyecto evolucione de manera orgánica y se adapte a las necesidades cambiantes y a las mejores prácticas de desarrollo de software, a medida que se disponga de más recursos.

12.2. Conclusiones para el Sistema de información

La arquitectura de la base de datos ha sido diseñada cuidadosamente, generando una estructura que percibimos como altamente escalable y legible para cualquier persona que pueda contribuir al desarrollo del proyecto en el futuro. Aunque se presenta como una arquitectura sencilla, su simplicidad es una fortaleza, ya que proporciona una base sólida que permite el crecimiento gradual y sostenible de la solución a lo largo del tiempo. Cada tabla y relación ha sido cuidadosamente concebida para optimizar el rendimiento y facilitar la comprensión de nuestro sistema de información, garantizando que incluso a medida que la aplicación se expanda, la base de datos pueda adaptarse sin perder claridad en su estructura. Esta elección de diseño busca no solo cumplir con las necesidades actuales, sino también anticipar y facilitar futuras expansiones y modificaciones en el sistema.

Una oportunidad clave de mejora se centra en el despliegue de la base de datos. Similar a las restricciones encontradas para los microservicios, la arquitectura actual concentra toda la información

en una única base de datos. Para una mejora sustancial en escalabilidad y eficiencia operativa, se podría considerar la opción de separar la arquitectura, distribuyendo la información entre múltiples bases de datos. Esta estrategia facilitaría operaciones más ágiles y tiempos de respuesta más eficientes, especialmente cuando se enfrenta a un aumento significativo en las peticiones de usuarios. La separación de bases de datos permitiría una gestión más especializada y optimizada de conjuntos específicos de datos, mejorando la capacidad del sistema para manejar cargas de trabajo intensivas y proporcionando una base sólida para futuras expansiones y optimizaciones en la infraestructura de datos.

12.3. Conclusiones para el Frontend

El desarrollo del FrontEnd de la aplicación comenzó con la compresión completa de los requerimientos de los usuarios interesados en el sistema, esto permitió que las funcionalidades implementadas estuvieran siempre alineadas con las necesidades reales de los usuarios finales. Una vez realizado lo anterior, fué mucho más fácil poder escoger las tecnologías que permitierion tener un desarrollo ágil, eficiente y escalable. La adopción de frameworks cómo Next JS permitieron simplificar la arquitectura de la aplicación como también tener un rendimiento óptimo, además, la utilización de librerías como React hicieron mucho más fácil la mantenebilidad del código y la reutilización de componentes en las diferentes páginas de la aplicación.

La adopción de tecnologías FrontEnd de vanguardia permite la mejora y ampliación de funcionalides en la aplicación web, ya que al usar varias de las tecnolgías más demandadas actualmente existe gran documentación en internet que permite la implementación de nuevas características que se ajusten a las necesidades emergentes del sistema. En adición a lo anterior, cabe resaltar que este trabajo de grado se documentaron todas las características principales con las que posee el sistema en su versión inicial.

En conclusión, la adopción de prácticas de desarrollo actuales ha demostrado ser esencial para la entrega de un producto tecnológico que se ajusta a las necesidades específicas de los usuarios. Desde la identificación de los requerimientos hasta la elección de tecnologías y la implementación de librerías especializadas, seguir prácticas de desarrollo modernas ha sido clave para desarrollar este proyecto. La agilidad lograda mediante frameworks como Next JS, la atención a la experiencia del usuario con librerías específicas, y la iteración constante en respuesta a la retroalimentación, han sido elementos críticos en la creación de este trabajo de grado. La importancia de estas prácticas radica en su capacidad para mantener la relevancia y la eficacia del producto en un entorno tecnológico en constante evolución, garantizando así un impacto positivo y duradero en los usuarios finales.

Capítulo 13

Trabajo Futuro

En cuanto a futuras mejoras y expansiones de nuestra solución, se sugiere realizar una serie de pasos para optimizar la estructura y el despliegue del backend, el sistema de información, y el sistema en términos generales. En primer lugar, se debería avanzar hacia la independencia total de cada microservicio, separando el código de cada uno para facilitar el mantenimiento y desarrollo independiente. Posteriormente, se propone desplegar cada microservicio de manera individual, lo que permitiría una gestión más eficiente y escalable de cada componente.

En el ámbito del sistema de información, se recomienda explorar un despliegue mÃ¡s separado y específico para cada microservicio, evitando centralizar toda la información en una única base de datos. Este enfoque modular y distribuido no solo mejoraría el rendimiento al permitir una gestión más especializada de los datos, sino que también fortalecería la seguridad y la resiliencia del sistema al limitar el impacto de posibles fallos en un componente aislado.

