DESARROLLO DE SOFTWARE PRODUCTO MÍNIMO VIABLE (MVP) PARA EMPRESA DE TAPETES KÁFE

MACEA BERNAL MARIA FERNANDA

INSTITUCIÓN UNIVERSITARIA PASCUAL BRAVO

FACULTAD DE INGENIERÍA

TECNOLOGÍA DESARROLLO DE SOFTWARE

MEDELLÍN

2022

DISEÑO Y DESARROLLO DE MVP PARA EMPRESA DE TAPETES

MACEA BERNAL MARIA FERNANDA

Trabajo de grado para optar al título de Tecnólogo en Desarrollo de Software

Asesor
Liliana María García Aguirre
Magister en Ingeniería de Software

INSTITUCIÓN UNIVERSITARIA PASCUAL BRAVO
FACULTAD DE INGENIERÍA
TECNOLOGÍA DESARROLLO DE SOFTWARE
MEDELLÍN
2022

Contenido

Pág.

Resumen 8
Abstract 9
Glosario 10
Introducción 11
1. Planteamiento del problema 12
1.1 Descripción 12
1.2 Formulación 12
2. Justificación 13
3. Objetivos 14
3.1 Objetivo general 14
3.2 Objetivos específicos 14
4. Marco teórico 15
4.1 Antecedentes 15
4.2 Bases teóricas: 17
4.2.2 Modelo vista controlador 17
5. Metodología 20
5.1 Tipo de proyecto 20
5.2 Método 20
5.3 Instrumentos de recolección de información 22
5.3.1 Fuentes primarias. 22
6. Resultados 23
6.1. Análisis 23
6.1.1. Requisitos 23
6.1.2. Diagramas UML 25
6.2 Diseño 27
6.2.1 Arquitectura de la aplicación web 27
6.2.2. Diagrama entidad relación 30
6.3 Implementación 31
7. Conclusiones 50

8. Recomendaciones 51
9. Referencias bibliográficas 52
10. Bibliografía 53

Lista de figuras

Pág.

Figura 1:Sitio web 16
Figura 2: Casos de usos 25
Figura 3: Diagrama de clases 26
Figura 4: Diagrama de clase seguridad 26
Figura 5:Arquitectura general 27
Figura 6: Arquitectura del frontend 28
Figura 7: Arquitectura del backend 29
Figura 8: Diagrama entidad relación 30
Figura 9: Estructura del backend 31
Figura 10: ProductController 32
Figura 11: Métodos de ProductController 33
Figura 12: UserController y sus métodos 34
Figura 13: DTO y sus métodos 35
Figura 14: ProductRequestDTO y sus validaciones 36
Figura 15: ExceptionManager 37
Figura 16: ProductService y sus métodos 38
Figura 17: UserService y sus métodos 39
Figura 18: UserRepository 40
Figura 19: Entidad User 40
Figura 20: Configuraciones de seguridad 41
Figura 21: SecurityConfig. Método configure 42
Figura 22: Authentication Filter 42
Figura 23: Authorization Filter. 43
Figura 24: PasswordEncoder Bean. 43
Figura 25: Estructura de carpetas frontend 44
Figura 26: App.vue 45
Figura 27: Template App-vue 46
Figura 28: Login 47
Figura 29: Token refresh y fetch de productos 48
Figura 30: Funcionalidad carrito de compras 48
Figura 31: Funcionalidad vaciar carrito y comprar carrito 49
Figura 32: Mensaje de compra de productos 49

Lista de tablas

Pág.

Tabla 1: Internet, web y página web 15
Tabla 2: Requisitos funcionales 23
Tabla 3: Requisitos no funcionales 24

Lista de anexos

Pág.

Anexo 1. Jira Software 54
Anexo 2. Capa lógica 55
Anexo 3. DTO 56
Anexo 4. Métodos 57
Anexo 5. HTTP Codes 58
Anexo 6. Trabajo final 59

Resumen

DESARROLLO DE SOFTWARE PRODUCTO MÍNIMO VIABLE (MVP) PARA EMPRESA DE TAPETES KÁFE

MACEA BERNAL MARIA FERNANDA

Actualmente se ha convertido en una necesidad que las empresas tengan presencia digital mediante aplicaciones web. Es por ello que, en este proyecto se realizará un sitio web para la empresa Káfe utilizando un patrón de diseño que se basa en una estructura en capas cuyo fin es separar el código de acuerdo a sus funcionalidades. Este patrón es conocido como Modelo Vista Controlador (MVC). Se va a utilizar esta estructura ya que es de las mejores alternativas que existen para la creación de software con procesos eficaces. Este estilo de arquitectura permite crear un sitio web escalable y con facilidad de mantenimiento a futuro, además de ser orientado a los clientes debido a su sencillo soporte.

Para este trabajo utilizamos diversas herramientas tecnológicas como: Vue.js, Java, Spring, MySQL, entre otras. En el presente informe se encuentra el uso que se le dio a las mismas para lograr el objetivo de obtener un sitio web para la empresa Kafe. Todo esto se logró utilizando la metodología ágil de SCRUM, esta nos permitió optimizar la ejecución del proyecto mediante la fragmentación de tareas.

Palabras claves: Modelo Vista Controlador, aplicación web, Producto Mínimo Viable, arquitectura web, Objeto de transferencia de datos.

Abstract

MINIMUM VIABLE PRODUCT SOFTWARE DEVELOPMENT (MVP) FOR KAFE RUG BUSINESS

MACEA BERNAL MARIA FERNANDA

Nowadays it has become a need for the business to have a digital presence through web applications. It is therefore, that in this project, a website will be created using a design pattern based on a tier architecture. This will allow us to break up the base code according to its functionalities. This pattern is known as the Model View Controller (MVC). This architecture will be used because it is one of the best alternatives to create software efficiently. This architecture allows us to create a scalable, maintainable, and client oriented website thanks to its simple support.

To complete this job, we will use different technologies, such as: Vue.js, Java with Spring, MySQL, among others. In this report, you will find the use of those technologies to accomplish the objective of creating a website for Kafe business. All of this has been accomplished using the SCRUM agile methodology. This methodology allowed us to optimize the project execution through task fragmentation.

Keywords: Model View Controller, web application, Minimum Viable Product, web architecture, data transfer object.

Glosario

Java: es un lenguaje de programación que será utilizado en este trabajo para la codificación de la aplicación web.

MVC: Modelo Vista Controlador. Se encarga de separar las interfaces de usuario, lógica y datos.

POO: programación orientada a objetos. Es un modelo de programación basado en clases y objetos

Káfe: empresa de lencería para hogar. Ofrece productos como alfombras, cojines, cobijas, entre otros.

DTO: Objeto de transferencia de datos, su propósito es transportar datos mediante la comunicación de procesos

Aplicación web: programa que funciona en internet y es ejecutado en un servidor web. Son accesibles mediante el navegador.

Introducción

El trabajo presente desarrolla una aplicación web para la empresa Kafe. Se hace porque mediante este se va a evidenciar un producto mínimo viable de software web para ventas digitales, las cuales son cada vez más necesarias en el mercado actual. Se realizará el levantamiento de requerimientos necesarios con los usuarios finales, se elaborará mediante la arquitectura en capas MVC y utilizando tecnologías de alto nivel como Java, Spring, MySQL, entre otras.

La metodología empleada para realizar este trabajo se basa en la herramienta de desarrollo ágil SCRUM, el marco de esta metodología permite aplicar un conjunto de buenas prácticas para obtener los mejores resultados posibles en un proyecto, pues los pilares de la misma son: la innovación, la competitividad, la flexibilidad y la productividad.
Un factor que también se tuvo en cuenta a la hora de elegir esta metodología es que permite obtener resultados con prontitud, lo cual es importante porque si las entregas se alargan esto implica más costos y menos calidad

Debido a que el entorno web es tan amplio y está en rigor, este trabajo se limita únicamente a programas web. No se tocarán temas relacionados con aplicaciones móviles o de escritorio. Nuestro enfoque especial por la web se debe a que la aplicación que se va a desarrollar está dirigida a los clientes de una empresa, y las aplicaciones web permiten que, desde cualquier lugar, el cliente pueda acceder sin tener que instalar nada y sin necesitar un sistema operativo determinado, además de que no se presentan problemas de incompatibilidad entre versiones. Es pocas palabras, las aplicaciones web se llevan el reconocimiento por su portabilidad, pues solo requieren un navegador que esté presente en cualquier dispositivo.

1. Planteamiento del problema

1.1 Descripción

La empresa Káfe inició sus operaciones en el año 2020 ofreciendo productos de tapetes como decoración para una variedad de espacios. En esta época la empresa solo vendía en su entorno más cercano, con el tiempo, empezó a utilizar las redes sociales para ampliar su mercado, y al día de hoy, ha crecido hasta el punto en que ya no se limita únicamente a la venta de tapetes sino de una variedad más de productos de lencería para los hogares y hace envíos a nivel nacional.

Sin embargo, la empresa ha dejado de crecer, tiene nueva competencia, por lo que requiere actualizarse en tecnología.

Además, actualmente las personas se están acostumbrando a consultar y realizar pedidos a través de plataformas web, generando más confianza en muchos de ellos y hay evidencias que este tipo de aplicaciones ayuda al incremento de las ventas, la productividad y el valor de sus servicios.

En otras palabras, hoy día no debe haber empresas sin sitio web, y es ahí donde está el problema de Káfe, pues una empresa sin web es sinónimo de desconfianza, una empresa sin web es una empresa sin futuro (Solera, 2021)

1.2 Formulación

¿El desarrollo de un software web para la empresa de tapetes Káfe, permitirá mejorar sus procesos, ser más competitivo y generar confianza en antiguos y nuevos clientes?

2. Justificación

Actualmente las empresas que no hacen parte de las últimas tendencias en tecnología como aplicaciones web y móviles han perdido su ventaja competitiva en su sector de mercado, perdiendo clientes potenciales, sin crecimiento por no promover sus productos y servicios e incluso han desaparecido. Es por eso que la empresa Káfe necesita contar con una aplicación web que le sirva como una vitrina digital de sus productos para incrementar sus ingresos, ayudar a sus clientes a mejorar los niveles de conocimiento de sus productos y de la propia empresa como tal, mantener satisfechos a sus actuales consumidores y captar nuevos, incrementar rentabilidad, reducir costos a futuro, estar al frente de la competencia con mayor facilidad de comunicación, imagen confiable e impacto en el marketing online.

Esta aplicación se realizará en una arquitectura MVC que permite dividir una aplicación en tres módulos para el manejo de la lógica del negocio (Modelo), los formularios y/o presentación (Vista) y el enlace entre ellos (Controlador).

3. Objetivos

3.1 Objetivo general

Desarrollar un producto mínimo viable de software web para las ventas y servicios de la empresa de tapetes Káfe, utilizando el lenguaje de programación Java Web.

3.2 Objetivos específicos

Realizar el levantamiento de los requerimientos con los usuarios finales que serían el gerente y auxiliar de la empresa de tapetes.

Utilizar la arquitectura de modelo vista controlador en el desarrollo de sitios web, ya que permitirá los mantenimientos y ajustes necesarios en el futuro.

Implementar la aplicación utilizando un lenguaje de programación de alto nivel y el sistema manejador de bases de datos MySQL

Diseñar la interfaz gráfica de acuerdo a los requerimientos y necesidades específicas del negocio, como colores institucionales entre otros.

