PROYECTO DE GRADO

SISTEMA DE GESTIÓN ADMINISTRATIVA PARA MEJORAR LA EFICIENCIA EN EL REGISTRO Y PRÉSTAMO DE PARTITURAS.

PARTE 1: BACK-END

CASO: FUNDACIÓN PROMÚSICA, DIRECCIÓN DE CULTURA-ULA

Por

Br. Francisco Javier Peña Ruiz

Tutor: Dr. Rafael Rivas Estrada

Abril de 2024

RESUMEN

En el contexto actual, la eficiencia en la gestión administrativa se vuelve fundamental para optimizar procesos en diferentes áreas. La Fundación Promúsica, reconocida por su labor en la promoción y difusión de la música clásica, enfrenta desafíos en la administración manual de su amplia colección de partituras. La gestión manual ha resultado en pérdida de partituras, retrasos en préstamos y dificultades en el seguimiento detallado de su uso. Ante esta situación, se plantea el desarrollo de un sistema que optimice la gestión y préstamo de partituras, mediante una solución tecnológica que permita un seguimiento detallado y actualizado. El objetivo de este proyecto es desarrollar el backend de un sistema de gestión administrativa para mejorar la eficiencia en el registro y préstamo de partituras en la Fundación Promúsica, adscrita a la Dirección de Cultura de la Universidad de Los Andes. Se busca analizar los procesos actuales, diseñar e implementar una base de datos eficiente y una API Rest para la comunicación entre la base de datos y la interfaz de usuario. Para lograr este objetivo, se adopta la metodología de desarrollo dirigido por pruebas (TDD), que permite una implementación cuidadosa y una validación continua del sistema. Se establecen pruebas para garantizar la funcionalidad óptima del sistema, asegurando su eficiencia y fiabilidad. Se lleva a cabo una investigación para comprender las necesidades específicas de la Fundación Promúsica, con el fin de diseñar e implementar una solución que agilice los procesos de registro y préstamo de partituras. La implementación del sistema propuesto se considera una solución factible que contribuirá significativamente a la preservación y difusión del valioso repertorio musical promovido por la Fundación.

Palabras clave: Gestión administrativa, TDD, desarrollo de software, aplicación WEB, gestión de préstamos.

ÍNDICE GENERAL

ÍNDICE DE TABLAS Y FIGURAS

CAPÍTULO 1: INTRODUCCIÓN

El desarrollo de sistemas eficientes de gestión administrativa ha sido fundamental para optimizar diversos procesos en diferentes ámbitos. En paralelo, las interfaces de programación de aplicaciones (APIs) han desempeñado un papel crucial al facilitar la comunicación fluida entre sistemas informáticos, marcando un hito en el desarrollo de los sistemas administrativos. En esta línea, la Fundación Promúsica, reconocida por su labor en la promoción y difusión de la música clásica desde su establecimiento en 1998, se ha enfrentado a desafíos en la administración manual de su amplia colección de partituras.

La gestión manual de estas valiosas piezas musicales ha generado problemas como la pérdida de partituras, retrasos en los préstamos y dificultades en el seguimiento detallado de su uso. Ante esta situación, se ha identificado la necesidad apremiante de desarrollar un sistema que optimice la gestión y préstamo de las partituras en esta entidad. Este proyecto se ha planteado como una solución factible, la cual agilice los procesos de registro y préstamo de partituras, mediante el desarrollo e implementación de un sistema de gestión administrativa que permita al mismo tiempo un seguimiento detallado y actualizado de su uso.

En tal sentido, se ha optado por la metodología de desarrollo dirigido por pruebas (TDD), la cual ha permitido una implementación cuidadosa y una validación continua del sistema. A través de la metodología TDD, se han establecido pruebas para garantizar la

funcionalidad óptima del sistema, asegurando la eficiencia y la fiabilidad del proceso de desarrollo.

Para lograr este objetivo, se ha llevado a cabo una investigación para comprender las necesidades y requerimientos específicos de la Fundación Promúsica. Asimismo, se ha diseñado e implementado una base de datos eficiente y una API que facilita la comunicación entre la base de datos y la interfaz de usuario. En este contexto, se explorará cómo la implementación de un sistema de gestión administrativa eficiente puede contribuir de manera significativa a la preservación y difusión del valioso repertorio musical promovido por la Fundación Promúsica.

1.1 Antecedentes

La Fundación Promúsica, establecida en 1998, adscrita a la dirección de cultura de la Universidad de Los Andes, se ha destacado por su labor en la promoción y difusión de la música clásica, particularmente en la formación pedagógica y la promoción de la música coral en la región. A lo largo de los años, ha contribuido significativamente a la difusión de la música venezolana, latinoamericana y universal, fomentando el desarrollo humano y artístico de sus miembros, con especial énfasis en niños y jóvenes provenientes de diversos contextos socioeconómicos. A pesar de sus esfuerzos sobresalientes, la organización ha enfrentado desafíos en la administración de su extensa colección de partituras, lo que ha dado lugar a problemas como la pérdida de partituras, retrasos en los préstamos y dificultades para mantener un registro detallado del uso de las mismas.

En el contexto más amplio de la gestión administrativa, la implementación de sistemas eficientes ha demostrado ser fundamental para optimizar los procesos en diversas áreas. Estos sistemas han desempeñado un papel crucial en la mejora de la eficiencia y la

gestión efectiva de recursos y datos. La interconexión de componentes y el flujo dinámico de información son elementos esenciales para lograr esta eficacia. Las interfaces de programación de aplicaciones (APIs) han surgido como una herramienta esencial para facilitar esta comunicación fluida (Jin et al., 2018). A lo largo de la historia, las APIs han marcado un hito en la evolución de los sistemas administrativos, al permitir la integración de diferentes interfaces y la admisión de diversos tipos de programación (Mikula, 2023). Estas interfaces han demostrado su eficacia al posibilitar una conexión efectiva entre sistemas informáticos, convirtiéndose en una parte integral del desarrollo de software moderno.

Estos avances tecnológicos, específicamente en la gestión administrativa se han evidenciado en diversos estudios previos que han abordado problemas similares, algunos ejemplos de ello se describen a continuación:

1.1.1. Desarrollo de un sistema de gestión de biblioteca en la Institución Educativa Técnico Industrial Pedro A. Oñoro de Baranoa.

Este sistema fue diseñado en la plataforma Visual Studio 2017, Asp.Net y el motor de base de datos MySQL. El propósito del sistema era permitir al encargado de la biblioteca llevar un seguimiento y control de toda la información almacenada en el sistema; así como conocer de manera rápida y segura los datos de los libros, estudiantes, usuarios, datos de entrada y salida de los libros que se realizan diariamente, sin tener que recurrir a los archivos manuales obsoletos que se utilizaban en aquel momento. La propuesta buscaba solucionar un problema que se había presentado por mucho tiempo en la institución, ya que profesores y estudiantes no sabían realmente qué libros, textos y otros recursos estaban disponibles en la biblioteca escolar (Ruiz, 2020).

1.1.2. Sistema de gestión y digitalización bibliotecaria. Caso: carrera de odontología U.P.E.A.

La investigación describe la propuesta de un sistema de gestión y digitalización bibliotecaria para mejorar el servicio al personal y usuarios de la biblioteca de la carrera de odontología, que tenía un control semiautomático y limitaciones en el acceso a la información y préstamo de materiales bibliográficos. El sistema se desarrolló utilizando la metodología UWE y tecnología Web, con PHP como lenguaje de programación y MariaDB como gestor de base de datos. Además, se realizó una evaluación de la calidad de producto de software con ISO/IEC 9126 y una estimación de costo de software con el modelo COCOMO II (Gutiérrez ,2020).

1.1.3. Diseño e Implementación de una aplicación para la gestión de partituras.

Esta investigación describe un trabajo que se enfocó en el desarrollo e implementación de una aplicación web para la gestión de partituras, con el objetivo de almacenar las partituras subidas por los usuarios y desarrollar una capa social y un motor de búsqueda. La aplicación cuenta con una arquitectura en capas, con la capa del servidor implementada a través del framework Django y la capa del cliente implementada a través de la biblioteca React. El proyecto se desarrolló utilizando la metodología ágil Scrum (Pérez, 2022).

Estos antecedentes enfatizan la importancia de implementar sistemas de gestión avanzados en diferentes contextos para optimizar los procesos y mejorar la eficiencia en diversas áreas de estudio y trabajo.

1.2. Planteamiento del problema

La Fundación Promúsica, dedicada a la promoción y difusión de la música clásica, se ha enfrentado a desafíos significativos en la gestión manual de su extensa colección de partituras. Este proceso tradicional ha llevado a problemas tales como pérdida de partituras, demoras en los préstamos y dificultades para mantener un registro detallado del uso de las mismas. La ausencia de un sistema de gestión administrativa eficiente ha agravado estas dificultades, resultando en una base de datos desactualizada que no permite un seguimiento adecuado de la disponibilidad de las partituras y su historial de préstamos. En este contexto, surge la necesidad de desarrollar un sistema que agilice el proceso de registro y préstamo de partituras, al tiempo que facilite un seguimiento detallado y actualizado del uso de las mismas en la Fundación.

1.3. Justificación

Considerando los desafíos actuales en la gestión manual de la extensa colección de partituras de la Fundación Promúsica, es evidente la necesidad urgente de mejorar la eficiencia en el registro y préstamo de estas valiosas piezas musicales. La implementación de un sistema de gestión administrativa eficiente se vuelve crucial para optimizar los procesos de mantenimiento y préstamo de las partituras, evitando extravíos, retrasos y deficiencias en el registro del uso de las mismas.

Este proyecto se plantea como un producto factible y se justifica en su propósito de desarrollar una solución tecnológica que permita una gestión más efectiva y automatizada de las partituras en la Fundación Promúsica. La implementación de este sistema no solo contribuirá a agilizar los procesos internos, sino que también garantizará un acceso más

rápido y seguro a las partituras, mejorando así la experiencia de los usuarios y optimizando la preservación y difusión del valioso repertorio musical en la región.

1.4. Objetivos

1.4.1. Objetivo general

Desarrollar el back-end de un sistema de gestión administrativa para mejorar la eficiencia en el registro y préstamo de partituras en la fundación Promúsica, adscrita a la dirección de cultura de la Universidad de Los Andes.

1.4.2. Objetivos específicos

Para cumplir con el objetivo general, se plantean los siguientes objetivos específicos:

- Analizar los procesos actuales de registro y préstamo de partituras en la Fundación Promúsica para identificar las necesidades y requerimientos del sistema de gestión administrativa.
- Diseñar e implementar una base de datos que permita almacenar la información de los usuarios, partituras, y préstamos de manera eficiente, organizada y segura.
- Desarrollar una API Rest que permita la comunicación entre la base de datos y una interfaz de usuario.
- Implementar el sistema de gestión administrativa en la fundación Promúsica y capacitar al personal encargado en su uso.

1.5. Metodología

Para el desarrollo del Producto Factible orientado a optimizar la gestión de las partituras en la Fundación Promúsica, se ha adoptado la Metodología de Desarrollo Dirigido por Prueba (TDD, por sus siglas en inglés: Test Driven Development). Esta metodología se ha elegido por su enfoque proactivo y su capacidad para garantizar la calidad del sistema a lo largo de todo el proceso de desarrollo.

Herranz (2023), describe el "Test Driven Development" (TDD) como una metodología de desarrollo de software que se basa en escribir primero las pruebas, después escribir el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito.

En este orden de ideas, el TDD se fundamenta en un enfoque iterativo que se inicia con la creación de pruebas automatizadas antes de implementar cualquier funcionalidad. Estas pruebas se diseñarán para validar la funcionalidad y asegurar que el sistema cumpla con los requisitos definidos previamente. Posteriormente, se desarrollará la funcionalidad mínima necesaria para que la prueba pase satisfactoriamente. Este proceso, conocido como ciclo TDD, se repetirá de forma iterativa, integrando nuevas funcionalidades y ampliando las pruebas existentes a lo largo de las distintas fases de desarrollo, este ciclo se puede observar en la figura 1.

Se prestará especial atención a la escritura de pruebas claras y específicas que cubran todos los casos posibles de uso y que aborden los potenciales problemas que podrían surgir durante la implementación. Las pruebas se automatizarán para garantizar una verificación continua y exhaustiva del sistema en cada etapa del desarrollo.

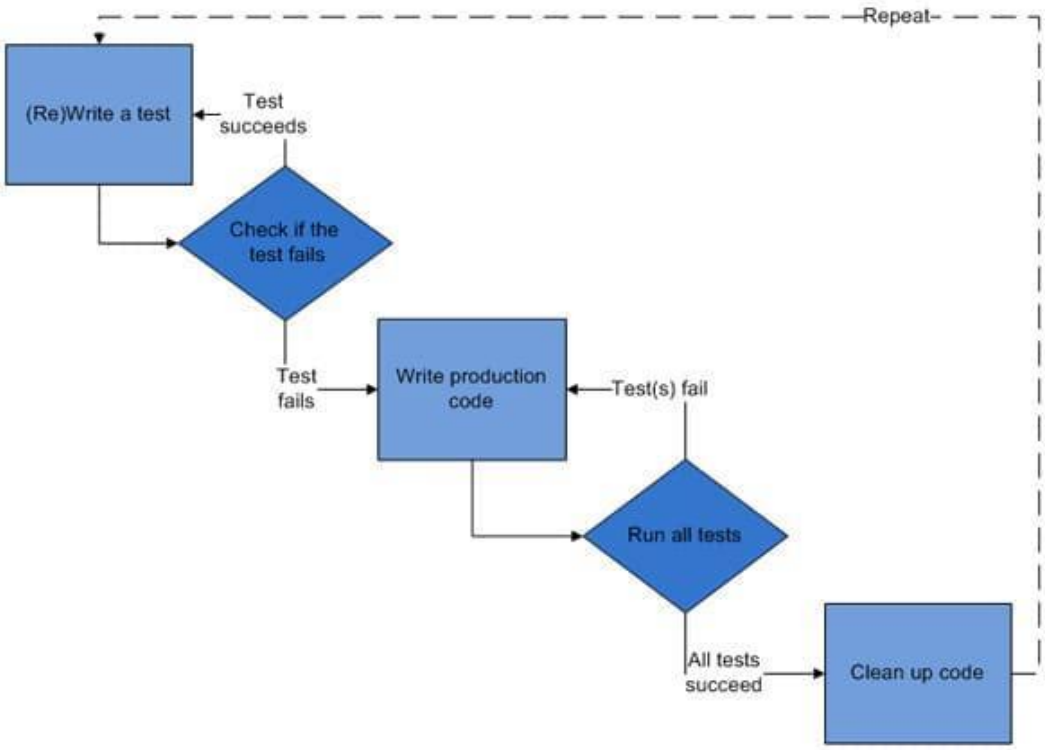


Figura 1. Metodología de desarrollo TDD (Herranz, 2023)

1.5.1. Algoritmo TDD

Según Blé (2010), el algoritmo TDD sólo tiene tres pasos fundamentales:

- Escribir la especificación del requisito (el ejemplo, el test).
- Implementar el código según dicho ejemplo.
- Refactorizar para eliminar duplicidad y hacer mejoras.

A continuación, se presentan con más detalle cada uno de estos pasos:

Paso 1 - Escribir la especificación del requisito primero: Antes de implementar cualquier código, se redactarán pruebas claras y específicas que describen los requisitos del sistema.

Las pruebas se diseñarán para abordar casos de uso o historias de usuario y escenarios específicos, lo que permitirá clarificar el comportamiento esperado del sistema.

En otras palabras, el primer paso es **"escribir una prueba que falle"**. Esta prueba debe especificar el comportamiento esperado del software. Para escribir una prueba, debemos identificar el comportamiento que queremos probar. Una vez que hayamos identificado el comportamiento, podemos escribir una prueba que verifique ese comportamiento. La prueba debe ser lo más simple posible. No se debe hacer nada más que verificar el comportamiento que queremos probar.

Paso 2 - Implementar el código que haga funcionar la especificación: Se procederá a desarrollar el código mínimo necesario para que las pruebas definidas anteriormente se cumplan. Durante esta etapa, se evitará la implementación de cualquier código adicional que no sea estrictamente necesario para satisfacer la especificación actual. Se prestará especial atención a la eficiencia y a la concentración en la implementación precisa de la funcionalidad requerida.

Básicamente, el siguiente paso es **"escribir el código necesario para que la prueba pase"**. Este código debe ser lo más simple posible. Para escribir el código, se debe pensar en cómo implementar el comportamiento que queremos probar. Una vez que se tenga una idea de cómo implementar el comportamiento, se procede a escribir el código necesario. No se debe hacer nada más que implementar el comportamiento que queremos probar.

Paso 3 - Refactorización: A continuación, se llevará a cabo una revisión exhaustiva del código para identificar y eliminar cualquier duplicidad. Además, se verificará que el código cumpla con los principios de diseño pertinentes. Se realizarán ajustes y optimizaciones necesarios para mejorar la claridad y la mantenibilidad del código.

Dicho en otras palabras, el último paso es **"refactorizar el código escrito"**. La refactorización se puede realizar de muchas maneras diferentes. Una forma de refactorizar el código es eliminar el código duplicado o simplificar la estructura y lectura del mismo.

Otra forma de enumerar las tres fases del ciclo es:

- Rojo
- Verde
- Refactorizar

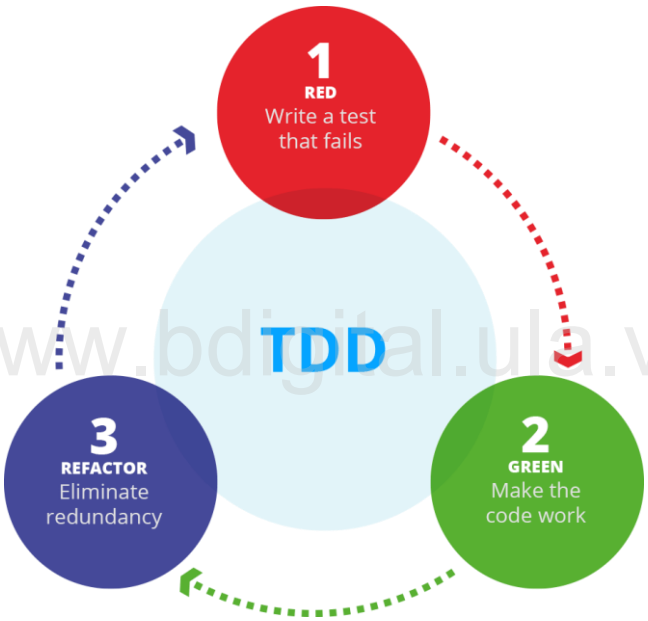


Figura 2. Ciclo TDD (Fuentes, 2021)

Es una descripción metafórica (Figura 2), ya que los frameworks de tests suelen colorear en rojo aquellas especificaciones que no se cumplen y en verde las que lo hacen. Así, cuando escribimos el test, el primer color es rojo porque todavía no existe código que implemente el requisito. Una vez implementado, se pasa a verde (Blé, 2010).

