UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PLATAFORMA PARA CREAR Y ADMINISTRAR UNA TIENDA VIRTUAL

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

CATALINA PAZ VILCHES VILCHES

PROFESORA GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
FRANCISCO GUTIÉRREZ FIGUEROA
PABLO GONZÁLEZ JURE

SANTIAGO DE CHILE
2022

Resumen

Para este trabajo de título se desarrolló una aplicación web que permite a usuarios administradores de un emprendimiento de productos tangibles, configurar una tienda virtual propia para que potenciales clientes puedan revisar el catálogo de los productos que ofrecen y realizar pedidos a través de ella.

En este informe se menciona la motivación para realizar esta plataforma, el contexto en el que se desenvuelve, el objetivo general y los objetivos específicos, se explica en detalle en qué consiste la solución desarrollada, se describe la metodología que se utilizó durante el semestre que se realizó el proyecto, se explican los conceptos fundamentales de las tecnologías utilizadas para entender el código desarrollado y se describen plataformas similares existentes en el mercado.

Las etapas del análisis del proyecto fueron: primero realizar un análisis del escenario escogido, confeccionando una lista de necesidades que tienen los emprendedores; luego se investigaron plataformas que permiten solucionar el problema abordado, obteniendo ideas sobre funcionalidades que se quieren implementar; finalmente se hizo el levantamiento de requisitos y se diseñó el prototipo de una solución al problema usando una aplicación web.

En este informe se muestra parte relevante y representativa del código que utiliza el sistema desarrollado para que los lectores puedan entender la complejidad del software y tener una idea sobre los aprendizajes obtenidos por la memorista para lograr el producto final.

Al final del informe se hace una validación funcional del sistema, mencionando e ilustrando en una tabla, cuáles fueron los requisitos que se abordaron con cada una de las interfaces que tiene la plataforma, concluyendo que se cumplió con la mayoría de los requisitos propuestos (15 de 18), obteniendo un software funcional que cumple las expectativas que se tenían al principio del semestre.

Agradecimientos

Agradezco a mi mamá y papá que me apoyaron en todos los aspectos durante mi formación académica y quienes son los mas felices con la finalización de este periodo. A mi hermana Rocío, quien siempre está ahí para ayudarme en lo que necesite.

Agradezco a mi profesora guía, Jocelyn, quien me brindó su ayuda en un momento muy complicado para mí, gracias a su experiencia, disposición y buena voluntad fue capaz de guiar mi trabajo logrando que pudiera desarrollar el proyecto y la escritura de este documento de forma amena, poniéndome metas realistas, siendo flexible y practica a la hora de tomar decisiones y priorizar las tareas que se debían realizar.

Agradezco a mis amigos Pancho, Juanma, Rodri, Cristi, Oleita y todos mis amigos que me acompañan en las buenas y en las malas, con quienes he compartido momentos que me hacen valorar la vida y me motivan a cumplir las metas que me propongo. También agradezco a los amigos que conocí en la U, Pelao, Dania, Fran, Nelson, Boro, y muchas otras personas que me acompañaron en las distintas etapas de mi vida universitaria.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Contexto

Las personas que deciden emprender un negocio deben encontrar una forma para llegar a sus clientes. Antes de que las redes sociales se popularizaran, la compra y venta de productos en línea sólo se podía realizar a través de plataformas especializadas para ello como Mercado Libre [8] y Yapo.cl [15] o a través de páginas web particulares de cada empresa, las cuales se han hecho accesibles con plataformas como Shopify [12] y Jumpseller [6].

1.2. Problema abordado

Los microempresarios que quieren tener una tienda virtual deben dedicar tiempo a investigar como crear una página web, cotizar las distintas alternativas y probar la que más le guste. Herramientas como Shopify y Jumpseller, ofrecen la posibilidad de crear una tienda virtual en pocos pasos y con una prueba gratuita, sin embargo, para usar este tipo interfaces se requiere tener experiencia en el uso de herramientas digitales, el cual muchas veces un microempresario no posee [10] por lo que les es más factible invertir dinero en contratar a alguien que construya la página web utilizando estas mismas herramientas.

Este trabajo de título pretende desarrollar un sistema que ayude a microempresarios que poseen poco manejo en herramientas digitales, a través de una plataforma que le permita a cualquier persona crear una tienda virtual de una forma rápida e intuitiva.

Esta solución se diferencia de las demás en que está enfocada en microempresas Chilenas y en que la plataforma no contará con tantas opciones como las que ofrecen Shopify y Jumpseller. Además tendrá un enfoque en ayudar a emprendedores que necesitan recopilar datos automáticamente de las ventas que realizan, para continuar trabajando en su negocio de una manera más informada.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo general de este trabajo de título es desarrollar una plataforma que permita a microempresarios sin conocimientos en programación, administrar una tienda virtual y que potenciales clientes puedan comprar sus productos a través de ella.

1.3.2. Objetivos Específicos

1. Investigar y evaluar las soluciones existentes en el mercado.
2. Diseñar la experiencia de usuario para que microempresarios sin conocimientos en programación, puedan crear y administrar una tienda virtual.
3. Diseñar la experiencia de usuario para que clientes puedan comprar productos a través de una tienda virtual.
4. Implementar una aplicación web en base a las experiencias de usuario diseñadas en los dos objetivos específicos anteriores.
5. Validar la aplicación web desarrollada, a través de una validación funcional de los requisitos levantados en la primera etapa del proyecto.

1.4. Solución desarrollada

Para la realización de este trabajo de título se desarrolló una plataforma que permite a dueños y/o trabajadores de una empresa de venta de productos tangibles administrar una tienda virtual. Por otro lado, la plataforma permite que usuarios puedan comprar productos a través de las tiendas virtuales creadas previamente.

Se decidió que la plataforma sería una aplicación web ya que de esta forma los usuarios pueden utilizar un navegador desde su computador, tablet o smartphone para acceder a la plataforma, sin tener que instalar un software extra. Para desarrollar la aplicación web se utilizó el framework Django para el backend y React para el frontend. Se hizo uso de la herramienta Django REST framework que sirve para simplificar la utilización de llamadas REST y de esta forma conectar el backend con el frontend.

La plataforma tiene dos tipos de usuarios: administradores y clientes. Los administradores deben tener una cuenta en el sistema para poder utilizarlo. Los clientes son usuarios del sistema que no necesitan tener una cuenta creada previamente. Para acceder al sistema deben utilizar la url de una tienda virtual configurada previamente por un administrador.

Para este trabajo de título se priorizó que los clientes puedan realizar el flujo completo de una compra, con lo que lo primero que se desarrolló fueron las funcionalidades necesarias para que los clientes puedan elegir y comprar productos. Una vez que se tuvo un producto

mínimo viable para poder realizar compras, se continuó desarrollando un módulo para que los administradores puedan monitorear los pedidos que han realizado sus clientes a través de una tabla que muestra los pedidos pendientes.

Para realizar el deployment del sistema se usó la plataforma Heroku [11], a través de la cual se puede usar un hosting gratuito que servirá para que administradores y clientes puedan validar la usabilidad del sistema. Para esta primera etapa del proyecto, se creará una aplicación en Heroku por cada tienda virtual, separando así las bases de datos de cada negocio. En todas las aplicaciones se utilizará el gestor de bases de datos PostgreSQL.

Se creó una tienda virtual ficticia llamada *Rushop*, en la cual hubo que crear una app de Heroku para el backend y otra para el frontend. Se utilizó Git para el versionamiento del código y la plataforma Gitlab [5] para almacenar el código en la nube y poder actualizarlo tanto en Heroku como en los computadores que utilice la desarrolladora.

1.5. Metodología

Para el desarrollo del proyecto se utilizó la metodología SCRUM [2], por lo que el semestre se dividió en bloques de dos semanas llamados sprints.

Antes de iniciar cada sprint se planificaron las tareas que se debían realizar, las cuales estaban estimadas con su respectivo puntaje. El puntaje se midió en días de trabajo, un punto equivale a un día entero de trabajo. La suma de los puntajes de las tareas planificadas para un sprint no debía pasar de 8.5 puntos, ya que en dos semanas se tenía diez días de trabajo y se consideró que 1.5 puntos podría usarse en: dedicar tiempo a tareas administrativas relacionadas al proyecto, si alguna tarea tomaba más tiempo del estimado o si la memorista tenía algún inconveniente que no le permitiera trabajar los 10 días del sprint.

Cada sprint tenía un objetivo que se debía cumplir y la idea es que se desarrollaran correctamente todas las tareas que fueron asignadas. Al final de cada sprint se hizo una retrospectiva y una demostración a la profesora guía de los logros alcanzados.

Para mantener el orden de las tareas, priorizarlas y asignarlas a cada sprint, se utilizó la plataforma Jira, la cual cuenta con las funcionalidades necesarias para realizar proyectos usando la metodología SCRUM.

1.6. Estructura del documento

En el Capítulo 2, se tiene un marco teórico donde se nombra y explica brevemente en que consisten las tecnologías, herramientas y conceptos utilizados en este documento. Además se muestran plataformas que proponen una solución similar al problema abordado.

En el Capítulo 3 se describe un análisis y diseño del producto implementado. En este capítulo se muestra la primera etapa del proyecto, en la cual se hizo un levantamiento de

requisitos a partir de una lluvia de ideas de necesidades detectadas en el escenario escogido. Luego se muestra el diseño de la solución propuesta a través de diagramas de arquitectura y del modelo de datos, además de mockups realizados con la plataforma Balsamiq [1].

En el Capítulo 4 se detalla como esta implementada la solución desarrollada. Primero se muestra la estructura del proyecto, tanto del backend como frontend. Luego se explica en qué consiste el código desarrollado, introduciendo los conceptos técnicos de cada tecnología utilizada y mostrando líneas de código relevantes y ejemplificadoras.

En el Capítulo 5 se hizo una descripción del proceso de desarrollo del proyecto. Para esto se muestra la metodología que se utilizó durante el semestre, el cual se dividió en sprints de dos semanas de duración. En este capítulo se detalla que se hizo en cada sprint y luego se hace una retrospectiva general de la metodología utilizada.

En el Capítulo 6 se hizo una validación funcional del sistema, mostrando las interfaces que tiene el producto final y haciendo una evaluación de si se cumplieron o no los requisitos levantados en la primera etapa del proyecto.

Por último, en el Capítulo 7 se tienen las conclusiones del trabajo realizado.

Capítulo 2

Marco teórico

2.1. Conceptos y herramientas utilizadas

2.1.1. Conceptos generales

- **URL:** sigla de *Uniform Resource Locator* o localizador de recursos uniforme en español, es la dirección que se utiliza para acceder a un recurso web. Está compuesta por el nombre del dominio y por la dirección relativa del recurso.

  Generalmente los navegadores muestran la URL de las páginas web en una barra de direcciones ubicada en la parte superior de la pantalla. Una URL típica tiene la forma `http://www.example.com/index.html`, la cual indica el protocolo (`http`), el nombre del host (`www.example.com`) y el nombre del archivo (`index.html`).

- **API REST:** Una API (sigla de *Aplication Program Interface*) es un programa que permite que dos sistemas independientes, se comuniquen entre sí. Una API REST es una interfaz de programación de aplicaciones que utiliza la arquitectura REST (sigla de *Representational State Transfer*), la cual permite utilizar eficientemente y de forma segura los recursos web.

- **ORM:** sigla de *Object-Relational Mapping* en inglés, es una técnica que permite consultar y manipular una base de datos utilizando el paradigma de programación orientada a objetos.

2.1.2. Django

En esta sección se explican los principales objetos que se utilizan en Django para crear una API. Estos son los modelos, los serializadores y las vistas. En la figura 2.1 se puede ver un diagrama de cómo se relacionan estos distintos objetos entre ellos, con la base datos, la API y el frontend.

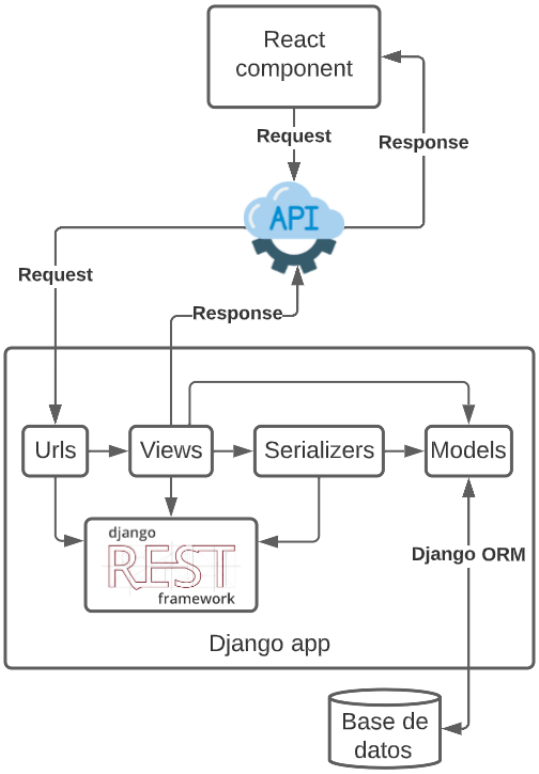

Figura 2.1: Diagrama de una app de Django

- **Modelos:** En Django un modelo es la única fuente de información sobre los datos del proyecto, conteniendo todas las tablas, sus atributos y cómo se relacionan entre ellos. Los modelos de cada aplicación se guardan en un archivo llamado `models.py`. Un modelo es una clase de Python, al correr el comando: `python manage.py migrate` se traducen los cambios realizados en los modelos a sentencias SQL para crear automáticamente las tablas en la base de datos configurada previamente en el archivo `settings.py`.

- **Migraciones:** Cuando se modifica un modelo, el comando `makemigrations` del modulo `manage.py` permite crear archivos llamados migraciones, los cuales contienen el código necesario para actualizar la base de datos. Las migraciones se guardan en el directorio `migrations` de cada app. Estos archivos sirven para tener una historia de los cambios que se realizan al esquema de la base de datos a lo largo del desarrollo del proyecto. En la figura 2.2 se puede ver un diagrama de la interacción entre los modelos, la base de datos y las migraciones.

- **Django REST Framework** [4]: Es una librería de Python que permite crear una API REST a través del framework Django. Contiene diversas clases y objetos que permiten facilitar la implementación de una API REST.

- **Serializadores:** Los serializadores permiten convertir objetos complejos como modelos y `querysets` a datos nativos de Python que pueden ser convertidos fácilmente a JSON, XML, entre otros tipos de contenido. Los serializadores también permiten des-serializar, es decir, realizar la transformación inversa: convertir datos de tipo JSON u otros, a datos complejos, después de realizar una validación.

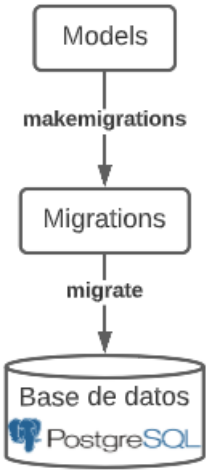

Figura 2.2: Django ORM

- **Vistas:** En Django una vista o `view`, es una función de Python que recibe como argumento una petición a través del protocolo HTTP (`request`) y retorna una respuesta a través del mismo protocolo (`response`). Esta respuesta puede ser contenido en distintos formatos (HTML, XML, un error 400, etc). La vista en sí contiene una lógica arbitraria necesaria para retornar la respuesta. Por convención, las vistas se ubican en un archivo llamado `views.py` de cada aplicación.
- **ViewSet:** Django REST framework permite combinar la lógica de varias vistas relacionadas en la misma clase, llamada `ViewSet`. En otros frameworks este tipo de clase se conoce como recurso (*resource*) o controlador (*controller*). Una instancia `ViewSet` es una clase que tiene métodos como `list` para listar objetos de un modelo y `create` para crear un objeto.

2.1.3. React

React es una librería de Javascript que permite construir interfaces de sistemas interactivos, tiene distintas características que permiten facilitar el desarrollo del frontend y mejorar la experiencia de usuario. En React, las interfaces se pueden encapsular en pequeñas partes conocidas como componentes, los cuales se renderizan eficientemente y pueden ser reutilizables en distintas partes de la plataforma.

- **Elementos:** React permite generar elementos, conocidos en inglés como `elements`, a través del lenguaje JSX. Este lenguaje es una combinación entre Javascript y HTML. Por ejemplo, se puede generar el elemento que contiene el texto "Hola mundo" con la siguiente línea de código: `const element = <h1>Hola mundo!</h1>;`.
- **Componentes:** Un componente es una función o una clase de Javascript que recibe un único argumento llamado `props` y retorna un elemento que puede contener otros componentes. Los componentes pueden ser utilizados en distintas partes del código, se llama padre e hijo a la relación que se genera entre un componente que llama a otro.

- **Contexto:** En una aplicación típica de React, los datos se pasan de un componente a otro (de un padre a un hijo) a través de un argumento que se conoce como *props*. Para algunos casos, como para preferencias locales de un usuario o el tema de la interfaz, el uso de props se puede volver incómodo y/o poco viable. El uso de contexto sirve para compartir valores a través de distintos componentes que no necesariamente estén relacionados entre sí, sin tener que pasar el valor explícitamente en las props de cada componente que lo requiera.
- **Material UI [7]**: Es una librería para React que contiene distintos componentes que se pueden utilizar para facilitar el desarrollo de las distintas interfaces de un sistema.

2.2. Soluciones existentes

2.2.1. Shopify

Shopify [12] es una plataforma para crear, personalizar y gestionar una tienda virtual sin la necesidad de saber programar. Puede ser utilizada en 175 países. Para poder crear la tienda debes crear un usuario en la plataforma y contratar una suscripción mensual con precios que van desde los $29 USD mensuales más 2,5 % de comisión por venta, a los $299 USD mensuales + 0,5 % de comisión por venta.

Para tener un diseño personalizado o agregar funcionalidades extra se debe contratar a alguien que tenga conocimientos técnicos (CSS, HTML, Javascript, etc). En general la plataforma es muy completa, pues se tienen muchas opciones que son útiles para los emprendedores.

2.2.2. Jumpseller

Jumpseller [6] es una plataforma similar a Shopify, creada el año 2010, puede ser utilizada en 20 países, sin embargo como tiene oficinas en Chile está orientado a los países latinoamericanos. Sirve para crear una página web que cuente con una tienda virtual, sin la necesidad de saber programar. Los precios van desde los $14.000 mensuales hasta los $180.000 mensuales, variando según la cantidad de productos que se quieran tener en el catálogo. No cobra comisión por venta. Se puede pedir una prueba gratis por 14 días para familiarizarse con el sistema y ver si se adecua al negocio.

Las funcionalidades y diseño que se le quiere dar a la página web vienen predeterminados por la plataforma. Para poder agregar alguna funcionalidad extra o tener un diseño personalizado se debe tener conocimientos técnicos (CSS, HTML, Javascript, etc).

2.2.3. Woocomerce

WooCommerce [14] es una herramienta de código abierto para páginas web creadas con WordPress. La compañía comenzó en 2008 como WooThemes y en 2017 decidió focalizarse exclusivamente en E-Commerce. Precio: Gratuito, solo se debe pagar el hosting, por lo que el precio es muy variable (la página ofrece por defecto un plan de $59 USD mensual). Se debe tener conocimientos técnicos (Wordpress) para crearla y personalizarla.

También hay empresas dedicadas a crear E-Commerce para pequeñas y medianas empresas, Justo es una marca que cuenta con más de 2 mil partners en Chile, esta empresa en un inicio estaba enfocada en crear tiendas online para restaurantes y dado su crecimiento es que ahora también tiene partners de rubros distintos al gastronómico, como empresas de tecnología y papelería.

Capítulo 3

Análisis y Diseño

En este capítulo se presenta la primera etapa del proyecto. En la sección 3.1 se hace un análisis de cuáles fueron las necesidades detectadas y un levantamiento de requisitos del producto desarrollado. Luego, en la sección 3.2 se muestra el diseño de la solución a través de un diagrama de la arquitectura del sistema, un diagrama de entidad-relación del modelo de datos y mockups que muestran una idea de las interfaces que se desarrollaron.

3.1. Análisis

Para iniciar el trabajo se definió un escenario en el cual se quería trabajar. El escenario que se eligió fue el de pequeñas y medianas empresas que venden productos tangibles principalmente por redes sociales (desde este momento se utilizará indistintamente *empresa*, *emprendimiento* o *negocio* para referirse a una pequeña o mediana empresa).