4. Marco teórico

4.1 Antecedentes

Con el fin de entender de dónde nace este ilimitado campo de las aplicaciones web, se hace necesario investigar qué es internet y que es la web. Laviosa (2006) explica esto en su propuesta de diseño web para un jardín de infancia. Menciona que Internet es una “red de redes”, es decir, un sistema mundial de redes de computadoras, lo que significa que muchas redes operadas por una multitud de organizaciones están interconectadas. Por su lado, la Web, hace referencia a computadoras conectadas a Internet que pueden comunicarse entre sí utilizando el protocolo computacional HTTP. Todos los navegadores utilizan el HTTP para solicitar y recibir páginas Web de otras computadoras, es por ello que en todos los computadores se puede entender la información. Es importante resaltar la diferencia entre Internet y la Web, aunque ambos términos están relacionados, corresponden a dos nociones diferentes: mientras que Internet es la colección de redes entrelazadas o las computadoras conectadas unas a otras y asignadas a una dirección única; la web es la sección de Internet que contiene información multimedia, es una aplicación gigante, que utiliza la red Internet y hace posible el intercambio de información sobre esta red. Un Sitio Web es un conjunto de páginas Web que hacen referencia a un asunto en particular, normalmente incluye una página inicial de bienvenida, denominada home page, con un nombre de dominio y dirección en Internet específicos. Las instituciones, organizaciones e individuos las emplean para comunicarse con el mundo entero. En el caso de las empresas, tratan de demostrar a manera de tarjeta de presentación la oferta de sus bienes y servicios a través de Internet, y en general para promover sus funciones de publicidad y mercadeo (Laviosa, 2006).

Internet	Web	Página web - Sitio web
Red inmensa de computadoras conectadas entre sí alrededor de todo el mundo	Colección de páginas que se sitúan sobre esa red de computadoras	Una página es la unidad más básica de la Web. Un sitio web es una agrupación de páginas web

Tabla 1: Internet, web y página web

Fuente: diseño propio

Después de tener claros estos importantes términos, la pregunta qué surge es: ¿Cómo influye el internet y la web en el crecimiento para las empresas actualmente? Gómez y Aversano (2018) explican esto en su artículo Marketing en la web, donde mencionan que el crecimiento explosivo de internet y la tecnología ha cambiado radicalmente las tradicionales prácticas empresariales. Pues la web ha dado lugar al nacimiento del llamado marketing 2.0, digital u online, que incide en las acciones de comunicación de las empresas a nivel tecnológico. Explican que el sitio web es el pilar fundamental del marketing digital, pues allí la empresa ofrece y vende sus productos y servicios; tener un sitio web profesional genera la confianza suficiente para que los clientes potenciales se animen a tener relación comercial con la empresa. El sitio web debe ser fácil de navegar y tener elementos que faciliten la participación de los visitantes, estas investigaciones y antecedentes apoyan el proyecto de la aplicación web para Káfe y muestran la importancia de que este se ejecute.

Las estadísticas también muestran el incremento en la demanda de los sitios web, en el trabajo Arquitecturas de aplicaciones web, Guillén y Leandro (2019) hablan sobre cómo el tráfico web es el responsable de un buen porcentaje del tráfico de Internet. Esta tendencia ha ido creciendo gradualmente desde que apareció la web (protocolo HTTP), y hoy día el tráfico HTTP predomina respecto del resto de los protocolos, hay una gran población de usuarios navegantes que pueden generar una cantidad inmensa de peticiones si el contenido es interesante, tal y como muestra el siguiente gráfico

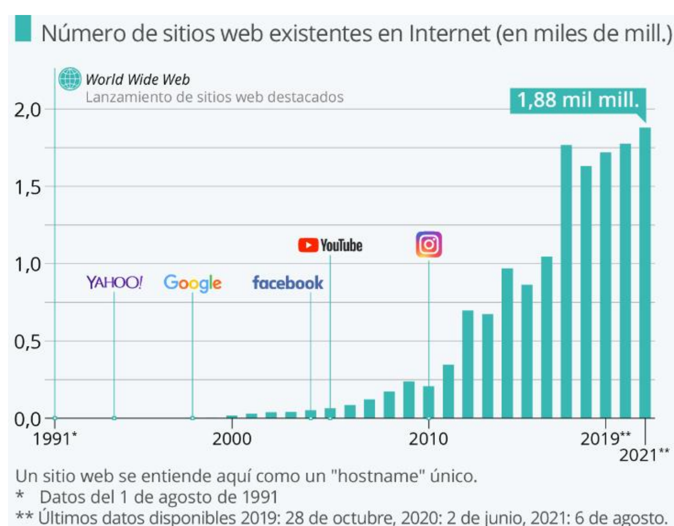


Figura 1:Sitio web

Fuente: Internet Live Stats

4.2 Bases teóricas:

4.2.2 Modelo vista controlador

La aplicación web para Káfe va a ser desarrollada siguiendo el patrón MVC. Buscando un poco de información histórica, es posible afirmar que el patrón Modelo/Vista/Controlador o MVC(Model/View/Controller) fue descrito por primera vez en 1979 por Trygve Reenskaug e introducido como parte de la versión Smalltalk-80 del lenguaje de programación Smalltalk. Hoy por hoy es usado en la enseñanza de las Universidades e Institutos de Desarrollo y también se aplica en el Desarrollo de Proyectos de Software.

Fue diseñado para reducir el esfuerzo de programación necesario en la implementación de sistemas. Cada una de las capas de este modelo son utilizadas como entidades separadas y esto permite que los cambios que se generen en la capa de Modelo puedan ser reflejados en todas las vistas.

Modelo: El modelo está conformado por un conjunto de clases las cuales se encargan de procesar la información que debe recibir el sistema.

Quien tiene la responsabilidad de acceder a la capa donde están los datos es el modelo, que normalmente es independiente del sistema de almacenamiento. El modelo es quien se encarga de definir las reglas o la funcionalidad que el sistema debe tener. Un ejemplo de regla es: "Si el tapete pedido no está en el almacén, consultar el tiempo de entrega estándar del proveedor". La responsabilidad de tener un registro de vistas y controladores del sistema también cae sobre el modelo y notifica a las vistas los cambios que en los datos pueda producir un agente externo.

Vista: Las vistas están conformadas por un conjunto de clases que muestran al usuario la información que hay en la capa modelo. Todas las vistas están asociadas a un modelo, o varias vistas pueden estar asociadas al mismo modelo, por ejemplo, se puede tener una vista mostrando la información de un tapete (medida, color, forma, cantidad, precio) y otra vista mostrando la misma información, pero con descuentos especiales. El modelo únicamente le da a una vista la información que necesita

En otras palabras, las vistas son una interfaz de usuario que tienen la tarea de mostrar la

información que se envía al cliente y de manejar los mecanismos de interacción con éste.

Controlador: El controlador es un objeto responsable de guiar el flujo de la aplicación mediante mensajes externos, es decir, los datos enviados por el usuario. Según el mensaje, el controlador modifica el modelo o abre y cierra vistas. Es importante aclarar que las vistas y el modelo no tienen acceso al controlador, mientras que el controlador si tiene acceso a las mistas y el modelo.

Los eventos de entrada los recibe el controlador es responsable de recibir los eventos de entrada, estos eventos pueden ser un click en un control como un botón. Muchas de estas acciones suponen peticiones al modelo o a las vistas. Un ejemplo de una petición a las vistas es: llamar al método "Actualizar()". Un ejemplo de una petición al modelo es: "Obtener_tiempo_de_entrega (nueva_orden_de_venta)".

Esta estructura basada en capas que separa el código en función de sus responsabilidades ha sido catalogada por muchos ingenieros como una de las mejores soluciones para mejorar los procesos de creación de software. Por lo tanto, el Modelo Vista Controlador (MVC) es eficaz para ayudarnos a crear aplicaciones de mayor calidad.

Este modelo es especialmente útil para la creación de sitios web como portafolios para empresas, pues facilita la escalabilidad y mantenimiento del sitio, su soporte es más sencillo y es orientado a los clientes. Este proyecto muestra cómo usar este modelo en un ejemplo práctico y real.

Modo de operación MVC

Hay diferentes formas de implementar MVC, sin embargo, el uso más común es el siguiente:

1. El usuario interactúa con la interfaz de usuario (mediante un control).

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la acción solicitada por el usuario y gestiona ese evento.

3. El controlador ingresa al modelo y se actualiza, normalmente de acuerdo a la acción solicitada

por el usuario.

4. El controlador le encarga la responsabilidad de desplegar la interfaz de usuario a los objetos de la vista, mientras que esta obtiene los datos del modelo para generar una interfaz apropiada para el usuario. El controlador puede proveer cierta interacción entre el modelo y la vista, así el modelo puede notificar cualquier cambio.

5. La interfaz de usuario espera nuevas interacciones y comienza de nuevo el ciclo.

5. Metodología

5.1 Tipo de proyecto

Como el objetivo de este proyecto es solucionar un problema real, concreto y práctico para la empresa Kafe, el tipo de metodología utilizada es la aplicada, ya que esta es la que se enfoca en emplear la ciencia a los problemas que se presentan en la realidad de las empresas, con el fin de mejorar los procesos productivos de las mismas.

5.2 Método

La metodología de software utilizada es SCRUM, siguiendo esta metodología se hicieron entregas parciales del producto final. El proceso comenzó con la elaboración del Sprint Backlog donde se recogieron un conjunto de tareas con sus respectivos requerimientos y funcionalidades utilizando la plataforma Jira Software (Anexo 1)

Esta metodología es utilizada con el objetivo de solucionar los problemas y necesidades de la empresa Kafe, se aplicará SCRUM mientras se diseña el sitio web para cumplir con los requerimientos deseados. La estrategia con la que se desarrollará la solución requiere una planificación de las necesidades específicas del cliente. En este caso, la empresa desea poder publicar sus productos y que estos tengan un mayor alcance en internet. Además, la empresa necesita que se pueda tener una especie de carrito de compras, un modo de autenticación de usuarios, y una forma de contacto con sus clientes para atenderlos de manera personalizada.

Luego de haber realizado esta primera etapa, con el objetivo de saciar las necesidades anteriormente mencionadas, se procederá a realizar el diseño de las arquitecturas necesarias para el funcionamiento de la aplicación. Además, se realizará la selección de la paleta de colores que llevará el sitio web, los logotipos, las fotografías de los productos y los iconos para mejorar la experiencia del usuario. Adicionalmente se realizarán los mockups correspondientes

para hacerse una idea del “look and feel” final del sitio web.

Después de haber concluido con la etapa de diseño, maquetado y prototipado, se procederá a realizar el diseño del modelo entidad relación que nos dará la oportunidad de mejorar la forma de vender productos en la empresa. El motor de base de datos que se elegirá en esta ocasión será MySql.

Una vez terminado el diseño del modelo que usará la base de datos, procederemos a realizar el diseño e implementación de la capa del backend. La implementación será realizada en el lenguaje de programación de Java. Como se pudo observar en el diagrama de arquitectura de backend, este constará de un modelo de servicios, el cual nos dará la robustez necesaria para el caso de uso dado.

Pasada esta etapa, se realizará el diseño y la implementación de la capa de frontend, en la cual se utilizará el lenguaje de programación javascript, más concretamente con su popular framework para frontend llamado vue.js. Como se pudo observar en el diagrama de arquitectura del frontend, esta capa se desarrollará siguiendo un “modelo vista controlador”. Esta capa será la última en la etapa de desarrollo.

Finalmente, se desplegará la aplicación en una plataforma que se usa como servicio llamada Heroku. Esta plataforma nos permitirá mantener nuestro sitio web en la nube, dando paso a que otras personas puedan acceder a él. Adicionalmente, se realizarán pruebas funcionales a la aplicación para comprobar su integridad y funcionamiento.

Una vez pasadas estas pruebas funcionales, y luego de haber comprobado todos los aspectos fundamentales relacionados con el funcionamiento de la aplicación, se dará por terminado el proyecto entregando un “minimum viable product (MVP)” que servirá como base para un sitio más robusto en el futuro.