1.5.2. TDD como metodología ágil

La integración del Desarrollo Dirigido por Pruebas (TDD) como metodología ágil, se traduce en un proceso estructurado que aborda la creación de software de manera eficiente (Herranz, 2023). Este enfoque se manifiesta en una serie de pasos coherentes, delineados de la siguiente manera:

1. El cliente escribe su historia de usuario.
2. En colaboración con el cliente, se definen los criterios de aceptación asociados con la historia de usuario, seccionándolos en elementos más simples para facilitar el proceso.
3. Se escoge el criterio de aceptación más simple y se traduce en una prueba.
4. Se comprueba que esta prueba falla.
5. Se escribe el código que hace pasar la prueba.
6. Se ejecutan todas las pruebas automatizadas.
7. Se refactoriza y se limpia el código.
8. Se lleva a cabo una revisión exhaustiva a través de la ejecución de todas las pruebas automatizadas, garantizando que la funcionalidad del sistema se mantenga intacta y sin errores.
9. Volver al punto 3 con los criterios de aceptación que faltan y se repite el ciclo una y otra vez hasta completar la aplicación.

1.6. Alcances y Limitaciones

1.6.1. Alcances:

- El estudio se centrará en el desarrollo de un sistema de gestión administrativa para la fundación Promúsica.

- El sistema permitirá la optimización del proceso de registro y préstamo de partituras, así como el registro detallado y actualizado del uso de las mismas.
- Se identificarán las necesidades y requerimientos de la fundación Promúsica para el desarrollo del sistema.
- Se diseñará y desarrollará una base de datos que permita el registro y préstamo de partituras de manera eficiente y organizada.
- Se desarrollará una API Rest, la cual podrá ser consumida por una interfaz de usuario.
- Se implementará el sistema de gestión administrativa en la fundación Promúsica y se capacitará al personal encargado en su uso.

1.6.2. Limitaciones:

- El estudio se enfoca en el desarrollo del sistema de gestión administrativa para la fundación Promúsica y no se considerarán otras organizaciones dedicadas a la promoción y difusión de la música.
- El sistema a desarrollar estará enfocado en la gestión de partituras y no se incluirán otros elementos de la organización, como la gestión de recursos humanos o la gestión financiera.
- El estudio no considerará la infraestructura tecnológica disponible en la fundación Promúsica y se asumirá que se cuenta con los recursos necesarios para la implementación del sistema.

CAPÍTULO 2: MARCO TEÓRICO

En este capítulo, se sientan las bases teóricas para el desarrollo del proyecto, presentando las herramientas de desarrollo que se utilizarán. Aquí se establece el fundamento conceptual necesario para comprender y ejecutar de manera efectiva la investigación y el desarrollo propuestos.

2.1. Sistema de Información

O'Brien y Marakas (2006), en su libro Sistemas de información gerencial, definen un sistema de información como un conjunto de componentes interrelacionados que recopilan, procesan, almacenan y distribuyen información con el fin de apoyar la toma de decisiones, la coordinación y el control en una organización. En esta definición, se destaca la importancia de los componentes interrelacionados y la función del sistema de información para apoyar la gestión de la organización.

La definición de sistema de información de estos autores, es relevante en este contexto, debido a que, un sistema de gestión administrativa para la gestión de partituras es, en esencia, un sistema de información que recopila, procesa, almacena y distribuye información relevante para la toma de decisiones y el control dentro de la organización.

2.2. Tecnología de la información (TI)

Langer (2018), define la tecnología de la información como la combinación de hardware, software y servicios que se utilizan para gestionar y procesar la información dentro de una organización.

La definición de Langer destaca la importancia de la tecnología de la información en la administración y el procesamiento de la información dentro de una organización. La tecnología de la información es fundamental para la construcción y el funcionamiento de un sistema porque ofrece las herramientas necesarias para la gestión de datos y el intercambio de información entre los numerosos componentes del sistema.

Los sistemas pueden procesar, almacenar y transportar información de manera confiable y efectiva gracias al hardware, software y servicios de tecnología de la información, que respaldan la toma de decisiones corporativas y la formulación de estrategias. La tecnología de la información es crucial para la seguridad del sistema porque ofrece instrumentos para la protección de la información contra el acceso no autorizado y la recuperación de la información en caso de falla del sistema.

2.3. Sistema de gestión administrativa

Un sistema de gestión administrativa se refiere al conjunto de herramientas tecnológicas que utilizan las empresas para la automatización y mejora de sus procesos administrativos, con el objetivo de incrementar su eficiencia y productividad. (Overview of the Administrative Management Systems (AMS), 2016).

Debido a que permitiría la automatización y optimización de los procesos administrativos asociados a la gestión de partituras de la fundación Promúsica, la creación de

un sistema para aumentar la eficiencia en el registro y préstamo de partituras es crucial porque conduciría a un mayor nivel de productividad en la gestión de estos recursos.

Herramientas como bases de datos de clientes, sistemas de monitoreo de préstamos y devoluciones, software de gestión de inventario y otros pueden ser parte de un sistema de gestión administrativa. Al automatizar estas operaciones, sería posible acelerar la gestión de recursos, disminuir la cantidad de tiempo requerido para los procesos de registro y préstamo, y eliminar por completo los errores humanos.

Un sistema de gestión administrativa también permitiría mantener un registro preciso y actualizado del inventario de partituras, lo que ayudaría en la toma de decisiones y la planificación a largo plazo. Además, podría ayudar a maximizar la capacidad de almacenamiento de las partituras, así como a mejorar la seguridad y la administración del acceso a las partituras.

2.4. Sistemas integrados de gestión bibliotecaria (SIGB)

La gestión eficiente de recursos y servicios es fundamental para el éxito de cualquier organización, incluyendo las bibliotecas. Los sistemas de gestión de bibliotecas (SIGB) son herramientas tecnológicas que permiten gestionar de manera efectiva los recursos y servicios de una biblioteca.

Gavilán (2008), define los sistemas integrados de gestión bibliotecaria (SIGB) como herramientas tecnológicas que permiten la gestión automatizada de los recursos y servicios de una biblioteca, incluyendo la adquisición, catalogación, circulación, préstamo y control de inventario de los materiales bibliográficos.

Este tipo de programas surgen como un intento de conseguir que las unidades de información se conviertan en centros más eficaces, con capacidad de poder gestionar de

manera más eficiente sus recursos y la posibilidad de comunicación más viable con los usuarios.

Un SIGB, integra en un solo programa informático un conjunto de aplicaciones específicas que se denominan módulos, pensados para la facilitación de las tareas específicas de este, las cuales están directamente relacionadas unas con otras. Toda la información reunida, se almacena en una misma base de datos que permite el mejor intercambio de la información y el aprovechamiento de los recursos con el menor esfuerzo posible.

2.5. Biblioteca musical

Smiraglia (2001), propone la siguiente definición para biblioteca musical: es una biblioteca especializada en la adquisición, procesamiento, almacenamiento y acceso de materiales relacionados con la música en cualquier formato.

La definición implica que una biblioteca musical tiene un enfoque específico en la música y sus materiales relacionados, lo que incluye partituras en diferentes formatos (por ejemplo, partituras impresas, partituras digitales, archivos de audio, etc.). Al tener una comprensión clara de las necesidades de una biblioteca musical, se puede diseñar un sistema de gestión de partituras para la fundación Promúsica que incluya las características y funcionalidades específicas necesarias para satisfacer sus necesidades.

Un sistema de gestión de partituras debe ser capaz de adquirir, procesar, almacenar y permitir el acceso a las partituras en diferentes formatos, ya sea en forma impresa o digital. Debe contar con herramientas de catalogación, metadatos y búsqueda que permitan a los usuarios buscar y encontrar las partituras de manera eficiente y precisa. También debe tener herramientas de visualización y reproducción de partituras digitales para permitir a los usuarios ver y escuchar las partituras de manera clara y precisa.

2.6. Plataforma digital

Ross et al. (2019), definen la plataforma digital como un conjunto de tecnologías, estándares y acuerdos comerciales que, cuando se combinan con el contenido digital, proporcionan una experiencia integral y personalizada para los clientes, usuarios y empleados. Estos autores, argumentan que una plataforma digital exitosa debe ofrecer una solución integral a las necesidades de los usuarios, que incluye la entrega de contenido y la capacidad de interactuar y realizar transacciones en línea. También destaca la importancia de la colaboración entre las empresas y los proveedores de tecnología para construir y mantener una plataforma digital efectiva.

2.7. Gestión de procesos de negocio

La gestión de procesos de negocio (BPM, por las iniciales de la expresión en inglés Business Process Management) constituye uno de los tópicos más pronunciados cuando se abordan las Tecnologías de Información (TI) aplicadas al entorno empresarial. BPM se considera un enfoque multidisciplinario ya que presenta conectores con elementos empresariales y tecnológicos altamente relacionados entre sí. Bajo el paradigma BPM estos procesos se conciben en un ciclo donde son modelados electrónicamente y pueden ser analizados y mejorados como resultado de varias instancias de procesos ejecutados. Este ciclo BPM se sustenta por los sistemas BPM (BPMS). Las BPMS ofrecen componentes de software integrados en un entorno único que se pueden clasificar en: herramientas de modelado, herramientas de simulación, motores de ejecución, integración de aplicaciones, portales web y monitorización (Cruz et al., 2020).

2.8. Ingeniería de requisitos

La ingeniería de requisitos es un enfoque sistemático a través del cual el ingeniero de software recopila requisitos de diferentes fuentes y los implementa en los procesos de desarrollo de software. Las actividades de ingeniería de requisitos cubren todo el ciclo de vida del desarrollo de sistemas y software. El proceso de ingeniería de requisitos es un proceso iterativo que también indica que la gestión de requisitos se entiende como un aspecto del proceso de ingeniería de requisitos (An Effective Requirement Engineering Process Model for Software Development and Requirements Management, 2010).

Es común clasificar los requisitos en funcionales y no funcionales.

2.8.1. Requisitos funcionales

Los requisitos funcionales son los que definen las funciones que el sistema será capaz de realizar, describen las transformaciones que el sistema realiza sobre las entradas para producir salidas (Alarcón, 2006). Por consiguiente, estos requisitos establecen la base sobre la cual se construirá el sistema. Los requisitos funcionales especifican las funcionalidades, tareas y procesos que el software debe cumplir, lo que significa que son fundamentales para el diseño, implementación, pruebas y validación del software.

Por tanto, los requisitos funcionales son esenciales para garantizar que el software cumpla con las necesidades y expectativas de los usuarios finales. Si los requisitos funcionales no se definen adecuadamente, es probable que el software no cumpla con los objetivos previstos, lo que puede llevar a la insatisfacción del usuario, a la necesidad de realizar cambios costosos y a la posible pérdida de oportunidades de negocio. En

consecuencia, es fundamental que los requisitos funcionales sean claros, precisos y completos para que el software pueda ser desarrollado de manera efectiva.

2.8.2. Requisitos no funcionales

Los requisitos no funcionales tienen que ver con características que de una u otra forma puedan limitar el sistema, por ejemplo, el rendimiento (en tiempo y espacio), interfaces de usuario, fiabilidad (robustez del sistema, disponibilidad de equipo), mantenimiento, seguridad, portabilidad, estándares, etc. Son restricciones de los servicios o funciones ofrecidos por el sistema (Alarcón, 2006).

Es por ello, que los requisitos no funcionales proporcionan limitaciones y especificaciones importantes que se deben cumplir para garantizar que el sistema cumpla con las expectativas de los usuarios. Si estos requisitos no se cumplen, el sistema puede no ser efectivo, eficiente o seguro para su uso, por lo que es fundamental tenerlos en cuenta desde el inicio del proyecto y considerarlos durante todo el ciclo de vida del software.

2.9. Arquitectura de software

La arquitectura de software es la estructura de un producto de software. Esto incluye elementos, las propiedades visibles externamente de los elementos y las relaciones entre los elementos (Bass et al., 2012).

En base a lo anterior, Lilienthal (2019) dice que esta definición habla deliberadamente de elementos y relaciones en términos muy generales. Estos dos materiales básicos se pueden utilizar para describir una amplia variedad de vistas arquitectónicas. La vista estática (módulo) contiene los siguientes elementos: clases, paquetes, espacios de nombres, directorios y proyectos; en otras palabras, todos los contenedores que puede usar para programar código en ese lenguaje de programación en particular. En la vista de distribución,

se pueden encontrar los siguientes elementos: archivos (JAR, WAR, ensamblados), computadoras, procesos, protocolos y canales de comunicación, etc. En la vista dinámica (tiempo de ejecución) estamos interesados en los objetos de tiempo de ejecución y sus interacciones.

2.9.1. Arquitectura cliente-servidor

La arquitectura cliente-servidor de una red informática es aquella en la que muchos clientes (procesadores remotos) solicitan y reciben servicios de un servidor centralizado (computadora host). Las computadoras cliente proporcionan una interfaz para permitir que un usuario de computadora solicite servicios del servidor y muestre los resultados que devuelve el servidor. Los servidores esperan que lleguen las solicitudes de los clientes y luego las responden (The Editors of Encyclopaedia Britannica, 2023).

2.9.1.1. Servidor

Un servidor es un sistema que contiene datos o proporciona recursos a los que deben acceder otros sistemas de la red. Los tipos de servidor comunes son servidores de archivos que almacenan archivos, servidores de nombres que almacenan nombres y direcciones, servidores de aplicaciones que almacenan programas y aplicaciones y servidores de impresión que planifican y dirigen los trabajos de impresión al destino (IBM Documentation, 2021).

2.9.1.2. Cliente

Un cliente es un sistema que solicita servicios o datos de un servidor. Un cliente puede solicitar código de programa actualizado o el uso de aplicaciones de un servidor de código. Para obtener un nombre o una dirección, un cliente se pone en contacto con un

servidor de nombres. Un cliente también puede solicitar archivos y datos para la entrada de datos, las consultas o la actualización de registros de un servidor de archivos (IBM Documentation, 2021).

2.9.2. MVC (Modelo-Vista-Controlador)

Es un patrón en el diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Enfatiza una separación entre la lógica de negocios y su visualización. Esta "separación de preocupaciones" proporciona una mejor división del trabajo y una mejora de mantenimiento. Algunos otros patrones de diseño se basan en MVC, como MVVM (Modelo-Vista-modelo de vista), MVP (Modelo-Vista-Presentador) y MVW (Modelo-Vista-Whatever) (MVC - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN, 2022).

2.10. Back-end

El back-end, también conocido como el lado del servidor, es la parte del software que se encarga de procesar y almacenar los datos y de gestionar la lógica de negocio. Es la columna vertebral de cualquier aplicación web o móvil, ya que es responsable de proporcionar la funcionalidad y los servicios necesarios para que la aplicación pueda funcionar correctamente.

En relación al desarrollo de software, el back-end es crucial para garantizar la seguridad, escalabilidad y rendimiento de una aplicación. Un back-end bien diseñado y desarrollado puede mejorar significativamente la experiencia del usuario y la eficiencia de la aplicación. (Eseme, 2021)

2.11. ¿Qué es un Framework o marco de trabajo de backend?

Un Framework o marco de trabajo de software es una base donde los desarrolladores pueden crear aplicaciones de una manera más rápida y estandarizada. El siguiente ejemplo, disponible en Stack Overflow (s. f.), es bastante útil para comprender el concepto de marco de trabajo.

> "... Si te dijera que cortes un trozo de papel con unas dimensiones de 5 m por 5 m, seguramente lo harías. Pero supón que te pido que cortes 1000 hojas de papel de las mismas dimensiones. En este caso, no harás la medición 1000 veces; obviamente, harías un marco de trabajo de 5 m por 5 m, y luego con su ayuda podrías cortar 1000 hojas de papel en menos tiempo. Entonces, lo que hiciste fue crear un marco de trabajo que haría un tipo específico de tarea. En lugar de realizar el mismo tipo de tarea una y otra vez para el mismo tipo de aplicaciones, creas un marco de trabajo que tiene todas esas facilidades juntas en un paquete agradable, proporcionando de esta forma la abstracción para tu aplicación y, lo que es más importante, muchas aplicaciones".

En este orden de ideas, un marco de trabajo de software se puede considerar como una plantilla predefinida que permite a los desarrolladores construir aplicaciones de manera más rápida y eficiente. De manera similar, un marco de trabajo de backend proporciona un conjunto de herramientas y funcionalidades integradas que simplifican el desarrollo de aplicaciones, permitiendo a los desarrolladores enfocarse en la lógica específica de su aplicación en lugar de reinventar constantemente la rueda. Este enfoque estandarizado y eficiente no sólo acelera el proceso de desarrollo, sino que también fomenta una mayor consistencia y confiabilidad en las aplicaciones resultantes (Clark, 2021).

2.11.1. Evaluación de Frameworks para Desarrollo de Backend

Los marcos de trabajo de backend son esenciales para el desarrollo de aplicaciones para innumerables compañías en todo el mundo en la actualidad. Encontrar el marco de trabajo de backend adecuado puede ser crucial para que los desarrolladores garanticen un rendimiento y escalabilidad óptimos. Con tantas opciones disponibles hoy en día, elegir las más relevantes puede ser complicado.