En una fase más avanzada, se podría considerar la implementación de un balanceador de cargas para distribuir eficientemente las solicitudes entre los microservicios. Además, la opción de utilizar servicios que admitan un escalado horizontal podría proporcionar beneficios significativos en términos de rendimiento y disponibilidad. Un enfoque de este tipo permitiría manejar de manera más efectiva aumentos inesperados en la demanda, garantizando una experiencia fluida para los usuarios finales.

Adicionalmente, una mejora considerable es plantear la posibilidad de integrar el sistema central de autenticación de la universidad como el sistema de autenticación que use nuestra aplicación. En lugar de manejar la autenticación como lo hacemos actualmente, podriamos usar los servicios que todas las aplicaciones de la universidad usan, favoreciendo la homogeneidad entre estas aplicaciones y la nuestra. Así mismo, esto favoreceria la experiencia de usuario pues no tendría que crea un usaurio adicional sino que con uno solo podria acceder a todas las aplicaciones existentes dentro del ecosistema informático de la universidad.

Por otro lado, también se podría explorar la posibilidad de compartir información relevante con los otros sistemas de la universidad que podrían enriquecer la experiencia del usuario. La interoperabilidad con plataformas como la gestión de cursos, calendarios académicos y bases de datos estudiantiles podría facilitar la obtención de información contextual y mejorar la eficiencia en la gestión global de actividades académicas.

Por otro lado el desarrollo de este proyecto ha proporcionado no solo avances técnicos sino también lecciones valiosas. Entre ellas, destaca la importancia de una comunicación efectiva dentro del equipo y la necesidad de mantener un equilibrio adecuado entre la innovación tecnológica y las restricciones presupuestarias. Además, se resalta la importancia de considerar de manera proactiva aspectos como la escalabilidad y la modularidad en la arquitectura del sistema. Estas enseñanzas nos quedan como pilares fundamentales para futuros proyectos, subrayando la necesidad de una gestión eficiente, la anticipación de desafíos potenciales y el aprendizaje continuo en el ámbito del desarrollo de software.

Además, el trabajo en equipo ha sido fundamental para el éxito del proyecto. La colaboración entre los miembros del equipo, cada uno aportando sus habilidades y conocimientos únicos, ha permitido superar desafíos y alcanzar metas de manera eficiente. Esta experiencia resalta la importancia de la comunicación efectiva, la coordinación y el apoyo mutuo en la consecución de objetivos comunes. Asimismo, ha demostrado la relevancia de mantener un ambiente de trabajo positivo y colaborativo, donde se fomente la creatividad y se promueva la confianza entre los integrantes del equipo. Estas lecciones sobre trabajo en equipo son transferibles a cualquier entorno profesional y representan un valor a ñadido más allá de los límites del proyecto actual.

En conclusión, antes de abordar nuevas funcionalidades, se sugiere priorizar las acciones anteriormente expuestas para que a partir de una base más sólida de la que estamos entregando puedan construir funcionalidades que aseguren la adaptabilidad a medida que evolucionan los requisitos y las condiciones operativas de la solución.

Bibliografía

[Abo23] About postgresql. Disponible en: https://www.postgresql.org/about/, Nov 2023.

[amz23] What is a restful api? Disponible en: https://aws.amazon.com/what-is/restful-api/, 2023.

[App23] Routing fundamentals. Disponible en: https://nextjs.org/docs/app/building-your-application/routing, Nov 2023.

[BGT13] Shobha R. Bharamagoudar, R. B. Geeta, and Shashikumar G. Totad. Web based student information management system, 2013.

[BM11] Tim Berglund and Matthew McCullough. *Building and testing with Gradle*. .ºReilly Media, Inc.", 2011.

[dC22] Universidad Nacional de Colombia. Sistema de información hermes'. Disponible en: http://www.hermes.unal.edu.co/, 2022.

[Dis23] Kasun Dissanayake. Apr 2023.

[fro] Qué es frontend y backend? Disponible en: https://platzi.com/blog/que-es-frontend-y-backend/.

[Fug03] Alfonso Fuggetta. Open source softwareâân evaluation. *Journal of Systems and Software*, 66(1):77–90, 2003.

[G+12] Oliver Gierke et al. Spring data jpa-reference documentation. *URL https://docs. spring. io/spring-data/jpa/docs/current/reference/html/.[utoljára megtekintve: 2017. 04. 21.]*, 2012.