5.3 Instrumentos de recolección de información

5.3.1 Fuentes primarias.

Levantamiento de requerimientos encuesta:

Nombre de la compañía: Káfe
Servicio que produce: Lencería para hogares
Público al que se dirige: Principalmente mujeres
Técnica de recolección: Encuesta

Encuesta al administrador de la Empresa Kafe

¿Cómo gestionan los pedidos actualmente?: Mediante la aplicación WhatsApp Business
¿Cómo toman las órdenes?: Por medio de los chats de las redes sociales: WhatsApp, Instagram, Facebook
¿Qué tipo de información acerca de sus clientes manejan?: Nombre completo, celular, documento de identidad, dirección, ciudad.
¿Cuál es el problema?: No contar con un sitio web
¿Qué esperan de una aplicación web?: Poder mostrar a la empresa y sus servicios con el fin de incrementar ventas y posicionarnos en el mercado
¿Desean integrar descuentos en la aplicación para la fidelización de clientes?: Sí, pero queremos manejar nosotros mismos esos descuentos y poder modificarlos.
¿Qué es indispensable que incluya la aplicación?: Nuestros datos de contacto, los productos que ofrecemos y sus características
¿Qué tipos de usuarios tendrá la aplicación? Usuario para el gerente, usuario para administradores y usuario para clientes
¿Contará con acceso directo a sus redes sociales?: Si
¿Incluir diseño a la medida para posicionar la marca?: Si
¿Qué medios de pago manejan?: Transferencia bancaria: Bancolombia, nequi y Davivienda, Daviplata
¿Incluir chatbot?: No

6. Resultados

6.1. Análisis

6.1.1. Requisitos

Requisitos Funcionales:

Código	Descripción	Actor
RF-001	La aplicación permitirá tener 2 tipos de usuarios: Administrador y cliente.	**Administrador**
RF-002	La aplicación creará los usuarios con correo y contraseña	**Administrador**
RF-003	Los usuarios administradores podrán crear productos nuevos	**Administrador**
RF-013	Los usuarios administradores podrán modificar productos existentes	**Administrador**
RF-023	Los usuarios administradores podrán borrar productos existentes	**Administrador**
RF-033	La aplicación permitirá a los usuarios visualizar la descripción de los productos: medidas, precio, color, forma	**Cliente**
RF-004	La aplicación traerá un carrito de compras	**Cliente**
RF-005	La aplicación mostrará un campo de descuentos	**Cliente**
RF-006	La aplicación tendrá los datos de contacto	**Cliente**
RF-014	La aplicación permitirá al cliente tener varios productos en el carrito de compras	**Cliente**

Tabla 2: Requisitos funcionales

Fuente: diseño propio

Requisitos No Funcionales:

Código	**Descripción**	**Item**
RNF-001	Realizar pruebas con las salidas que proporciona el software Si se genera un error, se informará al usuario como solucionarlo de una forma rápida y eficiente.	Confiabilidad
RNF-002	La aplicación estará disponible el 99% del tiempo al año y en una plataforma web segura.	Disponibilidad
RNF-003	La aplicación solo creará usuarios que ingresen contraseñas validadas como seguras	Seguridad
RNF-004	La aplicación funcionará en entornos web	Portabilidad
RNF-005	La aplicación maneja el tema de los pagos redireccionando al cliente al WhatsApp y no en la misma aplicación	Restricciones

Tabla 3: Requisitos no funcionales

Fuente: diseño propio

Como resultado, los requisitos funcionales mencionados (Tabla 2) cumplen con el objetivo de ser completos y consistentes. Son completos al definir los servicios solicitados por el cliente, y son consistentes al no tener definiciones contradictorias. Los requisitos no funcionales (Tabla 3) también mencionados muestran las propiedades emergentes como la fiabilidad, tiempo y portabilidad (Sommerville, 2005)

6.1.2. Diagramas UML

Diagrama de casos de usos

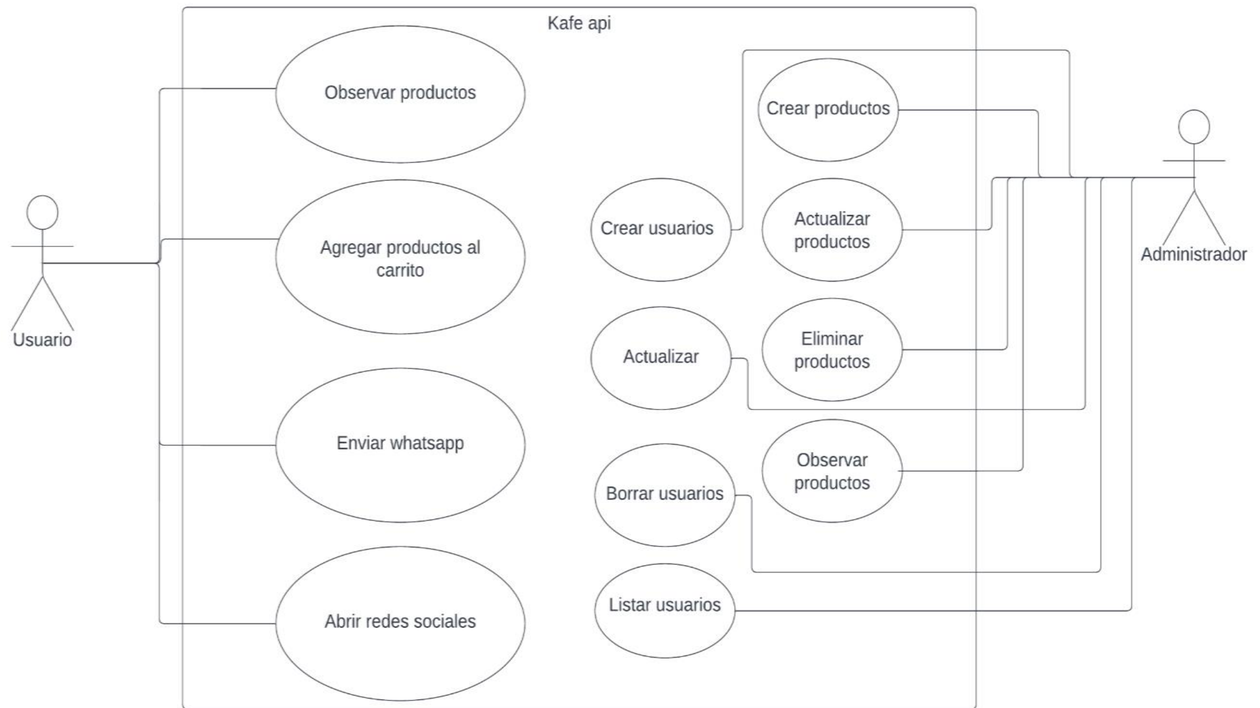


Figura 2: Casos de usos

Fuente: propia

Diagrama de clases:

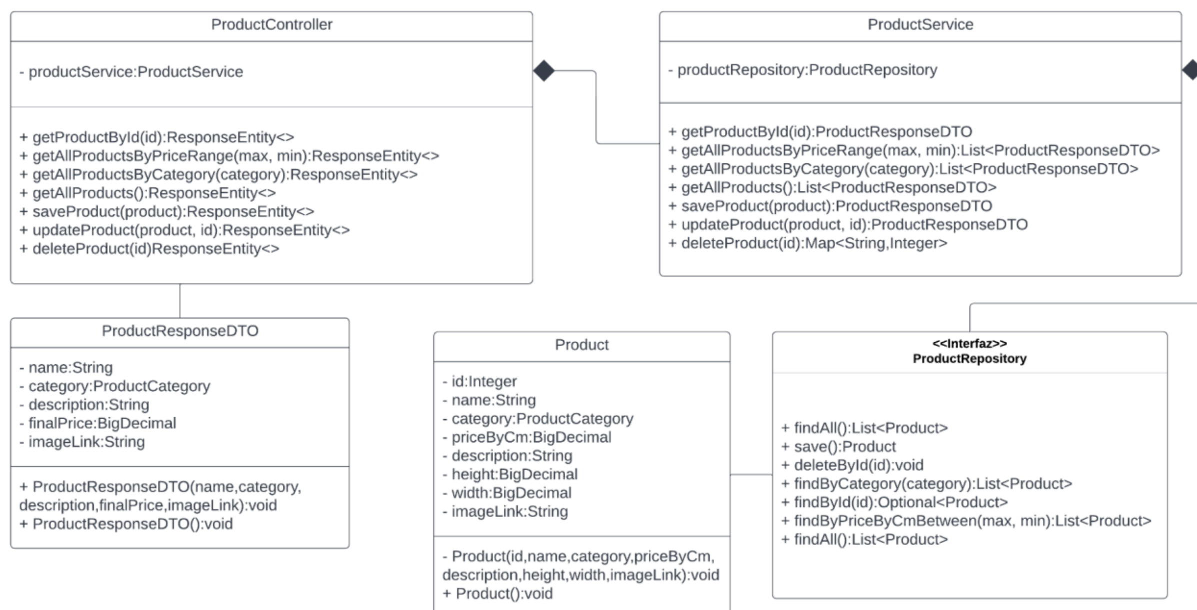


Figura 3: Diagrama de clases

Fuente: propia

Diagrama de clases: seguridad:

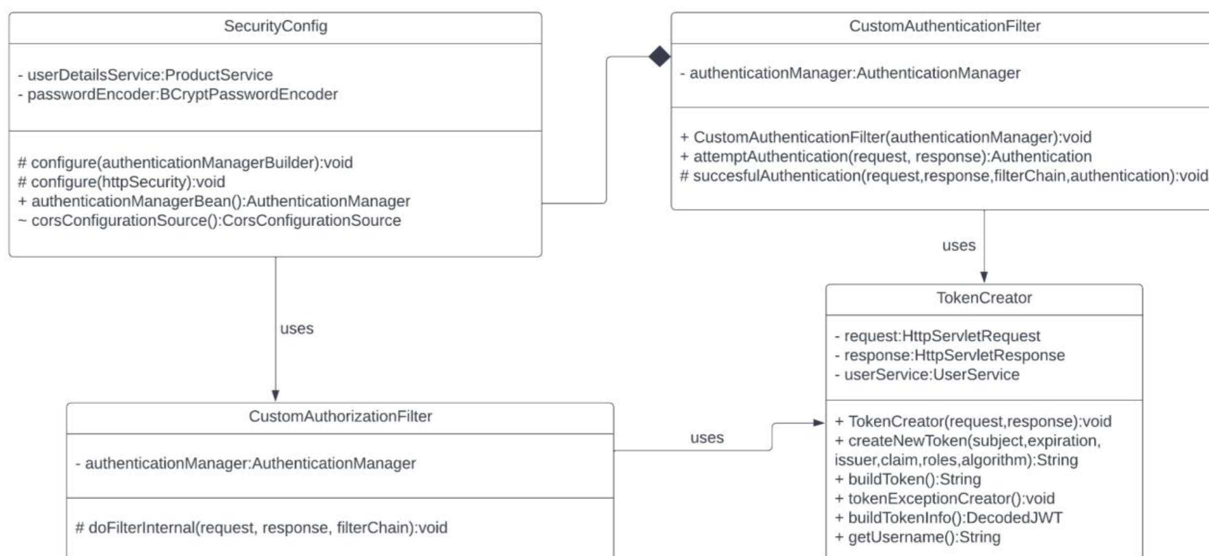


Figura 4: Diagrama de clase seguridad

Fuente: propia

6.2 Diseño

6.2.1 Arquitectura de la aplicación web

La arquitectura que tendrá la aplicación web será una arquitectura de 3 capas. Cada una de las capas tendrá su propia responsabilidad, y en conjunto, todas actuarán como un solo servicio que permitirá darle al usuario la funcionalidad deseada. Para la capa de presentación se utilizará el framework vue.js. Para la capa lógica o de negocio se utilizará el lenguaje de programación Java y su framework web más popular: Spring. Finalmente, para la capa de datos, se utilizará un gestor de base de datos relacional: MySQL.