En la presente revisión, se evalúan algunos de los frameworks más destacados, como Django, Laravel, Ruby on Rails, Spring Boot y Express, junto con una breve consideración de otros frameworks relevantes como ASP.NET Core, Node.js, Gin, Kotlin, Flask, CakePHP y Yii.

Django (Python): Conocido por su facilidad de uso y capacidad para crear aplicaciones web complejas, Django se destaca por su soporte para la arquitectura Modelo-Vista-Controlador (MVC), generación de código, enrutamiento de URL, validación de datos y seguridad.

Laravel (PHP): Reconocido por su elegancia y simplicidad, Laravel presenta un sólido soporte para la arquitectura MVC, generación de código, enrutamiento de URL, validación de datos y seguridad. Sus características clave incluyen una autenticación simplificada, una API flexible, soporte para varios backends de caché, registros y pruebas sencillas.

Ruby on Rails (Ruby): Destacado por su velocidad, escalabilidad y facilidad de aprendizaje, Ruby on Rails ofrece soporte para MVC, generación de código, enrutamiento de URL, validación de datos y seguridad.

Spring Boot (Java): Reconocido por su simplicidad y eficiencia, Spring Boot brinda soporte para MVC, generación de código, enrutamiento de URL, validación de datos y seguridad. Su flexibilidad y soporte para una amplia gama de tecnologías lo convierten en una opción atractiva para el desarrollo de backend en Java.

Express (JavaScript): Reconocido por su flexibilidad y rendimiento, Express ofrece soporte para MVC, generación de código, enrutamiento de URL, validación de datos y seguridad. Su capacidad para trabajar con una amplia gama de tecnologías lo hace popular entre los desarrolladores que buscan un rendimiento óptimo.

2.11.1.1. Framework Laravel

Laravel se destaca como un robusto marco de trabajo web PHP de código abierto, diseñado para el desarrollo de aplicaciones web basadas en Symfony que siguen la arquitectura Modelo-Vista-Controlador (MVC). Su versatilidad y diversas funcionalidades hacen de Laravel una opción atractiva para proyectos de desarrollo de backend. Algunas de las ventajas clave de Laravel incluye su sistema de autenticación simplificado, API flexible y versátil, soporte para varios backends de caché, registro y manejo de errores, funcionalidades de pruebas sencillas y características de seguridad avanzadas (*Documentación Laravel en español - El framework de PHP para artesanos de la WEB*, s. f.).

El marco de trabajo Laravel también presenta una serie de características notables, incluyendo un potente motor de plantillas que permite una generación de diseños eficiente, un sólido soporte para la arquitectura Modelo-Vista-Controlador (MVC) que facilita una separación efectiva de la lógica de presentación y de negocios, un Mapeo Objeto-Relacional Elocuente (ORM) para la construcción de consultas de bases de datos utilizando la sintaxis de

PHP, y una sólida seguridad que incluye modalidades de contraseña con hash y sal, entre otras características.

Estas características y ventajas demuestran la idoneidad de Laravel para el proyecto en cuestión, ya que su flexibilidad, facilidad de uso y potentes funcionalidades se alinean perfectamente con los objetivos del proyecto y la metodología de desarrollo prevista. Con su sólido soporte para MVC, ORM eficiente y características de seguridad avanzadas, Laravel es una opción destacada para garantizar un desarrollo de backend eficiente y seguro.

2.12. Bases de datos

Las bases de datos son simplemente una forma estructurada y sistemática de almacenar, acceder, analizar, transformar, actualizar y mover información (a otras bases de datos) (Dowsett, 2022).

Así pues, la relevancia de las bases de datos en el desarrollo de sistemas informáticos es fundamental, ya que la mayoría de los sistemas informáticos dependen de ellas para almacenar y administrar grandes cantidades de información. Las bases de datos, permiten a los desarrolladores de software diseñar sistemas más robustos, escalables y eficientes, al mismo tiempo que proporcionan una estructura organizada para acceder y administrar la información. Además, las bases de datos son esenciales para la toma de decisiones y el análisis de datos en los sistemas informáticos, lo que los hace imprescindibles en el mundo de la tecnología de la información.

2.12.1. Bases de datos relacionales

Una base de datos relacional organiza los datos en filas y columnas, que en conjunto forman una tabla. Los datos normalmente se estructuran en varias tablas, que se pueden unir a través de una clave principal o una clave externa. Estos identificadores únicos demuestran las

diferentes relaciones que existen entre las tablas, y estas relaciones generalmente se ilustran a través de diferentes tipos de modelos de datos. Los analistas utilizan consultas SQL para combinar diferentes puntos de datos y resumir el rendimiento empresarial, lo que permite a las organizaciones obtener información, optimizar los flujos de trabajo e identificar nuevas oportunidades (IBM, 2021).

2.13. ORM

El Mapeador Objeto-Relacional (ORM por sus siglas en inglés: Object-Relational Mapping) constituye un componente crucial en el desarrollo de aplicaciones, al permitir a los desarrolladores interactuar con bases de datos relacionales utilizando objetos de programación. Al esconder gran parte de la complejidad de las consultas SQL, los ORM simplifican el proceso de desarrollo y ayudan a mejorar la eficiencia y la productividad de los equipos de desarrollo. Al hacer que la interacción con la base de datos sea más intuitiva y menos propensa a errores, los ORM fomentan una programación más estructurada y organizada, lo que a su vez conduce a la creación de aplicaciones más estables y mantenibles (Deloitte-Spain, 2023).

En este contexto, la elección de Eloquent como ORM preferido para el proyecto se fundamenta en su sólido soporte y su destacada capacidad para simplificar el proceso de desarrollo en el entorno específico de PHP, garantizando así la eficacia y la eficiencia en la implementación de la solución propuesta. Laravel, el framework en el que se basa el proyecto, incluye Eloquent, el cual se encarga de crear para cada tabla en la base de datos su correspondiente "Modelo", lo que permite realizar operaciones como recuperar, insertar, actualizar y eliminar registros de manera intuitiva y eficiente. Gracias a esta integración estrecha y la funcionalidad avanzada proporcionada por Eloquent, el desarrollo de la

aplicación se lleva a cabo de manera ágil y efectiva, permitiendo enfocarse en la lógica de la aplicación en lugar de las complejidades de la base de datos (Documentación Laravel en español, s. f.-b).

2.14. API

Según Jin et al. (2018), una API (interfaz de programación de aplicaciones) se define como "una interfaz entre dos sistemas de software que les permite comunicarse entre sí". Ellos continúan explicando que las API permiten que diferentes sistemas de software interactúen e intercambien información entre sí, sin necesidad de que los sistemas subyacentes comprendan los detalles de implementación de los demás. Los autores enfatizan la importancia de diseñar API teniendo en cuenta las necesidades de los desarrolladores y brindan orientación práctica para crear API que sean fáciles de usar y comprender.

2.14.1. API Rest

Jin et al. (2018), definen una API RESTful (Representational State Transfer) como una API que se adhiere a un conjunto de restricciones arquitectónicas, incluido el uso de métodos HTTP (como GET, POST, PUT y DELETE) para realizar operaciones en recursos y usar URI (identificadores uniformes de recursos) para identificar recursos. Además, las API RESTful permiten a los desarrolladores crear APIs escalables, flexibles y mantenibles, y se utilizan ampliamente en el desarrollo web moderno.

2.15. Seguridad de los datos

La seguridad de los datos es un aspecto crítico en el desarrollo de aplicaciones y sistemas informáticos. Se refiere a la implementación de prácticas y medidas diseñadas para garantizar la confidencialidad, integridad y disponibilidad de la información. En un mundo cada vez más digital y conectado, la seguridad de los datos se ha vuelto esencial para proteger la privacidad de los usuarios y prevenir el acceso no autorizado a la información sensible.

Las aplicaciones web se han convertido en la columna vertebral del entorno digital. Su preferencia por el protocolo seguro HTTPS subraya la importancia de la seguridad en la transmisión de datos. En este contexto, el desarrollo de aplicaciones web se apoya en enfoques avanzados de intercambio de datos en línea, destacando REST (Representational State Transfer) por su flexibilidad. Este método, constituye la base para la evolución constante de aplicaciones que buscan optimizar el rendimiento y la experiencia del usuario (Cevallos, 2022).

2.15.1. Web Tokens

Los tokens web (Web Tokens) son una forma popular de garantizar la seguridad en las comunicaciones entre diferentes partes de una aplicación web. Estos son pequeños paquetes de información que contienen datos específicos y se utilizan para la autenticación y autorización. En un contexto de desarrollo web, los tokens web son comúnmente utilizados para validar la identidad de un usuario después de que este ha iniciado sesión. Los dos tipos principales de tokens web son JWT (JSON Web Tokens) y OAuth y otros mecanismos similares (Auth, s. f.).

2.15.1. Cookies

Las cookies son esenciales en las solicitudes web para el intercambio de información entre el servidor y el cliente, brindando la capacidad de almacenar datos de manera persistente durante un período. Su ubicación en la sección de cookies o en las herramientas de desarrollo de los navegadores permite su visualización. Ampliamente utilizado en las políticas de cookies, Google (s. f.) destaca su empleo para diversas funciones como preferencias de usuario, seguridad, autenticación y personalización de anuncios según las preferencias configuradas.

2.15.3. Sanctum

Laravel Sanctum ofrece un sistema de autenticación ligero diseñado para SPAs, aplicaciones móviles y APIs sencillas basadas en tokens. La flexibilidad de Sanctum permite a cada usuario generar múltiples tokens API, cada uno con habilidades y ámbitos específicos que determinan las acciones permitidas (Documentación Laravel en español - El framework de PHP para artesanos de la WEB, s. f.).

Funcionamiento de Laravel Sanctum:

- **Tokens de API:** Sanctum facilita la emisión de tokens de API sin la complejidad de OAuth. Inspirado en aplicaciones como GitHub, permite a los usuarios generar y gestionar tokens API desde la configuración de cuenta. Estos tokens, con una larga duración, pueden ser revocados manualmente por el usuario. La autenticación se realiza a través de la cabecera Authorization con un token API válido almacenado en una tabla de base de datos.

- **Autenticación de SPA:** Sanctum aborda la autenticación de aplicaciones de página única (SPA) que se comunican con una API Laravel. Utiliza servicios de autenticación de sesión basados en cookies incorporados en Laravel, sin necesidad de tokens. Aprovechando la guarda de autenticación web de Laravel, ofrece beneficios de protección CSRF, autenticación de sesión y seguridad contra fuga de credenciales por XSS.
- **Uso de Cookies:** Sanctum autentica mediante cookies sólo cuando la petición proviene del frontend SPA. Al examinar una solicitud HTTP entrante, verifica la presencia de una cookie de autenticación. Si no está presente, examina la cabecera Authorization en busca de un token API válido.

2.16. Protocolos HTTP y HTTPS

Los protocolos HTTP (Hypertext Transfer Protocol) y HTTPS (Hypertext Transfer Protocol Secure) son fundamentales para la comunicación en la World Wide Web. Estos protocolos definen cómo los mensajes se formatean y transmiten, permitiendo la transferencia de datos entre clientes y servidores de manera efectiva y segura.

2.16.1. HTTP (Hypertext Transfer Protocol)

El Protocolo de transferencia de hipertexto (HTTP: Hypertext Transfer Protocol) es el protocolo base para la comunicación en la web. Es un protocolo sin estado, lo que significa que cada solicitud entre un cliente y un servidor se trata de manera independiente, sin que el servidor recuerde el estado anterior. Las solicitudes HTTP pueden ser de diferentes tipos,

como GET para recuperar datos, POST para enviar datos al servidor, y otros métodos para diferentes acciones (DevDocs, s. f.).

Características Clave de HTTP:

- **Sin Estado:** Cada solicitud es independiente, sin conocimiento del estado anterior.
- **Basado en Texto:** Los mensajes HTTP son legibles para humanos y están compuestos principalmente de encabezados y cuerpo.
- **Conexiones No Seguras:** La información transmitida no está cifrada, lo que puede presentar riesgos de seguridad, especialmente para datos sensibles.

2.16.2. HTTPS (Hypertext Transfer Protocol Secure)

El Protocolo de transferencia de hipertexto seguro (HTTPS: Hypertext Transfer Protocol Secure) es la versión segura de HTTP y utiliza cifrado para proteger la integridad y confidencialidad de los datos transmitidos. Se basa en el protocolo SSL/TLS para proporcionar una capa adicional de seguridad (DevDocs, s. f.).

Características Clave de HTTPS:

- **Cifrado:** Utiliza SSL/TLS para cifrar los datos, lo que hace que sea más difícil para los atacantes interceptar y comprender la información transmitida.
- **Certificados SSL/TLS:** Requiere un certificado SSL/TLS válido para establecer la conexión segura.
- **Puerto 443:** HTTPS utiliza el puerto 443 en lugar del puerto 80 utilizado por HTTP.

Ventajas de HTTPS sobre HTTP:

- **Seguridad de Datos:** La información sensible está protegida mediante cifrado.

- **Integridad de Datos:** El cifrado garantiza que los datos no se alteren durante la transmisión.
- **Autenticación:** Los certificados SSL/TLS permiten verificar la autenticidad del servidor.

2.17. Herramientas de escalabilidad – Contenedores y Kubernetes

En el ámbito de desarrollo y despliegue de aplicaciones, la escalabilidad se ha convertido en un aspecto crítico. La capacidad de manejar el aumento de carga y asegurar un rendimiento consistente lleva al uso extendido de herramientas como contenedores y orquestadores como Kubernetes.

2.17.1. Contenedores

Los contenedores son entornos ligeros y portátiles que encapsulan una aplicación y todas sus dependencias, permitiendo su ejecución de manera consistente en cualquier entorno que admita contenedores. La tecnología de contenedores, liderada por Docker, ha transformado la forma en que las aplicaciones se desarrollan, empaquetan y despliegan (IBM. s. f.-b).

Características Clave de Contenedores:

- **Portabilidad:** Los contenedores incluyen todo lo necesario para ejecutar una aplicación, asegurando la consistencia entre entornos de desarrollo, prueba y producción.
- **Aislamiento:** Cada contenedor es independiente, evitando conflictos de dependencias entre aplicaciones.

- **Eficiencia:** Los contenedores comparten el mismo núcleo del sistema operativo, reduciendo la sobrecarga en comparación con las máquinas virtuales.

2.17.2. Kubernetes

Kubernetes, a menudo abreviado como K8s, es un sistema de orquestación de contenedores de código abierto que automatiza la implementación, escalabilidad y operación de aplicaciones en contenedores. Diseñado por Google, Kubernetes proporciona un entorno robusto y escalable para gestionar aplicaciones contenerizadas en un clúster (iKenshu, 2019).

Características Clave de Kubernetes:

- **Orquestación Automatizada:** Kubernetes automatiza la implementación, actualización y escalabilidad de las aplicaciones.
- **Escalabilidad Horizontal:** Permite la escalabilidad dinámica agregando o eliminando contenedores según la carga de trabajo.
- **Autoreparación:** Kubernetes detecta y reemplaza automáticamente contenedores o nodos defectuosos.
- **Gestión de Recursos:** Controla y asigna recursos, como CPU y memoria, para garantizar un rendimiento óptimo.

Las herramientas de escalabilidad, como los contenedores y Kubernetes, han revolucionado la forma en que las aplicaciones se desarrollan, despliegan y escalan. Estas tecnologías ofrecen soluciones eficientes y flexibles para abordar los desafíos de escalabilidad en entornos modernos, permitiendo a las empresas mejorar la eficiencia, la consistencia y la confiabilidad de sus aplicaciones.

2.18. Técnicas para digitalizar y almacenar documentos

La digitalización de documentos es un proceso esencial en entornos modernos que permite convertir documentos físicos en formato digital para facilitar su almacenamiento, gestión y acceso. Este proceso es especialmente valioso para documentos como partituras musicales, donde la preservación y manipulación digital son cruciales. A continuación, se describen varias técnicas y consideraciones relacionadas con la digitalización y almacenamiento de documentos, específicamente partituras.

2.18.1. Digitalización de Documentos

La digitalización implica la conversión de documentos físicos, como partituras en papel, a formatos digitales. En el caso de partituras, se puede realizar utilizando una cámara fotográfica para capturar imágenes de alta resolución. Posteriormente, estas imágenes se pueden convertir a formatos digitales comunes como JPEG, PNG o incluso documentos PDF. Este proceso no solo preserva la esencia de la partitura física, sino que también facilita su almacenamiento y distribución electrónica (Sydle, 2023).

2.18.2. Almacenamiento en el Sistema de Archivos

En esta técnica, las imágenes o documentos PDF generados se almacenan directamente en carpetas del sistema de archivos. Después de la digitalización, los archivos se guardan en ubicaciones específicas del sistema de archivos. La base de datos almacena referencias (rutas o nombres de archivo) para acceder a ellos.

Ventajas:

- **Simplicidad:** Es un enfoque directo y fácil de implementar.

- **Accesibilidad:** Los archivos son accesibles a través del sistema de archivos.

2.18.3. Almacenamiento en Servicios de Almacenamiento en la Nube

Los documentos digitalizados se pueden cargar en servicios de almacenamiento en la nube (por ejemplo, Google Drive, Dropbox). La base de datos almacena enlaces o referencias a los archivos almacenados en la nube.

Ventajas:

- **Escalabilidad:** Facilita la gestión de grandes volúmenes de documentos.
- **Accesibilidad Remota:** Permite acceder a los documentos desde cualquier ubicación.

2.18.4. Almacenamiento en Base de Datos usando el Tipo de Dato Binario

Las imágenes o documentos PDF se almacenan directamente como tipos de datos binarios en la base de datos. Los documentos digitalizados se guardan en columnas de tipo binario (BLOB o bytea según la base de datos).

Ventajas:

- **Consistencia:** La base de datos mantiene una relación uno a uno entre la entrada y el archivo.
- **Simplicidad:** Implementación directa y fácil gestión de los documentos.

Para el proyecto en cuestión, la técnica seleccionada para almacenar los documentos digitalizados, como las partituras, es la descrita en el punto 2.18.4: *"Almacenamiento en Base de Datos usando el Tipo de Dato Binario"*. En este enfoque, las imágenes y documentos PDF digitalizados se guardan directamente como tipos de datos binarios en la base de datos. Esta elección se alinea con los requisitos del proyecto, priorizando la simplicidad, la consistencia

y la gestión eficiente de documentos, asegurando un acceso rápido y seguro a las partituras digitalizadas.