Los actores relevantes del escenario escogido son los o las dueñas de un emprendimiento, sus trabajadores y sus clientes.

3.1.1. Necesidades

Para detectar necesidades y encontrar un problema a resolver del escenario escogido, se observaron tres empresas de venta de productos. Las observaciones se realizaron informalmente y en variadas ocasiones entre julio y diciembre de 2021, además se entrevistó a la dueña de un emprendimiento de artículos de papelería (el 12 de enero de 2022) para indagar más en detalle sobre el escenario de estudio y el comportamiento de la persona observada en este contexto.

El propósito de esta tarea era identificar cómo trabajan los actores relevantes del escenario escogido para conocer las prácticas habituales que realizan, los problemas que tienen, sus expectativas, preferencias, cómo interactúan con sus clientes y colegas. De esta forma se podría tener la información necesaria para abordar una solución factible y útil a un problema

común del escenario escogido.

Se hizo una lluvia de ideas con necesidades detectadas luego de realizar las observaciones, la entrevista y la propia experiencia de la memorista.

1. Tener una base de datos de productos
2. Calcular las utilidades de la micro empresa
3. Calcular automáticamente el precio de una venta en particular
4. Cotizar precios de insumos en distintos puntos de venta
5. Saber precios de los productos de la competencia
6. Mantener las redes sociales actualizadas
7. Responder todos los mensajes de potenciales clientes
8. Aumentar la llegada a nuevos clientes
9. Aparecer en la primera página de Google cuando se busca el nombre del emprendimiento
10. Concretar repartos a domicilio exitosamente
11. Aumentar la capacidad y/o disponibilidad de repartos
12. Fidelizar clientes
13. Optimizar stock de productos
14. Aumentar capacidad de almacenamiento de productos
15. Optimizar almacenamiento de productos

Se decidió abordar principalmente las necesidades 1, 2, 3 y 8, a través de una plataforma que permita a micro empresarios administrar una tienda virtual donde puedan tener una base de datos de sus productos, recopilar información de las ventas que están realizando para conocer las utilidades de su emprendimiento, calcular automáticamente el precio de una venta y aumentar la llegada a clientes que prefieren comprar a través de una tienda virtual.

3.1.2. Requisitos funcionales

Para esta primera etapa del proyecto se decidió trabajar con dos tipos de usuarios del sistema: los usuarios administradores y los clientes. Los administradores pueden ser él o la dueña de una empresa o bien alguno de sus trabajadores. Un cliente es cualquier persona que utilice una tienda virtual creada previamente por algún administrador.

A partir de las necesidades que se decidió abordar, se confeccionó una lista de requisitos a alto nivel para desarrollar una plataforma que permita cubrir las necesidades planteadas.

1. La dueña de una tienda (desde ahora administradora) quiere crear una página web para vender productos a través de ella.
2. La administradora quiere agregar, editar y borrar productos a la tienda virtual con su respectiva imagen, descripción y precio de venta para que estos sean mostrados en un catálogo.
3. La administradora quiere personalizar su página web con la información necesaria para que los usuarios que visitan la página quieran comprar desde la plataforma y sepan como realizar una compra.
4. Un cliente quiere ver los productos que hay a la venta para elegir que comprar.
5. Un cliente quiere conocer la información detallada de un producto: una imagen que lo muestre, su descripción, precio, unidad de venta, entre otros.
6. Un cliente quiere agregar un producto a su carro de compras, seleccionando la unidad o tamaño del producto y la cantidad que desea comprar.
7. Un cliente quiere revisar su carro de compras mientras ve el catálogo y/o la ficha de un producto para verificar si ya lo agregó a su pedido o para revisar cual es el precio total de la compra.
8. Un cliente quiere agregar o quitar productos de su carro de compras por si se arrepiente de comprar un determinado producto, por si el precio total es mayor al presupuesto que tiene y/o si desea aumentar o disminuir la cantidad de algunos productos.
9. Un cliente quiere conocer la información de una tienda: donde está ubicada, su teléfono, correo electrónico o redes sociales para contactarse a través de alguno de esos medios.
10. Un cliente quiere elegir el método de pago de su compra para pagar de la forma que mas le acomoda.
11. Un cliente quiere elegir si desea que la compra le llegue a su domicilio o si prefiere ir a buscarla a la tienda para recibir el pedido de la forma que más le acomoda.
12. Un cliente quiere revisar los productos que va a comprar antes de confirmar el pedido.
13. Un cliente quiere recibir un correo electrónico con la información de su pedido y/o como comprobante de la compra que realizó.
14. La administradora quiere ver los pedidos que se han hecho a través de la página web para monitorear los pedidos que debe entregar.
15. La administradora quiere recibir una notificación cada vez que un cliente haga una comprar a través de la tienda virtual para estar informada sobre los pedidos que debe entregar.
16. La administradora quiere tener información sobre los clientes que tiene para ver quienes compran mas de una vez, los pedidos que hacen y para saber sus datos (número de teléfono, dirección, correo electrónico, etc) por si necesita contactarlos cuando hacen un pedido.
17. La administradora quiere saber cuántas ventas ha realizado y cual es el monto de las ventas en un tiempo determinado para saber cuánto dinero ha ganado.

18. La administradora quiere saber cuánto dinero gana por cada producto vendido o en determinadas ventas considerando lo que gasta en los insumos o precio de compra de los productos.

3.1.3. Requisitos no funcionales

- Cada tienda virtual debe tener su base de datos de productos e información relevante propia. Los productos que se agreguen en una tienda no pueden ser accesibles por otra tienda virtual creada a través de la plataforma.
- Debe garantizarse la privacidad de los datos personales y bancarios de personas que realicen compras en línea para que no puedan ser utilizados maliciosamente por terceros.
- El sistema debe ser seguro para los administradores, sus datos deben ser privados y se debe usar un sistema que asegure que solo ellos pueden acceder a su cuenta y editar la información de su tienda virtual.

Los primeros requisitos que se abordaron son los relativos a la compra de productos, esto porque se consideró lo más importante, ya que sin la funcionalidad de que los clientes puedan comprar a través de la plataforma, los administradores no tendrían un incentivo para utilizar el sistema.

Una vez que ya se pudo realizar el flujo completo de una compra, se empezó a desarrollar el módulo para que los administradores puedan tener información sobre las ventas que realizan a través de la tienda virtual y/o a través de las ventas que realicen por otros medios e ingresen manualmente al sistema. Para esto se desarrolló una interfaz donde se muestra una tabla con los pedidos pendientes.

Posterior a este trabajo de título se continuará mejorando el software y desarrollando lo relativo a mejorar la experiencia de usuario de los administradores, creando interfaces intuitivas y usables por cualquier persona que no esté acostumbrada a manejar sistemas digitales.

3.2. Diseño

En esta sección se mostrará la etapa de diseño de la aplicación web implementada. Para esto se confeccionó un diagrama de la arquitectura que utiliza el sistema, luego se tiene un diagrama de entidad-relación para mostrar el modelo de datos y finalmente se muestran los mockups que se hicieron con el software *Balsamiq*.

3.2.1. Arquitectura

El producto desarrollado tiene como cliente directo a dueños/as o administradores de cualquier empresa que venda productos tangibles. La idea es que si alguien decide crear su

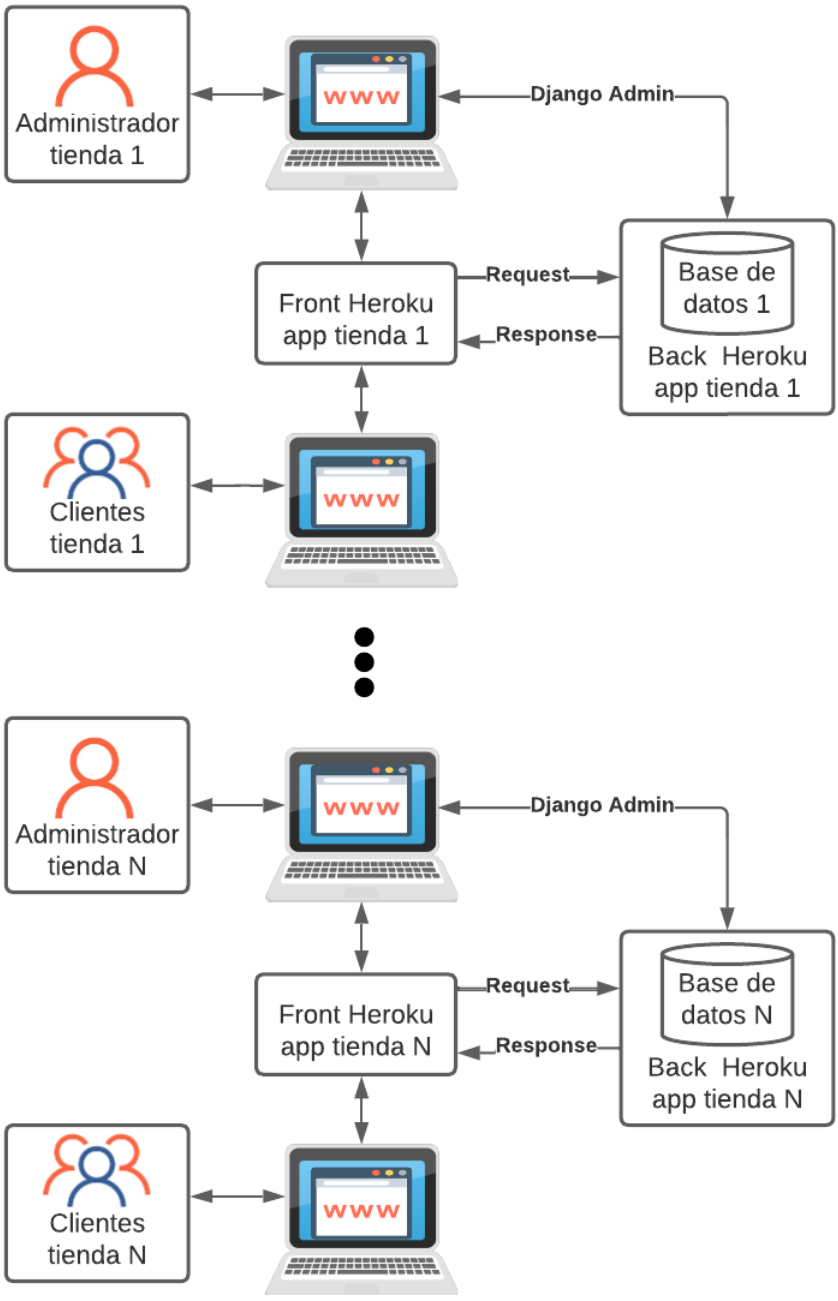

Figura 3.1: Arquitectura de N tiendas

tienda virtual a través de esta plataforma pueda hacerlo sin necesidad de saber programar y solo deba ingresar la información de su tienda y los productos que ofrece a través de la interfaz que se desarrolló para ello.

Para esto se decidió que la forma mas simple y factible de desplegar varias aplicaciones web simultáneamente sería levantar un servidor propio por cada tienda virtual. Esto permite simplificar el modelo de datos ya que solo hay que considerar los datos necesarios de una sola tienda, además permite simplificar el uso de recursos ya que si una tienda esta siendo visitada por muchos clientes se puede saber fácilmente donde hay que mejorar los recursos del servidor y que esto no afecte el costo de otra tienda.

Se decidió utilizar la arquitectura cliente-servidor para desarrollar el proyecto. Para esto se creó un proyecto Django para el backend y uno React para el frontend. Cada proyecto tiene su propio repositorio en Gitlab.

La plataforma *Heroku*[11] permite levantar una aplicación web en pocos pasos y alojarla en un servidor que esté funcionando las 24 horas del día gratuitamente. Se decidió implementar una tienda de prueba a través de esta plataforma. Para esto se tuvo que crear dos aplicaciones (conocidas como *apps*) de *Heroku*, una para el backend y otra para frontend.

En la figura 3.1 se puede ver un diagrama de la arquitectura de dos tiendas virtuales distintas, están enumeradas con un 1 y una N para indicar que la arquitectura es similar para N cantidad de tiendas distintas.

Se consideró que los dueños/as de las empresas gestionen su página a través del módulo de administración que ofrece Django (en un futuro se pretende desarrollar un módulo de administración propio), es por esto que para agregar la información de la tienda y los productos, los administradores deben usar directamente el backend.

3.2.2. Modelo de datos

Considerando los requisitos planteados en la sección anterior, la arquitectura mostrada previamente y los antecedentes dados sobre las funcionalidades que tendrá el sistema, se confeccionó un diagrama entidad-relación que representa el modelo de datos de la plataforma. Este se puede ver en la figura 3.2.

Se decidió separar el proyecto del backend en tres aplicaciones de Django, en una se tienen los modelos relacionados a la venta de productos (color rosado), en otra lo relacionado a la información y personalización de la tienda (en color celeste) y por último, una aplicación donde se desarrolle lo relacionado a los usuarios del sistema y sus características (color verde).

Se creó la entidad *ProductOrder* para guardar las opciones que elige un usuario al seleccionar un producto. De esta forma, a un pedido se le asocia una lista de *ProductsOrder* en vez de directamente productos.

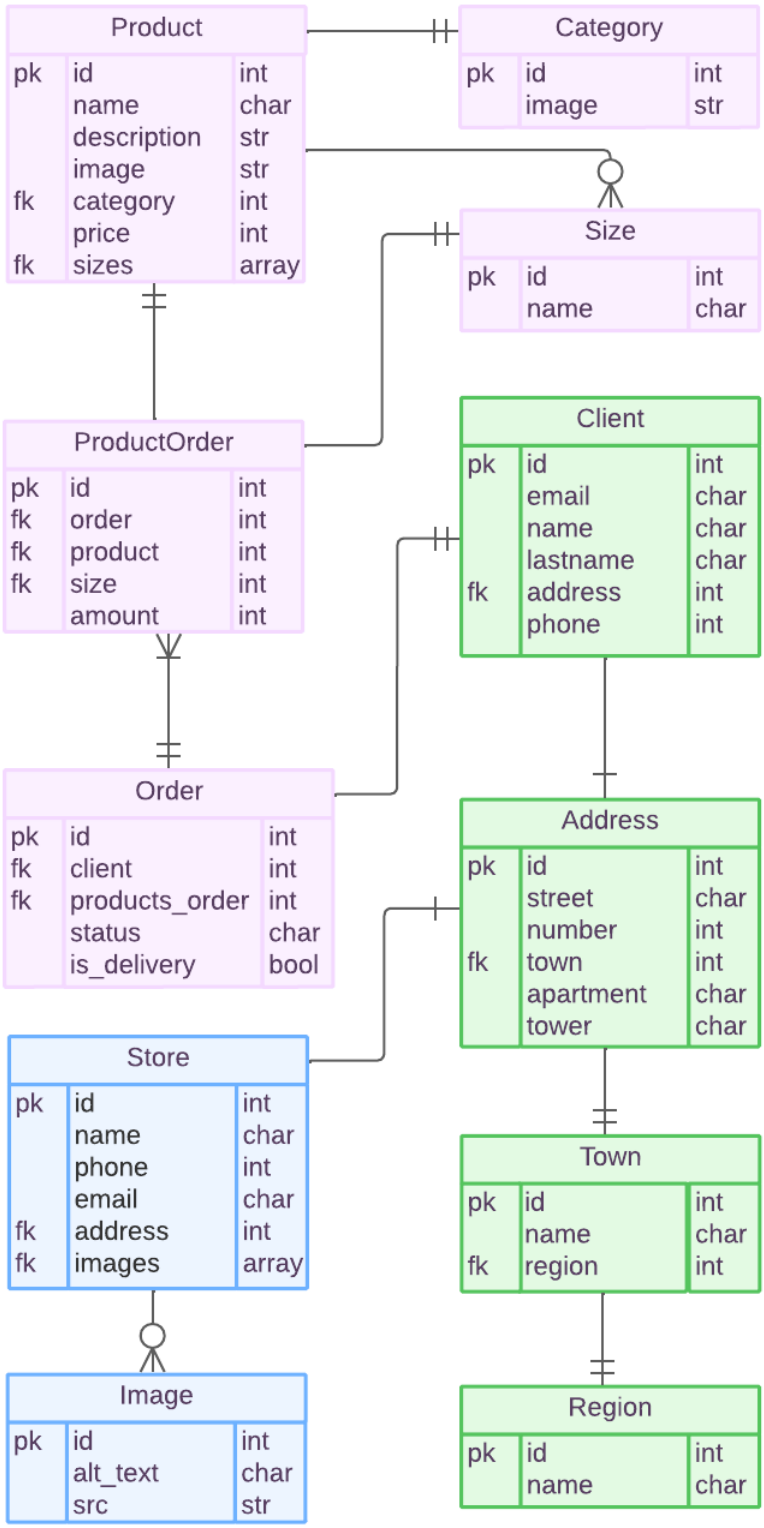

Figura 3.2: Modelo de Datos

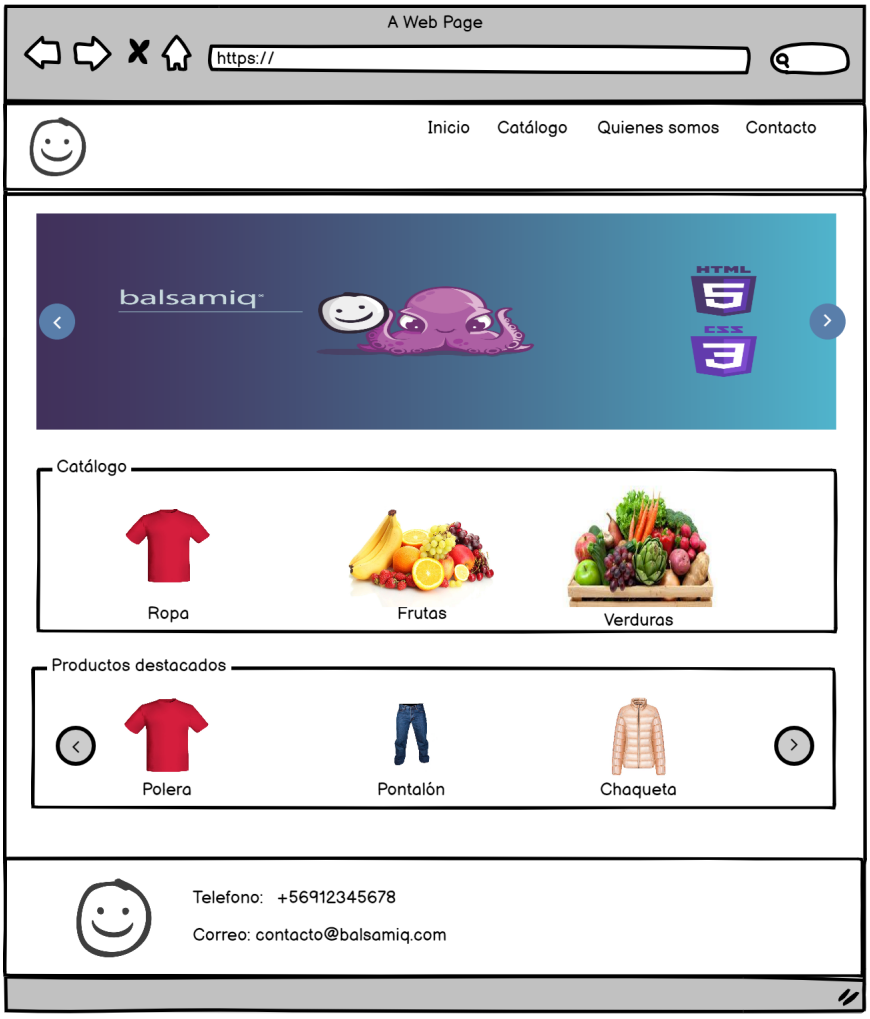

Figura 3.3: Interfaz de una tienda virtual vista por un cliente

3.2.3. Diseño de interfaces

Se confeccionó un prototipo de la solución a través de bocetos de las principales interfaces del sistema. Para crear y generar la imágenes se utilizó la plataforma *Balsamiq*.

En la figura 3.3 se puede ver el mockup de la vista de un cliente de la página principal de una tienda virtual. Esta interfaz contiene el catálogo de los productos y la información de contacto de la tienda por lo que sirve para cubrir los requisitos 4 y 9.