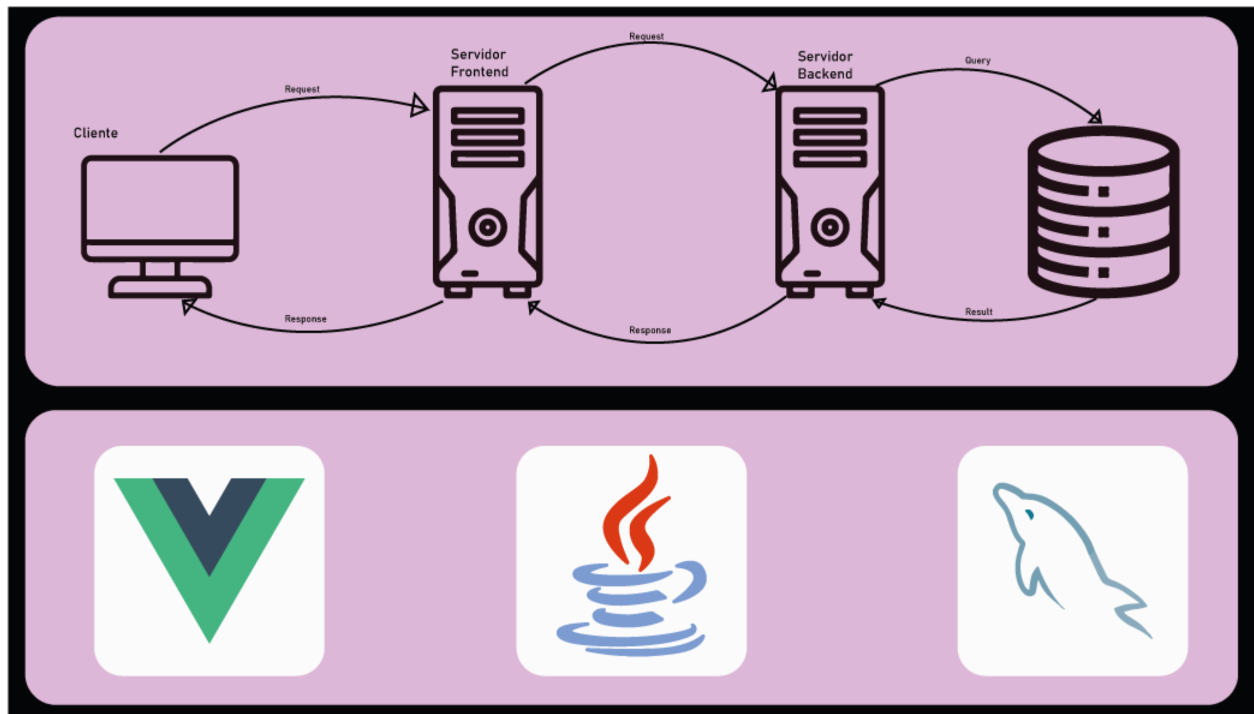


Figura 5:Arquitectura general

Fuente: diseño propio

La capa de presentación, o frontend de la aplicación, estará organizado internamente simulando una arquitectura MVC. Su lógica de funcionamiento va a estar dividida en 3 partes: Modelo, vista y controlador. Estas se encargarán de realizar peticiones a la capa lógica, también conocida como backend, y, a su vez, de disponibilizar la información para que el usuario pueda verla.

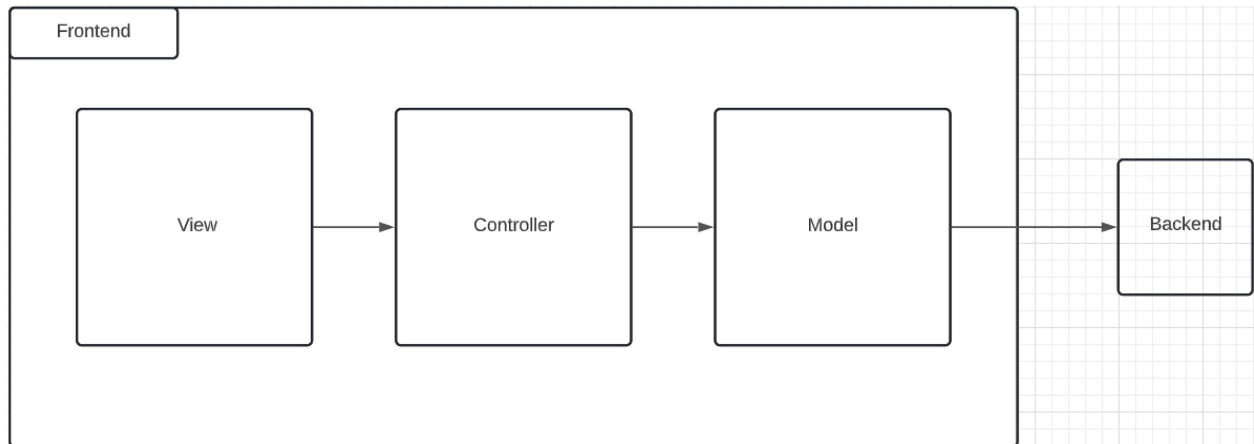


Figura 6: Arquitectura del frontend
Fuente: propia

La capa lógica estará organizada siguiendo un modelo de n capas. Primero tendremos un controlador que se encargará de recibir las peticiones provenientes del frontend. Este controlador llamará al servicio que requiera. El servicio llamará al repositorio que necesite para obtener la información, y este, finalmente, llamará a la base de datos para obtener los datos requeridos (Anexo 2). Cada dato obtenido de la base de datos será convertido en una entidad de Java, esta entidad será enviada desde el repositorio hasta el service. El service implementará la lógica que necesite para garantizar que el negocio funcione correctamente, y luego devolverá un DTO (Anexo 3) u objeto plano que contendrá los datos de la entidad que realmente sea necesario mostrar al usuario. Finalmente el controller devolverá la respuesta al frontend, y este la mostrará al usuario.

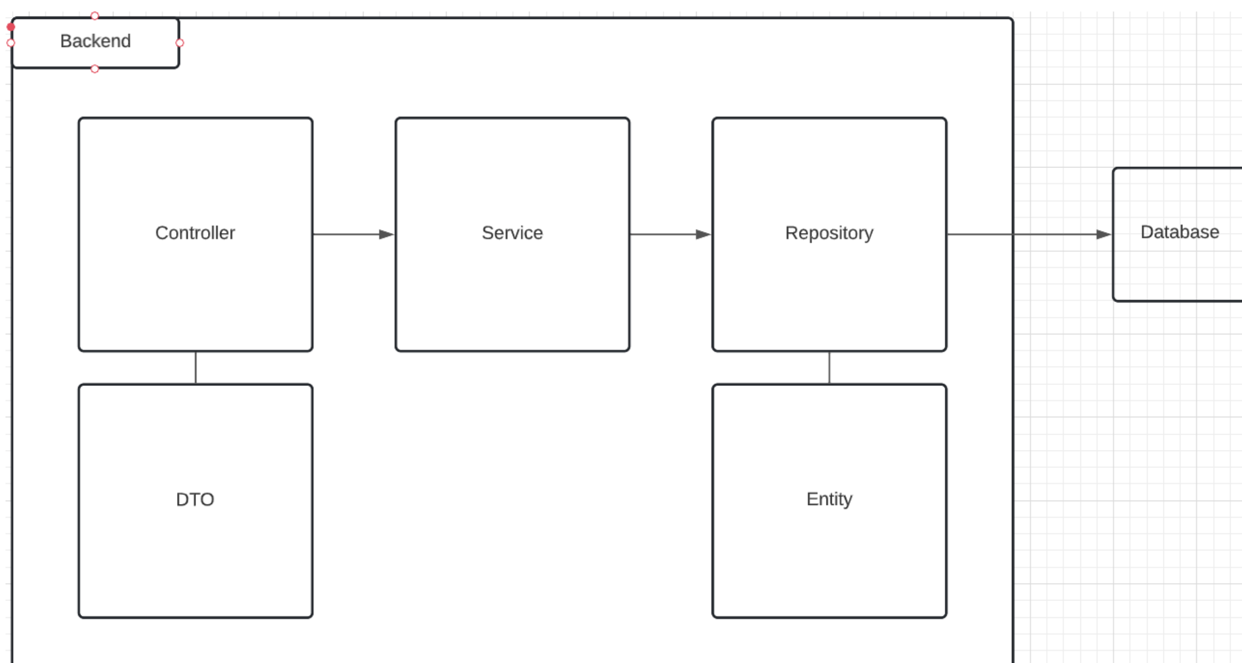


Figura 7: Arquitectura del backend
Fuente: propia

6.2.2. Diagrama entidad relación

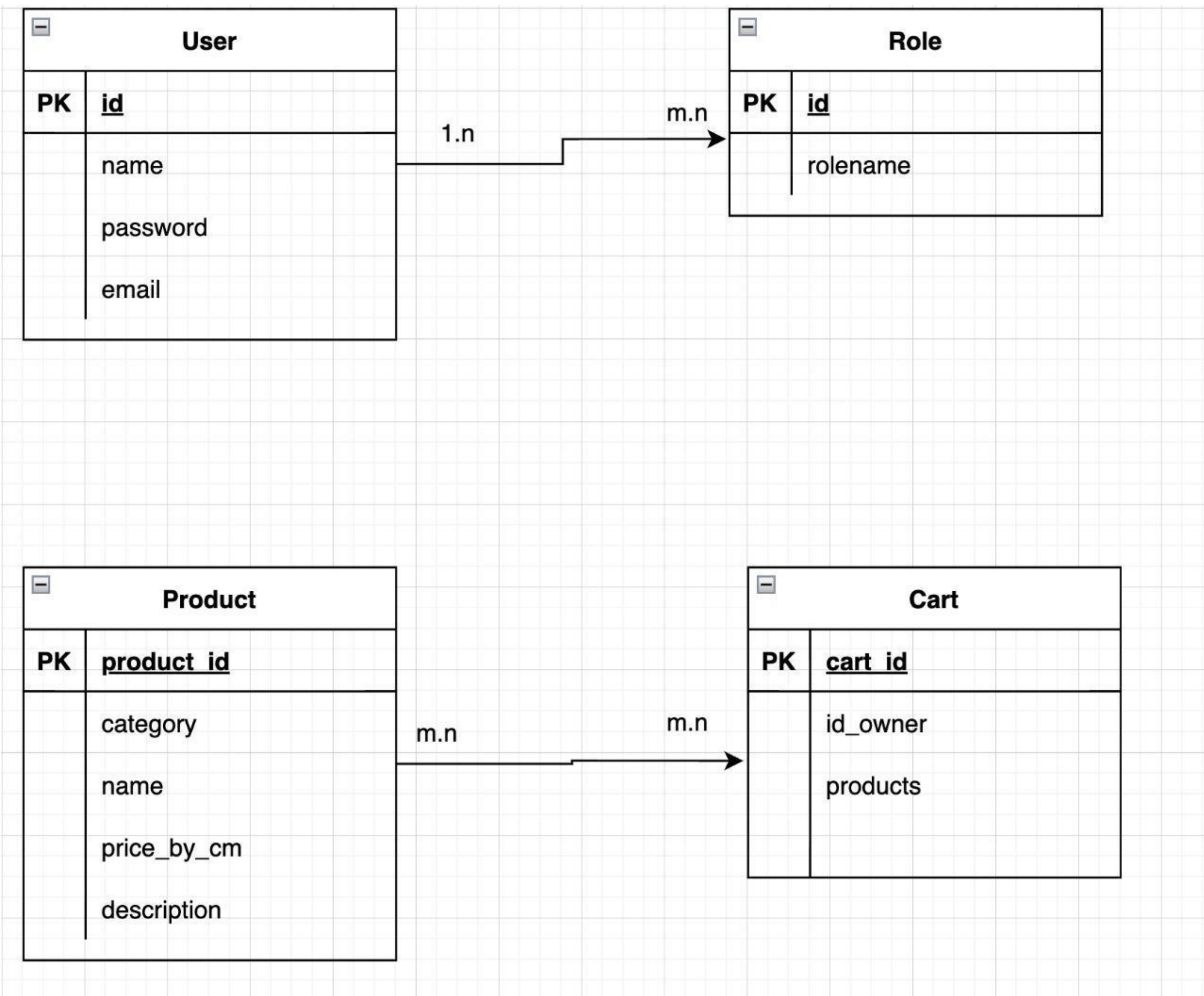


Figura 8: Diagrama entidad relación

Fuente: propia

6.3 Implementación

Siguiendo el diagrama que se tiene en la figura 4, se crearon los packages necesarios para el correcto funcionamiento de la aplicación que serviría de backend.