2.19. Herramientas tecnológicas

2.19.1. PHP

PHP es un lenguaje de secuencias de comandos de propósito general de código abierto ampliamente utilizado que es especialmente adecuado para el desarrollo web y se puede incrustar en HTML (*PHP: What is PHP? - Manual, s. f.*). Se hará uso de la versión 8.1.

2.19.2. Laravel

Laravel es un marco de trabajo o framework, basado en el lenguaje de programación PHP, gratuito y de código abierto que proporciona un conjunto de herramientas y recursos para crear aplicaciones PHP modernas. Con un ecosistema completo que aprovecha sus funciones integradas y una variedad de paquetes y extensiones compatibles. Laravel proporciona poderosas herramientas de base de datos que incluyen un ORM (Object Relational Mapper) llamado Eloquent y mecanismos integrados para crear migraciones de bases de datos y seeders. Con la herramienta de línea de comandos Artisan, los desarrolladores pueden iniciar nuevos modelos, controladores y otros componentes de la aplicación, lo que acelera el desarrollo general de la aplicación (Heidi, 2021). (Versión 10).

2.19.3. Eloquent

Eloquent, es un mapeador objeto-relacional (ORM) que hace que sea agradable interactuar con la base de datos. Cuando se utiliza Eloquent, cada tabla de la base de datos tiene su correspondiente "Modelo" que se utiliza para interactuar con esa tabla. Además de recuperar registros de la tabla de la base de datos, los modelos de Eloquent permiten insertar, actualizar y eliminar registros de la tabla (Documentación Laravel en español, s. f.-b).

2.19.4. SQL

SQL, conocido como lenguaje de consulta estructurada, ha transformado la gestión de bases de datos relacionales con su capacidad de interactuar con sistemas de gestión de bases de datos de manera eficiente. Su sintaxis clara y coherente simplifica la realización de consultas complejas y operaciones de administración. Dividido en comandos de creación y manipulación de datos, SQL permite la creación de objetos esenciales en la estructura de la base de datos y facilita la selección, inserción, actualización y eliminación de registros. Ampliamente utilizado en una variedad de aplicaciones, desde sistemas de inventario hasta aplicaciones web dinámicas, SQL sirve como una herramienta vital para desarrolladores de software y analistas de datos, brindando una interfaz efectiva para gestionar y analizar datos críticos en diferentes entornos operativos (AWS, s. f.).

2.19.5. PostgreSQL

PostgreSQL es un sistema de base de datos altamente estable y de código abierto que brinda soporte a diferentes funciones de SQL, como claves externas, subconsultas, disparadores y diferentes tipos y funciones definidos por el usuario. Aumenta aún más el lenguaje SQL y ofrece varias características que escalan y reservan cargas de trabajo de datos

meticulosamente. Se utiliza principalmente para almacenar datos para muchas aplicaciones móviles, web, geoespaciales y de análisis (Ravoof, 2023). (Versión 14.9).

2.19.6. Visual Studio Code

Visual Studio Code es un entorno de desarrollo integrado (IDE) de código abierto desarrollado por Microsoft. Es conocido por su amplia gama de funciones y su interfaz de usuario altamente personalizable que lo convierte en una herramienta popular entre los desarrolladores de software. Diseñado para admitir múltiples lenguajes de programación y plataformas, Visual Studio Code es compatible con sistemas operativos Windows, macOS y Linux (Visual Studio Code - Code editing. Redefined, 2021).

Entre las características clave de Visual Studio Code se encuentran su potente editor de código con resaltado de sintaxis, finalización de código y depuración integrada. Además, ofrece una amplia gama de extensiones y complementos que permiten a los usuarios personalizar su experiencia de desarrollo según sus necesidades específicas.

Visual Studio Code también es conocido por su amplio soporte para el control de versiones a través de sistemas como Git, lo que facilita el trabajo colaborativo y el seguimiento de cambios en el código. Asimismo, proporciona una integración fluida con servicios en la nube y herramientas de desarrollo web, lo que lo convierte en una opción versátil para una variedad de proyectos de desarrollo de software. (Versión 1.84).

2.19.7. Postman

Postman es una plataforma de colaboración para el desarrollo de API que simplifica el proceso de diseño, desarrollo, prueba y documentación de API. Se ha convertido en una herramienta esencial para los desarrolladores de software y equipos de desarrollo, ya que permite una fácil interacción con API y servicios web de una manera intuitiva y eficiente.

Con Postman, los desarrolladores pueden enviar solicitudes a una API, inspeccionar las respuestas y realizar pruebas exhaustivas para garantizar la funcionalidad y el rendimiento adecuados de la API. Además de sus capacidades de prueba, Postman ofrece herramientas para la creación de documentación detallada de API, lo que facilita la comprensión y el uso de las API por parte de otros miembros del equipo y de la comunidad en general (About Postman, 2023).

2.19.8. Docker

Docker es una plataforma de código abierto que facilita la creación, implementación y ejecución de aplicaciones en contenedores. Los contenedores son unidades ligeras y portátiles que encapsulan una aplicación y sus dependencias, permitiendo su ejecución de manera consistente en cualquier entorno compatible con Docker. Las características clave de Docker incluyen portabilidad, eficiencia y gestión de recursos. Permite a los desarrolladores empacar una aplicación y todas sus dependencias en un contenedor, lo que garantiza que se ejecute de manera coherente en cualquier entorno (InnovaciónDigital, 2022). (Versión 24.0.7)

Para integrar Docker con Laravel, generalmente se define un archivo Dockerfile para construir la imagen del contenedor de la aplicación Laravel y un archivo docker-compose.yml para configurar y ejecutar servicios adicionales. Laravel proporciona una estructura modular que se integra bien con el enfoque de contenedores de Docker (Documentación Laravel en español, s. f.-b).

CAPÍTULO 3: ANÁLISIS Y DISEÑO DEL SISTEMA

Este capítulo constituye una inmersión en la especificación y diseño del backend del sistema, concentrándose en los requerimientos (identificación de funcionalidades del software) y en la descripción de las historias de usuario. En este contexto, nos abocaremos al diseño de pruebas, estableciendo una base robusta para la validación continua del sistema.

Con un énfasis exclusivo en la lógica del servidor, exploramos estrategias detalladas de prueba que garantizarán la fiabilidad del back-end. Además, se abordarán aspectos cruciales como la arquitectura general del sistema (Cliente-Servidor), la elección del patrón de diseño (MVC) y el modelado de datos (Entidad-Relación), configurando así la infraestructura esencial para la implementación del sistema.

3.1. Requerimientos del Sistema

A continuación, se delinean los requisitos esenciales que guiarán el desarrollo del sistema. Es necesario definir los actores presentes en la aplicación y requisitos del sistema, los cuales abarcan tanto aspectos funcionales como no funcionales, estableciendo el marco estructural y los estándares de calidad a alcanzar.

3.1.1. Actores

Un actor es una persona que interactúa con el sistema para ejecutar y cumplir con los requerimientos planteados. Se ha realizado una investigación de los actores que forman parte del desarrollo de la aplicación, los cuales cumplen tareas específicas en el sistema. Para el desarrollo de la aplicación distinguimos dos actores: administrador y usuario prestatario.

- **Administrador**: Posee acceso a todas las funcionalidades del sistema. Es el encargado del registro y control de préstamos de partituras, así como el seguimiento y control de solicitudes de partituras. Puede registrar usuarios administradores y prestatarios.

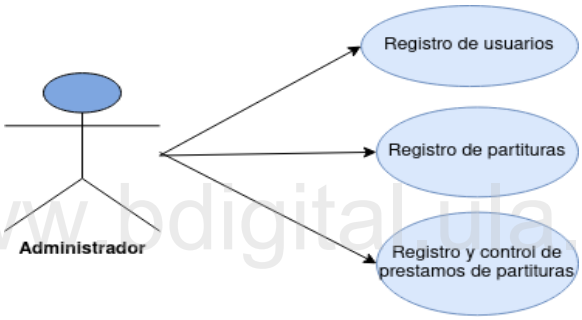


Figura 3: Usuario Administrador.

- **Usuario prestatario (Autenticado):** Posee permisos para solicitar préstamos de partituras dentro del sistema. Si lo desea puede registrarse en el sistema.

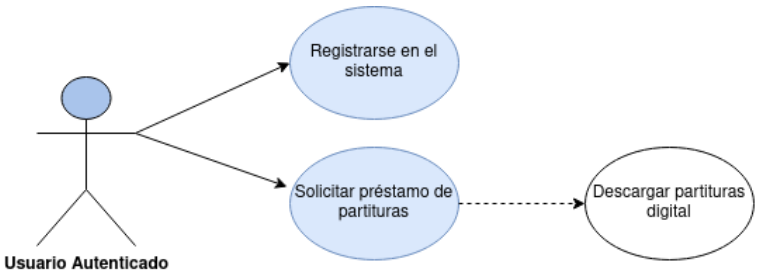


Figura 4: Usuario Prestatario (Autenticado).

3.1.2. Requisitos Funcionales

Un requisito funcional define las funcionalidades que debe realizar el sistema. A continuación, se presentan de manera general los principales requisitos de la aplicación:

- **Registro y autenticación de usuarios:** El usuario administrador puede registrar usuarios administradores y usuarios prestatarios del sistema. Además, los usuarios registrados en el sistema pueden realizar distintas acciones del sistema.
- **Registro inicial del sistema**: El sistema posee una carga inicial que permite configurar y registrar la información de las partituras, por ejemplo, autor, género musical, ubicaciones y archivadores.
- **Registro de partituras**: El usuario administrador puede realizar la carga de partituras dentro del sistema, indicando datos propios de la misma, como autor, género, título y ubicación.
- **Solicitud de préstamos de partituras:** Los usuarios prestatarios pueden solicitar préstamos de las partituras almacenadas en el sistema. Esta solicitud puede hacerse de dos formas, un préstamo digital y un préstamo físico.
- **Carga y descarga de partituras:** Mediante el sistema se pueden cargar y descargar las partituras, si la solicitud de préstamo es de forma digital.
- **Búsqueda, filtrado y ordenación de partituras:** Los usuarios pueden llevar a cabo búsquedas de partituras por el título, autor o género musical atendiendo a los criterios definidos en la aplicación.

3.1.3 Requisitos no funcionales

Los requisitos no funcionales imponen restricciones en el diseño o la implementación de la aplicación. Se describen a continuación los requisitos que definirán la manera de desarrollar la aplicación.

- **Confidencialidad**: La información manejada por el sistema está protegida de acceso no autorizado y divulgación.
- **Disponibilidad**: El sistema debe estar disponible en cualquier momento. Los usuarios tienen acceso garantizado a la información.
- **Mantenimiento:** El código fuente del sistema debe estar bien documentado y seguir las mejores prácticas de desarrollo para facilitar futuras mejoras y actualizaciones.
- **Portabilidad:** El sistema debe poder ejecutarse en diferentes plataformas con cambios mínimos, por lo que es necesario que posea un diseño “Responsive” a fin de garantizar la visualización en múltiples equipos electrónicos, computadoras, dispositivos móviles.
- **Fiabilidad:** El sistema debe ser confiable y cumplir con los requisitos planteados.
- **Usabilidad**: El sistema posee una interfaz sencilla y atractiva que garantiza el buen funcionamiento del sistema al usuario.

3.1.4. Historias de Usuario

La descripción detallada de las historias de usuario complementa este enfoque, proporcionando una comprensión exhaustiva de las funcionalidades clave que se deben implementar, junto con criterios de aceptación específicos que actúan como hitos claros para la finalización exitosa de cada característica. En la tabla 1 se exponen las historias de usuario

del sistema con una breve descripción, seguidamente se mostrará la estructura de las historias de usuarios más relevantes del sistema.

ID	Nombre de la historia de usuario	Descripción
001	**Registro de partitura**	Como administrador del sistema quiero poder registrar una nueva partitura en el sistema, para tener un registro organizado de todas las partituras disponibles en la fundación.
002	**Registro de autor**	Como administrador del sistema quiero poder registrar un nuevo autor de partitura para tener un registro organizado de los autores de partituras disponibles.
003	**Registro de géneros musicales**	Como administrador del sistema quiero poder registrar un nuevo género musical para tener un registro organizado de los géneros musicales disponibles.
004	**Registro de ubicaciones**	Como administrador del sistema quiero poder registrar una nueva ubicación para tener un registro organizado de los archivadores y/o gavetas.
005	**Registro de archivadores**	Como administrador del sistema quiero poder registrar un nuevo archivador para tener un registro organizado de los archivadores.
006	**Registro de gavetas**	Como administrador del sistema quiero poder registrar una nueva gaveta para tener un registro organizado de los archivadores.
007	**Registro de usuario**	Como administrador del sistema quiero poder registrar usuarios para que puedan solicitar préstamos de partituras digitales o físicas. Como usuario quiero poder registrarme en la aplicación y poder solicitar préstamos de partituras digitales.
008	**Registro de usuarios administrador**	Como administrador del sistema quiero poder registrar usuarios administradores
009	**Descargar partitura digital**	Como Prestatario quiero poder descargar una partitura digital.
010	**Préstamo de partitura físico**	Como administrador del sistema puedo registrar un préstamo de partitura asociado a un prestatario.

011	**Registrarse como prestatario**	Como prestatario quiero registrarme en el sistema para solicitar préstamos de partituras.

Tabla 1: Historias de usuario del sistema

A continuación, se presentan las historias de usuario más resaltantes del sistema.

Historia de usuario HU 001
Nombre de la historia de usuario: **Registro de partitura**
Usuario: Administrador del sistema
Descripción: Como administrador del sistema quiero poder registrar una nueva partitura en el sistema, para tener un registro organizado de todas las partituras disponibles en la fundación.
Validación: El usuario debe tener acceso al formulario de partitura. El usuario debe poder ingresar los siguientes campos: Titulo de la partitura(Campo de texto. Obligatorio) Autor de la partitura (Campo de selección. Obligatorio) Género musical al que pertenece(Campo de selección. Opcional) Ubicación física del documento(Opcional) Stock disponible (Campo de tipo numérico. Es opcional, pero se establecerá en 0 si no se proporciona)

Figura 5: Historia de usuario HU 001: Registro de Partituras

Historia de usuario HU 007
Nombre de la historia de usuario: **Registro de usuarios prestatario**
Usuario: Administrador del sistema o usuario
Descripción: Como administrador del sistema quiero poder registrar usuarios para que puedan solicitar prestamos de partituras digitales o físicas. Como usuario quiero poder registrarme en la aplicación y poder solicitar prestamos de partituras digitales.
Validación: El usuario debe tener acceso al formulario de registro de prestador. El usuario debe poder ingresar los siguientes campos: Nombre(Campo de texto. Obligatorio) Teléfono (Campo numérico. Obligatorio) Correo electrónico (Campo de texto. Obligatorio)

Figura 6: Historia de usuario HU 007: Registro de usuario prestatario.

Historia de usuario HU 008
Nombre de la historia de usuario: **Registro de usuario administrador**
Usuario: Administrador del sistema
Descripción: Como administrador del sistema quiero poder registrar usuarios administradores.
Validación: El usuario debe tener acceso al formulario de registro de administrador. El usuario debe poder ingresar los siguientes campos: Nombre(Campo de texto. Obligatorio) usuario (Campo de texto. Obligatorio) Contraseña (Campo de texto. Obligatorio)

Figura 7: Historia de usuario HU 008: Registro de usuario administrador.

Historia de usuario HU 009
Nombre de la historia de usuario: **Descargar partitura digital**
Usuario: Prestatario
Descripción: Como Prestatario quiero poder descargar una partitura digital.
Validación: Se debe tener un usuario registrado en el sistema. El usuario debe tener acceso al formulario de solicitud de préstamos. El usuario debe poder ingresar los siguientes campos: Partitura(Campo de selección. Obligatorio) Fecha del préstamo (Campo de tipo fecha. Obligatorio) Fecha de entrega (Campo de tipo fecha. Obligatorio) Cantidad (Campo de tipo numérico)

Figura 8: Historia de usuario HU 009: Descargar partitura digital.

Historia de usuario HU 010
Nombre de la historia de usuario: **Préstamo de partitura físico**
Usuario: Administrador del sistema
Descripción: Como administrador del sistema puedo registrar un préstamo de partitura asociado a un prestatario.
Validación: El usuario prestatario debe tener un usuario dentro del sistema. El usuario debe tener acceso al formulario de solicitud de préstamos. El usuario administrador puede aprobar préstamos de partituras. El usuario debe poder ingresar los siguientes campos: Partitura(Campo de selección. Obligatorio) Fecha del préstamo (Campo de tipo fecha. Obligatorio) Fecha de entrega (Campo de tipo fecha. Obligatorio) Cantidad (Campo de tipo numérico)

Figura 9: Historia de usuario HU 010: Préstamo de partitura físico.

3.2. Diseño de Pruebas

Este es el paso crucial para asegurar la robustez y confiabilidad del sistema. Siguiendo la metodología TDD, a partir de cada aspecto funcional del sistema identificado en la sección anterior, se traducirá en una prueba unitaria.

1. Test para registrar un usuario prestatario

Nombre: Registrar un usuario prestatario en el sistema.

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.
- El nombre de usuario no debe estar repetido.
- Validar que el sistema permita registrar una contraseña válida.
- Validar que sólo permita registros con correos electrónicos válidos.
- Validar que permita ingresar datos numéricos en el campo Teléfono.
- Validar envío de correo de confirmación.
- Validar que no permita registros con un correo electrónico duplicado.

2. Test para registrar un usuario administrador

Nombre: Registrar un usuario administrador en el sistema.

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.
- El nombre de usuario no debe estar repetido.
- Validar que el sistema permita registrar una contraseña válida.

3. Test para el Login de usuario

Nombre: Iniciar sesión en el sistema.

Descripción: Validar que se pueda ingresar con las credenciales registradas en el sistema.

- Validar acceso con credenciales correctas.
- Validar mensaje de error con credenciales incorrectas.
- Validar cierre de sesión.