Cuando un cliente selecciona un producto del catálogo, se muestra la ficha del producto, el boceto de esta interfaz se puede ver en la figura 3.4. Con esta interfaz se cumplen los requisitos 5 y 6, ya que con esta vista se puede ver la información detallada de un producto, seleccionar el tamaño y cantidad que desea comprar, y agregarlo a su pedido.

Se realizó un mockup de un carrito de compras de un cliente (ver la figura 3.5). Se consideró que el carrito podría ser una ventana que se muestre encima de las demás vistas para cumplir el requisito 7. Además con esta interfaz se cumpliría el requisito 8 pues el carrito muestra los productos que se han agregado, permite agregar y quitar productos, y ver

Figura 3.4: Ficha de un producto

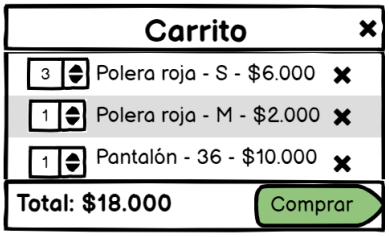

Figura 3.5: Interfaz de un carrito de compras

	Figura 3.3	Figura 3.4	Figura 3.5	Figura 3.6	Figura 3.7	Figura 3.8	Figura 3.9
1							✓
2						✓	
3							✓
4	✓						
5		✓					
6		✓					
7			✓				
8			✓				
9	✓						
10				✓			
11				✓			
12				✓			
13							
14					✓		
15							
16				✓	✓		
17					✓		
18					✓		

Tabla 3.1: Evaluación cuantitativa de los requisitos funcionales enumerados en la sección 3.1.2 que se cumplen en el prototipo de la solución.

el precio total de la compra.

En la figura 3.6 se puede ver el boceto de la interfaz donde se muestra el detalle de un pedido. Además en esta vista se tiene el formulario para que el cliente agregue sus datos de contacto, su dirección en caso de que pida reparto, elección del método de pago y confirmación del pedido. Este boceto se confeccionó para cumplir los requisitos 10, 11, 12 y 16.

En la figura 3.7 se puede ver el módulo de pedidos de la plataforma. Este módulo permite a los administradores de una tienda visualizar los pedidos pendientes que tienen, los pedidos por mes y por cliente. De esta forma pueden tener información sobre las ganancias de su negocio. Con esta interfaz se cumplen los requisitos 14, 16, 17 y 18.

En el módulo de administración se puede modificar el catálogo de productos, la interfaz para realizar esta acción se puede ver en la figura 3.8. Con esta se cumple el requisito 2.

Por último, para cumplir los requisitos 1 y 3 se confeccionó un mockup del módulo de administración y personalización de las tiendas virtuales. Este mockup se puede ver en la figura 3.9.

Para ilustrar los requisitos que se cumplen con cada boceto, se hizo una tabla que cruza los requisitos con los mockups (ver tabla 3.1). Cada fila corresponde al requisito que está siendo referenciado.

Los requisitos 13 y 15 no se cubren en los mockups ya que se consideró redundante mostrar los correos que se manden a los clientes y al administrador cuando se realiza un pedido pues la información que contienen es principalmente la lista de productos y el precio total de la compra, lo cual ya se muestra en el mockup del detalle de un pedido y en el carrito.

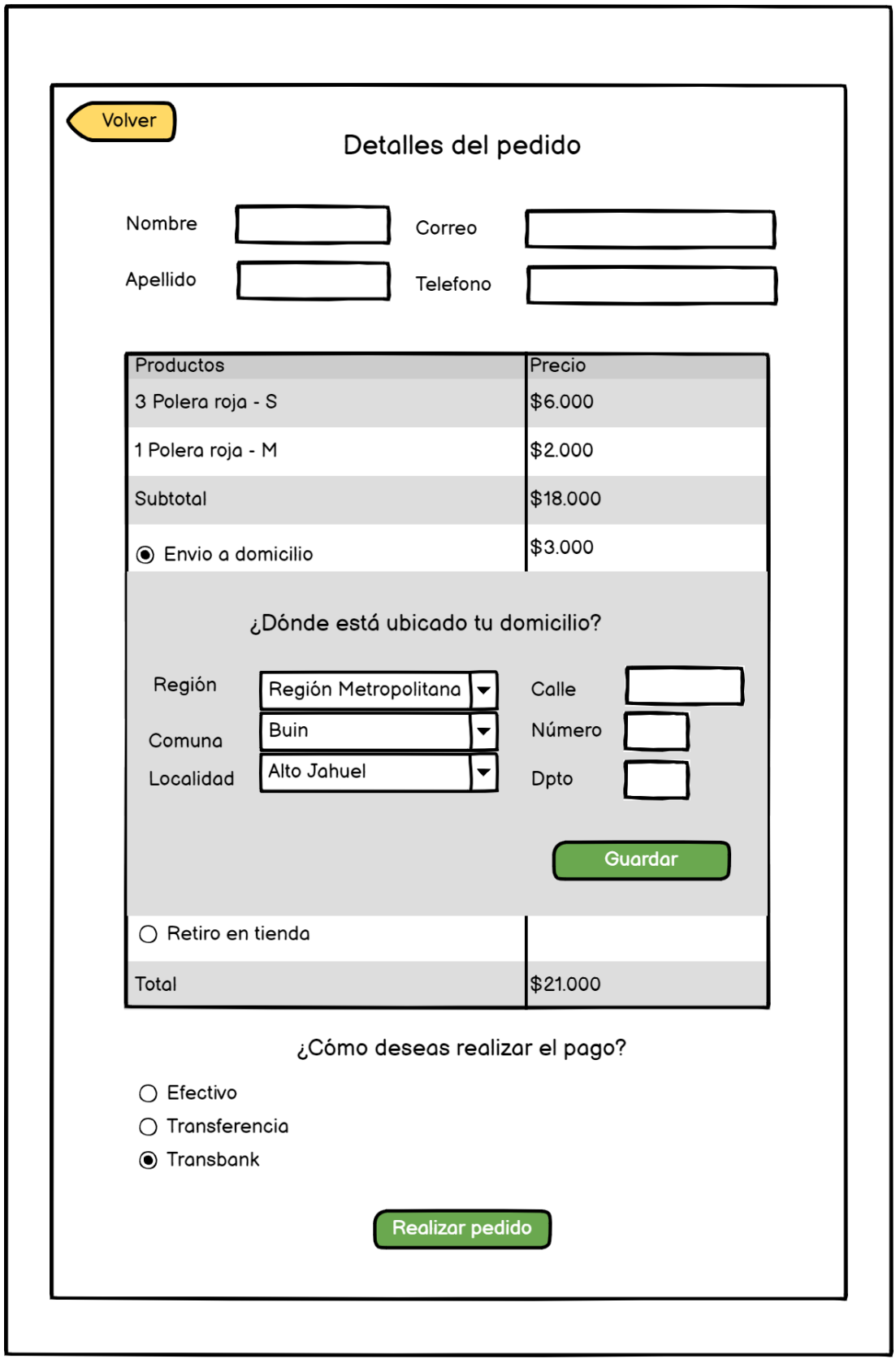

Figura 3.6: Interfaz donde se muestra el detalle de un pedido al cliente que lo realizó.

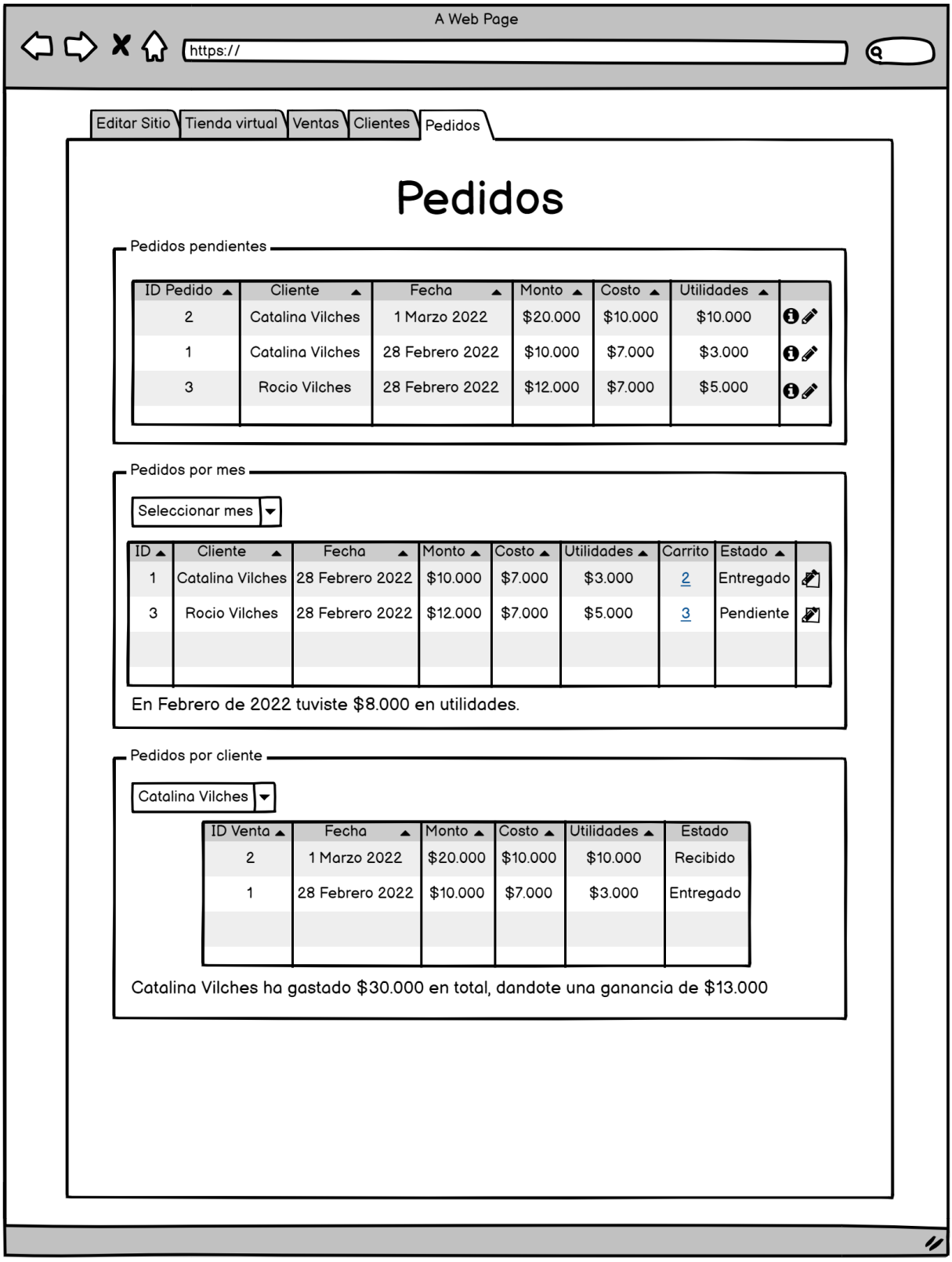

Figura 3.7: Módulo de pedidos para el administrador

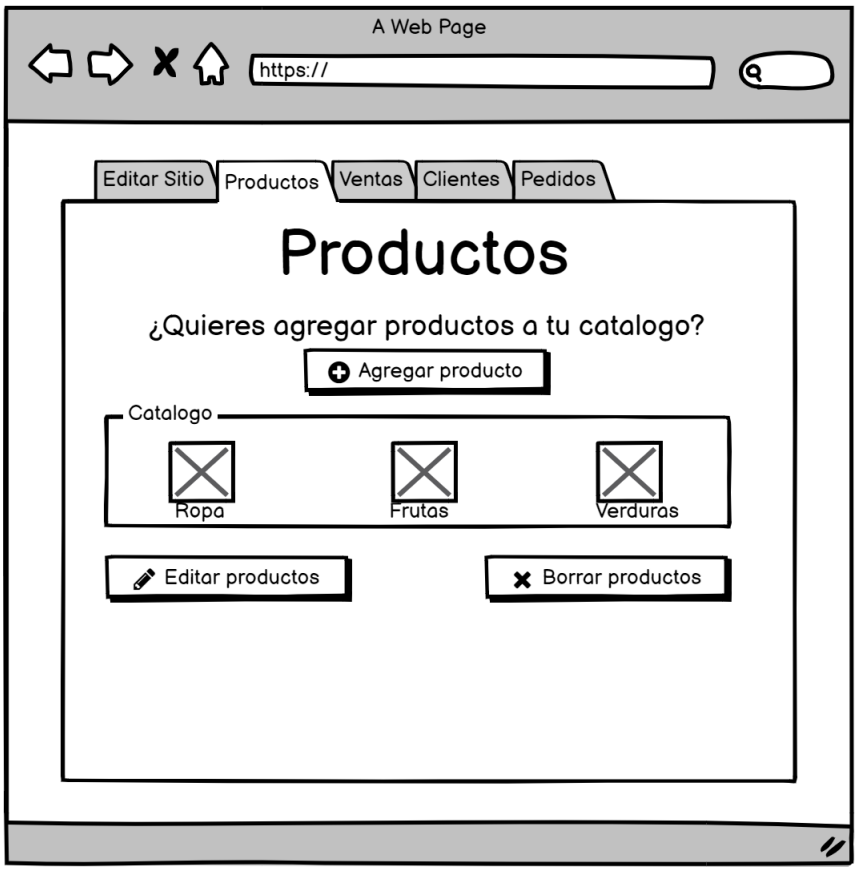

Figura 3.8: Interfaz para que los administradores agreguen productos a su tienda

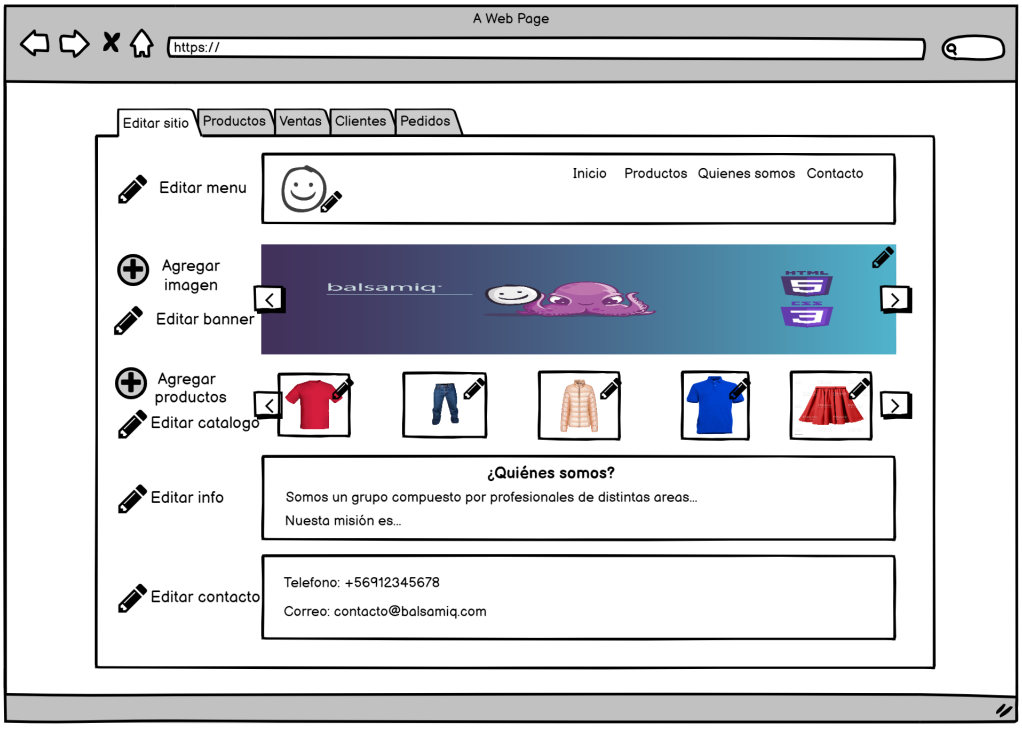

Figura 3.9: Módulo donde los administradores podrán editar su sitio web

Capítulo 4

Implementación

En este capítulo se muestra y explica la estructura del proyecto realizado, y las distintas partes que lo componen. Se explican los conceptos que se tuvieron que aprender de las tecnologías utilizadas y se muestra código representativo de los archivos que componen los dos repositorios creados.

En la sección 4.1 se muestra un diagrama de la arquitectura de una tienda virtual. Luego, se muestra el contenido de los directorios y archivos que contiene cada repositorio. En la sección 4.2 se explica cómo se construyeron las distintas instancias de Django desarrolladas y se muestra código representativo de ellas. En la sección 4.3 se explican los conceptos y herramientas más relevantes de React y Javascript, se muestra a través de diagramas todos los componentes desarrollados y se muestra parte del código.

Con este capítulo se puede tener una idea general de en qué consiste el código desarrollado, donde hubo que dedicar más tiempo para implementar las distintas funcionalidades de la plataforma y cual es la complejidad de este trabajo de título.

4.1. Estructura del proyecto

El código del proyecto se dividió en dos; se tiene un proyecto en Gitlab para el backend llamado *rushop-backend* y otro para el frontend llamado *rushop-frontend*. Como se mencionó en la sección 3.2.1 cada tienda utilizará el código de ambos proyectos en un servidor propio.

En la figura 4.1 se puede ver un diagrama de la arquitectura de una tienda virtual. En esta se puede ver que el backend se dividió en tres apps, se indica con flechas cómo se relacionan entre ellas, con la API, con el frontend y con los usuarios finales.

En esta sección se explicará la estructura de ambos proyectos, ejemplificando que función cumple cada directorio y archivo que se tiene en los repositorios.

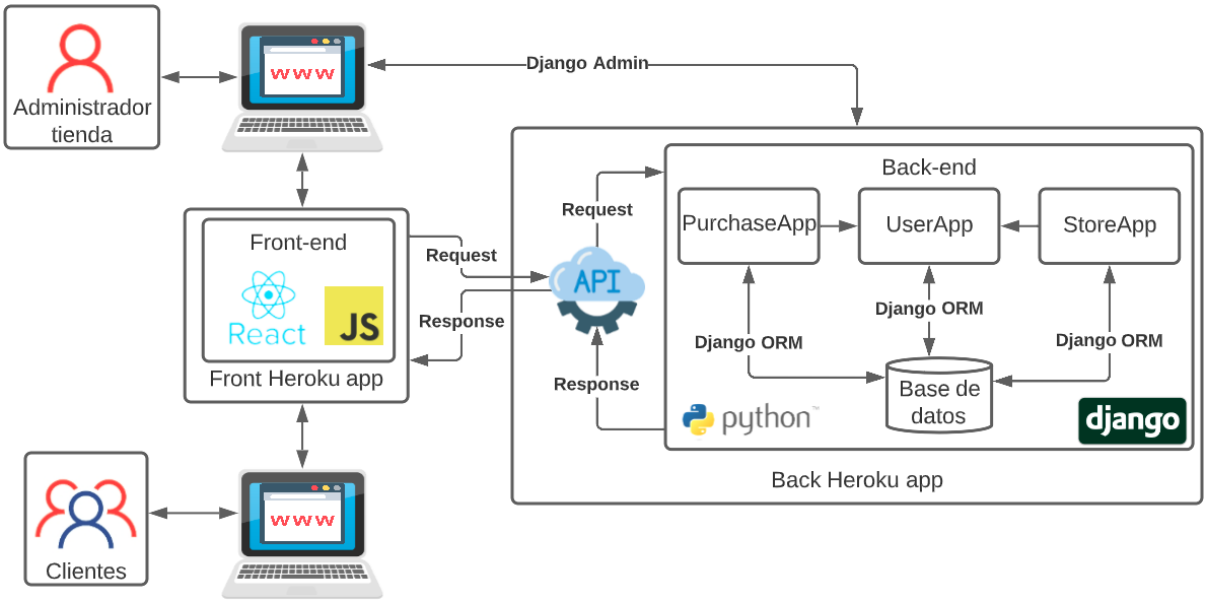

Figura 4.1: Diagrama de arquitectura del sistema

4.1.1. Backend

Se utilizó el lenguaje Python con el framework Django para el backend del proyecto. En particular se hizo uso de la librería Django Rest Framework para crear una API REST.

Para la base de datos se usó el gestor *Postgresql*. Se decidió utilizar este gestor ya que es más escalable que *SQLite*, el cual viene integrado con Django por defecto. Para esto se tuvo que configurar una base de datos nueva en el computador de la memorista y en el servidor de Heroku para hacer el deploy.

Para versionar el código tanto del backend como frontend se utilizó Git, haciendo uso de la plataforma Gitlab para tener el código en la nube.