Esta estructura se puede observar en la siguiente figura (Figura 5):

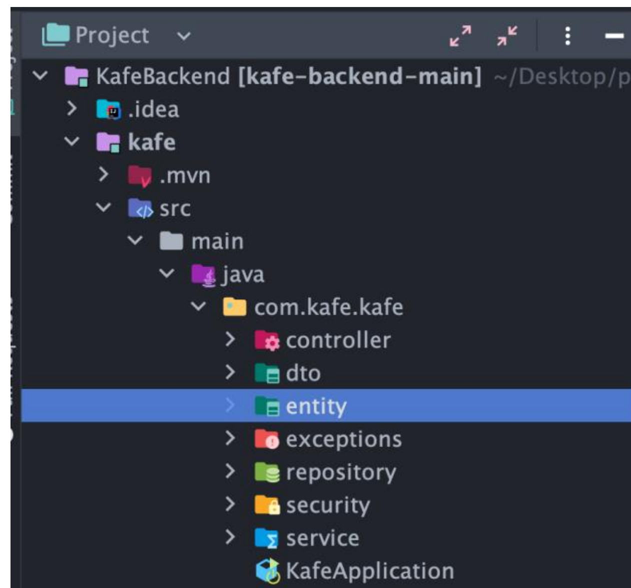


Figura 9: Estructura del backend

Fuente: Fotografía propia

Luego de haber organizado la estructura de las carpetas, se empezó con el desarrollo de la lógica que iba a tener la aplicación. Dentro del package controller se crearon dos clases que servirían de entry points para las aplicaciones que consumieran el api de backend. En este caso, se creó una Controller para manejar toda la lógica de los usuarios y otro controller para manejar la lógica de los productos.

```java
@RestController
@RequestMapping("/api/product")
public class ProductController {

    @Autowired
    ProductService productService;

    @GetMapping("/products")
    public ResponseEntity<List<ProductResponseDTO>> getAllProducts() {
        return new ResponseEntity<>(productService.getAllProducts(), HttpStatus.OK);
    }
```

Figura 10: ProductController

Fuente: Fotografía propia

En la figura 6 se puede observar la disposición normal de un controlador en Spring. Se tiene una anotación @RestController para indicarle a Spring que la clase ProductController efectivamente es un controlador de tipo rest que va a recibir las peticiones provenientes de los clientes que consuman la api. En la figura, también se observa el uso de la anotación @RequestMapping que se encarga de decirle a Spring, cuál es el path asociado a la clase ProductController. Es decir, le indica a Spring qué endpoint va a ser común a todos los métodos de la clase (Anexo 4). La anotación @Autowired, por su parte, se encarga de inyectar las dependencias de la clase en la que se usa la anotación. En este caso, esta anotación le dice a Spring que el servicio de productos, va a ser inyectado en la clase controlador. Con esta anotación, Spring se encarga de realizar la inyección de dependencias sin que a nosotros, como programadores, nos toque preocuparnos de los detalles de implementación. La anotación @GetMapping, le dice a Spring que el método que le sigue, en este caso, getAllProducts, se encargará de recibir las peticiones del endpoint llamado “/products”. Este endpoint, cómo su nombre lo indica, se encarga de devolver una lista que contiene todos los productos almacenados en la base de datos, y, a su vez, se encarga de devolver el status code correspondiente, en este caso, un “http 200 ok”.

```
@GetMapping("/category/{category}")
public ResponseEntity<List<ProductResponseDTO>> getAllProductsByCategory(
    @PathVariable ProductCategory category) {
    return new ResponseEntity<>(productService.getAllProductsByCategory(category), HttpStatus.OK);
}


@PostMapping("/save")
public ResponseEntity<ProductResponseDTO> saveProduct(@Valid @RequestBody ProductRequestDTO product) {
    return new ResponseEntity<>(productService.saveProduct(product), HttpStatus.CREATED);
}

@PutMapping("/{id}")
public ResponseEntity<ProductResponseDTO> updateProduct(@RequestBody ProductRequestDTO product,
    @PathVariable Integer id) {
    return new ResponseEntity<>(productService.updateProduct(product, id), HttpStatus.ACCEPTED);
}

@DeleteMapping("/{id}")
public ResponseEntity<Map<String, Integer>> deleteProduct(@PathVariable Integer id) {
    return new ResponseEntity<>(productService.deleteProduct(id), HttpStatus.OK);
}
```

Figura 11: Métodos de ProductController

Fuente: Fotografía propia

En la figura 7, se puede observar que, además de contar con un endpoint y su respectivo método para obtener todos los productos, también se cuenta con algunos endpoints que proveen funcionalidades muy útiles a la hora de realizar operaciones en la base de datos. Por ejemplo, se puede ver que se cuenta con anotaciones del tipo @PutMapping, @DeleteMapping y @PostMapping que se encargan de recibir las peticiones que tengan dichos verbos http. Dependiendo del tipo de petición se devuelve un tipo de respuesta diferente. En la figura queda claro que cuando se solicita guardar un producto, se devuelve un DTO con la información del producto guardado, y un “http 201 created”. Cuando se solicita actualizar un producto, se responde con el mismo tipo de DTO, pero, esta vez, con el status code “202 accepted” (Anexo 5). A la hora de eliminar un producto, simplemente se responde con un Map que contiene como clave un String y como valor un entero. Este Map contiene el id del producto que fue eliminado de la base de datos.

Cabe resaltar que para poder manejar la conversión de cada uno de los tipos de datos a valores de tipo json, se requiere el uso de la clase ResponseEntity. Esta se encarga de manejar toda la

respuesta del controlador al pasarle únicamente el tipo de dato que se desea manejar, y el status code que llevará la respuesta.

```
@RestController
@RequestMapping("/api")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @GetMapping("/users")
    public ResponseEntity<List<UserResponseDTO>> getAllUsers() {
        return new ResponseEntity<>(userService.getUsers(), HttpStatus.OK);
    }

    @GetMapping("/user/{username}")
    public ResponseEntity<User> getUser(@PathVariable String username) {
        return new ResponseEntity<>(userService.getUser(username), HttpStatus.OK);
    }

    @PostMapping("/user/save")
    public ResponseEntity<User> saveUser(@RequestBody User user) {
        return new ResponseEntity<>(userService.saveUser(user), HttpStatus.OK);
    }
```

Figura 12: UserController y sus métodos

Fuente: Fotografía propia

En la figura 8 se ve que, al igual que se hizo para los productos, se creó un controlador para los usuarios. La única diferencia que vale la pena resaltar, entre el ProductController y el UserController, es el uso de la anotación @RequiredArgsConstructor en vez de la anotación @Autowired. Esta anotación le dice a Spring que, donde quiera que encuentre una dependencia de la que se deba encargar, la inyecte en la clase que la necesita. En este caso, la anotación inyecta el UserService en la clase UserController. Se usó esta anotación para esta clase solo con fines de aprendizaje. El funcionamiento de las dos clases controladoras es exactamente el mismo aunque se usen estas dos anotaciones distintas.

Luego nos encontramos con las clases de tipo DTO. Estas clases se encargan de devolverle al usuario solo la información requerida, o de manejar la información enviada por el usuario

mediante métodos de tipo Post o Put. Como se puede observar en la figura 10 (ver abajo), se utilizan algunas anotaciones muy útiles provistas por la librería Lombok. La anotación @Data se encarga de generar automáticamente todo el código que se necesita normalmente en esta clase de DTO. Genera los getters, los setters, el toString y el equalsAndHashCode. Además se observa el uso de las anotaciones @NoArgsConstructor y @AllArgsConstructor que se encargan de generar los constructores de la clase. Generan un constructor vacío, y otro con todos los atributos de la clase respectivamente.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class LoginForm {
    private String username;
    private String password;
}
```

Figura 13: DTO y sus métodos

Fuente: Fotografía propia

Algunos DTO requerían el uso de anotaciones más específicas. En la figura 10 se observa un DTO con una composición un poco más compleja. Este contiene la anotación @JsonNaming, la cual a su vez cuenta con el valor PropertyNamingStrategies.SnakeCaseStrategy. Esta anotación, en combinación en la propiedad que recibe, se encarga de convertir los datos que vienen en el body de la petición de tipo snake case a tipo camel case. Esto se hace con el fin de seguir las buenas prácticas en Java, las cuales exigen que las variables sean nombradas en camel case.

También se puede observar en este DTO el uso de anotaciones tales como: @NotNull, @Min y @Size. Al tratarse de un DTO que maneja los datos que nos manda cualquier cliente mediante el cuerpo de una petición, se deben validar los datos para evitar que la aplicación falle en cualquier momento. Estas anotaciones validan que los datos enviados correspondan con lo que deseamos recibir. En este caso en específico, validan que el nombre, la categoría, la altura, el ancho, y el link del producto no estén vacíos. Además de esto, se valida que el precio no sea menor a 0, que la descripción del producto no tenga menos de 10 caracteres ni más de 255.

En caso de que en el body de la petición, venga cualquier dato que no concuerde con las validaciones realizadas, se lanzará una excepción de tipo MethodArgumentNotValidException. Esta excepción contendrá en su interior el mensaje que se había puesto a la hora de declarar la validación. Esta excepción, además, será capturada por un clase especializada (ver figura 12) que se encargará de devolver la respuesta al cliente indicando el error ocurrido , y además, tendrá un status code “400 bad request”.

```
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
@Data @NoArgsConstructor @AllArgsConstructor
public class ProductRequestDTO {
    @NotNull(message = "El nombre no puede estar vacío")
    private String name;

    @NotNull(message = "La categoría no puede estar vacía")
    private ProductCategory category;

    @NotNull(message = "El precio no puede estar vacío")
    @Min(value=0,message = "El precio no puede ser menor a 0")
    private BigDecimal priceByCm;

    @Size(min = 10, max = 255, message = "La descripcion debe tener entre 10 y 255 caracteres")
    private String description;

    @NotNull(message = "Agregue una altura")
    private Double height;

    @NotNull(message = "Agregue un ancho")
    private Double width;

    @NotNull(message = "Agregue un link de la imagen del producto")
    private String imageLink;
```

Figura 14: ProductRequestDTO y sus validaciones

Fuente: Fotografía propia

```
@ControllerAdvice
public class ExceptionManager {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handlerValidationExceptions(MethodArgumentNotValidException ex){
        Map<String, String> result = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((x)->{
            result.put(((FieldError)x).getField(),x.getDefaultMessage());
        });
        return new ResponseEntity<>(result , HttpStatus.BAD_REQUEST);
    }

}
```

Figura 15: ExceptionManager

Fuente: Fotografía propia

La clase ExceptionManager se encarga de convertir los errores de tipo MethodArgumentNotValidException en Strings legibles para el usuario. Como se ve en la figura 11, se devuelve un mensaje por cada validación que falló. La anotación @ControllerAdvice le dice a Spring que la clase ExceptionManager será la encargada de manejar cierto tipo de excepciones ocurridas en cualquier controlador. La anotación ExceptionHandler, en conjunto con la propiedad que se le pasa entre los paréntesis, le dicen a Spring que el método que aparece a continuación, realizará el manejo del tipo de excepción requerido. En este caso, solo se están capturando excepciones del tipo MethodArgumentNotValidException, pero se puede realizar el manejo de cualquier tipo de excepción.