[GG21] Jesse Griffin and Jesse Griffin. Hexagonal-driven development. *Domain-Driven Laravel: Learn to Implement Domain-Driven Design Using Laravel*, pages 521–544, 2021.

[GZ20] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[Har16] Jan L Harrington. *Relational database design and implementation*. Morgan Kaufmann, 2016.

[HT03] Alejandro Hernández Trasobares. Los sistemas de información: evolución y desarrollo, 01 2003.

[JBS15] Michael Jones, John Bradley, and Nat Sakimura. Json web token (jwt). Technical report, 2015.

[JJ18] Josh Juneau and Josh Juneau. The query api and jpql. *Java EE 8 Recipes: A Problem-Solution Approach*, pages 479–506, 2018.

[Lat23] Malhar Lathkar. Getting started with fastapi. In *High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python*, pages 29–64. Springer, 2023.

[LC09] Qing Li and Yu-Liu Chen. Entity-relationship diagram. In *Modeling and analysis of enterprise and information systems*, pages 125–139. Springer, 2009.

[LMS05] P Leach, Michael Mealling, and Rich Salz. Rfc 4122: A universally unique identifier (uuid) urn namespace, 2005.

[LZJ+21] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and software technology*, 131:106449, 2021.

[MA23] Paul Harvey Martinez Alcala. Tesis: Thesis bd design and backend. Disponible en: https://github.com/xoxtomon/tesis, 2023.

[MAMF23] Paul Harvey Martinez Alcala and Sebastian Mena Ferreira. Api-tesis-spec. Disponible en: https://app.swaggerhub.com/apis-docs/PAULMARTINEZ_1/API-TESIS-SPEC/1.3.0#/, 2023.

[Mar14] Robert C. Martin. The single responsibility principle, May 2014.

[MF23] Sebastian Mena Ferreira. Tesis-web-app-front. Disponible en: https://github.com/zGack/tesis-web-app-front, 2023.

[MFMA24] Sebastian Mena Ferreira and Paul Harvey Martinez Alcala. Video demostrativos - sistema gestor de trabajos de grado. Disponible en: https://www.youtube.com/watch?v=LyM2AR7D0E8, Feb 2024.

[MR16] Félix A. Jiménez Dany E. Díaz Sandra H. Morejón Rivera, Rogelio Cámara. Sisdam: Aplicación web para el procesamiento de datos según un diseño aumentado modificado, 2016.

[Pol13] Peter Pollock. *Web Hosting for dummies*. John Wiley & Sons, 2013.

[Pra09] Dhananjay Prasanna. *Dependency injection: design patterns using spring and guice*. Simon and Schuster, 2009.

[Pul13] Ville Pulkkinen. Continuous deployment of software. In *Proc. of the Seminar*, volume 58312107, page 46, 2013.

[Pyt21] Why Python. Python. *Python Releases for Windows*, 24, 2021.

[RBA17] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[RE03] Jorge Reyes Estévez. La ingeniería de requisitos en el desarrollo de aplicaciones informáticas, 01 2003.

[Rob11] Christina Robinson. Basic introduction into pgadmin iii and sql queries, 2011.

[Rou23] Route groups. Disponible en: https://nextjs.org/docs/app/building-your-application/routing/route-groups, Nov 2023.

[Say20] RaÅit SaygÄ±$l \pm .Manytomanyrelationshipsinrelationaldatabases, Oct$2020.

[SD21] Kishori Sharan and Adam L Davis. *Beginning Java 17 Fundamentals: Object-Oriented Programming in Java 17*. Springer, 2021.

[SS20] Edward Sciore and Edward Sciore. Jdbc. *Database Design and Implementation: Second Edition*, pages 15–47, 2020.

[sta] start.spring.io. Spring initializr.

[SWL+23] Dave Syer, Phillip Webb, Josh Long, StÃ©phane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, SÃ©bastien Deleuze, Michael Simons, and et al. Spring boot reference documentation. Disponible en: https://docs.spring.io/spring-boot/docs/current/reference/html/, Oct 2023.

[Var19] Balaji Varanasi. *Introducing Maven: A Build Tool for Today's Java Developers*. Apress, 2019.

[WFQ20] Xiangcheng Wu, Bowen Feng, and Wenmin Qi. Design and implementation of a novel student information management system, 2020.

[Wha] What is swagger? Disponible en: https://swagger.io/docs/specification/2-0/what-is-swagger/.

[Wha23] What is java? Disponible en: https://www.ibm.com/topics/java, Nov 2023.