Tal como se describió en la sección 3.2.1, el deployment del sistema desarrollado para este trabajo de título se hizo a través de la plataforma *Heroku*.

La estructura del directorio principal del backend se muestra en la figura 4.2. Si los nombres de las carpetas y/o archivos aparecen en blanco significa que están alojados en el proyecto de Gitlab, en gris aparece lo que esta guardado en el computador de la memorista y que es preferible no subir. A continuación se listarán las carpetas y archivos relevantes del proyecto.

1. `purchaseApp`: Directorio principal de la app llamada purchaseApp, se guarda el código relacionado al flujo de venta de productos. En la figura 4.3 se puede ver su contenido.
   (a) `fixtures`: Carpeta donde se guardan archivos JSON que contienen datos de prueba para el modelo. En el código 1 se puede ver a modo de ejemplo el contenido del archivo `products.json`
   (b) `migrations`: Directorio donde se guardan las migraciones creadas con el comando

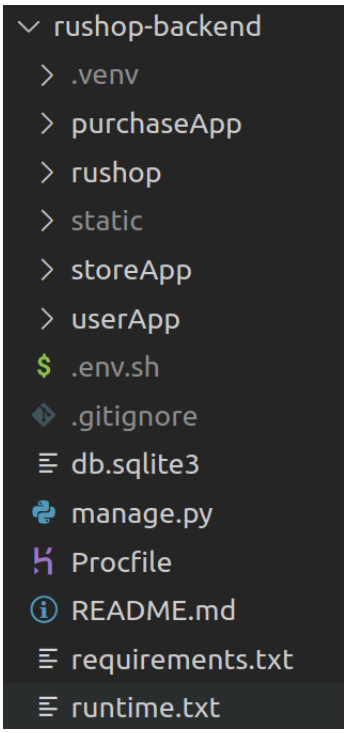

Figura 4.2: Estructura proyecto rushop-backend

```
[
  {
    "model": "purchaseApp.product",
    "pk": 1,
    "fields": {
      "name": "Polera roja",
      "description": "Una linda polera roja",
      "image": "https://i.ibb.co/BnwB9Rr/polera.png",
      "price": 3000,
      "category": 1,
      "sizes": [1, 2, 3, 4]
    }
  },
  {
    "model": "purchaseApp.product",
    "pk": 3,
    "fields": {
      "name": "Harry Potter y la piedra filosofal",
      "description": "Un lindo libro",
      "image": "https://i.ibb.co/BnwB9Rr/hp1.png",
      "price": 10000,
      "category": 2,
      "sizes": [],
    }
  }
]
```

Código 1: Extracto de archivo `products.json`

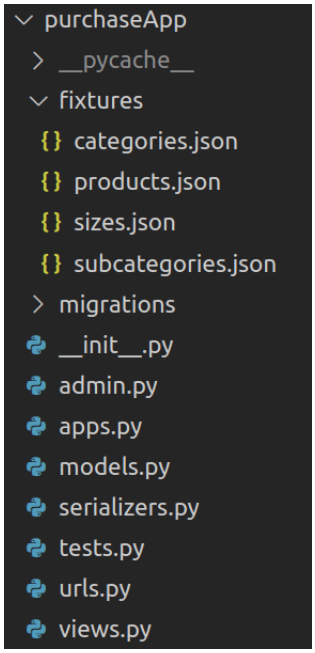

Figura 4.3: Contenido del directorio `purchaseApp`

`makemigrations`.

(c) `admin.py`: Archivo donde se registran y configuran los modelos que pueden gestionarse a través del módulo de administración de Django.

(d) `apps.py`: Archivo donde se configura el nombre de la aplicación, el cual se utiliza cuando se registra la app en el archivo `settings.py` y se muestra en el módulo de administración de Django.

(e) `models.py`: Archivo donde se registran y configuran los modelos de la aplicación.

(f) `serializers.py`:Archivo donde se registran y configuran los serializadores de la aplicación. En la próxima sección se describe que son los serializadores y cuales se desarrollaron para la plataforma.

(g) `tests.py`: Archivo donde se registran tests para la aplicación.

(h) `urls.py`: Archivo donde se registran y configuran las urls de las vistas de la aplicación. En la siguiente sección se describe cómo se configuraron las urls de la plataforma.

(i) `views.py`: Archivo donde se registran y configuran las vistas de la aplicación. En la próxima sección se describe qué son las vistas y cuales se desarrollaron para la plataforma.

2. `storeApp`: Directorio principal de la app llamada `storeApp`. Acá se guarda el código relacionado a la personalización de las tiendas de parte de los administradores. Las carpetas y archivos son similares a `purchaseApp` descritos en el item 2.

3. `userApp`: Directorio principal de la app llamada `userApp`. Acá se guarda el código relacionado a los usuarios del sistema. Las carpetas y archivos son similares a `purchaseApp` descritos en el ítem 2.

4. `.env.sh`: Archivo donde se setean las credenciales a la base de datos y otras variables privadas. Este archivo se debe ejecutar antes de iniciar el servidor django, de lo contrario este no funciona.

5. `Procfile`: Para desplegar una aplicación web con Heroku se debe incluir un archivo con este nombre, en el cual se especifican los comandos que se deben ejecutar en el momento del despliegue. Entre los tipos de comandos que se pueden incluir en este archivo está la ejecución del servidor web para la aplicación.

   El contenido de este archivo es: `web: gunicorn rushop.wsgi --log-file -`

   Esta línea de código utiliza la librería de Python llamada *Gunicorn* la cual sirve para desplegar un servidor HTTP WSGI. Se utiliza el módulo *wsgi* del directorio `rushop` del proyecto para configurar el despliegue. Además se agrega la opción `--log-file` para generar un archivo que guarde los logs del servidor.

6. `runtime.txt`: En este archivo se especifica el lenguaje y la versión de Python que se debe utilizar en Heroku para que la aplicación funcione correctamente. En este caso la versión utilizada es la 3.9.13, por lo que este archivo contiene la línea `Python3.9.13`

7. `.venv`: Es una carpeta donde se guarda el ambiente virtual que se utiliza para instalar las librerías de Python que se requieren. Es preferible instalar las librerías en un ambiente virtual en vez de directamente en el computador donde se desarrolla ya que así no se generan problemas de compatibilidad entre un proyecto y otro.

8. `rushop`: Este directorio se genera cuando se inicia un proyecto Django, el cual tiene el nombre del proyecto. Acá se guarda el archivo `settings.py`, un archivo llamado `urls.py` donde se configuran las urls de la aplicación web y un archivo llamado `wsgi.py` en el cual se realiza la configuración WSGI del proyecto.

9. `static`: En este directorio se guardan los archivos estáticos del proyecto. Se genera automáticamente al ejecutar el comando `python manage.py collecstatic`

10. `manage.py`: Archivo creado automáticamente por Django que sirve para ejecutar determinados comandos desde una terminal.

    Por ejemplo al ejecutar `python manage.py runserver` se inicia un servidor web de desarrollo en la máquina donde se ejecutó el comando.

4.1.2. Frontend

Para el frontend de la plataforma se utilizó React. El proyecto tiene un directorio llamado `Views`, donde se tienen las vistas principales de la plataforma y otro llamado `Components` donde se guardan los distintos componentes. La estructura del proyecto se puede ver en la figura 4.4.

Se hizo uso de las librerías *Reactstrap* y *Material UI* para personalizar las distintas interfaces de usuario. Se decidió usar ambas librerías en vez de solo una ya que así se podrían complementar y tener mayor cantidad de componentes disponibles para mejorar la experiencia de usuario.

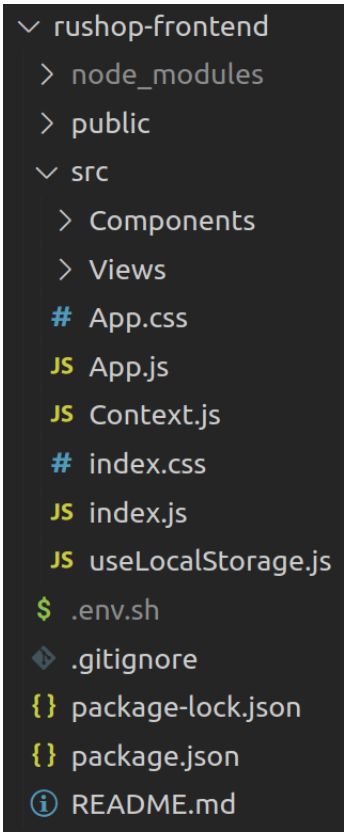

Figura 4.4: Estructura proyecto rushop-frontend

A continuación se listan los directorios y archivos más relevantes del proyecto, con una breve descripción de cada uno.

1. `Components`: Directorio donde se guardan los componentes de la aplicación.
2. `Views`: Directorio donde se guardan los componentes que corresponden a las vistas principales de la plataforma.
3. `App.js`: Componente principal de la plataforma. En este componente se llama a los componentes de la barra de navegación y el pie de página. Además se importa el archivo `Context.js` para obtener el número de la orden y precio total que es utilizado en distintos componentes.
4. `App.css`: Archivo CSS que se utiliza para modificar algunos estilos de textos y botones.
5. `Context.js`: En este archivo se inicializan los valores del contexto utilizado. Este sirve para que las variables que se tienen en este archivo puedan ser utilizadas en distintos componentes sin necesidad de pasarlas a través de props. En la sección 4.3 se ahondará mas sobre el concepto contexto en React.
6. `index.js`: En este archivo se hace el routing de la aplicación a través de la librería *React Router*. En la sección 4.3 se explica con mas detalle el contenido de este archivo.
7. `useLocalStorage.js`: Archivo que contiene una función desarrollada para acceder a valores alojados en el almacenamiento local del navegador de los usuarios que utilizan el sistema.

8. `.env.sh`: Archivo donde se setea el nombre de la url del backend. Este archivo se debe ejecutar antes de iniciar el servidor del frontend, de lo contrario este no funciona.

4.2. Implementación del backend

En Django, el término aplicación o simplemente *app* se usa para llamar a un paquete de Python que contenga módulos relacionados entre si por alguna funcionalidad. En este proyecto se decidió dividir el proyecto en tres aplicaciones distintas llamadas: `purchaseApp`, `storeApp` y `userApp`. Las tres aplicaciones tienen la misma estructura, mostrada en la figura 4.3.

En `purchaseApp` se guarda el código relacionado a la venta de productos, `storeApp` es para el código relacionado a la personalización y administración de las tiendas y `userApp` para el código de los usuarios del sistema y sus características.

4.2.1. Modelos

A continuación se nombran los modelos que se crearon en cada app, se explica la función de los modelos mas complejos y se muestra código representativo de ellos.

- `purchaseApp`: En esta aplicación se tienen los modelos relacionados con los productos y los pedidos que pueden realizar los clientes a través de una tienda virtual. Los modelos de esta aplicación son: `Category`, `Size`, `Product`, `ProductOrder` y `Order`. A modo de ejemplo se muestra el código del modelo `Product`, el cual representa a los productos en la base de datos.

  El modelo `ProductOrder` se creó para guardar la información que ingresa un usuario al agregar un producto a su carro de compras. Esta información es: el id del producto que escogió, la cantidad, el tamaño y la hora exacta de la agregación y/o modificación.

  El modelo `Order` representa los pedidos que realizan los clientes. Este modelo tiene una relación de uno a muchos con el modelo `ProductOrder` a través de un campo llamado `products_order`, es decir, cada pedido tiene un campo que guarda la información de los productos que pide el cliente y a su vez se tiene una tabla que relaciona los productos pedidos con cada orden.

- `storeApp`: En esta aplicación se tienen los modelos relacionados a la personalización y administración de las diferentes tiendas. Como el trabajo se enfocó en el flujo de venta de productos, en esta app se están usando solo dos modelos: `Image` y `Store`. En un desarrollo posterior a este trabajo de título se pretende usar modelos para guardar los datos de una sección de la página y para elegir los colores que se utilicen.

  El modelo `Store` se usa para que los administradores agreguen los datos de su tienda a través del módulo admin de Django, y esta información se muestre en su respectiva tienda virtual.

```python
from django.db import models


class Product(models.Model):
    name = models.CharField(max_length=100, verbose_name="nombre")
    description = models.TextField(verbose_name="descripción", null=True, blank=True)
    image = models.TextField(verbose_name="imagen", null=True, blank=True)
    category = models.ForeignKey(Category,
                                 related_name='products',
                                 verbose_name="categoria",
                                 on_delete=models.CASCADE,
                                 null=True,
                                 blank=True)

    price = models.IntegerField(verbose_name="precio", null=True, blank=True)
    sizes = models.ManyToManyField(Size,
                                   related_name='size_options',
                                   verbose_name="tamaños",
                                   blank=True)
```

Código 2: Modelo que define el esquema de la tabla de productos.

El modelo `Image` sirve para guardar la información necesaria de las imágenes que se usan en la plataforma. En particular, se utiliza para mostrar las imágenes del carrusel que se muestra en el landing page de las tiendas virtuales.

- `UserApp`: En esta aplicación se tienen los modelos relacionados a los usuarios del sistema y sus características. Se crearon los modelos `Client`, `Region`, `Town` y `Address`.

4.2.2. Serializadores

En cada una de las aplicaciones de Django se tiene un archivo llamado `serializers.py` donde se declaran los serializadores a partir de los distintos modelos de datos. Se pueden declarar los atributos de cada serializador explícitamente en el código especificando el tipo de cada atributo o crear los serializadores a partir de los modelos, los cuales ya contienen la información sobre los dominios de cada atributo. En este proyecto se utilizó el segundo método para crear los distintos serializadores, ya que es la forma mas concisa y pulcra de desarrollar el código.

Existen serializadores simples, donde solo se especifican los campos que se usarán de un determinado modelo, los cuales llamaremos serializadores sin anidación. Los serializadores con anidación son los que utilizan en uno de sus campos otro serializador.

A continuación se nombran los serializadores de cada aplicación que se desarrollaron para completar las distintas funcionalidades del sistema. También se muestra el código de los métodos de la instancia `ModelSerializer` que hubo que sobrescribir en determinados serializadores. En el siguiente capítulo se detalla el propósito de cada serializador y en qué etapa del proyecto fue desarrollado.

```python
class SizeSerializer(serializers.ModelSerializer):

    class Meta:
        model = Size
        fields = ['id', 'name']
```

Código 3: Serializador sin anidación

```python
class ProductSerializer(serializers.ModelSerializer):
    sizes = SizeSerializer(many=True)

    class Meta:
        model = Product
        fields = ['id', 'name', 'description', 'image', 'category', 'price', 'sizes']
```

Código 4: Serializador con campo anidado

- `purchaseApp`: En esta app es donde se desarrolló la mayor cantidad de serializadores ya que es la parte donde se enfocó el trabajo de esta memoria.

  Se crearon los serializadores simples o sin anidación llamados: `SizeSerializer`, `ShowProductSerializer`, `CatalogueProductSerializer`, `CatalogueSerializer`, `ProductOrderSerializer`, `OrderDetailSerializer` y `ShoppingCartSerializer`.

  En el código 3 se puede ver el serializador llamado `SizeSerializer`. Los demás serializadores simples tanto de esta app como de las otras dos, tienen un código similar, donde sólo varía el nombre del objeto y del modelo utilizado, además de los distintos atributos especificados en `fields`.

  Para simplificar las llamadas a la API desde el frontend, se crearon serializadores anidados según las distintas necesidades que se fueron presentando. Por ejemplo, en el código 4 se muestra el código del serializador `ProductSerializer`, el cual tiene un campo llamado `sizes` que usa el serializador `SizeSerializer` (línea 2).

  Los serializadores con anidación que se crearon en esta app son: `ProductSerializer`, `ShowProductOrderSerializer`, `CategorySerializer`, `OrderClientSerializer`, `OrderAddressSerializer`, `OrderSummarySerializer`.

  Para evitar crear un pedido cada vez que un cliente acceda a una tienda virtual sin necesariamente agregar productos a su carro de compras, se decidió crear los pedidos cuando un cliente agregue por primera vez un producto a su carro de compras. Para lograr este comportamiento a través de una sola llamada a la API, se sobreescribió el método `create` de `OrderSerializer`.

  El método `create` se utiliza cuando se hacen llamadas POST a una url de la API que usa un determinado serializador. En el código 5 se muestra el método `create` de

```python
def create(self, validated_data):
    products_data = validated_data.pop('products')
    order = Order.objects.create(**validated_data)
    for product in products_data:
        ProductOrder.objects.create(order=order, **product)
    return order
```

Código 5: Método `create` de serializador de modelo `Order`

```python
def create(self, validated_data):
    order = validated_data.pop('order')
    product = validated_data.pop('product')
    try:
        size = validated_data.pop('size')
    except KeyError:
        size = None
    amount = validated_data.pop('amount')
    products = ProductOrder.objects.filter(order=order)
    for product_order in products:
        if product_order.product == product and product_order.size == size:
            product_order.amount = product_order.amount + amount
            product_order.save()
            return product_order
    product_order = ProductOrder(order=order, product=product, size=size, amount=amount)
    product_order.save()
    return product_order
```

Código 6: Método `create` de serializador `ListProductOrderSerializer`

```python
def create(self, validated_data):
    try:
        email = validated_data.pop('email')
        name = validated_data.pop('name')
        lastname = validated_data.pop('lastname')
        phone = validated_data.pop('phone')
        client = Client.objects.get(email=email)
        client.name = name
        client.lastname = lastname
        client.phone = phone
        client.save()
        return client

    except Client.DoesNotExist:
        client = Client(name=name, lastname=lastname, phone=phone, email=email)
        client.save()
        return client
```

Código 7: Método *create* de serializador *ClientSerializer*

`OrderSerializer`. Este permite crear instancias del modelo `ProductOrder` cuando se hace una request del tipo POST a la url de pedidos.

Cuando un cliente agrega por primera vez un producto al carrito de compras, se crea el pedido en la base de datos con el `ProductOrder` ingresado. Si el cliente vuelve a agregar el mismo producto con el mismo tamaño al carrito, en vez de crear un nuevo `ProductOrder`, se suma la cantidad que se agregó del producto con la que ya se tenía. Para lograr esto se tuvo que sustituir el método `create` de `ListProductOrderSerializer`, esto se muestra en el código 6.

- `storeApp`: En esta aplicación se crearon los serializadores simples: `StoreSerializer` e `ImageSerializer`. El primero sirve para acceder a los datos de una tienda desde la API y el segundo para listar las imágenes que se han agregado al modelo.

  También se crearon dos serializadores con anidación: `StoreImagesSerializer` y `StoreAddressSerializer`, el primero utiliza el serializador `ImageSerializer` para acceder a las imágenes que se mostrarán en el carrusel a partir del id de la tienda. El segundo sirve para mostrar y modificar la dirección de la tienda a partir de su id.

- `UserApp`: En esta aplicación se desarrollaron cinco serializadores simples: `AddressSerializer`, `RegionSerializer`, `TownSerializer`, `ClientSerializer` y `ClientAddressSerializer`. También se creó un serializador anidado llamado `TownRegionSerializer`.