```java
@Service
public class ProductServiceImpl implements ProductService{

    @Autowired
    ProductRepository productRepository;
    ModelMapper modelMapper = new ModelMapper();

    @Override
    public List<ProductResponseDTO> getAllProducts() {
        List<Product> products = productRepository.findAll();
        return getProductResponseDTOS(products);
    }

    @Override
    public ProductResponseDTO saveProduct(ProductRequestDTO product) {
        Product productToSave = modelMapper.map(product, Product.class);
        computeTotalPrice(productToSave);
        ProductResponseDTO result = modelMapper.map(productRepository.save(productToSave), ProductResponseDTO.class);
        result.setFinalPrice(computeTotalPrice(productToSave));
        return result;
    }
```

Figura 16: ProductService y sus métodos

Fuente: Fotografía propia

En la figura 12 vemos la implementación normal de un servicio en Spring. Se usa la anotación @Service, para decirle a Spring que esta clase es precisamente eso, un servicio. Igual que se había hecho en el controller, se realiza la inyección de las dependencias necesarias. En la figura, se nota cómo se inyecta el repositorio de productos en el servicio. Adicionalmente vemos como se usa un atributo del tipo ModelMapper. Este se encargará de realizar la transformación de los datos de las entidades en objetos de tipo DTO. Esto se hace para no exponer información que sea sensible para el usuario, y para devolver solo los campos necesarios para los clientes. También se observa en el servicio, que se realiza la implementación de una interfaz. Esta interfaz se usa con la finalidad de que, si tenemos más de una implementación de servicio que maneje los usuarios, podamos usar cualquiera de las dos en el controller, ya que en este, se realiza la inyección de la dependencia por medio de la interfaz, y no de una clase en específico.

```java
@Service @RequiredArgsConstructor @Transactional @Slf4j
public class UserServiceImpl implements UserService, UserDetailsService {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;
    private final PasswordEncoder passwordEncoder;
    ModelMapper modelMapper = new ModelMapper();

    @Override
    public User saveUser(User user) {
        log.info("Saving new user {} to database", user.getName());
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        return userRepository.save(user);
    }
```

Figura 17: UserService y sus métodos

Fuente: Fotografía propia

En la figura 13 se usan algunas anotaciones un poco más específicas, ya que esta clase que se muestra en la imagen, se encarga de manejar la información de los usuarios, la cual, es un poco más sensible que la de los productos. Por esta razón se usan aplicaciones como @Transactional y @Slf4j. Estas son usadas para decirle a Spring que en este servicio, se realizarán algunas operaciones de tipo transaccional, y que usaremos un logger preestablecido para registrar estas transacciones, y permitir a los desarrolladores una mejor experiencia a la hora de depurar la aplicación. Además, el uso del logger nos deja una pequeña trazabilidad que nos permitirá detectar algunos errores de una mejor manera. Adicional a esto, el atributo de tipo PasswordEncoder nos permitirá realizar la encriptación de las contraseñas antes de guardarlas en la base de datos. Esto con el fin de tener un nivel adicional de seguridad para los usuarios. Al heredar las funciones de la clase UserDetailsService, podemos realizar todo el manejo de las funciones del usuario en cuanto a inicio de sesión y los roles que este usuario tendrá.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

Figura 18: UserRepository

Fuente: Fotografía propia

En la figura 14 se puede observar la implementación normal de un repositorio en Spring. Este repositorio se encargará de realizar todas las consultas a la base de datos. Por defecto, al heredar de la interfaz JpaRepository ya se tienen por defecto algunos métodos útiles en la interfaz UserRepository. Los métodos que sean un poco más específicos, como en este caso, el método de buscar un usuario por su nombre de usuario, se pueden realizar con la ayuda de JPA. Con tan solo nombrar el método siguiendo el estándar de JPA se pueden realizar queries a la base de datos sin escribir ni una sola línea de SQL.

```java
@Entity
@Getter
@Setter
@NoArgsConstructor @AllArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @Column(unique = true)
    private String username;
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    private Collection<Role> roles = new ArrayList<>();
```

Figura 19: Entidad User

Fuente: Fotografía propia

La query realizada en la figura 14 traerá como resultado, un objeto de tipo User, el cual se puede observar en la figura 15.

En esta figura se ve que la anotación @Entity es usada. Esta le dice a Spring que la clase usuario va a ser tratada como una entidad, o una tabla, de la base de datos. La anotación @Id le dice a Spring que el atributo que le sigue a esta anotación será tratado como la llave primaria de la tabla en la base de datos. La anotación @GeneratedValue con su valor GenerationType.AUTO se encarga de generar ese id de manera automática y secuencial. @Column con su valor unique en true, le dice a Spring que esa columna en la base de datos contendrá sólo valores únicos. Por último, la anotación @ManyToMany le indica a Spring que el usuario podrá tener muchos roles, y que un rol puede pertenecer a múltiples usuarios. JPA se encarga automáticamente de crear las tablas necesarias junto con sus relaciones en la base de datos.

Pasemos ahora a hablar un poco sobre el esquema de seguridad que usa la aplicación. Se puede observar en la figura 16 que se usó una clase con las anotaciones @Configuration y @EnableWebSecurity. Estas anotaciones le dicen a Spring que esta clase se encargará de realizar la configuración de toda la seguridad de la api. Además, al heredar de la clase WebSecurityConfigurerAdapter, se puede realizar la sobreescritura de ciertos métodos que nos permitirán dotar de seguridad a nuestra aplicación. En esta imagen se puede observar que se usa un password encoder y un servicio de detalles de usuario. El método configure que recibe una autenticación por parámetros, se encarga de configurar el sistema de autenticación del usuario, usando el encoder y el servicio de detalles mencionados anteriormente.

```
@Configuration @EnableWebSecurity @RequiredArgsConstructor
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;
    private final BCryptPasswordEncoder passwordEncoder;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
    }
```

Figura 20: Configuraciones de seguridad

Fuente: Fotografía propia

En la figura 17 se observa con claridad la configuración establecida de seguridad. Se puede ver que se usarán algunos cors para evitar que se manden peticiones desde cualquier lugar que no tenga los permisos necesarios. Además se configurará cada grupo de endpoints para que solo pueda ser accedidos por los usuarios que realmente tengan los permisos para hacerlo. Además se requerirá que cualquier petición que se realice deberá ser realizada por un usuario autenticado.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors();
    CustomAuthenticationFilter customAuthenticationFilter = new CustomAuthenticationFilter(authenticationManagerBean());
    customAuthenticationFilter.setFilterProcessesUrl("/api/login");
    http.csrf().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests().antMatchers( ...antPatterns: "/api/login/**", "/api/token/refresh/**").permitAll();
    http.authorizeRequests().antMatchers(HttpMethod.GET,  ...antPatterns: "/api/product/**").permitAll();
    http.authorizeRequests().antMatchers(HttpMethod.GET, ...antPatterns: "/api/user/**").hasAuthority("ROLE_USER");
    http.authorizeRequests().antMatchers(HttpMethod.POST, ...antPatterns: "/api/user/save").hasAuthority("ROLE_ADMIN");
    http.authorizeRequests().antMatchers(HttpMethod.POST,  ...antPatterns: "/api/product/save").hasAuthority("ROLE_ADMIN");
    http.authorizeRequests().antMatchers(HttpMethod.PUT,  ...antPatterns: "/api/product/**").hasAuthority("ROLE_ADMIN");
    http.authorizeRequests().antMatchers(HttpMethod.DELETE,  ...antPatterns: "/api/product/**").hasAuthority("ROLE_ADMIN");
    http.authorizeRequests().anyRequest().authenticated();
    http.addFilter(customAuthenticationFilter);
    http.addFilterBefore(new CustomAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

Figura 21: SecurityConfig. Método configure

Fuente: Fotografía propia

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
                                        FilterChain chain, Authentication authentication) throws IOException,
    User user = (User) authentication.getPrincipal();
    Algorithm algorithm = Algorithm.HMAC256("secret".getBytes());

    String subject = user.getUsername();
    Date accessTokenExpiration = new Date(System.currentTimeMillis() + 10 * 60 * 1000);
    Date refreshTokenExpiration = new Date(System.currentTimeMillis() + 24 * 60 * 60 * 1000);
    String issuer = request.getRequestURI();
    String claim = "roles";
    List<String> roles = user.getAuthorities().stream().map(GrantedAuthority::getAuthority).toList();

    String accessToken = TokenCreator.createNewToken(subject, accessTokenExpiration, issuer, claim,
            roles, algorithm);
    String refreshToken = TokenCreator.createNewToken(subject, refreshTokenExpiration, issuer, claim,
            roles, algorithm);
```

Figura 22: Authentication Filter.

Fuente: Fotografía propia

Estas peticiones, además, serán procesadas por algunos filtros personalizados. Estos filtros (como se observa en la figuras 18 y 19) procesan las peticiones de inicio de sesión y los roles de los usuarios, y devuelven los tokens de tipo jwt que luego serán requeridos por la aplicación.

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                FilterChain filterChain) throws ServletException, IOException {

    if(request.getServletPath().equals("/api/login") || request.getServletPath().equals("/api/token/refresh")) {
        filterChain.doFilter(request, response);
    } else {
        String authorizationHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (authorizationHeader ≠ null && authorizationHeader.startsWith("Bearer ")) {
            TokenCreator tokenCreator = new TokenCreator(request, response);
            try {
                String username = tokenCreator.getUsername();
                String[] roles = tokenCreator.buildTokenInfo().getClaim( s: "roles").asArray(String.class);
                Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
                Arrays.stream(roles).forEach(role → {
                    authorities.add(new SimpleGrantedAuthority((role)));
                });

                UsernamePasswordAuthenticationToken authenticationToken =
                    new UsernamePasswordAuthenticationToken(username, credentials: null, authorities);
                SecurityContextHolder.getContext().setAuthentication(authenticationToken);
                filterChain.doFilter(request, response);
            } catch (Exception e) {
                log.error("Error logging in: {}", e.getMessage());
                tokenCreator.tokenExceptionCreator(e);
            }

        } else {
            filterChain.doFilter(request, response);
        }
```

Figura 23: Authorization Filter.

Fuente: Fotografía propia

Estos filtros son indispensables para el manejo de los permisos de los usuarios.