4. Test para registrar partituras

Nombre: Registrar partituras en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.
- Un título perteneciente a un autor no debe estar repetido.
- Si no se ingresa la cantidad de partituras, el sistema la establece en 0.
- En el sistema tienen que existir registros almacenados de género musical y autor de la partitura.

5. Test para registrar autor

Nombre: Registrar autor en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.

- El campo nombre no debe estar repetido.

6. Test para registrar géneros musicales

Nombre: Registrar género musical en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.
- El campo nombre no debe estar repetido.

7. Test para registrar ubicaciones

Nombre: Registrar ubicaciones en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.
- En el sistema tienen que existir registros almacenados de gaveta y archivador.

8. Test para registrar gavetas

Nombre: Registrar gavetas en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos
- El campo nombre no debe estar repetido.

9. Test para descargar partituras en digital

Nombre: Descargar partituras en digital

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Validar que la fecha de préstamo de la partitura sea menor a la fecha de entrega.

- Validar que la partitura solicitada esté registrada en el sistema.
- Validar que la cantidad a solicitar sea mayor a 0.
- Validar que el sistema habilite la descarga de la partitura solicitada.

10. Test para solicitar préstamos de partituras en físico

Nombre: Solicitar préstamo de partitura en físico

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Validar que la fecha de préstamo de la partitura sea menor a la fecha de entrega.
- Validar que la partitura solicitada esté registrada en el sistema.
- Validar que la cantidad a solicitar esté disponible en el inventario de partituras.
- Validar que se puedan aprobar los préstamos de partituras.

11. Test para que el usuario prestatario pueda visualizar el historial de partituras

Nombre: Visualizar historial de partituras.

Descripción: Validar que un usuario pueda registrarse en el sistema y visualizar el historial o listado de partituras.

- Validar que el usuario pueda ingresar en el sistema.
- Validar que el usuario pueda visualizar el listado de partituras.
- Validar que el usuario pueda filtrar la información de las partituras.

3.3. Arquitectura General del Sistema (Cliente - Servidor)

La visión general del sistema, se configura bajo una arquitectura cliente-servidor. En esta estructura, el servidor se posiciona como el núcleo central, manejando la lógica de negocio y operaciones robustas, mientras que el cliente interactúa de manera eficiente para obtener y presentar la información. Esta disposición permite una gestión efectiva de

solicitudes y respuestas, asegurando un rendimiento optimizado en cada interacción del usuario.

A continuación, se describen las partes fundamentales de la arquitectura general de la aplicación, representada en el siguiente diagrama (Figura 10):

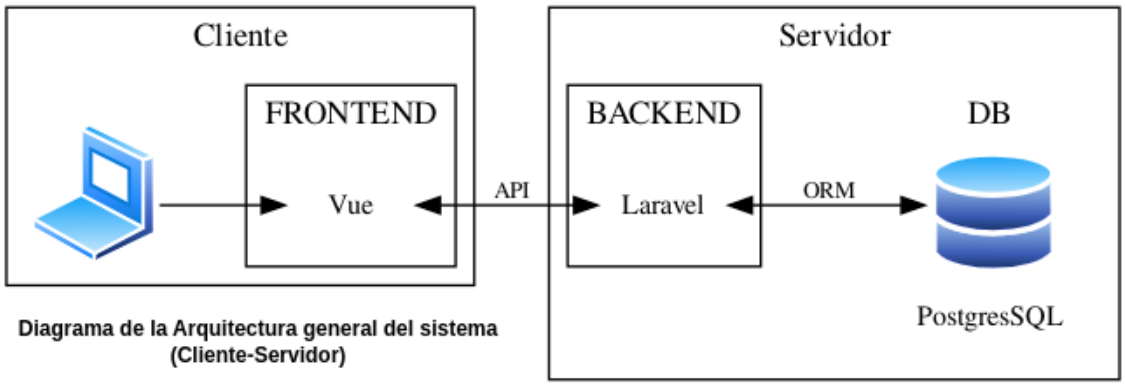


Figura 10. Arquitectura General (Cliente-Servidor)

Cliente: El usuario interactúa a través del frontend.

Frontend: La interfaz de usuario se construye con Vue.js, facilitando la creación de experiencias interactivas y atractivas.

Servidor: Compuesto por el backend y la base de datos, el servidor gestiona la lógica de la aplicación y el almacenamiento de datos.

Backend (Laravel): Desarrollado con Laravel, un framework PHP que utiliza la estructura MVC para organizar y gestionar la lógica de la aplicación.

Base de Datos (PostgresSQL): PostgresSQL es la elección para almacenar y gestionar los datos, facilitando la interacción mediante un Mapeador Objeto-Relacional (ORM).

Flujo de Datos:

- El cliente realiza acciones en la interfaz.
- Se generan solicitudes al backend mediante una API RESTful.

- El backend procesa las solicitudes, realiza operaciones en la base de datos y devuelve los resultados al frontend.

3.4. Patrón de Diseño de software – MVC

El patrón de diseño Modelo-Vista-Controlador (MVC) se refleja en la lógica de la aplicación, orientado al diseño del back-end, de la siguiente manera (Figura 11):

1. **Solicitud HTTP:** El proceso comienza con una solicitud HTTP que proviene de un cliente, representada por el nodo VIEW en el diagrama. Esta solicitud se dirige al nodo ROUTE a través de la funcionalidad de enrutamiento.
2. **Se direcciona la petición al controlador:** La función principal del enrutador representado por el nodo ROUTE, es dirigir la solicitud HTTP entrante al controlador correspondiente dado por el nodo CONTROLLER. El enrutador determina qué controlador debe manejar la solicitud, mediante una función específica.
3. **Lógica de negocio:** El controlador representado por el nodo CONTROLLER, es responsable de la lógica de negocio de la aplicación. Recibe la solicitud del enrutador y realiza las operaciones necesarias para procesarla. El controlador puede interactuar con el modelo (nodo MODEL) para realizar operaciones en la base de datos (nodo DB) y obtener la información necesaria.
4. **Se busca la data:** El modelo representado por el nodo MODEL, se encarga de interactuar con la base de datos (DB). Realiza consultas y operaciones en la base de datos para obtener o actualizar la información solicitada por el controlador. La base de datos contiene la información persistente de la aplicación.

5. **Se muestran los datos:** Una vez que el controlador ha procesado la solicitud y obtenido la información necesaria del modelo, se pasa esa información a la vista representada por el nodo VIEW. La vista se encarga de presentar la información al usuario final de la manera más adecuada.

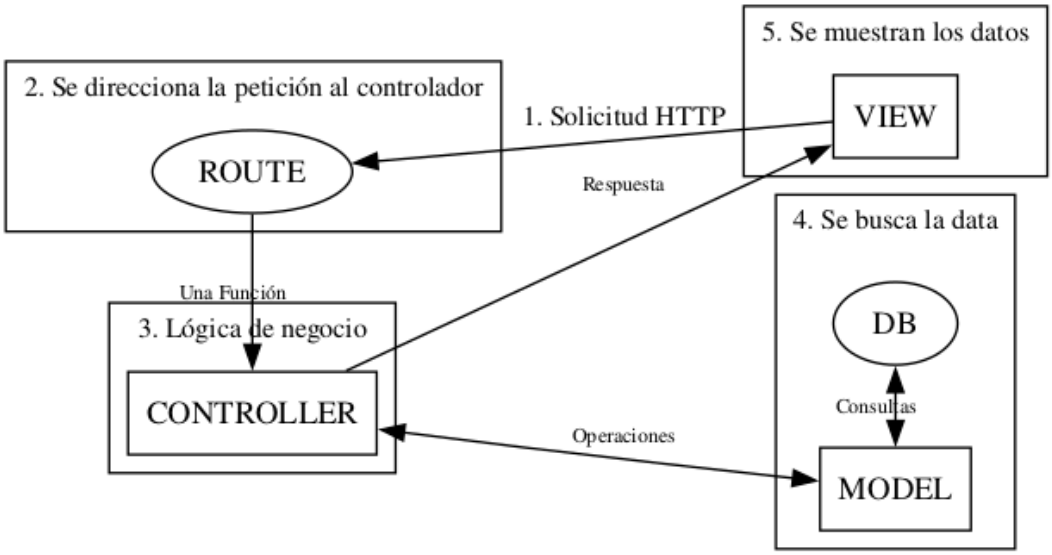


Figura 11. Patrón de Diseño MVC

Flujo General:

- La solicitud HTTP inicia el proceso y se dirige al controlador por medio de una ruta.
- El controlador realiza la lógica de negocio y puede interactuar con el modelo para obtener o actualizar datos.
- La información se pasa a la vista, que se encarga de presentarla al usuario.

Este diseño modular facilita el mantenimiento, la escalabilidad y la comprensión del código, ya que cada componente tiene una responsabilidad clara y separada.

3.5. Modelado de Datos (Diagrama de Entidad-Relación)

A continuación, se presenta una breve descripción del modelado de datos que respalda el sistema. La elección de PostgresSQL como sistema de gestión de bases de datos y Laravel como framework ofrece robustez y eficiencia en la manipulación de datos.

El modelado sigue las mejores prácticas, garantizando una estructura lógica y coherente. El Diagrama de Entidad-Relación (Figura 12) ilustra claramente las entidades, atributos y relaciones, proporcionando una guía visual del esquema de la base de datos. Este enfoque facilita la gestión de datos complejos y su manipulación dentro de la aplicación.

En tal sentido, en el diagrama entidad-relación que modela la base de datos, se resaltan las tablas fundamentales que capturan las relaciones clave en el sistema. La entidad principal, Partitura, actúa como eje central y se conecta de manera significativa con otras entidades cruciales, como Autor, Género, y Ubicación. Estas entidades adicionales permiten clasificar y organizar las partituras según sus autores, géneros musicales, y ubicaciones físicas.

Para respaldar la gestión eficiente de partituras digitalizadas, se encuentra la tabla “files”, la cual almacena representaciones digitales vinculadas directamente a las partituras. Además, la gestión de préstamos, una funcionalidad vital del sistema, se representa a través de las entidades Préstamo y Usuario Prestatario. Estas entidades capturan la información esencial sobre los préstamos realizados, como fechas y estados del préstamo, junto con la identificación de los usuarios que han solicitado las partituras.

Este esquema nos da un vistazo completo de cómo estas piezas están conectadas, dando una perspectiva más profunda sobre la estructura y las relaciones fundamentales en el sistema de gestión de partituras.

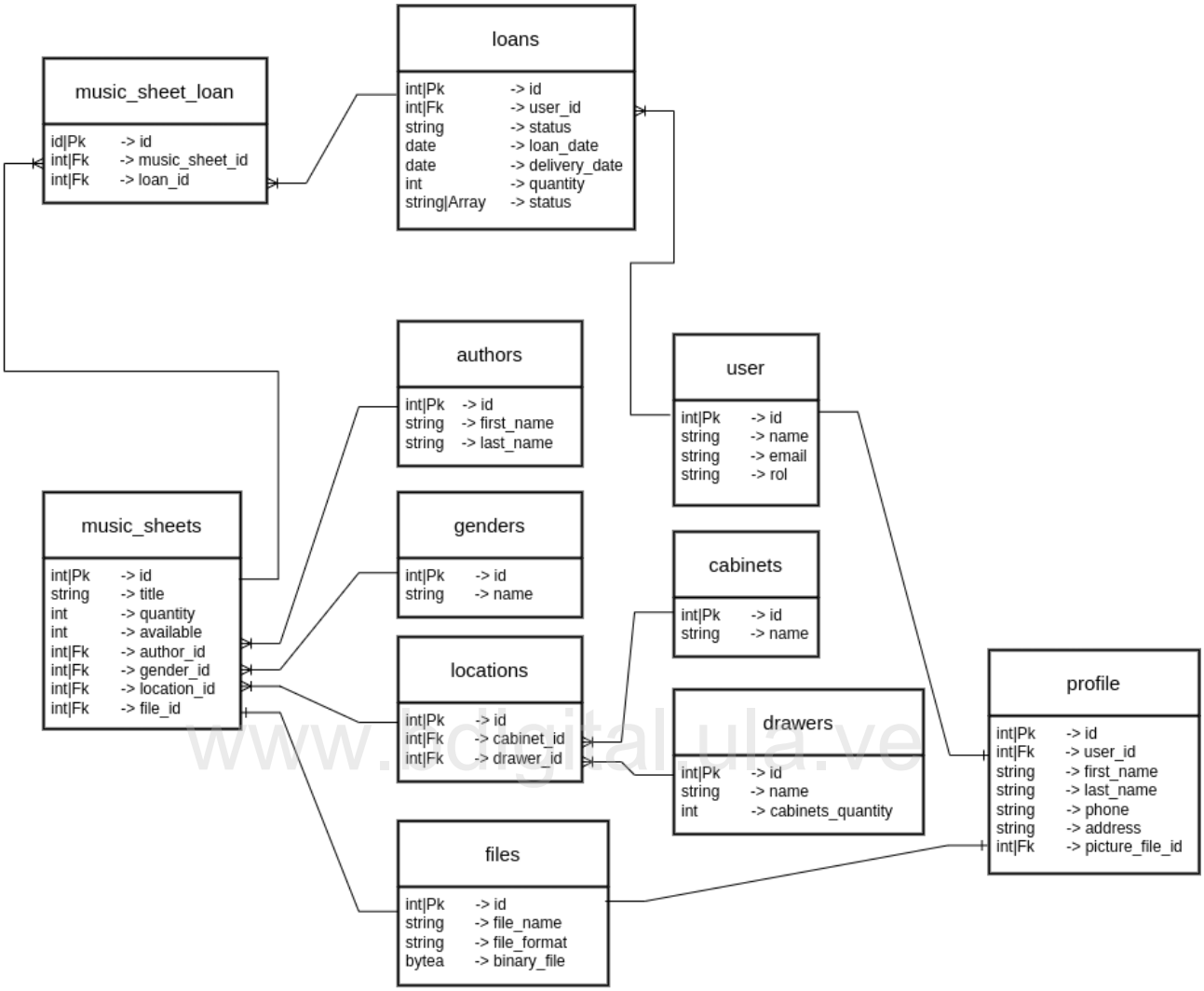


Figura 12. Diagrama Entidad-Relación

CAPÍTULO 4: IMPLEMENTACIÓN DEL SISTEMA

En este capítulo, se especifica cómo se estructuró el entorno de desarrollo, así como las herramientas y tecnologías utilizadas. De igual manera, se aborda la fase de implementación del sistema, donde se transforman los diseños y planes previos en código funcional. Se detallan los aspectos clave del entorno de desarrollo y se explora el desarrollo de la lógica del sistema, incluyendo la implementación de migraciones, la creación de modelos y controladores, y la configuración de las rutas para construir la API del sistema. Este capítulo es crucial para convertir la visión conceptual en una aplicación práctica y funcional.

4.1 Entorno de Desarrollo

En la fase de implementación del sistema, se utilizó un entorno de desarrollo cuidadosamente configurado para asegurar la eficiencia y la calidad del código. A continuación, se detallan las herramientas y tecnologías empleadas:

4.1.1 IDE y Control de Versiones

El código fuente fue desarrollado en Visual Studio Code, un entorno de desarrollo integrado (IDE) conocido por su ligereza y potencia.

La gestión de versiones se llevó a cabo utilizando Git, permitiendo un seguimiento detallado de los cambios a lo largo del desarrollo.

4.1.2 Lenguaje de Programación y Framework

La implementación se realizó en PHP 8.0, aprovechando las características más recientes del lenguaje. Para la estructuración y desarrollo eficiente de la aplicación, se utilizó Laravel, un framework PHP moderno y robusto que facilita la creación de aplicaciones web.

4.1.3. Gestor de Base de Datos y ORM

La persistencia de datos se gestionó mediante el sistema de gestión de bases de datos relacional Postgres. Laravel utiliza Eloquent, un poderoso ORM, para interactuar con la base de datos. Eloquent simplifica las operaciones de la base de datos al mapear objetos de la aplicación directamente a registros de la base de datos, proporcionando una capa de abstracción que facilita el manejo de datos de manera elegante y eficiente.

4.1.4. Configuración del Entorno local

Se estableció un servidor local mediante '*`php artisan serve`*' de Laravel para ejecutar pruebas unitarias de manera eficiente y automatizadas, con el comando '*`php artisan test`*'. Utilizando *localhost*, se garantiza un entorno uniforme y controlado para las pruebas.

4.2. Desarrollo de la Lógica del Sistema

En esta sección, abordamos la generación de migraciones para la base de datos, creación de modelos ORM y desarrollo de controladores para gestionar las operaciones asociadas a las entidades principales del sistema. Estos pasos son cruciales para establecer la infraestructura necesaria antes de proceder con la implementación de las pruebas.

4.2.1. Implementación de Migraciones Laravel

Este apartado describe detalladamente el proceso de migración, tomando como ejemplo la tabla `'music_sheets'`, una de las entidades fundamentales de la aplicación.

- Antes de iniciar la migración, se asegura de haber configurado correctamente la conexión a la base de datos en el archivo .env dentro de la carpeta del proyecto. Esto se logra definiendo los detalles de la base de datos, como el nombre de la base de datos, el usuario y la contraseña (Figura 13).


```
22
23  DB_CONNECTION=pgsql
24  DB_HOST=127.0.0.1
25  DB_PORT=5432
26  DB_DATABASE=nombre_de_datos
27  DB_USERNAME=nombre_de_usuario_db
28  DB_PASSWORD=contraseña_de_usuario_db
29
```

Figura 13. Configuración de la conexión a DB

- Iniciamos con la generación de la migración ejecutando el siguiente comando Artisan: '*php artisan make:migration create_music_sheets_table*'. Este comando crea un nuevo archivo de migración en el directorio database/migrations. Luego,

editamos este archivo (`create_music_sheets_table.php`) para definir la estructura de la tabla.

- Código de Migración: A continuación, se presenta el código de la migración que define la estructura de la tabla `'music_sheets'` (Figura 14). Este código incluye campos como título, cantidad de partituras, identificadores de autor, género, ubicación y archivo de partitura.