  Cuando los clientes realizan una compra deben ingresar sus datos al sistema para enviarles un correo electrónico con la información de su pedido y para poder contactarlos en caso de que sea necesario. Al ingresar los datos, se manda una request del tipo POST a la API, la API verifica si el cliente existe en el sistema revisando si el email que se

```python
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductViewSet(viewsets.Model ViewSet):
    """
    This viewset automatically provides `list`, `create`, `retrieve`,
    `update` and `destroy` actions.
    """
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

Código 8: `ViewSet` de productos

ingresó ya estaba en la base de datos. De esta forma se evita generar clientes repetidos en el sistema y validarlo en la interfaz del usuario.

Para lograr el comportamiento descrito en el párrafo anterior, se sobreescribió el método `create` del serializador `ClientSerializer`. En el código 7 se puede ver este método.

4.2.3. Vistas

A continuación se nombran las vistas creadas en cada aplicación, mostrando código relevante y representativo del desarrollo realizado en esta parte del proyecto.

- `purchaseApp`: En esta aplicación se desarrollaron las vistas llamadas `ProductViewSet`, `CategoryViewSet`, `CatalogueViewSet`, `OrderViewSet`, `ProductOrderViewSet`, `SizeViewSet`, `OrderDetailViewSet`, `OrderAddressViewSet`, `OrderClientViewSet` y `OrderSummaryViewSet`.

  En el código 8 se puede ver la clase `ProductViewSet`. En esta se define que la `queryset` es `Products.objects.all()` (línea 10) lo que significa que cuando esta clase reciba una request, esta hará una consulta en la tabla de productos de la base de datos pudiendo retornar todos los productos si así se desea. Además en la línea 11 se declara el serializador que se utiliza, el cual permite convertir los datos a un JSON que contiene las llaves definidas en la clase `ProductSerializer`.

  Se puede ver un ejemplo de respuesta de la clase `ProductViewSet` en el código 9. Esta respuesta se retorna cuando se utiliza el método `list`, el cual realiza la consulta a la tabla de productos y retorna un JSON que contiene los datos de todos los productos que encuentra.

  Las demás vistas mencionadas anteriormente son similares a `ProductViewSet`, en la única que hubo que sobrescribir uno de sus métodos fue en `OrderSummaryViewSet`. En el código 10 se puede ver el método `update` de esta clase, el cuál se modificó para

```
[
    {
        "id": 1,
        "name": "Polera roja",
        "description": "Una linda polera roja",
        "image": "https://i.ibb.co/BnwB9Rr/polera.png",
        "category": 1,
        "price": 3000,
        "sizes": [
            {
                "id": 1,
                "name": "S"
            },
            {
                "id": 2,
                "name": "M"
            },
            {
                "id": 3,
                "name": "L"
            },
            {
                "id": 4,
                "name": "XL"
            }
        ]
    },
    ...
]
```

Código 9: Ejemplo de respuesta de `ProductViewSet`

```
def update(self, request, *args, **kwargs):
    response = super(OrderSummaryViewSet, self).update(request, *args, **kwargs)
    try:
        if request.data["send_email"]:
            send_email(response.data)  # sending mail
        return response

    except:
        return response
```

Código 10: Método `update` de `OrderSummaryViewSet`

```python
from django.core.mail import EmailMultiAlternatives

def format(number):
  return '{:,}'.format(number).replace(',','.')

def send_email(data):
  client = data["client"]
  address = data["address"]
  products = data["products"]
  from_email, to = 'rushopcl@gmail.com', client["email"]
  subject = 'Confirmación Pedido N°' + {data["id"]}
  text_content = ('Hola '+{client["name"]}+', agradecemos tu preferencia.'+
                  'Tu pedido es el N°'+{data["id"]}+'\n')
  style = 'style="border: 1px solid #dddddd; padding: 8px;"'
  html_content = ('<p>Hola '+{client["name"]}+', agradecemos tu preferencia.'+
                  'Tu pedido es el N°'+{data["id"]}+'</p>'+
                  '<table style="border-collapse: collapse; width: 50%; '+
                  'text-align: left"><tr>'+
                  '<th '+{style}+'>Productos</th><th '+{style}+'>Cantidad</th>'+
                  '<th '+{style}+'>Valor</th></tr>')
  for product in products:
    html_content += ('<tr><td '+{style}+'>'+{product["name"]}+'</td>'+
                     '<td '+{style}+'>'+{product["amount"]}+'</td>'+
                     '<td '+{style}+'>$'+{format(product["total_price"])}+'</td></tr>')

  html_content += ('<tr><td'+{style}+'><b>Total</b></td>'+
                   '<td '+{style}+'>'+{data["total_amount"]}+'</td>'+
                   '<td '+{style}+'>\$'+{format(data["total_price"])}+'</td><tr></table>')
  msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
  msg.attach_alternative(html_content, "text/html")
  msg.send()
```

Código 11: Función para enviar correos electrónicos desde la API

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from purchaseApp import views

# Create a router and register our viewsets with it.
router = DefaultRouter()
router.register(r'products', views.ProductViewSet, basename="products")
router.register(r'categories', views.CategoryViewSet, basename="categories")
...
# The API URLs are now determined automatically by the router.
urlpatterns = [
    path('', include(router.urls)),
]
```

Código 12: Parte del archivo `urls.py` de `purchaseApp`

mandar un correo electrónico cuando se reciba una request con un valor `true` en la llave `send_mail`.

Cuando los clientes confirman un pedido, se envía un correo electrónico con la información de su pedido al correo que ingresaron. Para esto se creó una función llamada `send_mail`, la cual se utiliza en el método mencionado en el párrafo anterior. En el código 11 se puede ver esta función. Python cuenta con un módulo para enviar correos llamado *Smtplib*, en esta función se usó un wrapper de Django que facilita el uso de esta librería.

- `storeApp`: En esta aplicación se crearon las vistas `StoreViewSet`, `StoreImages` y `StoreAddress`. El código de estas clases es similar a `ProductViewSet`.
- `userApp`: En esta aplicación se crearon las vistas `UserViewSet`, `TownRegionViewSet` `GroupViewSet`, `ClientViewSet`, `AddressViewSet`, `RegionViewSet`, `TownViewSet` y `ClientAddressViewSet`. El código de estas clases es similar a `ProductViewSet`.

4.2.4. URLs

Se puede acceder a los recursos del proyecto a través de URLs, las cuales fueron configuradas en el archivo `urls.py` de cada aplicación e importadas en el archivo del mismo nombre ubicado en el directorio principal del proyecto.

Para registrar las vistas de una `ViewSet` en la configuración de URLs de Django, se utiliza una clase del tipo *router*, la cual permite determinar automáticamente la configuración de URLs.

En el código 12 se puede ver como se configuraron las URLs de `purchaseApp`. En la línea 7 se le otorga un nombre a la URL de `ProductViewSet`, en este caso `products` y automáticamente se nombran y configuran las vistas que contiene esta *ViewSet* a partir del nombre otorgado. Por ejemplo, la URL de la vista que retorna el json del código 9 es `/products/`. Las configuraciones de las otras dos aplicaciones son similares.

```
import React, { useState, useEffect } from 'react';
import { Container, UncontrolledCarousel } from 'reactstrap';

export function StoreImages(props) {
  const [images, setImages] = useState([]);

  useEffect(() => { ... } )

  return (
    <Container>
      <UncontrolledCarousel items={images}/>
    </Container>
  )
}
```

Código 13: Extracto de componente `StoreImages`

4.2.5. Módulo de Administración de Django

Una de las características más potentes de Django es su módulo de administración. Este permite que los usuarios puedan agregar, editar y borrar datos de determinadas tablas de la base de datos a través de una interfaz intuitiva y usable que se configura con pocas líneas de código.

Se configuró este módulo para que los administradores de las tiendas puedan utilizarlo para agregar y editar la información necesaria de sus tiendas virtuales. La forma que se usó para poder utilizar un determinado modelo en el administrador es registrándolo en el archivo `admin.py` a través de la línea `admin.site.register(Product)` donde `Product` es el nombre del respectivo modelo. El objeto `admin` se importa del módulo `django.contrib`.

4.3. Implementación del frontend

En este proyecto se utilizaron sólo funciones para desarrollar los distintos componentes ya que así se pueden usar las funciones de React conocidas como *Hooks*.

Un ejemplo de un componente simple utilizado en este proyecto es el que se muestra en el código 13. Este utiliza los componentes `Container` y `UncontrolledCarousel` de la librería `Reactstrap`. En la función `useEffect` se cargan los datos de las imágenes que utiliza el carrusel desde el servidor.

4.3.1. Componentes principales

El componente principal de un proyecto desarrollado con React comúnmente se llama `App`. Este se utiliza en `index.js` que es donde se realiza la renderización del componente. Desde `App` se llama a los distintos componentes que se desarrollaron para generar el producto

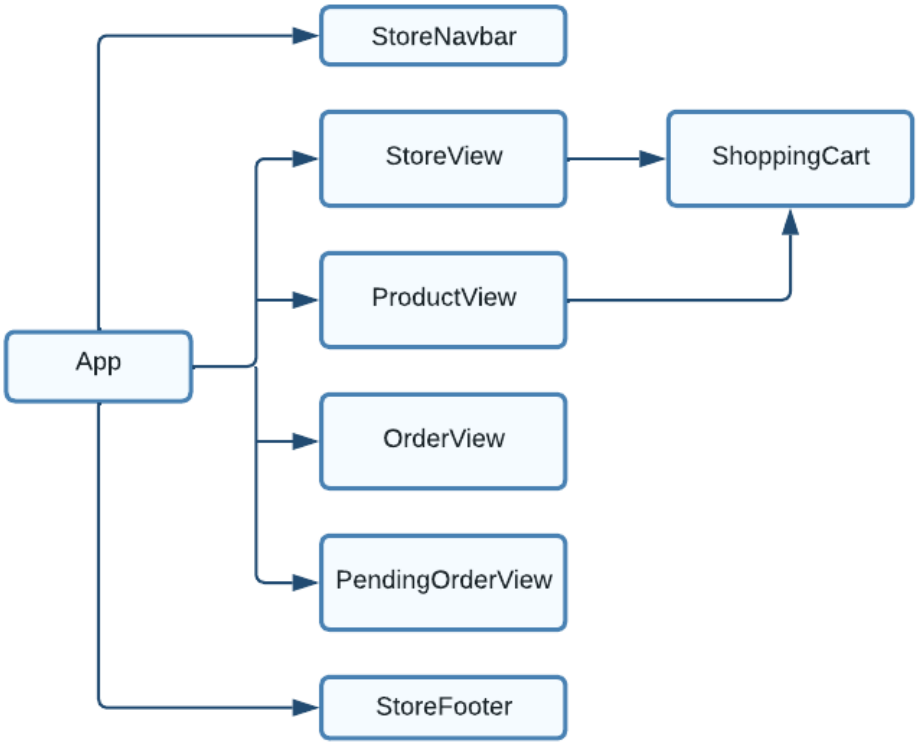

Figura 4.5: Jerarquía de componentes de React

final.

En la figura 4.5 se muestra un diagrama que grafica la jerarquía de los principales componentes de la plataforma. Las flechas indican que componentes se utilizan en cada componente.

Los principales componentes de la plataforma se llaman `Views` en el código. Estas son: `StoreView`, la cual corresponde a la página principal de la tienda virtual; `ProductView`, para la ficha de un producto; `OrderView`, la cual permite ver los productos del pedido, ingresar los datos del cliente, su dirección y confirmar la orden; por último se creó un componente llamado `PendingOrdersView` que muestra una tabla con los pedidos en estado pendiente para que puedan ser vistos por los administradores de las tiendas.

En la figura 4.5 se puede ver que en `App` se utilizan las cuatro vistas nombradas en el párrafo anterior, además se utilizan los componentes `StoreNavbar` y `StoreFooter`, los cuales corresponden a las barras de navegación y el pie de página respectivamente. El componente `ShoppingCart` es un panel o *drawer* que se ubica a la derecha de la pantalla y se abre cada vez que se presiona el botón con un icono de un carro de compras ubicado en la esquina inferior derecha de las vistas `StoreView` y `ProductView`.

- `StoreView`: Este componente representa la interfaz de la página principal de una tienda virtual. En la figura 4.6 se puede ver la jerarquía de los componentes que se utilizan en esta vista. En el diagrama se muestran en color rosado los componentes importados de las librerías *Reactstrap* y *React Slick*, el resto de componentes fueron desarrollados por la memorista.

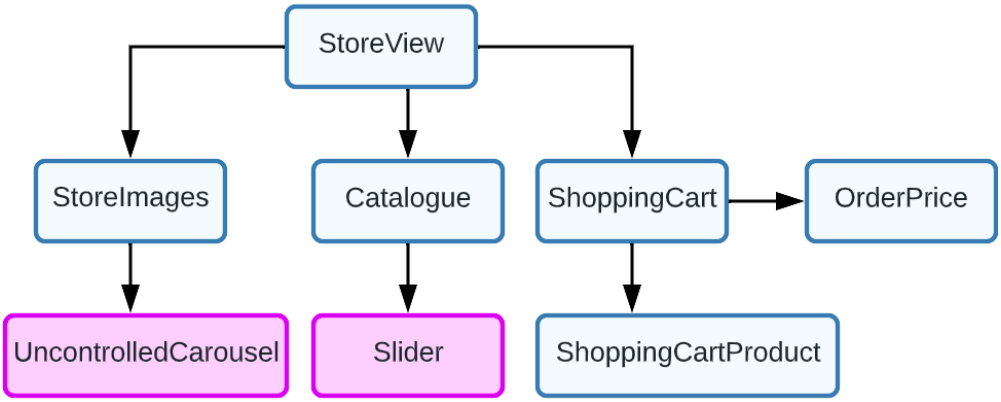

Figura 4.6: Diagrama de jerarquía de los componentes de `StoreView`

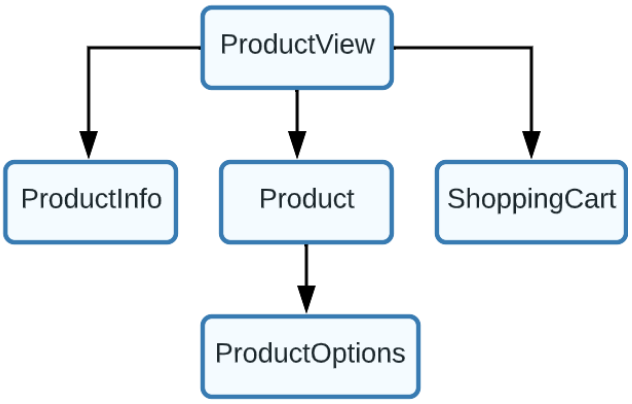

Figura 4.7: Diagrama de jerarquía de los componentes de `ProductView`

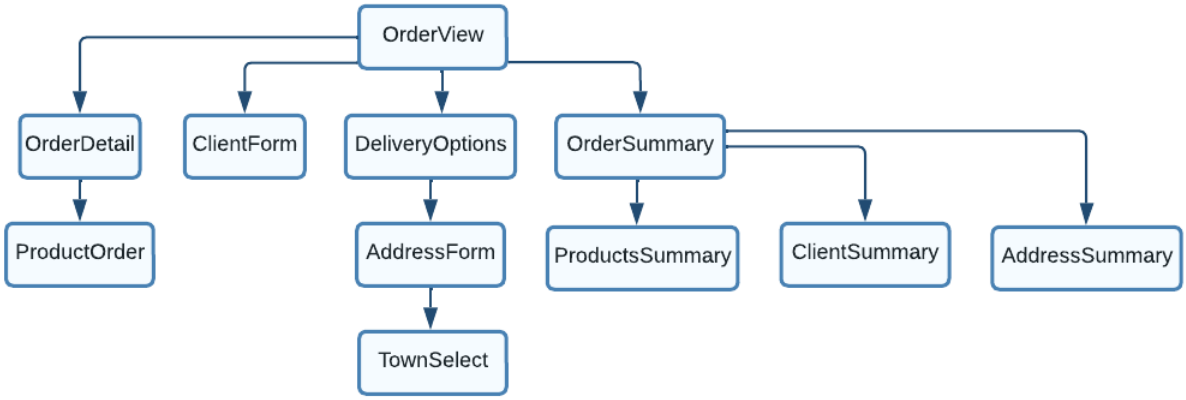

Figura 4.8: Diagrama de jerarquía de los componentes de `OrderView`

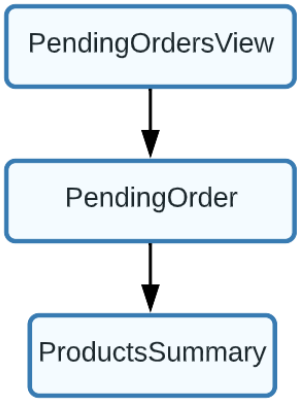

Figura 4.9: Diagrama de jerarquía de los componentes de `PendingOrdersView`

- `ProductView`: Este componente representa la interfaz de la ficha de un producto, donde se puede ver la información detallada de cada producto y agregar al carrito de compras. En la figura 4.7 se pueden ver los componentes que se desarrollaron para implementar esta interfaz. Para simplificar el diagrama y no ser redundante, no se muestran los hijos del componente `ShoppingCart` pues son los mismos del diagrama anterior.
- `OrderView`: Este componente representa la vista de la información de un pedido (figura 4.8). Este es el componente que contiene la mayor cantidad de componentes ya que contiene la lista de productos del pedido, un formulario para agregar los datos del cliente, otro formulario para ingresar la dirección y una vista que muestra el resumen del pedido.
- `PendingOrdersView`: Este componente representa la tabla de pedidos pendientes para los administradores de las tiendas. En la figura 4.9 se pueden ver los componentes que se utilizan en esta interfaz.

4.3.2. Características y herramientas de React utilizadas

- ***Hooks***: En React existen funciones llamadas *hooks* que permiten utilizar funcionalidades típicas de los componentes basados en clases en componentes basados en funciones.

```
1  import React from 'react'
2
3  export const OrderContext = React.createContext({
4    order: 0,
5    setOrder: () => {}
6  });
7
8  export const TotalPriceContext = React.createContext({
9    totalPrice: 0,
10   setTotalPrice: () => {}
11 })
```

Código 14: Contenido de `Context.js`

Los *hooks* mas importantes y que se utilizan en la mayoría de los componentes desarrollados son `useState` y `useEffect`.

El hook `useState` sirve para que los componentes tengan un estado, lo que significa que tendrán variables que se inicialicen con un valor por defecto cuando se renderice el componente y puedan cambiar a través de funciones privadas del componente y/o interacciones con el usuario. En la línea 5 del código 13 se puede ver un ejemplo de inicialización de la variable de estado `images`. Para modificar el valor de `images` se utiliza la función `setImages`, entregándole el nuevo valor en el argumento.

El hook `useEffect` sirve para generar un ciclo de vida a los componentes basados en funciones, esto quiere decir que cada vez que se renderiza (cuando nace) un componente este puede realizar determinadas acciones declaradas en el argumento de `useEffect` y también se pueden realizar determinadas acciones cuando el componente deje de renderizarse (cuando muere).

- ***Context***: Se decidió integrar el uso de contexto para obtener el número del pedido de un determinado usuario y el precio total de la compra en los distintos componentes que lo requieren.

  Para usar contexto se utilizó el hook llamado `useContext` el cual permite utilizar el o los valores que estén guardados en un contexto en el componente que llame a la función `useContext`. Por ejemplo, en el componente `ShoppingCart` se tiene la línea de código: `const { order, setOrder } = useContext(OrderContext);` con esto se crean las variables `order` y `setOrder` en este componente, las cuales pudieron haber sido inicializadas con anterioridad en cualquier componente que también haya hecho esta llamada.

  En el código 14 se puede ver el contenido del archivo `Context.js`. En este archivo se crea el contexto con un valor por defecto de las variables `order` y `totalPrice`. Cuando un componente utiliza el contexto a través de la llamada mencionada en el párrafo anterior, este puede cambiar el valor de estas variables, cuyo cambio se verá reflejado en todos los componentes que utilicen este contexto.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import BrowserRouter, Routes, Route from "react-router-dom";
import ProductView from './Views/ProductView';
import StoreView from './Views/StoreView';
import Catalogue from './Components/Catalogue';
import OrderView from './Views/OrderView';
import PendingOrdersView from './Views/PendingOrdersView'

ReactDOM.render(
    <BrowserRouter>
        <Routes>
            <Route path="/" element={<App />}>
                <Route path="/" element={<StoreView />} />
                <Route path="products" element={<Catalogue />} />
                <Route path="products/:productId" element={<ProductView />} />
                <Route path="order" element={<OrderView />} />
                <Route path="pending-orders" element={<PendingOrdersView />} />
            </Route>
        </Routes>
    </BrowserRouter>,
    document.getElementById('root')
);
```

Código 15: Contenido de `index.js`

- ***Routers***: En React, los routers ayudan a crear y navegar a través de las distintas URLs de la aplicación web que se este desarrollando. Permiten que los usuarios naveguen a través de los distintos componentes mientras conservan el mismo estado, y sirven para generar URLs únicas para cada componente permitiendo que sean mas fácil de compartir.

  En este proyecto se utilizó la librería *React Router* para generar las distintas URLs de la plataforma. Esta librería se usa principalmente en el archivo `index.js`. En `index` se nombra y asocia cada URL con un componente, se puede ver la parte relevante de este archivo en el código 15.

4.3.3. Características y herramientas de Javascript utilizadas

- **Almacenamiento local**: En Javascript existe un objeto llamado `localStorage` el cual permite guardar y utilizar llaves con un determinado valor en el navegador.

  Se desarrolló un hook llamado `useLocalStorage` que sirve para utilizar el objeto `localStorage` en el navegador que este renderizando la aplicación. Contiene un función llamada `getLocalStorageOrDefault` para revisar si una llave ya existe en el almacenamiento local, en caso contrario, se crea y se le entrega un valor por defecto. En el código 16 se puede ver el desarrollo del hook `useLocalStorage`.