```
@Bean
PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
```

Figura 24: PasswordEncoder Bean.

Fuente: Fotografía propia

Por último, pero no menos importante, se registra un Bean (Figura 20) que se encarga de devolver el encoder encargado de cifrar las contraseñas de los usuarios. Este bean es usado luego por Spring para guardar los usuarios en la base de datos con una contraseña encriptada.

Pasemos ahora a hablar un poco de la estructura del frontend. En la figura 3 se puede observar cómo se planteó la construcción del frontend de nuestra aplicación. Para poder conseguirlo se usó el framework vue.js. Este framework permite manejar todos los aspectos necesarios del DOM para poder darle un aspecto visual un poco más atractivo a nuestra aplicación. En vue, cada archivo contiene un segmento llamado script, otro segmento llamado template y otro segmento llamado style. Cada uno de estos segmentos cumple su labor a la hora de darle personalidad a la aplicación web.

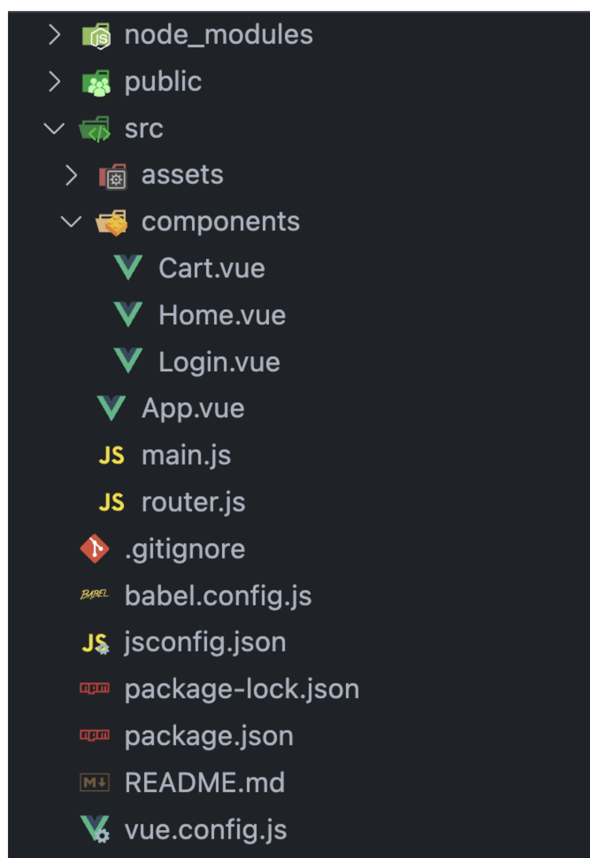


Figura 25: Estructura de carpetas frontend

Fuente: Fotografía propia

En la figura 21 se observa una estructura de carpetas común en una aplicación hecha en vue. La carpeta components contiene todos los archivos que actuarán como nuestras diferentes vistas. La carpeta node_modules contiene todas las dependencias y archivos necesarios para que nuestro proyecto pueda ejecutarse normalmente. La carpeta public contiene el icono que tendrá la aplicación en el navegador, y la carpeta src contendrá todos los archivos creados por el

desarrollador, y que contendrán la lógica del programa. El archivo App.vue será quien funcione como un controlador. Y cada componente tendrá en su interior cierta lógica que se usará como modelo. Todo esto para cumplir con el estándar MVC.

```
<script>
export default {
  name: "App",

  data: function () {
    return {
      isAuth: false,
      isAdmin: localStorage.getItem("isAdmin"),
      adminPanelActive: false,
    };
  },

  components: {},

  methods: {
    verifyAuth: function () {
      this.isAuth = localStorage.getItem("isAuth");
      if (!this.isAuth) {
        this.$router.push({ name: "login" });
      } else {
        this.$router.push({ name: "home" });
      }
    },

    loadLogin: function () {
      this.$router.push({ name: "login" });
    },

    loadSignup: function () {
      this.$router.push({ name: "signup" });
    },

    loadHome: function () {
      this.$router.push({ name: "home" });
    },

    loadAdminPanel: function () {
      this.adminPanelActive === true
        ? (this.adminPanelActive = false)
        : (this.adminPanelActive = true);
      localStorage.setItem("isAdminPanelActive", this.adminPanelActive);
    },
```

Figura 26: App.vue

Fuente: Fotografía propia

La funcionalidad que se observa en la figura 22 es muy sencilla. Esta es del componente App.vue. Aquí se puede observar cómo vue.js, por medio de sus métodos, nos permite realizar

ciertas operaciones sobre ciertos datos. Además, por medio del uso de un router incorporado en vue, podemos cambiar las diferentes vistas, dependiendo de algunos estados o señales recibidas.

```
<template>
  <div id="app" class="app">
    <div class="header">
      <nav>
        <button class="nav-button" v-if="isAuth" v-on:click="loadHome">
          Inicio
        </button>
        <button
          class="nav-button"
          v-if="isAuth && isAdmin"
          v-on:click="loadAdminPanel"
        >
          Administrar
        </button>
        <button class="nav-button" v-if="isAuth" v-on:click="logOut">
          Cerrar Sesión
        </button>
        <button class="nav-button" v-if="!isAuth" v-on:click="loadLogin">
          Iniciar Sesión
        </button>
      </nav>
    </div>

    <div class="main-component">
      <router-view
        v-on:completedLogin="completedLogin"
        v-on:logOut="logOut"
      ></router-view>
    </div>
  </div>
</template>
```

Figura 27: Template App-vue

Fuente: Fotografía propia

El template del App.vue, nos deja ver cómo vue se encarga de realizar el llamado de las diferentes vistas de la aplicación, por medio de la etiqueta <router-view>. Como se puede ver en la figura 23, se exporta una constante llamada router, la cual contiene todas las rutas asociadas a cada componente. Cuando cada ruta es llamada en App.vue, el router llama al componente indicado, y renderiza la página web correspondiente. El registro de cada ruta y cada componente se realiza en este archivo (router.js).

En la figura 24 se puede observar cómo Login.vue hace una petición de tipo post a nuestra api de backend con los datos que el usuario nos provee. A su vez, la respuesta obtenida por backend es

almacenada temporalmente en el localStorage del navegador para su posterior uso en otras funcionalidades. Si la autenticación fue exitosa, se realiza una emisión (se envía una especie de mensaje a App.vue) para que se ejecuten ciertas instrucciones necesarias para el funcionamiento de la aplicación.

Como se puede ver en la figura 25, los datos obtenidos en la figura 24 son usados para realizar diferentes peticiones a nuestra api, y de esta manera poder mostrar al usuario la información que necesita.

```
methods: {
  processLoginUser: function () {
    axios
      .post(
        "https://kafe-backend-api.herokuapp.com/api/login",
        JSON.stringify(this.user),
        {
          headers: {},
        }
      )
      .then((result) => {
        console.log(result);
        let dataLogin = {
          username: result.data.username,
          tokenAccess: result.data.access,
          tokenRefresh: result.data.refresh,
        };
        localStorage.setItem("tokenAccess", dataLogin.tokenAccess);
        localStorage.setItem("tokenRefresh", dataLogin.tokenRefresh);
        localStorage.setItem("username", dataLogin.username);
        this.$emit("completedLogin");
      })
      .catch((error) => {
        alert("No fue posible loguearse");
        if (error.response.status == "401") {
          alert("ERROR 401: credenciales incorrectas.");
        }
      });
```

Figura 28: Login

Fuente: Fotografía propia

```
  axios
    .get("https://kafe-backend-api.herokuapp.com/api/token/refresh", {
      headers: {
        Authorization: `Bearer ${localStorage.getItem("tokenRefresh")}`,
      },
    })
    .then((result) => {
      localStorage.setItem("tokenAccess", result.data.access);
    });
},
mounted: function () {
  axios
    .get("https://kafe-backend-api.herokuapp.com/api/product/products")
    .then((result) => {
      for (const product of result.data) {
        this.products.push(product);
      }
    })
    .catch((error) => {
      alert(error);
    });
```

Figura 29: Token refresh y fetch de productos

Fuente: Fotografía propia

La imagen 26 contiene la lógica usada para realizar la funcionalidad del carrito de compras. Permitiendo así que el usuario pueda almacenar temporalmente sus productos, para luego comprarlos si desea.

```
methods: {
  addProductToCart: function (product) {
    let productsAlreadyInCart =
      JSON.parse(localStorage.getItem("cart")) || [];
    console.log("Obtained", productsAlreadyInCart);
    productsAlreadyInCart.push(product);
    console.log("Updated", productsAlreadyInCart);
    this.cart = productsAlreadyInCart;
    localStorage.setItem("cart", JSON.stringify(this.cart));
    let itemsInCart = JSON.parse(localStorage.getItem("cart"));
    localStorage.setItem("itemsInCart", itemsInCart.length);
    this.localStorageCartSize = itemsInCart.length;
  },
  loadCart: function () {
    let carItems = JSON.parse(localStorage.getItem("cart")) || 0;
    if (carItems.length > 0) {
      this.$router.push("cart");
    } else {
      alert("No tienes ningún artículo en la bolsa de compras");
    }
  },
```

Figura 30: Funcionalidad carrito de compras

Fuente: Fotografía propia

La funcionalidad que se observa en esta imagen está muy relacionada con las funciones del componente Cart.vue. Como se observa en la figura 27 se ven dos funcionalidades más. Estas se encuentran en el componente anteriormente mencionado.

```
data: function () {
  return {
    products: JSON.parse(localStorage.getItem("cart")),
  };
},