- Finalmente, ejecutamos la migración para aplicar los cambios en la base de datos, con el siguiente comando desde la terminal: '*php artisan migrate*'.

Es importante destacar que las tablas referenciadas por las llaves foráneas (autor, género, ubicación, archivo de partitura) se crearon previamente para mantener la coherencia de la base de datos.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

You, hace 4 semanas | 2 authors (José Briceño and others) | Comment Code
class CreateMusicSheetsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    Codeium: Refactor | Explain
    public function up()
    {
        Schema::create('music_sheets', function (Blueprint $table) {
            $table->id();
            $table->string('title')->nullable()->comment('Título de la partitura');
            $table->integer('cuantity')->comment('Cantidad de partituras existentes');
            $table->integer('available')->comment('Cantidad de partituras disponibles');
            $table->foreignId('author_id')->nullable()->constrained()->onUpdate('cascade')->onDelete('restrict');
            $table->foreignId('gender_id')->nullable()->constrained()->onUpdate('cascade')->onDelete('restrict');
            $table->foreignId('location_id')->nullable()->constrained()->onUpdate('cascade')->onDelete('cascade');
            $table->foreignId('music_sheet_file_id')->nullable()->constrained()->onUpdate('cascade')->onDelete('cascade');
            $table->softDeletes();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    Codeium: Refactor | Explain
    public function down()
    {
        Schema::dropIfExists('music_sheets');
    }
}
```


Figura 14. Migración create_music_sheets_table.php

4.2.2. Creación de Modelos

En Laravel, los modelos sirven como una interfaz orientada a objetos para interacción entre la aplicación y la base de datos, simplificando las operaciones CRUD. A modo de ilustración se mostrará el proceso de creación del modelo para la entidad de "Partituras Musicales" (`music_sheets`).

- Creación del Modelo '`MusicSheet`'. Laravel facilita la creación de modelos mediante el uso de Artisan. Ejecutando el siguiente comando en la terminal: *'php artisan make:model MusicSheet'*.

- Este comando generará un archivo '`MusicSheet.php`' en el directorio app/Models, dentro de la carpeta del proyecto. En este archivo, se pueden especificar tanto los campos de la tabla en base de datos, que podrán ser accedidos por el modelo; así como las relaciones con otras entidades. A continuación, se muestra un ejemplo básico del contenido de este modelo (Figura 15).

Este modelo sigue las convenciones de nombres de Laravel. La tabla asociada se asume como pluralizada y en minúsculas (`music_sheets`), y el modelo en singular y en CamelCase (`MusicSheet`). Mantener estas convenciones facilitará la coherencia y mantenimiento en el desarrollo de la aplicación.

```php
class MusicSheet extends Model
{
    use HasFactory, SoftDeletes;

    /**
     * Undocumented variable
     *
     * @var array
     */
    protected $with = ['author', 'gender', 'location', 'musicSheetFile'];

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = ['author_id', 'gender_id', 'location_id', 'title', 'available', 'music_sheet_file_id'];

    /**
     * Attributes to be hidden from arrays
     *
     * @var array
     */
    protected $hidden = ['author_id', 'gender_id', 'location_id'];

    /**
     * One Music sheets belongs to one author
     *
     * @return void
     */
    Codeium: Refactor | Explain
    public function author()
    {
        return $this->belongsTo(Author::class);
    }

    /**
     * One Music sheets belongs to one gender
     *
     * @return void
     */
    Codeium: Refactor | Explain
    public function gender()
    {
        return $this->belongsTo(Gender::class);
    }

    /**
     * Undocumented function
     *
     * @return void
     */
    Codeium: Refactor | Explain
    public function location()
    {
        return $this->belongsTo(Locations::class);
    }

    /**
     * Retrieves the related MusicSheetFile model.
     *
     * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
     */
    Codeium: Refactor | Explain
    public function musicSheetFile()
    {
        return $this->belongsTo(MusicSheetFile::class);
    }
```


Figura 15. Modelo MusicSheet.php

4.2.3. Creación de Controladores

Con la infraestructura configurada y los modelos listos para interactuar con la base de datos, ahora procedemos con el desarrollo de los controladores, en este apartado sólo se mostrará la declaración de las funciones que intervienen en la lógica de negocio, la cual se especificará con mayor detalle en el siguiente capítulo. En tal sentido, abordaremos la creación de controladores, que actuarán como intermediarios entre las solicitudes del usuario

y las operaciones en la base de datos, haciendo uso de los modelos, seguiremos ilustrando este proceso con la entidad ‘Partituras musicales’.

- Generación del Controlador: Utilizando Artisan, el comando ‘*php artisan make:controller MusicSheetController’* se ejecutó para generar el archivo ‘MusicSheetController.php’. Este archivo se encuentra en el directorio app/Http/Controllers, dentro de la carpeta del proyecto.

- Estructura del Controlador: El archivo ‘MusicSheetController.php’ incluye métodos específicos para realizar diversas operaciones, (Figura 16):

  `index()`: Muestra todas las partituras musicales.

  `show($id)`: Muestra una partitura musical específica según su identificador.

  `store(Request $request)`: Almacena una nueva partitura musical en la base de datos.

  `update(Request $request, $id)`: Actualiza la información de una partitura existente.

  `destroy($id)`: Elimina una partitura musical.

Con la creación del controlador, se sientan las bases para la implementación de las pruebas diseñadas en el capítulo anterior. Estas pruebas verificarán la interacción correcta entre los controladores y los modelos con la base de datos, asegurando respuestas adecuadas a las solicitudes HTTP.

```php
namespace App\Http\Controllers;

use App\Models\MusicSheet;
use Illuminate\Http\Request;

class MusicSheetController extends Controller
{
    public function index()
    {
        // Lógica para mostrar todas las partituras musicales
    }

    public function show($id)
    {
        // Lógica para mostrar una partitura específica
    }

    public function store(Request $request)
    {
        // Lógica para almacenar una nueva partitura musical
    }

    public function update(Request $request, $id)
    {
        // Lógica para actualizar una partitura existente
    }

    public function destroy($id)
    {
        // Lógica para eliminar una partitura
    }
}
```


Figura 16. Controlador MusicSheetController.php

4.3. Configuración de Rutas: Construyendo la API del Sistema

Las rutas son la columna vertebral de cualquier API, definiendo cómo los usuarios interactúan con el sistema. Seguidamente, se detalla la creación y configuración de las rutas en Laravel, estableciendo endpoints que serán accesibles para el cliente. Estos endpoints, conectados a través de las rutas, representan la interfaz de programación de aplicaciones (API) del sistema. Cada ruta dirigirá las solicitudes del usuario a los controladores correspondientes, llevando a cabo acciones específicas y formando así la base para la

funcionalidad completa de la aplicación. Este paso es crucial para la construcción de un sistema coherente y fácilmente accesible.

A continuación, se especifican algunas rutas:

Middleware de Autenticación Sanctum:

- ‘`Route::middleware(['auth:sanctum'])->group(function () {...}`’: Este bloque asegura que las rutas dentro de él requieran autenticación utilizando el middleware Sanctum, que proporciona un sistema de autenticación de API simple y eficiente.

Rutas de Gestión de Usuarios:

- ‘`Route::get('/user', function (Request $request) {...}`’: Esta ruta permite obtener la información del usuario autenticado. La función anónima devuelve los detalles del usuario actual.

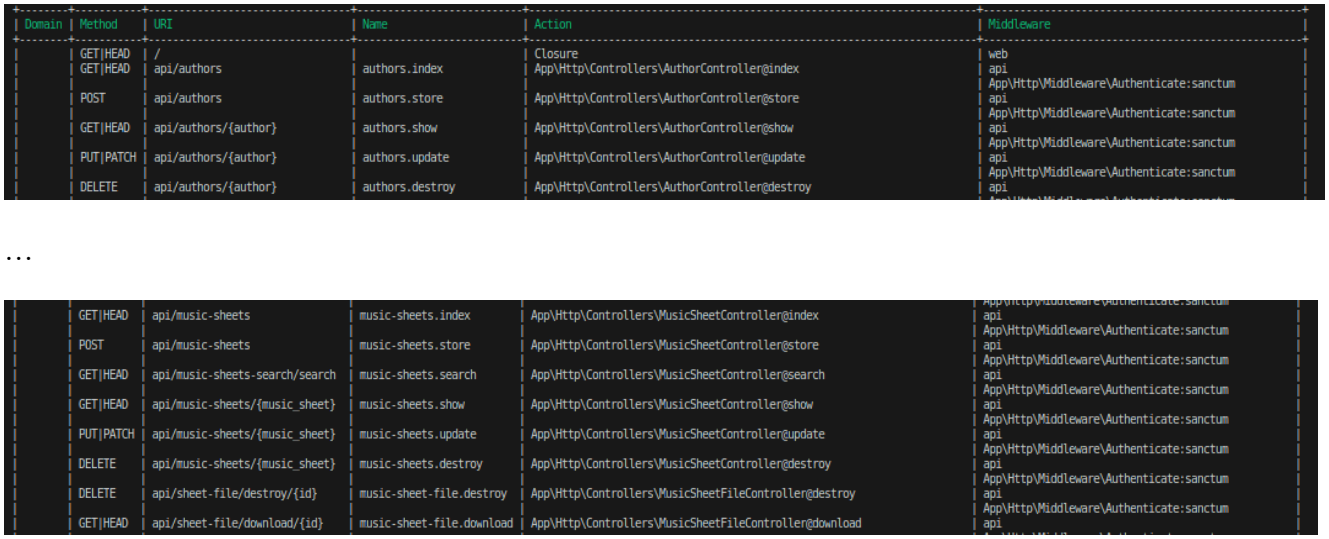
Rutas de Gestión de Partituras Musicales (/music-sheets):

- ‘`Route::get('/music-sheets', [MusicSheetController::class, 'index'])->name('music-sheets.index')`’: Devuelve la lista de partituras musicales. Utiliza el método `index` del controlador MusicSheetController.
- ‘`Route::post('/music-sheets', [MusicSheetController::class, 'store'])->name('music-`

`sheets.store')`: Almacena una nueva partitura musical en la base de datos. Utiliza el método `store` del controlador.

- `'Route::put('/music-sheets/{music_sheet}', [MusicSheetController::class, 'update'])->name('music-sheets.update')`: Actualiza una partitura musical existente. Utiliza el método `update` del controlador.
- `'Route::get('/music-sheets/{music_sheet}', [MusicSheetController::class, 'update'])->name('music-sheets.show')`: Devuelve una partitura musical en específico dado su ID. Utiliza el método `show` del controlador MusicSheetController.
- `'Route::delete('/music-sheets/{music_sheet}', [MusicSheetController::class, 'destroy'])->name('music-sheets.destroy')`: Elimina una partitura musical por su ID. Utiliza el método `destroy` del controlador.

Ejecutando en la terminal el comando de Artisan *'php artisan route:list'*, se listarán todas las rutas de la aplicación, como se puede observar en la imagen siguiente (Figura 17):



Domain	Method	URI	Name	Action	Middleware
	GET|HEAD	/		Closure	web
	GET|HEAD	api/authors	authors.index	App\Http\Controllers\AuthorController@index	api App\Http\Middleware\Authenticate:sanctum
	POST	api/authors	authors.store	App\Http\Controllers\AuthorController@store	api App\Http\Middleware\Authenticate:sanctum
	GET|HEAD	api/authors/{author}	authors.show	App\Http\Controllers\AuthorController@show	api App\Http\Middleware\Authenticate:sanctum
	PUT|PATCH	api/authors/{author}	authors.update	App\Http\Controllers\AuthorController@update	api App\Http\Middleware\Authenticate:sanctum
	DELETE	api/authors/{author}	authors.destroy	App\Http\Controllers\AuthorController@destroy	api

…

	GET|HEAD	api/music-sheets	music-sheets.index	App\Http\Controllers\MusicSheetController@index	api App\Http\Middleware\Authenticate:sanctum
	POST	api/music-sheets	music-sheets.store	App\Http\Controllers\MusicSheetController@store	api App\Http\Middleware\Authenticate:sanctum
	GET|HEAD	api/music-sheets-search/search	music-sheets.search	App\Http\Controllers\MusicSheetController@search	api App\Http\Middleware\Authenticate:sanctum
	GET|HEAD	api/music-sheets/{music_sheet}	music-sheets.show	App\Http\Controllers\MusicSheetController@show	api App\Http\Middleware\Authenticate:sanctum
	PUT|PATCH	api/music-sheets/{music_sheet}	music-sheets.update	App\Http\Controllers\MusicSheetController@update	api App\Http\Middleware\Authenticate:sanctum
	DELETE	api/music-sheets/{music_sheet}	music-sheets.destroy	App\Http\Controllers\MusicSheetController@destroy	api App\Http\Middleware\Authenticate:sanctum
	DELETE	api/sheet-file/destroy/{id}	music-sheet-file.destroy	App\Http\Controllers\MusicSheetFileController@destroy	api App\Http\Middleware\Authenticate:sanctum
	GET|HEAD	api/sheet-file/download/{id}	music-sheet-file.download	App\Http\Controllers\MusicSheetFileController@download	api

Figura 17. Lista de Rutas de la API

CAPÍTULO 5: IMPLEMENTACIÓN Y EJECUCIÓN DE PRUEBAS

En este capítulo, abordamos la fase de pruebas para validar la solidez y confiabilidad del sistema. Siguiendo la metodología de Desarrollo Dirigido por Pruebas (TDD), llevaremos a cabo la implementación y ejecución de las pruebas diseñadas en el capítulo 3. Desde las pruebas de características, hasta un análisis de rendimiento. En paralelo a este proceso, se desarrollará la lógica de negocio incrustada en los controladores declarados en el capítulo anterior, actuando como el puente esencial entre la conceptualización del sistema y su materialización práctica.

5.1. Pruebas de Características

Aquí se explora la implementación de pruebas de características (Feature Testing) para evaluar el comportamiento de extremo a extremo del sistema. Se describen los casos de prueba y se siguen las historias de usuario definidas en el Capítulo 3, para verificar la integración adecuada de los componentes y la satisfacción de los requisitos del usuario. Entonces, entramos en el ciclo TDD, donde se escoge un criterio de aceptación muy simple y se traduce a una prueba.

A continuación, se muestra el proceso de implementación de las pruebas, junto con el desarrollo de la lógica de negocio. Se tomará como ejemplo el registro de partituras para simplificar este procedimiento.

Laravel nos brinda un comando artisan para crear pruebas, por lo que ejecutando en la terminal ‘`php artisan make:test NombrePruebaTest`’, en este caso nombramos la clase de la prueba como ‘`MusicSheetTest`’, el comando completo quedaría: ‘`php artisan make:test MusicSheetTest`’, generando el archivo `MusicSheetTest.php` ubicado en la carpeta `tests/Feature`, donde se incluirán todos los criterios de aceptación en forma de test para la funcionalidad a desarrollar. En principio el código generado se ve como sigue:

```php
class MusicSheetTest extends TestCase
{
    use DatabaseTransactions;
    /**
     * A basic feature test example.
     * @return void
     */
    public function test_example()
    {
        $response = $this->get('/');
        $response->assertStatus(200);
    }
}
```


Recordando el diseño del Test 4:

Test para registrar partituras

Nombre: Registrar partituras en el sistema

Descripción: Validar que los datos de entrada cumplan con los requerimientos necesarios para ser almacenados en la base de datos.

- Los campos requeridos no deben llegar nulos.

- Un título perteneciente a un autor no debe estar repetido.
- En el sistema tienen que existir registros almacenados de género musical y autor de la partitura.

Primer criterio de aceptación: Los campos requeridos no deben llegar nulos.

- Los campos Título (`‘tilte’`), id del autor (`'authorId'`), id del género musical (`'genderId'`), los datos de la ubicación física de la partitura como el id del estante (`'cabinetId'`) y el id del la gaveta (`'drawerId'`), así como la cantidad (`'quantity'`) de partituras a registrar, son requeridos, por lo que se escribe una prueba con estas característica con el nombre de `‘required_fields_cannot_be_null’` y adañaade a la clase `‘MusicSheetTest’`:


```php
/** @test
     * Los campos requeridos no pueden ser nulos
    */
    public function required_fields_cannot_be_null()
    {
        $responseUser = $this->getAuthenticated();
        $this->assertAuthenticated();


        //Dat
        $response = $this->postJson('/api/music-sheets', [
            'title' => null,
            'authorId' => null,
            'genderId' => null,
            'cabinetId' => null,
            'drawerId' => null,
            'quantity' => null,
        ]);
```

```
    // Assert
    // El código 422 es para validaciones fallidas
    $response->assertStatus(422);
    $response->assertJsonValidationErrors([
      'title', 'authorId', 'genderId', 'cabinetId',
      'drawerId', 'quantity'
    ]);
  }
```

Este test utiliza el método `postJson` para simular una solicitud de registro de partitura con campos nulos usando la ruta `'/api/music-sheets'` la cual hace un llamado al método `store` en el controlador `'MusicSheetController'` y verifica que la respuesta tenga un estado HTTP 422 (falla de validación) y que los errores de validación específicos se encuentren en la respuesta JSON.

- Si intentamos probar este test, corriendo el comando de artisan `'php artisan test'` desde la terminal, podemos verificar que la prueba falla, debido a que no se han definido las reglas de validación en el controlador `'MusicSheetController'` (Figura 18. a).

```
FAIL Tests\Feature\MusicSheetTest
✓ example
× required fields cannot be null
```

Figura 18. a. Red Test – primer criterio de aceptación

- Definimos las reglas de validación que deberán estar contenidas en el controlador `'MusicSheetController'`, es decir, escribimos un poco de código para hacer que esta prueba pase:

```php
    // Get the validation rules that apply to the request.
    // @return array
    public function rules()
    {
        return [
            'title'             => ['required'],
            'authorId'          => ['required'],
            'genderId'          => ['required'],
            'drawerId'          => ['required'],
            'cabinetId'         => ['required'],
            'quantity'          => ['required'],
        ];
    }
```

- Corremos de nuevo el test para comprobar que realmente funcionan las reglas de validación añadidas al controlar y hacer que la prueba pase (Figura 18. b):

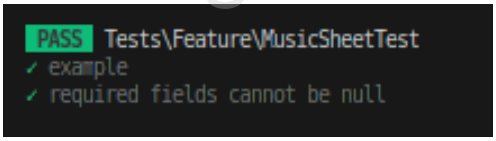


Figura 18. b. Green Test – primer criterio de aceptación

- En este caso no es necesario refactorizar el código en el controlador, puesto que todo se ve bien hasta este punto.

Segundo criterio de aceptación: Un título perteneciente a un autor no debe estar repetido.

- Aquí debemos validar que no se permita registrar de manera repetida el título de una partitura perteneciente a un autor, es decir, que el par título-autor deben ser únicos. Traducimos este criterio de aceptación a una prueba.