```
1  import { useState, useEffect } from 'react';
2
3  function getLocalStorageOrDefault(key, defaultValue) {
4    const stored = localStorage.getItem(key);
5    if (!stored || stored == null) {
6      return defaultValue;
7    }
8    return JSON.parse(stored);
9  }
10
11 export function useLocalStorage(key, defaultValue) {
12   const [value, setValue] = useState(
13     getLocalStorageOrDefault(key, defaultValue)
14   );
15
16   useEffect(() => {
17     localStorage.setItem(key, JSON.stringify(value));
18   }, [key, value]);
19
20   return [value, setValue];
21 }
```

Código 16: Contenido de archivo `useLocalStorage.js`

- `Fetch`: Este método permite realizar requests por el protocolo HTTP a servidores web. Se utiliza esta función en múltiples componentes del código para comunicarse con el backend.

  En el código 17 se puede ver un ejemplo de la utilización del método `fetch` en el componente `ProductView`. Este se utiliza en el cuerpo de una función llamada `fetchData()` que es utilizada en el argumento del hook `useEffect`. La mayoría de los componentes utilizan esta forma de cargar los datos desde el backend.

  La URL del backend es seteada a través de una variable de ambiente, esto se hizo así ya que puede variar donde este siendo ejecutado el backend según desde donde se este ejecutando el frontend. En la línea 13 del código mencionado en el párrafo anterior se guardan los datos obtenidos desde la API del backend en la variable de estado `product`.

4.3.4. Formularios

Los formularios forman parte importante de las funcionalidades que tiene la plataforma. Cada vez que un usuario interactúa con la aplicación a través de un botón o un campo de texto, se realiza una llamada al backend con la información que ingresó el usuario en el cuerpo de la request.

En el código 18 se muestra el componente `ClientForm`, el cual corresponde al formulario donde los usuarios que realizan un pedido ingresan su información de contacto. En la línea 18 se puede ver que se utiliza la etiqueta de HTML `form` y que el botón de la línea 36 es del tipo `submit`, por lo que cuando se hace click en él, se llama a la función `handleSubmit`, qué

```
const fetchData = () => {
    const backendUrl = process.env.REACT_APP_SERVER;
    fetch(backendUrl + "/products/" + productId)
    .then(response => {
        if (!response.ok) {
            throw new Error(
              "This is an HTTP error: The status is " + response.status
            );
        }
        return response.json();
    })
    .then(data => {
        setProduct(data);
        setIsLoaded(true);
    })
    .catch((error) => {
        setError(error.message);
    });
}
```

Código 17: Uso de `fetch` en componente `ProductView`.

es la encargada de mandar la request al backend para ingresar la información entregada en la base de datos del sistema.

Los campos de texto utilizan una función llamada `handleChange` para que cada vez que se realice un cambio en ellos, se actualice el valor de la variable de estado del respectivo campo. En el código 19 se puede ver la función `handleChange` del componente `ClientForm`. En este se puede ver que existen cuatro campos de texto: `name`, `lastName`, `email` y `phone`.

La función `handleSubmit` mostrada en el código 20 crea el cuerpo de la request a partir de las variables de estado del componente, también define que se utilice el método POST y que el contenido es un json. Luego se utiliza la función `fetch` explicada en la subsección anterior, se actualizan los datos del cliente y se le avisa a través de un cuadro de alerta, que se ingresaron correctamente sus datos.

4.4. Resumen

En este capítulo primero se mostró la estructura del proyecto para tener una idea general de que es lo que se desarrolló. Luego, en cada parte del proyecto se explicaron los conceptos importantes de las tecnologías utilizadas para que se pudiera entender el código mostrado. Finalmente, se mostró el código representativo de ambos repositorios.

En el proyecto de backend se puede ver que donde más se tuvo que desarrollar código fue en los serializadores y vistas de cada endpoint. Por otro lado, en el frontend, la mayor parte del código se concentra en los componentes de React.

```jsx
import React, { useState, useEffect } from "react";
import Box from '@mui/material/Box'
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';

export function ClientForm(props) {
  const [name, setName] = useState("");
  ...
  const [phone, setPhone] = useState("");

  useEffect(() => { ... } )

  function handleChange(event) { ... }

  function handleSubmit(event) { ... }

  return (
    <form onSubmit={handleSubmit}>
    <Box sx={{'& .MuiTextField-root': { m: 1, width: '25ch' },}}>
      <div>
        <TextField
          id="name"
          label="Nombre"
          onChange={handleChange}
          value={name}
        />
        ...
        <TextField
          id="phone"
          label="Telefono"
          onChange={handleChange}
          value={phone}
        />
      </div>
      <div>
        <Button sx={{ m: 1}} variant="contained" color="success" type="submit">
          Ingresar datos
        </Button>
      </div>
    </Box>
    </form>
  )}
```

Código 18: Componente `ClientForm`

```
function handleChange(event) {
    if (event.target.id === "name") {
      setName(event.target.value);
    }
    else if (event.target.id === "lastName") {
      setLastName(event.target.value);
    }
    else if (event.target.id === "email") {
      setEmail(event.target.value);
    }
    else if (event.target.id === "phone") {
      setPhone(event.target.value);
    }
}
```

Código 19: Función `handleChange` de componente `ClientForm`