methods: {
  clearCart() {
    localStorage.removeItem("cart");
    this.$router.go(0);
  },
  purchaseOrder() {
    window.open(
      `https://api.whatsapp.com/send/?phone=573043597697&text=Hola%2C+estoy+interesado+en+u
      "_blank"
    );
  },
},
```

Figura 31: Funcionalidad vaciar carrito y comprar carrito

Fuente: Fotografía propia

Si el usuario desea eliminar los productos de su carrito, puede vaciarlo tranquilamente. Pero si el usuario desea comprar los productos que contiene el carrito de compras, con un solo clic, puede enviar inmediatamente información al whatsapp de la empresa. En el método purchaseOrder que se observa en la imagen anterior, se puede ver cómo se abre una nueva pestaña enviando un mensaje al whatsapp de la empresa. Dependiendo del tipo de producto que hubiera en el carrito, se llenaría la información del mensaje.. En ese momento, se continuaría el proceso de compra directamente con un vendedor de la empresa. (Anexo 6)

Tapetes Peludos Káfe

Ir al chat

Hola, estoy interesado en uno de tus productos: Manta gris, que vale 60000, y se ve así https://i.ibb.co/v1kfgvT/IMG-20220614-142601-515.jpg

Figura 32: Mensaje de compra de productos

Fuente: Fotografía propia

7. Conclusiones

Proyectos como el presente muestran que el patrón MVC es probado y funciona, permitiendo que se desarrollen aplicaciones más flexibles y mantenibles. Al separar las funciones de la aplicación por capas esta es más ligera y permite mantenimientos a futuro. Así que es un patrón ideal para desarrollar aplicaciones para determinadas empresas donde puedan ofrecer sus productos y/o servicios.

Esta aplicación web para la empresa Kafe es de suma importancia si quiere mantener e incluso aumentar su presencia en el mercado. Por eso se desarrolló de tal manera que se tuvo en cuenta los deseos de la organización mediante el levantamiento de requerimientos y la definición de requisitos funcionales y no funcionales

8. Recomendaciones

La aplicación anteriormente explicada a detalle, aún carece de algunas funcionalidades del lado del frontend. Al ser un MVP, es normal que esto suceda, sin embargo, se planea incluir algunas de las funcionalidades ya existentes en back, para que la aplicación empiece a ser más robusta. En el futuro se podría incluir un checkout o pasarela de compras y recibir pagos directamente desde la página. Además se planea dejar que el usuario realice todo el proceso de pedido del producto por medio del sitio web, y que, a su vez, pueda personalizar el producto a su gusto sin necesidad de que haya un vendedor disponible.

9. Referencias bibliográficas

Laviosa, A. (2006, Mayo 01). *Diseño de un prototipo de página web para el jardín de infancia Colegio Schönthal*. Repositorios Universidad Metropolitana. Retrieved October 31, 2022, from http://repositorios.unimet.edu.ve/docs/90/P.GIE2006L3D5.pdf

Vilajosana Guillén, X. (2019). *Arquitectura de aplicaciones web*. Arquitectura de aplicaciones web. Retrieved November 15, 2022, from http://190.57.147.202:90/jspui/bitstream/123456789/465/1/Arquitectura-de-aplicaciones-web-M2.pdf

Sommerville, I. (2005). Requerimientos del software. Ingeniería del software, 7a ed., PEARSON EDUCACIÓN, Madrid, SPA, 109-110.

Gómez, L. J., & Pierini Aversano, M. (2018, 12 27). Marketing 2.0. Marketing en la web, marketing digital, marketing online. *public knowledge project*, *Vol. 2*(Núm. 2). https://revistas.uns.edu.ar/cea/article/view/1348

Mena, M. (2021, Agosto 6). *Infografía: ¿Cuántos sitios web hay en el mundo?* Statista. Retrieved November 15, 2022, from https://es.statista.com/grafico/19107/numero-de-sitios-web-existentes-en-internet/

10. Bibliografía

Fernández Romero, Y., & Díaz González, Y. (2012, abril 1). Patrón Modelo-Vista-Controlador. Revista Telematica, Vol. 11(No. 1), 47-57. revistatelematica.cujae.edu/index.php/tele

Andalucia, J. d. (n.d.). Patrón Modelo Vista Controlador | Marco de Desarrollo de la Junta de Andalucía. Junta de Andalucia. Retrieved November 15, 2022, from https://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/122

Alicante, U. d. (n.d.). Modelo vista controlador (MVC). Servicio de Informática ASP.NET MVC 3 Framework. Servicio de Informática. Retrieved November 16, 2022, from https://si.ua.es/es/documentacion/asp-net-mvc-3/1-dia/modelo-vista-controlador-mvc.html

Bascon Pantoja, E. (2004, 12). El patrón de diseño Modelo-Vista-Controlador (MVC) y su implementación en Java Swing. SciELO Bolivia. Retrieved November 16, 2022, from http://www.scielo.org.bo/scielo.php?script=sci_arttext&pid=S1683-07892004000100005

11. Anexos

Anexo 1. Jira Software

Jira es un software de gestión de proyectos que facilita el orden y seguimiento del trabajo.

Proyectos / mvp-kafe

MVP-KAFE Sprint 1

Restantes: 0 días

Termina

MB

Epic

AGRUPAR POR Nada

POR HACER 1 INCIDENCIA

Realizar scafolding ambos proyectos

DISEÑO Y PLANEACION PROYECTO

MK-10 MB

EN CURSO 3 INCIDENCIAS

Realizar prototipos de front

DISEÑO Y PLANEACION PROYECTO

MK-5 MB

Seleccionar las fotos para hacer los esqueletos

DISEÑO Y PLANEACION PROYECTO

MK-6 MB

Planificar la arquitectura del sitio y hacer plano

DISEÑO Y PLANEACION PROYECTO

MK-7 MB

IN REVIEW

LISTO 3 INCIDENCIAS

Realizar mockups de pagina

DISEÑO Y PLANEACION PROYECTO

MK-4 MB

Seleccionar las tecnologías a trabajar

DISEÑO Y PLANEACION PROYECTO

MK-8 MB

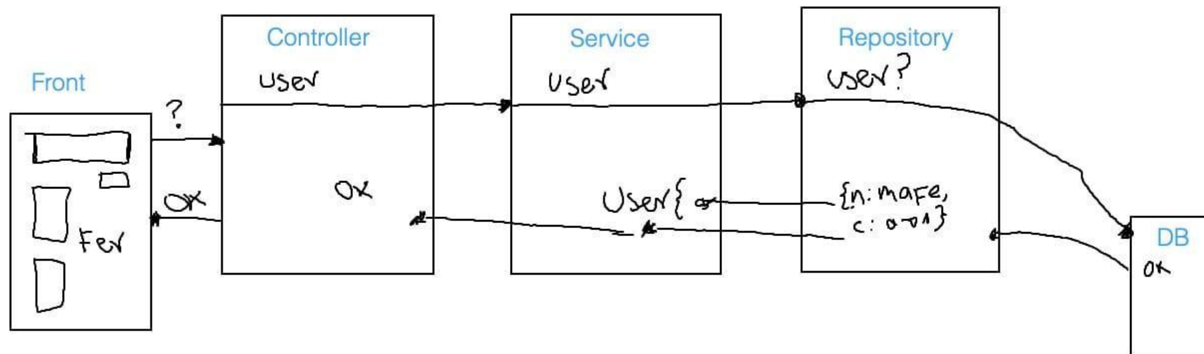
Crear proyectos en github para backend y frontend

DISEÑO Y PLANEACION PROYECTO

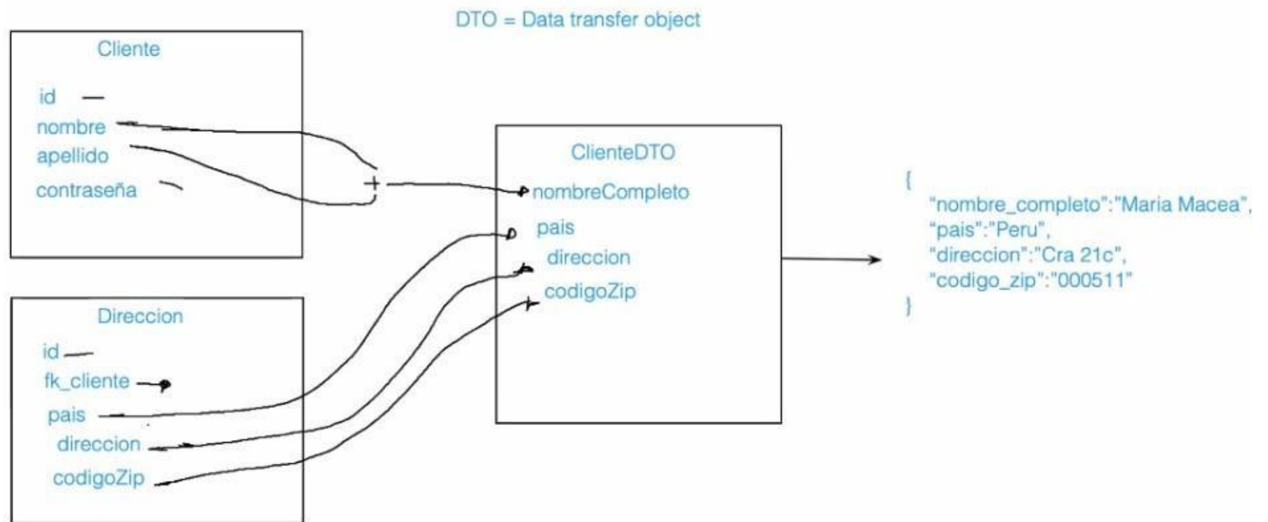
MK-9 MB

Anexo 2. Capa lógica

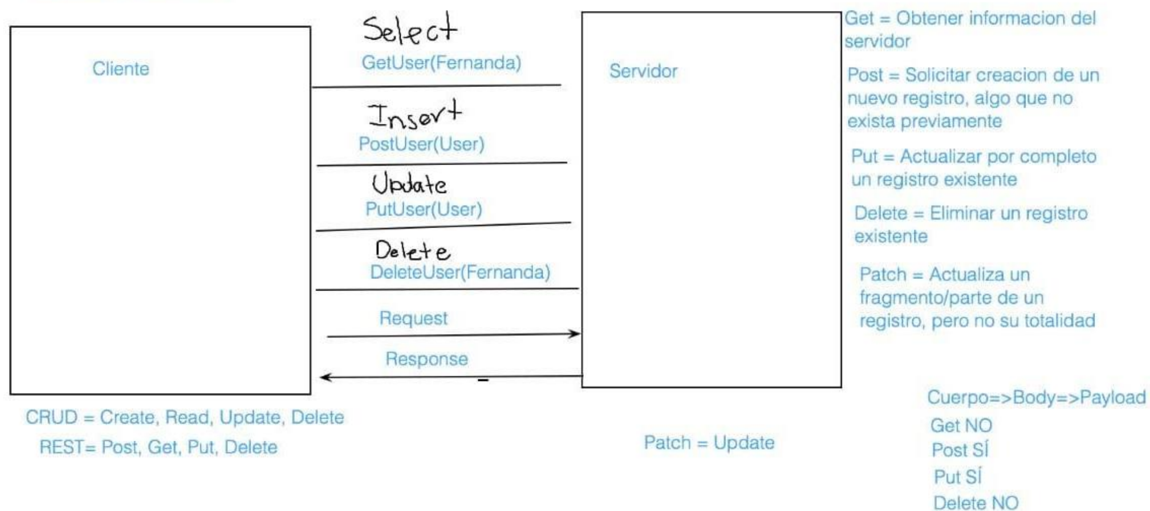
En el gráfico un ejemplo de la secuencia lógica que seguirá el backend



Anexo 3. DTO

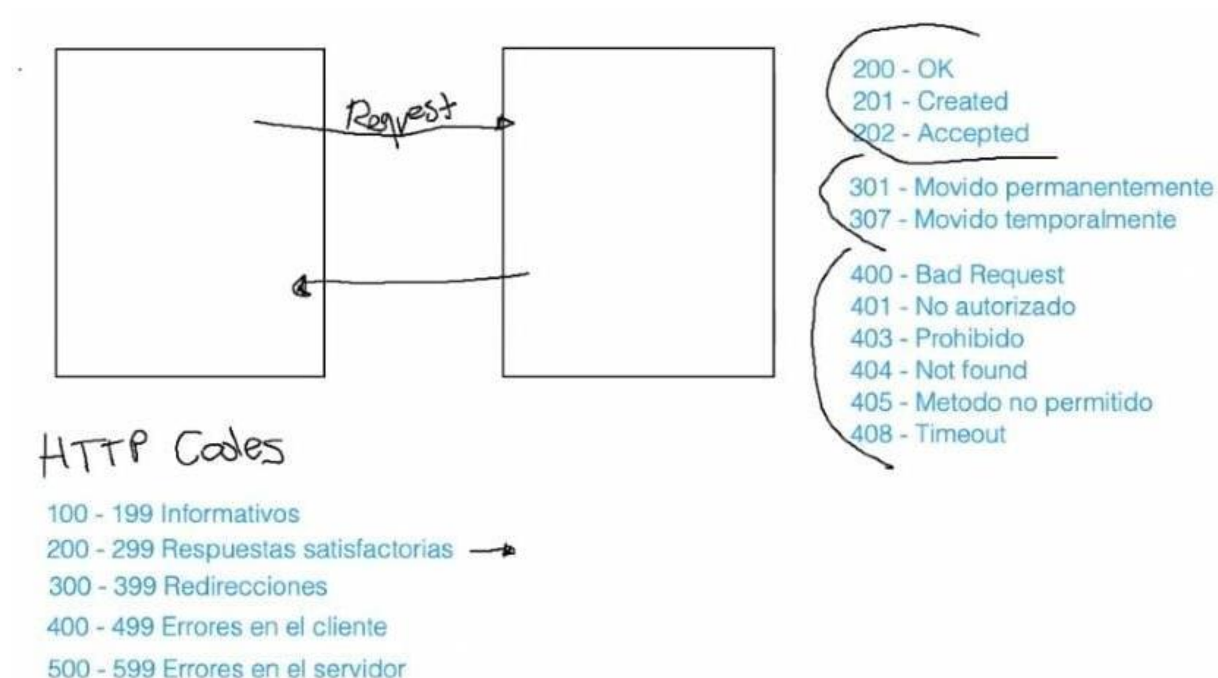


Anexo 4. Métodos



Anexo 5. HTTP Codes

Los códigos de estado de respuesta HTTP son mensajes del servidor que se insertan en una página web e indican si se ha completado satisfactoriamente una solicitud HTTP específica. En este trabajo se hizo uso de: "http 201 created" para responder a la solicitud de guardar un producto. "202 accepted" para responder a la solicitud de actualizar un producto. "400 bad request" para indicar un error.



Anexo 6. Trabajo final

Repositorios en GitHub

Backend: https://github.com/mariafmaceab/kafe_backend

Frontend: https://github.com/mariafmaceab/kafe_frontend

Aplicación web:

Credenciales para prueba: Usuario: mariafmaceab, Contraseña: sasha

Protocolo de operación e instalación:

MANUAL DE INSTALACION Y OPERACION

Colección Postman:

https://drive.google.com/file/d/1ZMX8Pslu1IuPeEh2k41t0z5Tc7fwayNQ/view?usp=share_link

https://kafe-frontend.herokuapp.com/login