```php
//  Un título perteneciente a un autor no debe estar repetido
  public function title for same author cannot be repeated()
  {
      $responseUser = $this->getAuthenticated();
      $this->assertAuthenticated();
      /**
       * Preparación: Crear un autor y una partitura con un título
      */
      $author = Author::find(1);
      MusicSheet::create([
          'title' => 'Partitura Existente',
          'author id' => $author->id,
      ]);

      /** Act: Intentar crear una nueva partitura
       * con el mismo título y autor*/
      $response = $this->postJson('/api/music-sheets', [
          'title' => 'Partitura Existente',
          'authorId' => $author->id,

          'genderId' => 1,
          'cabinetId' => 1,
          'drawerId' => 1,
          'quantity' => 10,
      ]);

       /** Assert: Verificar que la respuesta indique
       * una falla de validación */
      $response->assertStatus(422);

      /**Verificar que el error de validación
       * específico está presente en la respuesta JSON */
      $response->assertJsonValidationErrors(['title']);
  }
```


Este test simula el intento de registrar una nueva partitura con un título que ya existe para el mismo autor. La prueba verifica que la aplicación responde con un código de estado 422 (falla de validación) y que el error de validación específico para el campo 'title' está presente en la respuesta JSON.

- Como es de suponer, al intentar ejecutar este test, fallará, debido a que no se ha creado una regla de validación que tome en cuenta este criterio (Figura 19. a):

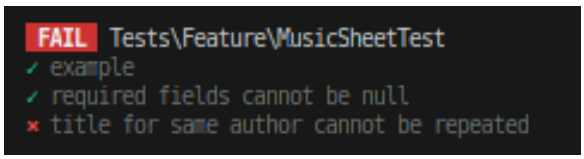


Figura 19. a. Red Test – segundo criterio de aceptación

- De la misma manera que en el criterio anterior, se debe escribir el código necesario para hacer que la prueba pase, en este caso, se agrega la regla de validación correspondiente `'"title" => Rule::unique('music_sheets', 'title')->where('author_id', $this->authorId)`.

```
public function rules()
    {
        return [
            'title'          => ['required', Rule::unique('music_sheets',
'title')->where('author_id', $this->authorId)],
            'authorId'       => ['required'],
            'genderId'       => ['required'],
            'drawerId'       => ['required'],
            'cabinetId'      => ['required'],
            'quantity'       => ['required'],
        ];
    }
```


- El siguiente paso es verificar que efectivamente la prueba pasa, al agregar el código mínimo necesario (Figura 19. b)

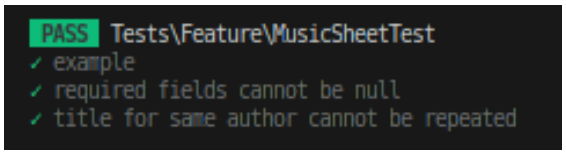


Figura 19. b. Green Test – segundo criterio de aceptación

- El paso de refactorización del código se ha cumplido en este caso, debido a que se ha modificado el método `Rules` del controlador `'MusicSheetController'.`

Tercer criterio de aceptación: En el sistema tienen que existir registros almacenados de género musical y autor de la partitura.

- Para probar que en el sistema existan registros almacenados de género musical y autor de la partitura antes de intentar registrar una nueva partitura, se crea la prueba correspondiente agregándola a la clase ‘`MusicSheetTest`’ con el nombre ‘`system_must_have_records_of_music_gender_and_author`’.

```
    /** En el sistema tienen que existir registros almacenados
     * de género musical y autor de la partitura. */
    public function
system_must_have_records_of_music_gender_and_author()
    {
        $responseUser = $this->getAuthenticated();
        $this->assertAuthenticated();
        // Act: Intentar crear una nueva partitura sin género musical y
autor existentes
        $response = $this->postJson('/api/music-sheets', [
            'title' => 'Nueva Partitura',
            // Supongamos que el ID dado para para género y autor
            //no existe en la base de datos
            'authorId' => 500,
            'genderId' => 200,
            // Otros campos necesarios para pasar la validación
            'cabinetId' => 1,
            'drawerId' => 1,
            'quantity' => 10,
        ]);

        // Assert: Verificar que la respuesta indique una falla del servidor
        $response->assertStatus(500);
    }
```


- Ejecutamos este test, y verificamos que la prueba falla (Figura 20. a)

```
FAIL Tests\Feature\MusicSheetTest
✓ example
✓ required fields cannot be null
✓ title for same author cannot be repeated
× system must have records of music gender and author
```

Figura 20. a. Red Test – tercer criterio de aceptación

- El siguiente paso es agregar a la función `store` del controlador el código necesario para hacer esta prueba pase:

```php
public function store(Request $request)
{
    $this->validate(
        $request,
        $this->Rules(),
        $this->Messages()
    );
    //se busca en base de datos los registros
    // de género musical y autor
    $authorId = Author::find($request->authorId);
    $genderId = Gender::find($request->genderId);

    return response()->json(['message' => 'success'], 200);
}
```

Intentamos buscar en base de datos el ID del género musical y del autor y retornamos un estatus 200 sí todo salió bien, de lo contrario se espera un estatus 500, que será un error de servidor sí no hay éxito en la búsqueda.

- ejecutamos nuevamente el test para verificar que esta prueba funciona (Figura 20. b)

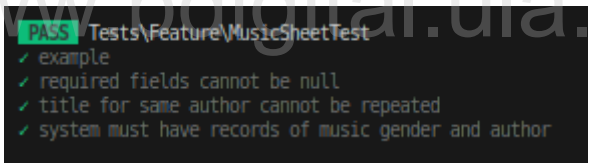


Figura 20. b. Green Test – tercer criterio de aceptación

En última instancia, se implementó una prueba general que verifica que se puede registrar una partitura musical con éxito.

Cuarto criterio de aceptación: Almacenar una nueva partitura con éxito.

- En este test se verifica que el usuario se ha autenticado, se envía una solicitud `POST` a la ruta `'api/music-sheets'` con datos simulados para almacenar una nueva partitura. Se asegura que la respuesta `HTTP` tenga un código de estado `200`, indicando un procesamiento exitoso de la solicitud y por último se verifica que la respuesta

JSON tenga la estructura esperada, que debe incluir las claves `'item'` y `'message'`.

```php
/** @test
 * Almacenar una nueva partitura.
 */
public function test_store_new_music_sheet()
{
    // Autenticación del usuario logueado
    $responseUser = $this->getAuthenticated();
    $this->assertAuthenticated();

    // Intento de almacenar una nueva partitura
    $response = $this->post('api/music-sheets', [
        'title' => 'Test Title',
        'authorId' => 1,
        'genderId' => 1,
        'quantity' => 10,
        'cabinetId' => 1,
        'drawerId' => 1,
    ]);

    // Verificación del estado de la respuesta HTTP
    $response->assertStatus(200);

    // Verificación de la estructura de la respuesta JSON
    $response->assertJsonStructure(['item', 'message']);
}
```


- Ejecutamos la prueba para verificar que falla (Figura 21. a)

```
FAIL Tests\Feature\MusicSheetTest
✓ example
✓ required fields cannot be null
✓ title for same author cannot be repeated
✓ system must have records of music gender and author
✖ store new music sheet
```

Figura 21. a. Red Test – cuarto criterio de aceptación

- Agregamos al controlador `'MusicSheetController'`, específicamente en la función `store`, la lógica y el código necesario para hacer que esta prueba pase, es decir, refactorizar la función ya existente, quedando de la siguiente manera.

```php
public function store(Request $request)
{
    $this->validation(
        $request,
        $this->Rules(),
        $this->Messages()
    );
    // Se busca el autor
    $authorId = Author::find($request->authorId);
    // Se busca el género musical
    $genderId = Author::find($request->authorId);
    // Se crea una nueva instancia de partitura musical
    $musicSheet = new MusicSheet();
    // Se crea una nueva instancia de ubicación
    $location = new Locations();

    // Se almacenan los datos de la partitura
    $musicSheet->title = $request->title;
    $musicSheet->author_id = $authorId->id;
    $musicSheet->gender_id = $genderId->id;
    $musicSheet->quantity = $request->quantity;
    $musicSheet->available = $request->quantity;
    // Se almacenan los datos de la ubicación de la partitura
    $location->cabinet_id = $request->cabinetId;
    $location->drawer_id = $request->drawerId;
    $location->save();

    // Se almacena el ID de la ubicación
    $musicSheet->location_id = $location->id;
    // Se guarda la partitura
    $musicSheet->save();

    return response()->json([
        'item' => $musicSheet,
        'message' => 'success']
        , 200);
}
```


- Verificamos que este código funciona, ejecutando de nuevo el test para hacer que pase (Figura 21. b):

```
PASS Tests\Feature\MusicSheetTest
✓ example
✓ required fields cannot be null
✓ title for same author cannot be repeated
✓ system must have records of music gender and author
✓ store new music sheet
```

Figura 21. b. Green Test – cuarto criterio de aceptación

- El siguiente paso implica la refactorización del código, especialmente para mejorar la seguridad durante el almacenamiento en la base de datos del registro de partituras musicales. Dado que este proceso implica varios modelos, optamos por encapsular toda la lógica en una transacción de base de datos. Esta elección nos proporciona la garantía de un almacenamiento coherente, ya que enfrentamos múltiples consultas a base de datos de forma secuencial.

```php
public function store(Request $request)
{
    // Se validan los datos
    $this->validation(
        $request,
        $this->Rules(),
        $this->Messages()
    );

    $musicSheet = DB::transaction(function () use ($request) {
       // Se busca el autor
        $authorId = Author::find($request->authorId);
        // Se busca el género musical
        $genderId = Author::find($request->authorId);
        // Se crea una nueva instancia de partitura musical
        $musicSheet = new MusicSheet();
        // Se crea una nueva instancia de ubicación
        $location = new Locations();

        // Se almacenan los datos de la partitura
        // ... Resto de la lógica ….
        $musicSheet->save();

        return $musicSheet;
    });

    return response()->json([
        'item' => $musicSheet,
        'message' => 'success']
        , 200);
}
```


- Ahora ejecutamos todos los test implementados y verificamos que todo sigue funcionando después del cambio en el código (Figura 21. c):

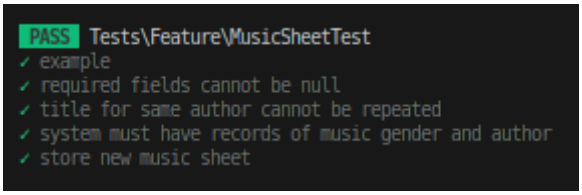


Figura 21. c. Green Test – todos los test de la función `store`

- Continuando en este ciclo de desarrollo, podemos refactorizar el código para incluir un bloque try-catch. Esto es necesario para gestionar eficazmente excepciones durante la ejecución, mejorando la robustez y la capacidad de respuesta del sistema frente a posibles errores imprevistos.

```php
public function store(Request $request)
{
    // Se validan los datos
    $this->validation(
        $request,
        $this->Rules(),
        $this->Messages()
    );

    try{
        $musicSheet = DB::transaction(function () use ($request) {
            // Se crea una nueva instancia de partitura musical
            $musicSheet = new MusicSheet();
            // Se almacenan los datos de la partitura
            // ... Resto de la lógica ….
            $musicSheet->save();

            return $musicSheet;
        });