```
function handleSubmit(event) {
    event.preventDefault();
    const fetchData = () => {
      const url = backendUrl + "/clients/"
      const body = {
        "name": name,
        "lastname": lastName,
        "email": email,
        "phone": phone
      }
      const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(body)
      };
      fetch(url, requestOptions)
      .then(response => {
        return response.json()
      })
      .then(client => {
        setClientId(client.id);
        ...
        .then(() => {
          alert("Se ingresaron los datos del cliente correctamente.")
        })
    })}
    fetchData();
}
```

Código 20: Extracto de la función `handleSubmit` de componente `ClientForm`

Capítulo 5

Desarrollo del proyecto

En este capítulo se detalla la metodología que se utilizó durante el semestre. Tal como se explica en la sección 1.5, el semestre se dividió en sprints de dos semanas, donde cada sprint tenía determinadas tareas que se debían realizar. A continuación se listan las tareas que se realizaron en cada sprint, se explican detalles del código y los desafíos que se tuvieron que enfrentar para lograr desarrollar el producto implementado.

5.1. Primer sprint

Para el primer sprint se planificó que se harían las configuraciones necesarias para empezar el desarrollo de la plataforma además se desarrollaría una primera interfaz de usuario. Las tareas fueron las siguientes:

- Crear proyecto Django agregando el modelo de datos.
- Crear y conectar base de datos Postgresql.
- Crear proyecto React.
- Conectar backend con frontend.
- Configurar versionamiento.
- Ficha de un producto.
- Crear barra de navegación.

Se creó una base de datos Postgresql alojada en el computador de la memorista para el desarrollo. En el proyecto Django se utilizarían variables de entorno para acceder a la base de datos, de esta forma se asegura la privacidad de las credenciales al no mostrarse directamente en el código.

Para el frontend se creó el proyecto de React con la herramienta Create React App, la cual permite configurar fácilmente un proyecto nuevo a través de un comando.

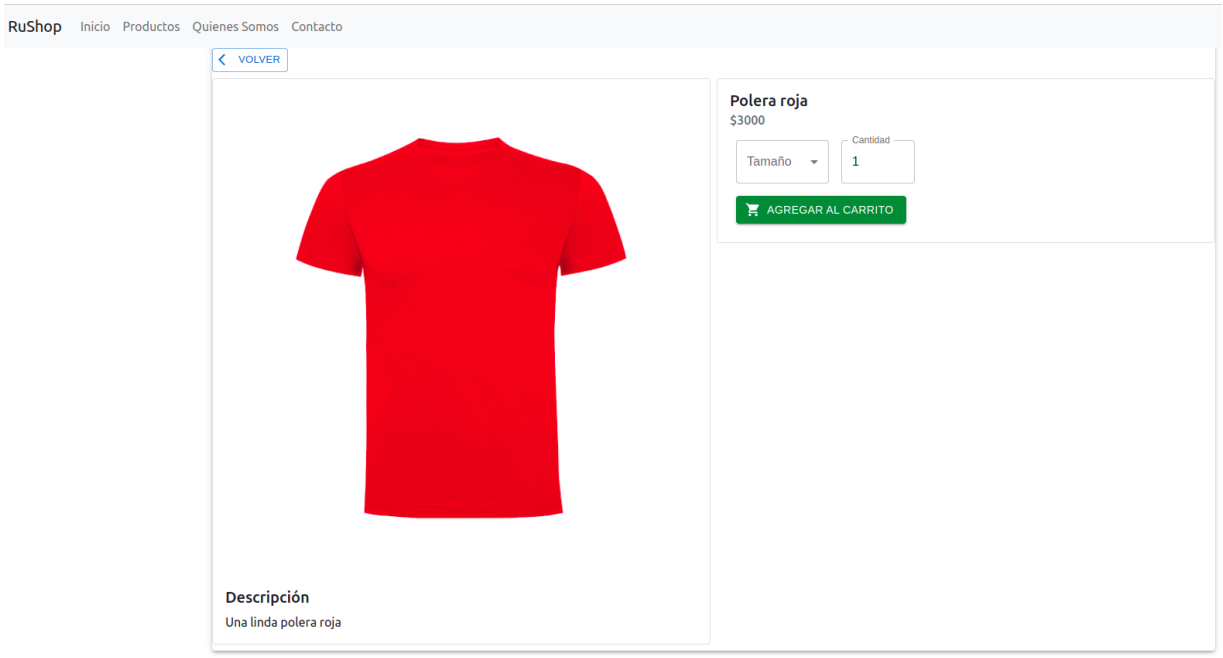

Figura 5.1: Primera versión de vista de un producto

Desde el frontend se pueden hacer peticiones a través de requests a la API desarrollada en el proyecto Django. Para que se pudieran realizar correctamente las llamadas se tuvo que configurar el mecanismo de uso compartido de recursos entre orígenes, también conocido como CORS.

Para versionar el código se crearon dos proyectos en Gitlab, uno para el backend y otro para el frontend, de esta forma se podrían administrar los proyectos en paralelo y que los errores que pudiera tener un proyecto no afectaran al otro. Ambos tendrían una rama llamada *dev* para el ambiente de desarrollo y otra llamada *main* para el código del ambiente de producción. Desde la rama *dev* se crearían distintas ramas para desarrollar las funcionalidades y utilizar la metodología *Gitflow*.

Se decidió partir con la vista de un producto ya que es una interfaz simple, desde la cual se debe hacer una llamada desde el frontend al backend para obtener los datos de cada producto. La primera versión de la vista de un producto desarrollada en este sprint se puede ver en la figura 5.1.

Para la vista de un producto se creó el componente llamado `ProductView` el cual contiene el código del componente principal, es decir, toda la vista mostrada en la figura 5.1. La vista tiene dos componentes principales, uno llamado `ProductInfo` donde se tiene la imagen y descripción del producto y otro llamado `Product` donde se tiene la parte derecha de la vista.

En el componente `Product` se obtiene el nombre y precio del producto a través de las props otorgadas por el componente padre (`ProductView`), luego se entregan los datos del producto obtenido del padre al componente `ProductOptions`.

En `ProductOptions` se tiene un campo para seleccionar el tamaño del producto que se quiere comprar, la cantidad y un botón para agregarlo al carrito. En este sprint este

componente solo muestra estos objetos, no son funcionales, ya que había mucha incertidumbre sobre cómo desarrollar la funcionalidad para crear una orden y luego agregar productos a una respectiva orden. Además esto escapaba del alcance del sprint y podría desarrollarse una vez que se empezara a hacer la funcionalidad para administrar el carrito de compras.

En la parte del backend se tuvo que crear un endpoint para listar la información de los productos. Para esto se crearon los primeros serializadores de `PurchaseApp`: `ProductSerializer` y `SizeSerializer`. Se utilizó `ProductSerializer` en `ProductViewSet` y se configuró la URL de esta `ViewSet` en el archivo `urls.py` de la aplicación. Con esto desde el componente `ProductView` se utiliza el endpoint `/products/n` a través del método GET para obtener la información del producto con número de identificación `n`.

Además en este sprint se desarrolló el componente `StoreNavbar` para mostrar una barra de navegación en la parte superior de la página. Esta se puede ver en la figura 5.1. Se utilizó *Reacstrap* para el estilo.

5.2. Segundo sprint

- Agregar admin de Django
- Crear fixtures
- Configurar routing

Se incorporó el módulo de administración de Django, el cual permite visualizar y editar la base de datos a través de una interfaz de usuario creada automáticamente a partir del modelo de datos. Esta funcionalidad podría usarse por los dueños de las tiendas durante la primera etapa del desarrollo de la plataforma para enfocar el trabajo en las funcionalidades necesarias para que los clientes puedan hacer compras.

Durante este sprint se decidió crear fixtures para contar con datos necesarios en el sistema para poder probar las funcionalidades que se empezaran a desarrollar. En el capítulo anterior se muestra un ejemplo de fixture en el código 1.

Se crearon datos de una tienda ficticia llamada Rushop, una usuaria administradora de la tienda, su dirección, las regiones de Chile y algunas comunas. Para la tienda virtual se crearon cinco categorías de productos y tres productos con sus respectivas imágenes y datos.

Para el proceso de routing se usó la librería *React Router*, la cual permite definir las múltiples rutas necesarias para la plataforma. Se tuvo que aprender a utilizar esta librería para incorporarla correctamente, lo cual tomó gran parte del tiempo del sprint.

En este sprint se creó el componente `StoreView`, se desarrolló parte del código de `index.js` mostrado en la sección anterior y una primera versión del componente `App` que permitía mostrar la barra de navegación en los componentes creados hasta ese momento y navegar a través de ellos.

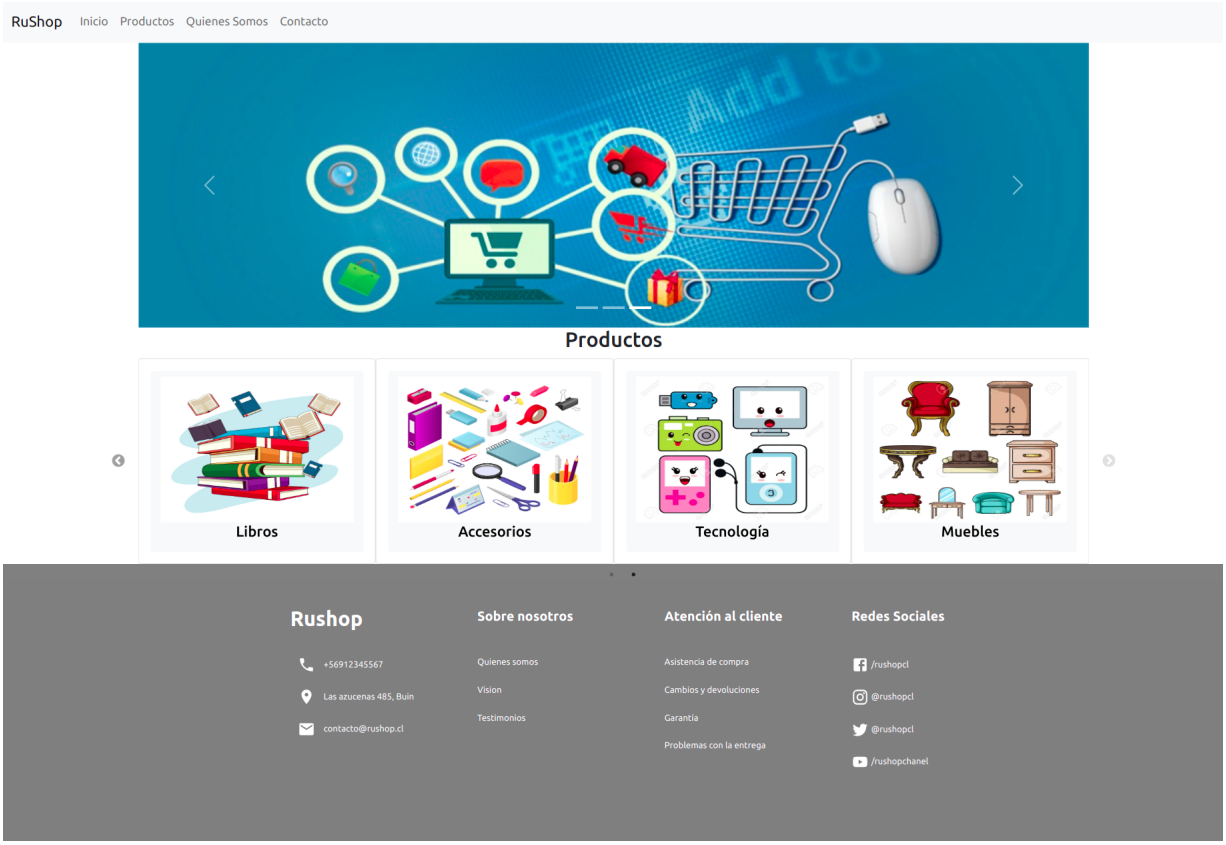

Figura 5.2: Primera versión de página principal de una tienda

5.3. Tercer sprint

- Catálogo de productos
- Agregar pie de página
- Página principal de una tienda

Como se puede ver en la figura 3.3 del capítulo 3, el catálogo de productos debía verse como una lista de las categorías con su respectiva imagen, donde al hacer click en una se muestran los productos pertenecientes a esa categoría. Para permitir el desplazamiento a través de los productos con un botón para avanzar y otro para retroceder se utilizó la librería *React-Slick* en el componente `Catalogue`, el cual se utiliza en `StoreView`.

En el backend se tuvieron que crear endpoints para listar las categorías, los productos a partir de una determinada categoría y un endpoint para listar las imágenes de una tienda. Para esto se crearon los serializadores `CatalogueSerializer`, `CatalogueProductSerializer`, `CategorySerializer`, `StoreSerializer`, `ImageSerializer` y `StoreImagesSerializer`. Se agregaron las vistas de `CategoryViewSet`, `CatalogueViewSet`, `StoreImages` y `StoreViewSet` con su respectiva configuración de URLs.

En la figura 5.2 se puede ver la primera versión del catálogo de productos, junto al carrusel que se agregó para que las tiendas puedan mostrar imágenes que inviten a los visitantes a

comprar a través de la página. Además se puede ver el pie de página con información y links que podrían ser útiles para los clientes.

Durante este sprint se desarrollaron los componentes `Catalogue`, `StoreImages` y `StoreFooter`, utilizando los dos primeros en `StoreView`. `StoreFooter` se utiliza directamente en `App` ya que se muestra en todas las vistas principales de la plataforma.

5.4. Cuarto sprint

- Agregar productos al carrito de compras
- Borrar productos del carrito
- Utilizar almacenamiento local
- Usar hook de contexto
- Calcular precio total del carrito

El objetivo de este sprint era implementar la funcionalidad para que los clientes puedan agregar y quitar productos de un carrito de compras. Para lograr esto se tuvo que trabajar con el modelo `Order`, creando un endpoint para agregar productos a una orden, incluyendo el caso en que la orden aún no exista.

En el backend se crearon los serializadores: `OrderSerializer`, `ProductOrderSerializer`, `ShowProductSerializer`, `ShowProductOrderSerializer` y `ShoppingCartSerializer`. Además se agregaron las vistas de `ShoppingCartViewSet`, `ShoppingCartProductViewSet`, `OrderViewSet` y `ProductOrderViewSet` con su respectiva configuración de URLs.

Cuando un usuario accede a la página por primera vez no hay una orden asociada a su visita y solo cuando se hace click en agregar un producto se crea una orden con el producto que agregó. Para esto se tuvo que desarrollar el método `create` de `OrderSerializer`, lo cual se explicó con más detalle en el capítulo anterior.

Para el frontend se crearon los componentes `ShoppingCart`, `ShoppingCartProduct` y `OrderPrice`. Además se agregaron las funciones que permiten seleccionar el tamaño de un producto y la cantidad que se desea agregar al pedido en la vista de la ficha de un producto.

El número identificador de la orden se guarda en el almacenamiento local del cliente para que cuando este cierre la ventana del navegador no pierda los datos de su pedido. Se tuvo que aprender a utilizar el objeto `localStorage` de javascript y crear el hook `useLocalStorageOrDefault` para poder realizar esta funcionalidad.

También se tuvo que aprender a usar el contexto de React descrito en la sección anterior, para acceder al id y precio total de un pedido desde los componentes que lo requieren. Estos componentes son: `OrderDetail, OrderPrice, ShoppingCart`, `ShoppingCartProduct`, `ProductOrder` y `ProductOptions`.

El carrito de compras corresponde a un panel (*Drawer* en inglés) ubicado a la derecha de la ventana. Este se abre cuando se presiona un botón que se ubica en la esquina inferior derecha de las vistas `StoreView` y `ProductView`. En este panel se muestra la lista de productos que se han agregado al pedido, se tienen botones en cada producto para agregar más o quitarlos, se muestra el precio total de la compra y un botón que redirige a la vista de la información del pedido.

Al final de este sprint se logró tener una primera versión funcional del carro de compras. Este permitía agregar y quitar productos del pedido y mostrar la información suficiente para que los usuarios puedan realizar pedidos más fácilmente.

5.5. Quinto sprint

- Formulario para agregar datos del cliente
- Formulario para agregar dirección
- Vista detalle de un pedido
- Usar pasos en el flujo de compra

Una vez que un cliente agrega los productos que desea comprar a su carrito de compras, debe hacer click en el botón comprar para continuar con el flujo de compra. Para mejorar la experiencia de usuario, se decidió usar pasos en el flujo de compra en vez de tener una sola vista que contenga toda la información del pedido como se había planificado en la etapa de diseño.

Se utilizó el componente `Stepper` de Material Ui para hacer la interfaz de pasos en el flujo de compras. En `OrderView` se usa el componente `Stepper` el cual llama a los componentes `OrderDetail, ClientForm, DeliveryOptions` y `OrderSummary`.

El primer paso consiste en una tabla donde se muestran los productos que se tienen en el carro. Desde esta tabla se puede modificar la cantidad que se quiere de cada producto o borrarlos. Para esto se crearon los componentes `OrderDetail` y `ProductOrder`; en el primero se tiene la tabla y se utiliza el segundo que es el que se encarga de renderizar cada producto por sí solo. Para obtener los datos de los productos se utiliza el mismo endpoint que se usa en el carrito de compras ya que la información que se requiere es la misma.

El segundo paso es un formulario donde el cliente debe agregar su nombre, apellido, correo electrónico y número de teléfono. Este formulario se explicó con más detalle en la sección anterior.

En el tercer paso debe elegir si desea que su pedido llegue a su domicilio o si lo quiere retirar en la tienda. En caso de que quiera que llegue a su domicilio se despliega un formulario donde debe agregar su dirección.

El formulario de dirección cuenta con campos de texto para que el usuario agregue la calle, el número, el departamento y la torre del edificio si es que corresponde. Además tiene dos select, uno para agregar la región y otro para la comuna.

Se creó un componente llamado `AddressForm` para el formulario con los cuatro campos de texto descritos en el párrafo anterior. Para los dos select se creó un componente llamado `TownSelect`.

En `TownSelect` se usa la librería *React Select*, la cual permite crear selects a partir de un json con claves en un formato determinado. Para convertir los datos traídos desde la base de datos se creó una función que convierte el json que devuelve el endpoint `regions` y `towns-by-region` en el diccionario que pide react-select.

El usuario debe seleccionar la región primero, para esto cuenta con un buscador para facilitar la elección. Una vez que elige la región debe elegir su comuna. El select de comunas contiene las opciones de acuerdo a la región seleccionada, es decir, si el usuario elige la región metropolitana, el select de comunas mostrará solo las comunas de esa región. Si el usuario no elige región, no se mostrarán opciones para la comuna.

Desde el backend, para el formulario de los datos de contacto del cliente se creó el serializador `ClientSerializer`, en el cual hubo que sobreescribir el método `create` mostrado en la sección anterior. También se agregaron las vistas de `ClientViewSet` con su respectiva configuración.

Para el formulario de direcciones se agregaron los serializadores: `AddressSerializer`, `RegionSerializer`, `TownSerializer` y `TownRegionSerializer`, junto a las vistas: `AddressViewSet`, `RegionViewSet`, `TownViewSet` y `TownRegionViewSet` con su respectiva configuración de URLs.

Durante este sprint se desarrollaron los componentes mencionados por separado y una vez que se tuvieron listos se juntaron en el componente `OrderView` obteniendo la vista mostrada en la figura 5.3. Se consideró que esta vista contiene demasiada información y se alargaría mucho pues aún falta agregar el método de pago al flujo. Por esto se cambió el diseño que se tenía en el prototipo por una vista con pasos, la cual tomó poco tiempo en implementar y se desarrolló al final de este sprint.

5.6. Sexto sprint

- Vista resumen de un pedido
- Notificar por mail confirmación de un pedido
- Deploy en heroku
- Tabla de pedidos pendientes