        return response()->json([
            'item' => $musicSheet,
            'message' => 'success']
            , 200);
    } catch (\Throwable $th) {
        return response()->json([
            'error' => $th->getMessage()]
            , 500);
    }
}
```


- Así podemos ejecutar los test nuevamente para verificar que todo sigue en su lugar.

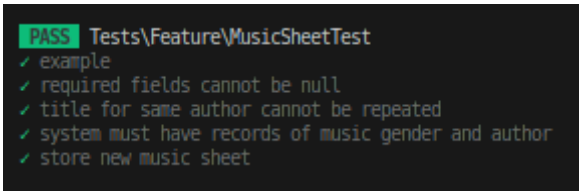


Figura 21. d. Última prueba (Green Test)

5.2 Pruebas de la API con Postman

La validación y prueba del API son elementos cruciales en el desarrollo de cualquier aplicación. Para asegurar que nuestro sistema de gestión de partituras cumple con los requisitos y ofrece una interfaz robusta, se empleó Postman, una herramienta versátil para el desarrollo de APIs. A continuación, se describe cómo se utilizó Postman para probar el API del sistema:

4.2.1 Configuración del Entorno

Antes de realizar las pruebas, se configuró un entorno en Postman para reflejar las diferentes variables y configuraciones que el API utiliza. Esto incluyó la URL base del API, claves de autenticación y cualquier otro parámetro necesario para las solicitudes.

4.2.2 Creación de Solicitudes

Postman permite crear y enviar solicitudes HTTP de manera sencilla. Se crearon diversas solicitudes para cada uno de los endpoints del API del sistema de gestión de partituras. Estas solicitudes incluyen operaciones como iniciar sesión, gestionar partituras y ejecutar acciones específicas del sistema, entre otras.

4.2.3 Gestión de Autenticación

Dado que nuestro sistema incorpora autenticación, se emplearon las funciones de Postman para gestionar diferentes métodos de autenticación, incluyendo el uso de tokens API para asegurar solicitudes protegidas.

4.2.4 Pruebas de Rendimiento

Postman permite realizar pruebas de rendimiento, evaluando cómo el API maneja cargas de trabajo diversas. Se llevaron a cabo pruebas para determinar la eficiencia y la escalabilidad del sistema bajo diferentes condiciones de carga.

4.2.5 Colecciones de Postman

Todas las solicitudes y escenarios de prueba se organizaron en colecciones de Postman. Estas colecciones proporcionan una estructura organizada para ejecutar pruebas individuales o suites completas, facilitando la repetición de pruebas durante el desarrollo y después de implementaciones importantes.

4.2.6 Documentación del API

Postman también facilita la generación de documentación del API. Se aprovechó esta funcionalidad para crear documentación clara y accesible que detalla cada endpoint, sus parámetros y las respuestas esperadas (*Promusica-ULA API*, s. f.).

El uso integral de Postman no solo aseguró la funcionalidad correcta de nuestra API, sino que también simplificó la colaboración entre desarrolladores y garantizó la consistencia en las pruebas a lo largo del desarrollo.

CAPÍTULO 6: ANÁLISIS Y RESULTADOS

En este capítulo, se presenta un análisis exhaustivo de los resultados obtenidos durante el desarrollo del sistema, así como una evaluación de la metodología de Desarrollo Dirigido por Pruebas (TDD) aplicada en el proceso.

6.1 Evaluación de los Resultados de las Pruebas de Rendimiento

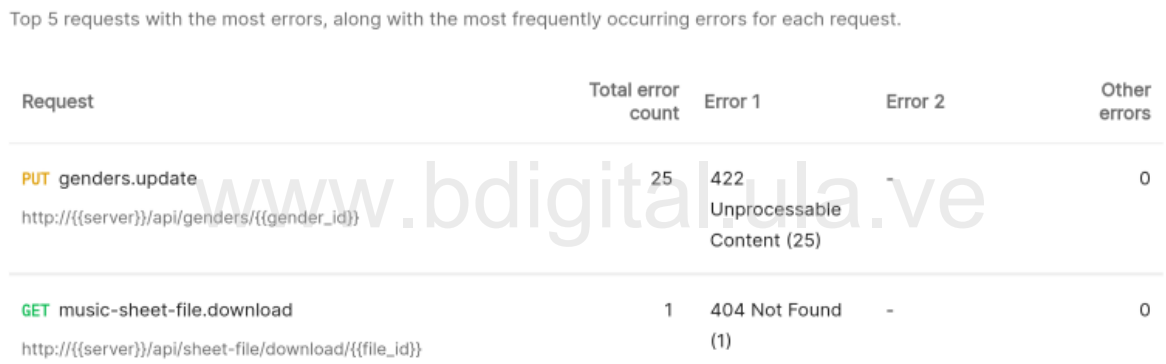
Durante la fase de pruebas, se realizaron pruebas exhaustivas de rendimiento de la API utilizando la herramienta Postman. Los resultados obtenidos proporcionaron una visión general del rendimiento del sistema bajo carga. Se observó un rendimiento satisfactorio, con una tasa de rendimiento promedio de 21.04 solicitudes por segundo y un tiempo de respuesta promedio de 408 milisegundos. Además, la tasa de error fue del 1.86%, lo que indica una buena estabilidad del sistema bajo carga. Se identificaron algunas rutas con tiempos de respuesta más lentos, lo que sugiere áreas potenciales de optimización para mejorar el rendimiento general del sistema (20User-1Min-Rampa-Promusica-ULA-Performance-Report-10000-1, 2023).

A continuación, se detallan los resultados de un ejemplo de prueba aplicada:

Performance test:

- Usuarios Virtuales: Se simularon 20 usuarios virtuales.
- Duración: La prueba tuvo una duración de 1 minuto.
- Perfil de Carga: Se implementó una carga en rampa durante 1 minuto.

- Total de solicitudes enviadas: Durante la prueba, se enviaron un total de 1397 solicitudes a la aplicación.
- Rendimiento (Throughput): La tasa de rendimiento fue de 21.04 solicitudes por segundo.
- Tiempo de Respuesta Promedio: El tiempo promedio de respuesta del sistema a una solicitud fue de 408 milisegundos.
- Tasa de Error: La tasa de error fue del 1,86%, lo que indica que aproximadamente el 1,86% de las solicitudes generaron algún tipo de error, siendo el más común el error 422 (Figura 22).



Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
PUT genders.update http://{{server}}/api/genders/{{gender_id}}	25	422 Unprocessable Content (25)	-	0
GET music-sheet-file.download http://{{server}}/api/sheet-file/download/{{file_id}}	1	404 Not Found (1)	-	0

Figura 22. Solicitudes con más errores.

Estos resultados proporcionan una visión general del rendimiento del sistema bajo carga simulada. Si bien los datos son considerados aceptables, existen áreas de mejora identificadas durante el análisis. Por ejemplo, se detectaron cinco rutas con respuestas más lentas (Figura 23), lo que sugiere la necesidad de optimización en esas áreas específicas.

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST music-sheets.store http://{{server}}/api/music-sheets	431	703	745	863	60	863
DELETE music-sheets.destroy http://{{server}}/api/music-sheets/{{music_sheet_id}}	429	705	740	803	56	803
DELETE genders.destroy http://{{server}}/api/genders/{{gender_id}}	427	736	756	869	56	869
PUT authors.update http://{{server}}/api/authors/{{author_id}}	426	707	782	836	52	836
DELETE authors.destroy http://{{server}}/api/authors/{{author_id}}	426	705	746	830	63	830

Figura 23. Top 5 de las solicitudes más lentas.

Además, al analizar los percentiles 90, 95 y 99, se observa que la mayoría de las solicitudes (hasta el 99%) se completan en 863 milisegundos o menos (Figura 24). Estas métricas estadísticas indican el tiempo necesario para que una cierta fracción de las solicitudes se complete, lo que permite identificar áreas que pueden ser optimizadas para mejorar el rendimiento general de la aplicación.

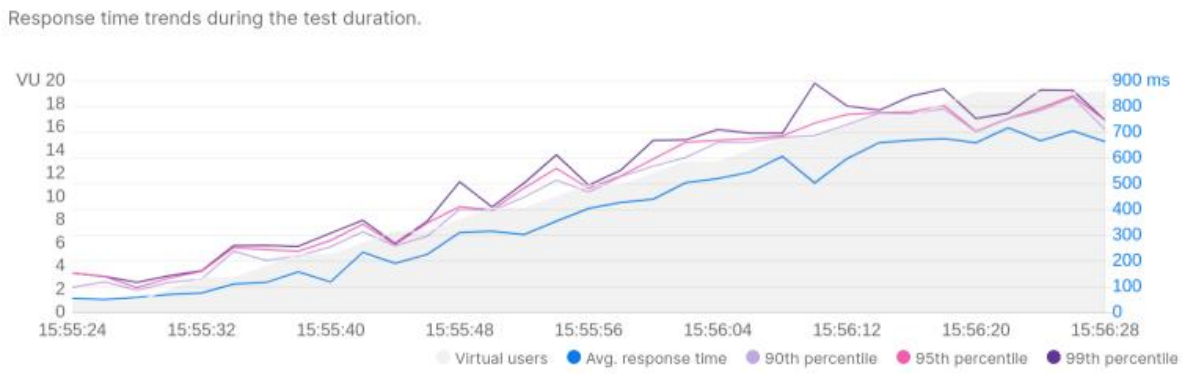


Figura 24. Tendencia del tiempo de respuesta durante la prueba.

En consideración de estos resultados, se plantean posibles mejoras y optimizaciones en el código y la configuración del sistema. Para una aplicación web que se estima tenga una carga constante de consultas, estas respuestas están dentro de los parámetros que se consideraron, si bien se pueden hacer optimizaciones de código y da una visión inicial para un despliegue en pruebas, como por ejemplo usar una base de datos como REDIS para el manejo de sesiones y caché, como servidor web usar NGINX que puede manejar el doble de conexiones que APACHE y es más flexible al momento de configurar.

6.2 Impacto en la Eficiencia del Registro y Préstamo de Partituras

El sistema propuesto tiene el potencial de mejorar significativamente la eficiencia en los procesos de registro y préstamo de partituras en la Fundación Promúsica. Mediante la automatización de tareas y la optimización de los flujos de trabajo, se espera reducir los tiempos de respuesta y minimizar los errores humanos. Esto llevará a una gestión más eficiente de las partituras y una mejor experiencia para los usuarios finales.

6.3 Análisis de la Metodología TDD

La aplicación de la metodología TDD durante el desarrollo del sistema demostró ser efectiva para garantizar la calidad y confiabilidad del código. La escritura de pruebas antes de la implementación del código permitió detectar y corregir errores tempranamente, lo que contribuyó a un desarrollo más ágil y robusto. Sin embargo, se encontraron algunos desafíos, como la curva de aprendizaje inicial y la necesidad de mantener un conjunto completo de pruebas a lo largo del ciclo de desarrollo.

6.4 Potenciales Áreas de Mejora

A pesar de los resultados satisfactorios, se identificaron algunas áreas en las que el sistema podría ser mejorado antes de su implementación. Esto incluye la optimización de las rutas con tiempos de respuesta más lentos, la mejora de la escalabilidad y la incorporación de funcionalidades adicionales según las necesidades específicas de la Fundación Promúsica. Se recomienda realizar una evaluación continua del sistema y realizar ajustes según sea necesario para garantizar su eficacia a largo plazo.

6.5 Plan de Implementación y Futuras Direcciones

Se propone un plan detallado para la implementación del sistema en la Fundación Promúsica, que incluye la asignación de recursos, la capacitación del personal y el cronograma de implementación. Además, se discuten posibles mejoras futuras, como la integración de tecnologías adicionales para mejorar el rendimiento y la funcionalidad del sistema a medida que evolucionan las necesidades de la organización.

El análisis y los resultados presentados en este capítulo respaldan la viabilidad y efectividad del sistema propuesto, así como proporcionan recomendaciones prácticas para su implementación exitosa y su mejora continua en el futuro.

CONCLUSIONES Y RECOMENDACIONES

La implementación de un sistema de gestión administrativa basado en la metodología de desarrollo dirigido por pruebas (TDD) ha demostrado ser una estrategia efectiva para garantizar la calidad y un mejor servicio a nuestra comunidad musical. La adopción de este enfoque ha permitido una implementación cuidadosa y una validación continua del sistema, asegurando su eficiencia y confiabilidad. Este proyecto ha logrado abordar de manera efectiva los desafíos existentes en la gestión manual de la extensa colección de partituras de la Fundación Promúsica.

El enfoque específico del estudio en el desarrollo de un sistema de gestión administrativa para la Fundación Promúsica ha permitido una atención dedicada a las necesidades particulares de esta organización. Al limitar el alcance a una única entidad, se ha facilitado la identificación precisa de requerimientos y la personalización del sistema según las especificaciones de la fundación. La delimitación del sistema a la gestión de partituras ha permitido una mayor focalización en las funcionalidades relevantes. Al no incluir elementos adicionales de gestión como recursos humanos o finanzas, se ha logrado una mayor claridad y eficiencia en el diseño y desarrollo del sistema, asegurando que cumpla con los objetivos establecidos de manera específica. Si bien el estudio no considera la infraestructura tecnológica disponible en la Fundación Promúsica, se asume que se cuenta con los recursos necesarios para la implementación del sistema. Esta delimitación resalta la importancia de tener en cuenta las capacidades y limitaciones tecnológicas de la organización al planificar e implementar soluciones tecnológicas, y destaca la necesidad de asegurar que se disponga de los recursos adecuados para el éxito del proyecto.

La configuración cuidadosa del entorno de desarrollo, empleando herramientas como Visual Studio Code y Git para la gestión del código fuente, ha desempeñado un papel

fundamental en la mejora notable de la eficiencia y calidad del proceso de desarrollo. Estas herramientas no solo proporcionan un entorno de trabajo robusto y colaborativo, sino que también simplifican tanto la creación como el mantenimiento del sistema. Además, la elección del lenguaje de programación PHP 8.0 y el framework Laravel ha resultado ser una decisión acertada para el desarrollo del sistema, permitiendo una implementación ágil y eficaz de las funcionalidades requeridas. Específicamente, Laravel ha demostrado brindar una estructura sólida y un conjunto de herramientas poderosas que han facilitado enormemente la creación de la API del sistema y la gestión de la base de datos.

La elección de Postgres como sistema de gestión de base de datos, junto con el uso del ORM Eloquent de Laravel, ha simplificado notablemente la interacción con la base de datos y ha proporcionado una capa de abstracción que facilita el manejo de datos de manera elegante y eficiente.

Asimismo, la configuración del entorno local utilizando 'php artisan serve' de Laravel ha permitido la ejecución de pruebas unitarias de manera eficiente y automatizada, garantizando un entorno uniforme y controlado para las pruebas.

El desarrollo de la lógica del sistema, que incluye la generación de migraciones, la creación de modelos y controladores, así como la configuración de las rutas para construir la API del sistema, ha establecido las bases para la funcionalidad completa de la aplicación. Estos pasos son cruciales para establecer la infraestructura necesaria y asegurar el correcto funcionamiento del sistema en su totalidad.

La ejecución de las pruebas realizadas mediante la herramienta Postman ha revelado resultados prometedores en cuanto al rendimiento, eficiencia y fiabilidad del sistema propuesto para la gestión administrativa en la Fundación Promúsica. Aunque se identificaron áreas de mejora, tales como la optimización de ciertas rutas y la incorporación de tecnologías adicionales; las pruebas de rendimiento, la metodología TDD y el plan detallado de

implementación respaldan la viabilidad y efectividad del proyecto. Se recomienda una evaluación continua y ajustes según sea necesario para garantizar el éxito a largo plazo del sistema en su objetivo de mejorar la gestión de partituras y la experiencia de los usuarios finales en la Fundación Promúsica.

REFERENCIAS

About Postman. (2023). Postman API Platform. https://www.postman.com/company/about-postman/

Alarcón, V. F. (2006). Desarrollo de sistemas de información: una metodología basada en el modelado. Ediciones UPC eBooks. https://dialnet.unirioja.es/servlet/libro?codigo=298995

An Effective Requirement Engineering Process Model for Software Development and Requirements Management. (2010b, octubre 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/5656776/

Auth. (s. f.). JSON Web Tokens. Auth0 Docs. https://auth0.com/docs/secure/tokens/json-web-tokens

AWS. (s. f.). ¿Qué es SQL? - Explicación de Lenguaje de consulta estructurado (SQL). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/sql/#:~:text=es%20importante%20SQL%3F-,El%20lenguaje%20de%20consulta%20estructurada%20(SQL)%20es%20un%20lenguaje%20de,los%20diferentes%20lenguajes%20de%20programaci%C3%B3n

Bass, L., Clements, P., y Kazman, R. (2012). Software Architecture in Practice. Addison-Wesley.

Blé Jurado, C. (2010). Diseño Ágil con TDD. Creative Commons. https://www.academia.edu/38401326/Diseno_Agil_Con_TDD

Britannica, T. Editors of Encyclopaedia (2021). client-server architecture. Encyclopedia Britannica. https://www.britannica.com/technology/client-server-architecture

Cevallos Muñoz, F. D. (2022). Propuesta de buenas prácticas de seguridad para creación, transporte y almacenamiento de JSON web token. [Tesis de pregrado, Pontificia Universidad Católica del Ecuador Sede Ambato]. https://repositorio.pucesa.edu.ec/bitstream/123456789/3505/1/77667.pdf

Clark, J. (2021, 13 septiembre). Los 10 mejores marcos de trabajo de Backend. Back4App Blog. https://blog.back4app.com/es/los-10-mejores-marcos-de-trabajo-de-backend/

Cruz, Y. E., Zamora, C., Paz, C., y Jorge, R. E. (2020). Adopción de tecnologías de gestión de procesos de negocio: una revisión sistemática. Ingeniare. Revista chilena de ingeniería, 28(1), 41-55. https://doi.org/10.4067/s0718-33052020000100041

DevDocs. (s. f.). HTTP documentation. https://devdocs.io/http/

Deloitte Spain (2023). ¿Qué es un ORM? https://www2.deloitte.com/es/es/pages/technology/articles/que-es-orm.html

Documentación Laravel en español. (s. f.-b). El framework de PHP para artesanos de la WEB. https://documentacionlaravel.com/docs/9.x/eloquent

Dowsett, C. (2022). What Is a Database? Built In. https://builtin.com/data-science/database

Eseme, S. (2021). Introduction to Backend Development - Backend Developers - Medium. Medium. https://medium.com/backenders-club/introduction-to-backend-development-3f3464afd815

Fuentes, J. P. (2021, 13 mayo). TDD: Desarrollo guiado por pruebas – Trifulcas. https://trifulcas.com/tdd-desarrollo-guiado-por-pruebas/

Gavilán, C. M (2008). SIGB. Catálogos y gestión de Autoridades. Diseño y prestaciones de OPACs. http://eprints.rclis.org/13188/1/sigb.pdf

Google. (s. f.). Cómo utiliza Google las cookies. Privacy & Terms – Google. https://policies.google.com/technologies/cookies?hl=es

Gutiérrez A., E. R. (2020). Sistema de gestión y digitalización bibliotecaria. http://repositorio.upea.bo/handle/123456789/96

Heidi, E. (2021, 3 febrero). What is Laravel? DigitalOcean Community. https://www.digitalocean.com/community/tutorials/what-is-laravel

Herranz, J. I. (2023, 13 abril). TDD como metodología de diseño de software. Paradigma Digital. https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

IBM. (s. f.-b). ¿Qué son los contenedores? https://www.ibm.com/es-es/topics/containers

IBM. (2021). What is a relational database? https://www.ibm.com/topics/relational-databases

IBM Documentation. (2021). https://www.ibm.com/docs/es/aix/7.1?topic=systems-client-server

iKenshu. (2019, 26 abril). ¿Qué es Kubernetes? Platzi. https://platzi.com/blog/que-es-kubernetes/?utm_source=google&utm_medium=cpc&utm_campaign=20290685455&utm_adgroup=&utm_content=&gclid=CjwKCAiAxreqBhAxEiwAfGfndDcbEg-y65QfwsUQ9KRHjhLHPjdQbhhS7wXP7ioeW91wA6HrjBEWPhoCmlgQAvD_BwE&gclsrc=aw.ds

InnovaciónDigital, R. (2022, 25 agosto). Docker: qué es y cómo funciona. Innovación Digital 360. https://www.innovaciondigital360.com/big-data/docker-que-es-y-como-funciona/?gclid=CjwKCAiAxreqBhAxEiwAfGfndH5YyRYDCbLG2TjCFHkewprjtrd1Mrin4khfCL6yBXi9l2Hl6YftHxoCWxsQAvD_BwE

Jin, B., Sahni, S., y Shevat, A. (2018). Designing Web APIs: Building APIs That Developers Love. "O'Reilly Media, Inc.". https://www.academia.edu/43452338/Designing_Web_APIs_BUILDING_APIS_THAT_DE VELOPERS_LOVE

Langer, A. M. (2018). Information technology and organizational learning: Managing behavioral change in the digital age. (3.a ed.). CRC Press Taylor & Francis Group. https://www.yourhomeworksolutions.com/wp-content/uploads/edd/2020/09/arthur_m._langer___information_technology_and_organizational_learning__managing_behavioral_change_in_the_digital_age_crc_press__2017___1_-1.pdf

Lilienthal, C. (2019). Sustainable Software Architecture: Analyze and Reduce Technical Debt. dpunkt.verlag.

Mikula, K. (2023, 30 agosto). The History and Evolution of APIs | Traefik Labs. Traefik Labs: Say Goodbye to Connectivity Chaos. https://traefik.io/blog/the-history-and-evolution-of-apis/

MVC - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN. (2022). https://developer.mozilla.org/es/docs/Glossary/MVC

O'Brien, J. A. y Marakas, M. H. (2006). SISTEMAS DE INFORMACIÓN GERENCIAL. https://www.academia.edu/91551151/SISTEMAS_DE_INFORMACION_GERENCIAL_OB rein_y_Marakas_McGraw_Hill

Overview of the Administrative Management Systems (AMS). (2016, 3 mayo). Financial Services. https://finance.utoronto.ca/policies/gtfm/financial-information-system-fis/overview-of-the- administrative-manaement-systems-ams/

Pérez, L. (2022). Diseño e Implementación de una aplicación para la gestión de partituras. https://ruc.udc.es/dspace/handle/2183/32088

PHP: What is PHP? - Manual. (s. f.-b). https://www.php.net/manual/en/intro-whatis.php

Promusica-ULA API. (s. f.). Promusica-ULA. https://documenter.getpostman.com/view/20766493/2s9YkgDkKZ

Ravoof, S. (2023, 17 febrero). What Is PostgreSQL? Kinsta®. https://kinsta.com/knowledgebase/what- is-postgresql/

Ross, J. W., Beath, C. M., y Mocker, M. (2019). Designed for Digital: How to Architect Your Business for Sustained Success. The MIT Press.

Ruiz P., F. R. (2021, 31 enero). Desarrollo de un sistema de gestión de biblioteca en la Institución Educativa Técnico Industrial Pedro A. Oñoro de Baranoa. 10596/39010. https://repository.unad.edu.co/handle/10596/39010?locale-attribute=en

Smiraglia, R. P. (2001). The nature of "a work": Implications for the organization of knowledge. Lanham, Md: Scarecrow Press.

Stack Overflow. (s. f.). What is a software framework? https://stackoverflow.com/questions/2964140/what-is-a-software-framework

Sydle. (2023). ¿Qué es la digitalización de documentos y cómo hacerla? Blog SYDLE. https://www.sydle.com/es/blog/digitalizacion-de-documentos-61b8e03c876cf6271dfbe88a#:~:text=La%20digitalizaci%C3%B3n%20de%20los%20documentos,una%20realidad%20en%20varias%20empresas

The Editors of Encyclopaedia Britannica. (2023). Client-Server Architecture | Definition, Characteristics, & Advantages. Encyclopedia Britannica. https://www.britannica.com/technology/client-server-architecture

Visual Studio Code - Code editing. Redefined. (2021, 3 noviembre). https://code.visualstudio.com/

20User-1Min-Rampa-Promusica-ULA-Performance-Report-10000-1. (2023, 6 de diciembre). Google Docs. https://drive.google.com/file/d/12oXqBgWfsdBBDWK72WAY6v9WMgGY96U7/view?usp=sharing