El último paso del flujo de compras consiste en una lista que resume los datos del pedido y un botón para confirmar la compra. Cuando se hace click en el botón confirmar, se manda un correo de confirmación al correo electrónico otorgado por el usuario en el formulario de datos del cliente.

Para mandar correos a través de la plataforma, se tuvo que crear un correo de prueba llamado *rushopcl@gmail.com*. El backend se encarga de mandar el correo a través de un

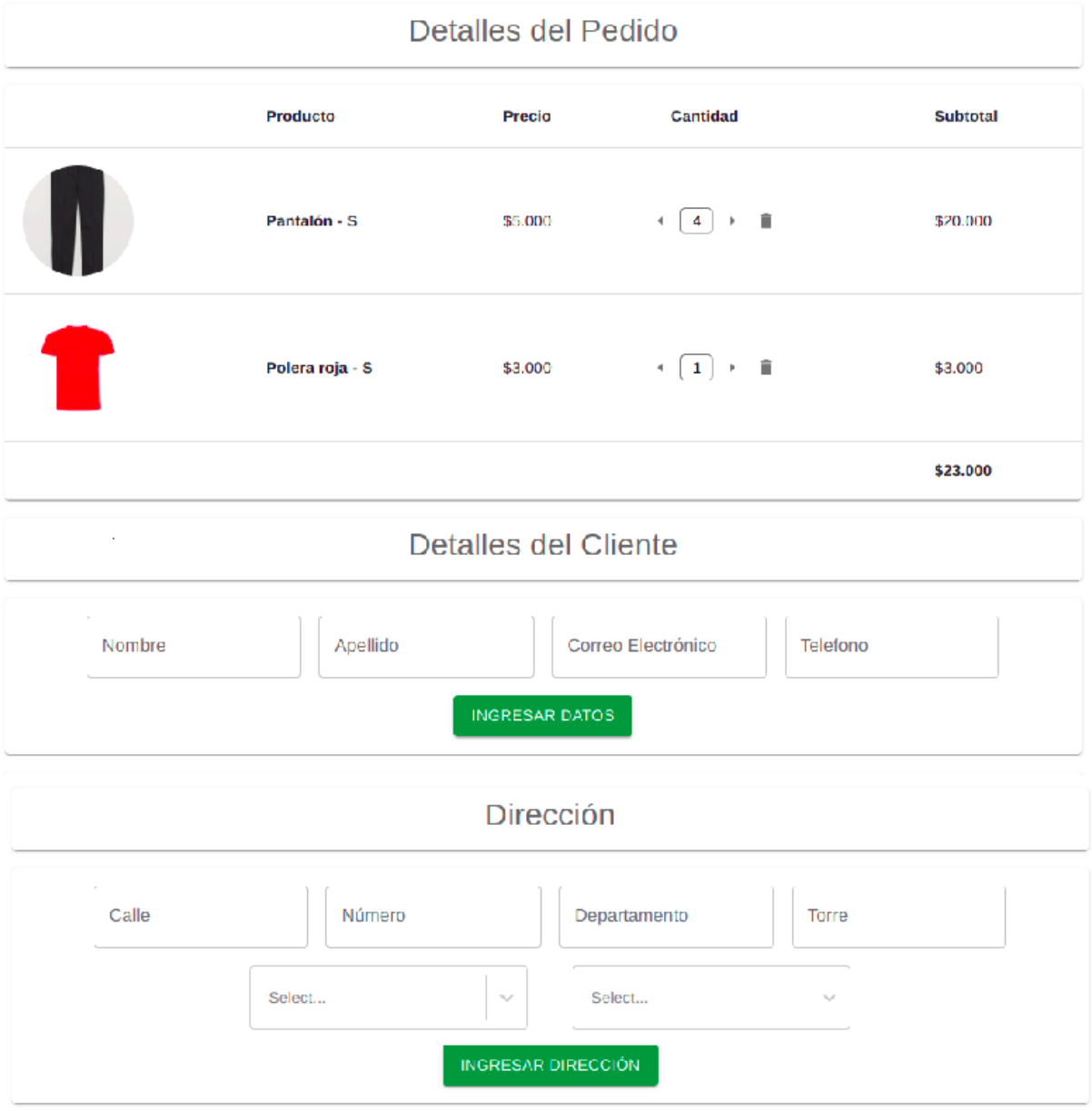

Figura 5.3: Primera versión de vista información de un pedido.

endpoint que recibe los datos de un pedido, si se tiene el campo "send_mail" se utiliza la función que parsea los datos y envía el mail. El código de esta función se mostró en el capítulo anterior.

Para el último paso del flujo de compras se crearon los componentes `OrderSummary`, `ProductsSummary`, `ClientSummary` y `AddressSummary`. Desde el primero se llama a los otros tres componentes mencionados.

En el backend se desarrollaron los serializadores `OrderSummarySerializer`, `OrderDetailSerializer` y `ListProductOrderSerializer`, además se agregaron las vistas de `ListProductOrderViewSet`, `OrderDetailViewSet` y `OrderSummaryViewSet` con su respectiva configuración de URLs.

Para facilitar la validación del sistema se hizo un despliegue a través de la plataforma *Heroku*. Se crearon dos apps, una para el backend y otra para el frontend. En ambas se tuvo que investigar las configuraciones necesarias para cada tipo de proyecto.

En el backend hubo que agregar los archivos `Procfile`, `runtime.txt` y `requirements.txt`. Además se tuvo que configurar el uso de la librería *Whitenoise* para manejar los archivos estáticos del proyecto, los cuales son necesarios para que el administrador de Django se vea correctamente.

Para el frontend hubo que modificar las URLs que se utilizaban en los componentes para realizar peticiones al backend ya que utilizaban el localhost, lo cual funcionaba en el ambiente de desarrollo. Esto se modificó a través de una variables de entorno. Ahora los componentes obtienen la URL del servidor desde una variable de entorno llamada "REACT_APP_SERVER". Además hubo que modificar el archivo `package.json` para entregarle a heroku la información necesaria para realizar el despliegue correcto de la aplicación.

La última funcionalidad que se decidió agregar a la plataforma para este trabajo de título fue la de la vista de pedidos pendientes. Esta se consideró importante ya que los administradores deben tener una fuente de información para saber cuales son los pedidos que se han realizado a través de la plataforma y poder entregarlos correctamente.

Para la vista de pedidos pendientes se crearon los componentes, `PendingOrdersView` y `PendingOrder`, desde este último se utiliza el componente desarrollado en el sprint anterior, `ProductsSummary` para mostrar una lista de productos de cada pedido. Desde el backend se creó una vista genérica que filtra los pedidos que tengan el estado pendiente para mostrar sólo los pedidos en ese estado. Esta vista se llama `PendingOrders`, el endpoint es "/pending-orders/".

5.7. Séptimo sprint

Durante este sprint se corrigieron bugs y se mejoraron funcionalidades que habían quedado incompletas durante sprints pasados. Los errores corregidos junto a su descripción fueron los siguientes:

- **Agrupar productos cuando se crean dos iguales**: se corrigió que cuando se agrega un producto al carrito de compras se agrupe en el caso que el producto con las mismas características ya exista.
- **Mostrar datos en formularios**: En la primera versión de los formularios del cliente y su dirección, no se mostraban los datos que ya se tenían en la base de datos en sus respectivos campos de texto. En este sprint se corrigió esto.
- **Enviar correo sólo cuando se confirma la orden**: Cuando se agregó la funcionalidad de enviar mails cuando se confirma una orden, se utilizó la función `send_email` directamente en el método `update` de `OrderSummaryViewSet`, el cual también se utiliza cuando se cambia el método de reparto de un pedido, por lo que se enviaban correos erráticamente antes de confirmar el pedido. Esto se corrigió teniendo que agregar un llave llamada "send_email" en la request de la función que se utiliza cuando se hace click en el botón confirmar.
- **Enviar correo desde cuenta Gmail**: Cuando se realizó la funcionalidad de notificar cuando se confirma una orden, se configuró un correo de prueba para enviar los mails desde una cuenta en Gmail. Gmail cambió las condiciones para poder enviar correos automáticamente desde una API propia al poco tiempo de haber desarrollado esta funcionalidad, por lo que dejó de funcionar cuando se realizó el cambio. Para corregir esto se tuvo que investigar cual era la nueva forma de realizar este tipo de envíos.
- **Seleccionar tamaño por defecto según BD**: En la primera versión de la ficha de un producto se consideró que el tamaño por defecto de los productos es el primero que se encuentre en la base de datos. Esto traía problemas cuando un producto no tiene tamaño, ya que en el pedido se guardaba con el tamaño por defecto. Esto se corrigió permitiendo en el modelo que el tamaño de un `ProductOrder` pueda ser nulo y entregando un valor por defecto solo a los productos que tienen tamaños.
- **Recargar carrito al agregar producto**: Cuando se agrega un producto al carro de compras, lo ideal es que este se muestre para que el usuario pueda tener feedback de la acción que realizó y pueda revisar su carrito sin tener que presionar un botón. Esto no era así en la primera versión del carrito y se corrigió durante este sprint.

5.8. Discusión de la metodología utilizada

En este capítulo se listaron las tareas que se desarrollaron en cada sprint, describiendo en qué consistían y los elementos que hubo que implementar para lograr las funcionalidades deseadas. Luego de esta descripción se puede tener una idea de cómo se distribuyó el tiempo del semestre en desarrollar las distintas funcionalidades de la plataforma.

Durante el semestre se utilizó metódicamente la plataforma Jira para planificar las tareas que se realizaría en cada sprint. Esta plataforma permite tener un backlog de las tareas, crear sprints y arrastrar fácilmente las tareas entre un sprint y otro.

Al final de cada sprint la memorista se reunió con la profesora guía, mostrando el trabajo que realizó durante las dos semanas que duraba cada sprint. En las reuniones la memorista

recibía feedback sobre el desarrollo realizado y sobre lo que debería priorizar para continuar con el trabajo, decidiendo en conjunto las tareas que se realizarían en el siguiente sprint. Esto permitió tener una planificación a corto plazo de lo que se debía realizar y a la vez, se tenía una planificación flexible ya que si una tarea tomaba más del tiempo estimado o no se alcanzó a realizar, fácilmente se podía modificar, arrastrar a un nuevo sprint o dejarla en el backlog.

Esta forma de trabajo funcionó bastante bien ya que al tener una idea de las tareas que se debían priorizar y una planificación concreta de los días que se tenían para poder ejecutarlas, se logró mantener un ritmo de trabajo fluido, donde no hubo momentos en que no se sabía que hacer o donde se haya estancado el desarrollo por alguna tarea que no se logró realizar.

Una de las funcionalidades importantes que se dejó para desarrollo posterior a este trabajo de título fue la implementación del método de pago. Esta funcionalidad es fundamental para completar el flujo de una compra ya que sin ella, los usuarios pueden crear pedidos sin comprometerse a pagar antes de recibirlo, lo cual puede implicar que menos compras se concreten.

Se decidió postergar la funcionalidad descrita en el párrafo anterior porque para implementar un método de pago en línea se debe usar la API de Transbank o Mercado Pago, las cuales exigen ciertos requisitos que se deben cumplir para utilizarlas en un ambiente de producción. Esto implicaba comprometer mucho tiempo en una funcionalidad que difícilmente se alcanzaría a completar, ya que había que aprender a utilizar alguna de las APIs, implementarla correctamente en el proyecto, luego cumplir los requisitos pedidos y esperar la aprobación de Transbank o Mercado Pago para efectivamente poder utilizarla.

Capítulo 6

Validación

En este capítulo se muestran las interfaces de la plataforma y se describen las funcionalidades implementadas, mencionando cuáles son los requisitos listados en la sección 3.1.2 que se cumple en cada interfaz para realizar una validación funcional del producto.

6.1. Descripción del sistema

La plataforma desarrollada es una aplicación web donde los usuarios pueden realizar compras a través de ella. Para que la página pueda funcionar los administradores de la tienda deben ingresar los datos de su tienda, los productos que ofrecen e imágenes que inviten a los usuarios a comprar.

6.1.1. Módulo de administración

Una de las características de Django es que ofrece un módulo para administrar el modelo de datos a través de una interfaz gráfica generada automáticamente. Para acceder a esta interfaz se debe tener un usuario con su respectiva contraseña por lo que no cualquier persona puede acceder a ella. Con la implementación de este módulo se cumplen los requisitos 1, 2 y 3.

En la figura 6.1 se puede ver la interfaz para gestionar la base de datos del sistema. Los administradores pueden agregar, editar y borrar los distintos valores de los campos del modelo de datos detallado en la sección 3.2.2. Como se puede ver en la imagen, se configuró el idioma español en este módulo ya que la plataforma está pensada para que sea utilizada en primera instancia por hablantes nativos de este idioma.

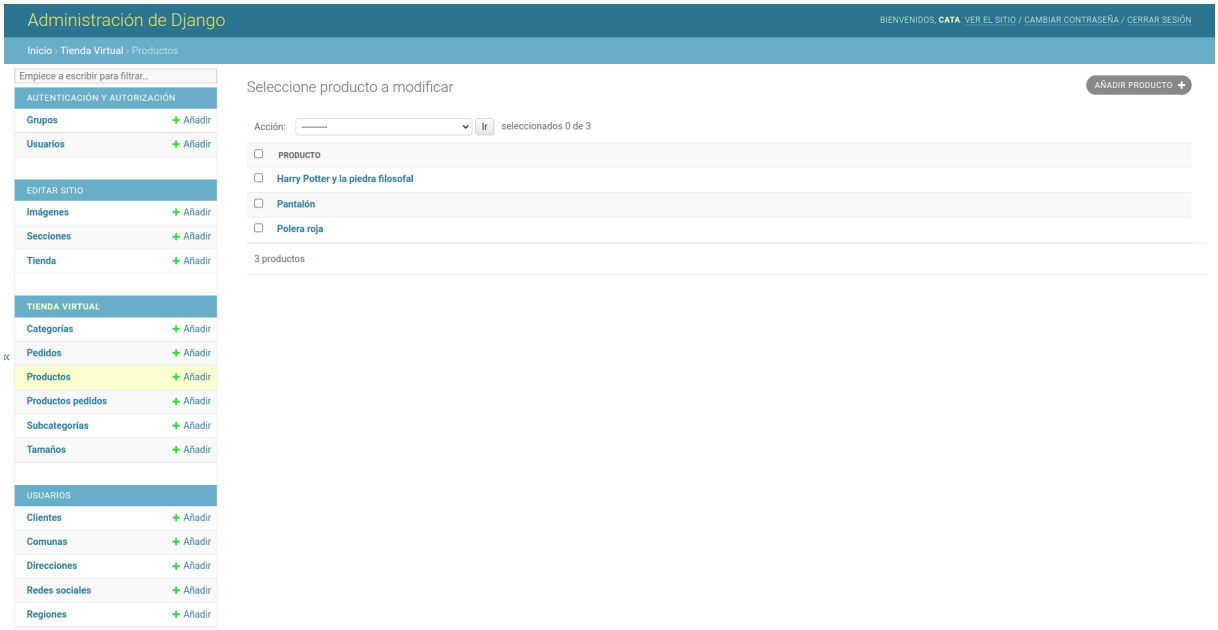

Figura 6.1: Módulo de administración

6.1.2. Página principal

En la figura 6.2 se puede ver la página principal de una tienda de prueba llamada Rushop. La interfaz cuenta con una barra de navegación en la parte superior donde se puede acceder a las distintas secciones de la página. Además cuenta con un carrusel de imágenes referenciadas previamente por el administrador de la tienda en el modelo de datos.

En esta vista se puede ver el catálogo de productos. En este se tienen las categorías de los productos que ofrece la tienda. La tienda que podemos ver en la figura 6.2 tiene 5 categorías: *Ropa*, *Libros*, *Accesorios*, *Tecnología* y *Muebles* (si se presiona el botón «▷» se puede ver la última categoría).

Cuando un usuario hace click en en una categoría el catálogo muestra los productos de la categoría seleccionada. Se puede navegar a través de los distintos productos con los botones «◁» y «▷» . En la figura 6.3 se muestra cómo se ve el catálogo cuando se elige la categoría *Ropa*.

Las distintas imágenes de los productos son clickeables, redirigiendo a la ficha del determinado producto. Para volver a ver las categorías se puede hacer click en «Productos».

En la figura 6.4 se puede ver el pie de página del sitio web, el cual se muestra al final de todas las secciones. Para que los links a las redes sociales funcionen los administradores deben haber agregado previamente los links en la base de datos.

Considerando que las tres imágenes nombradas en esta subsección es la misma interfaz, con esta se cumplen los requisitos 4 y 9.

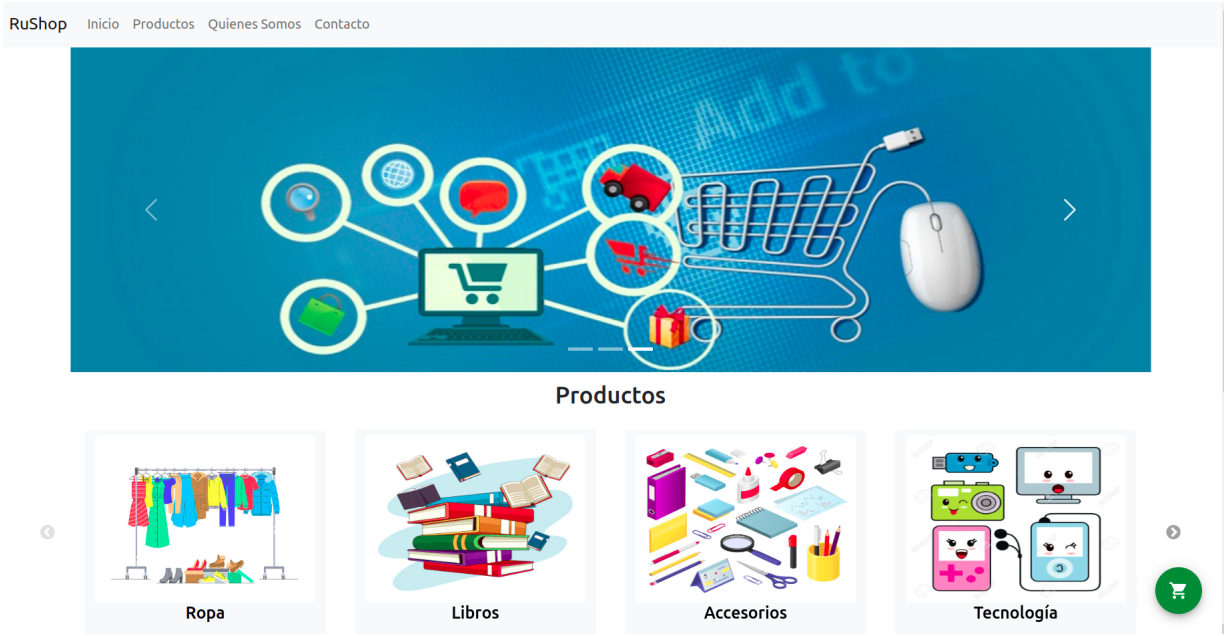

Figura 6.2: Página principal de una tienda virtual

Figura 6.3: Catálogo de productos.

Figura 6.4: Pie de página.

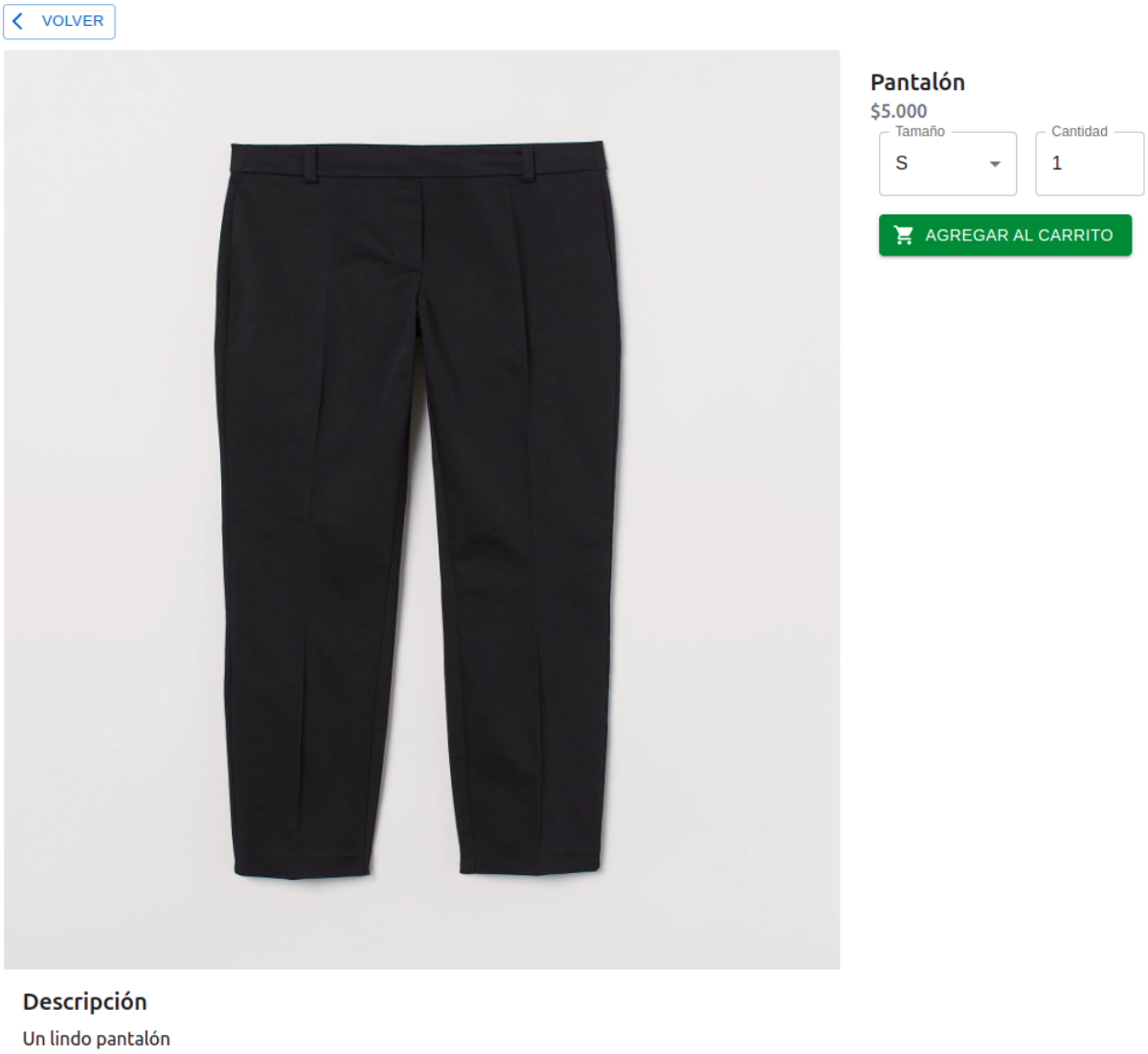

Figura 6.5: Ficha de un producto

6.1.3. Ficha de un producto

Al hacer click en un producto del catálogo se muestra una interfaz (figura 6.5) donde se puede ver la imagen en un tamaño más grande, además se muestra la descripción, el precio y los distintos tamaños que puede tener el producto. El usuario puede seleccionar el tamaño y la cantidad del producto que desea agregar a su pedido.

Si se presiona el botón «agregar al carrito», se añade el producto al pedido y se muestra el carrito de compras que se tiene hasta el momento. Con esta interfaz se cumplen los requisitos 5 y 6.

Figura 6.6: Carrito de compras

6.1.4. Carrito de compras

El carrito de compras es un panel, también conocido como *drawer*, a la derecha de la pantalla donde se listan los productos que se han agregado a la orden, se puede ver el ejemplo de un carrito en la figura 6.6. Este panel se muestra al hacer click en el botón circular de color verde situado en la esquina inferior derecha de la página y al agregar un producto al carrito.

La interfaz del carrito de compras permite agregar y disminuir la cantidad de cada producto y/o borrarlos. Además se muestra el precio de cada producto y el total, el cual se actualiza cuando se hace click en cualquiera de los botones disponibles. Cuando se hace click en el botón «comprar» se redirige a la vista *Información del pedido*. Con esta interfaz se cumplen los requisitos 7 y 8.

6.1.5. Información del pedido

En esta vista se muestran los productos que se agregaron al pedido, un formulario para agregar los datos del cliente y su dirección y una sección donde se puede ver el resumen del pedido y un botón de confirmación.

Se utilizaron pasos para ordenar y navegar a través de las distintas secciones de la vista (figuras 6.7, 6.8, 6.9 y 6.10). Se tienen los botones volver y siguiente para cambiar el paso, también se puede hacer click en el nombre y número del paso para acceder al determinado paso.

En el primer paso (figura 6.7) se muestra la lista de productos, pudiendo aumentar y disminuir la cantidad de cada producto. Con este paso se cumple el requisito 12.

El segundo paso consiste en un formulario para que los clientes agreguen sus datos y puedan ser contactados por el administrador de la tienda en caso que lo requieran (figura 6.8). Con este paso se cumple el requisito 16.

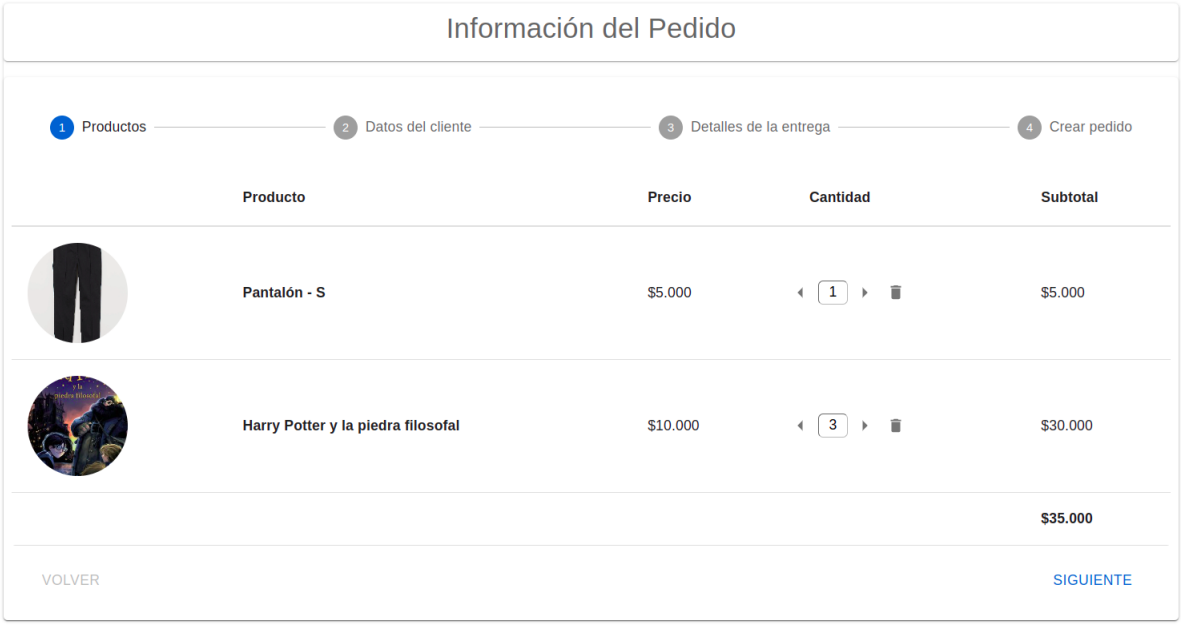

Figura 6.7: Lista de productos

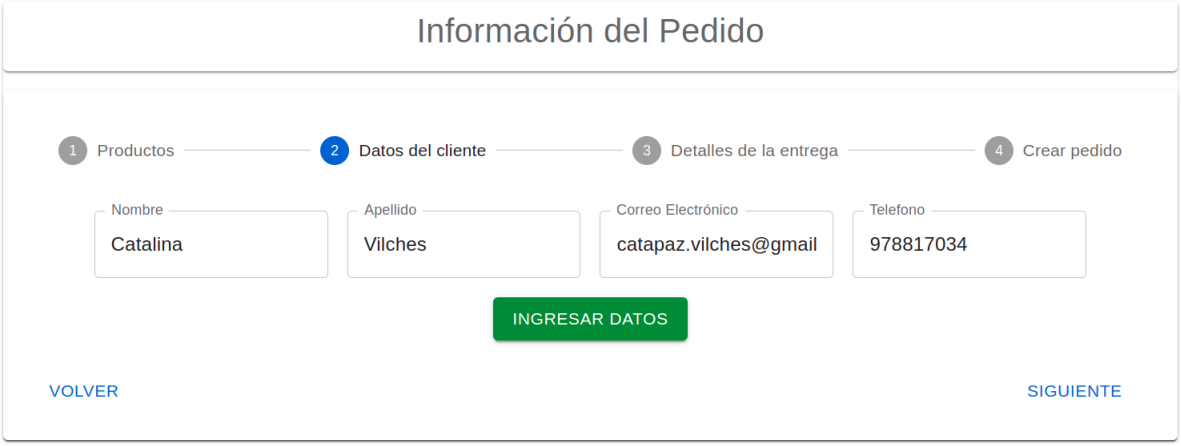

Figura 6.8: Formulario cliente

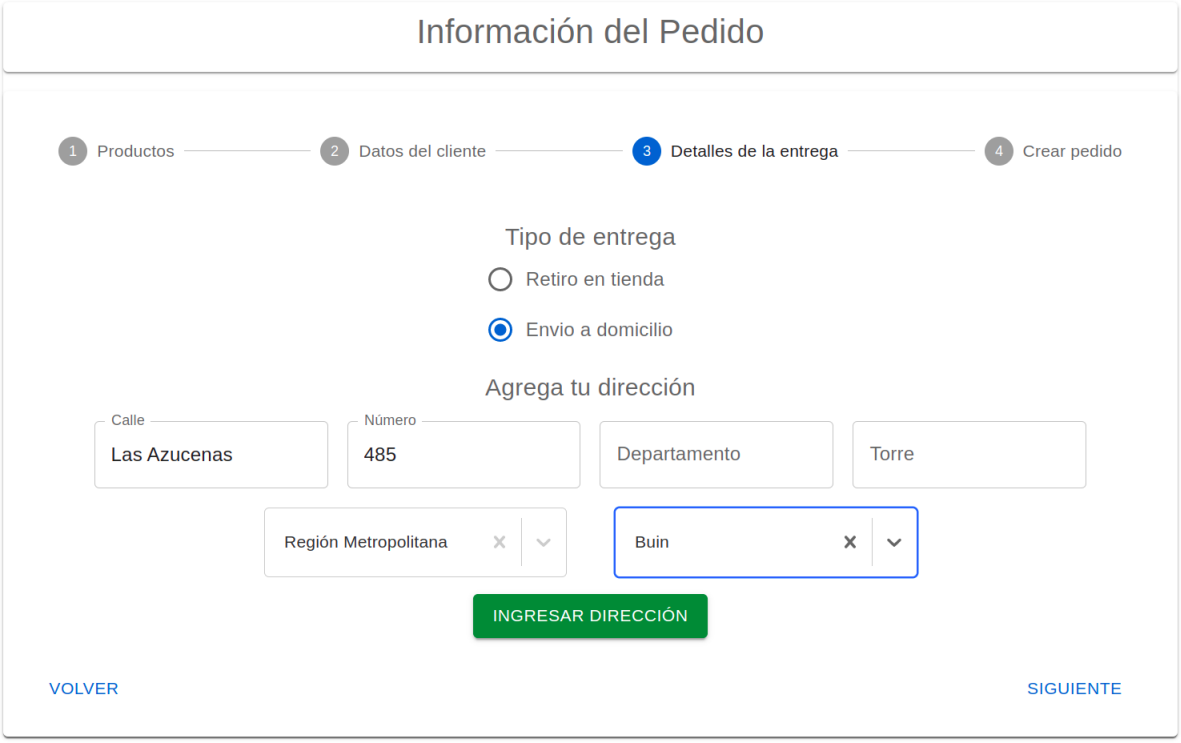

Figura 6.9: Formulario de dirección de entrega

El paso 3 (figura 6.9) es un formulario donde el cliente puede elegir si desea retirar su pedido en la tienda o si quiere que se le entregue en su domicilio. Si elige la segunda opción se despliegan los campos requeridos para ingresar la dirección del domicilio. Con este paso se cumple el requisito 11 y 16.

En el último paso (figura 6.10) se muestra una lista que resume el pedido para que el cliente pueda revisar y corregir la información antes de confirmar el pedido. Cuando el usuario hace click en el botón para confirmar el pedido se manda un correo con el resumen del pedido al correo electrónico ingresado en el formulario de datos del cliente y al correo del administrador de la tienda. Con esta interfaz y las notificaciones al correo electrónico se cumplen los requisitos 12, 13 y 15.

6.1.6. Pedidos pendientes

La última interfaz desarrollada fue una vista donde se tiene una tabla que muestra los pedidos en estado pendiente. Esta interfaz tiene como propósito que los administradores de las tiendas puedan revisar a través de la plataforma cuáles son los pedidos que deben concretar.

En la figura 6.11 se puede ver la interfaz mencionada en el párrafo anterior. Esta sería la primera del módulo propio de administración. Con esta se cumple el requisito 14.

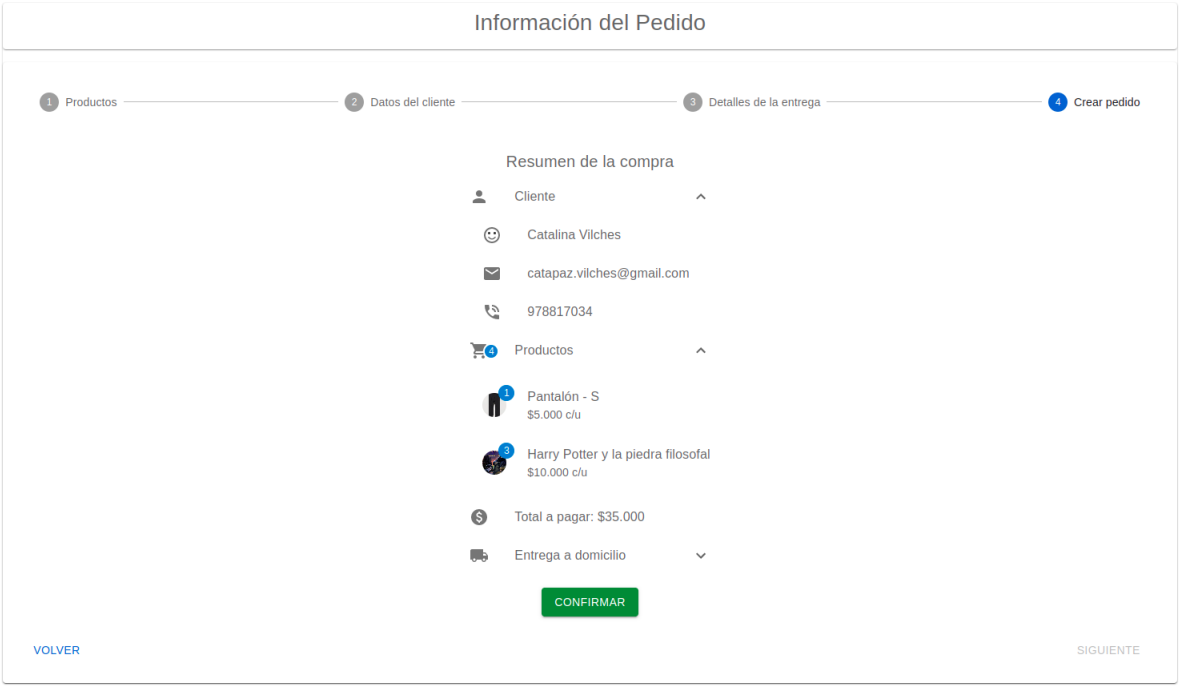

Figura 6.10: Confirmación del pedido

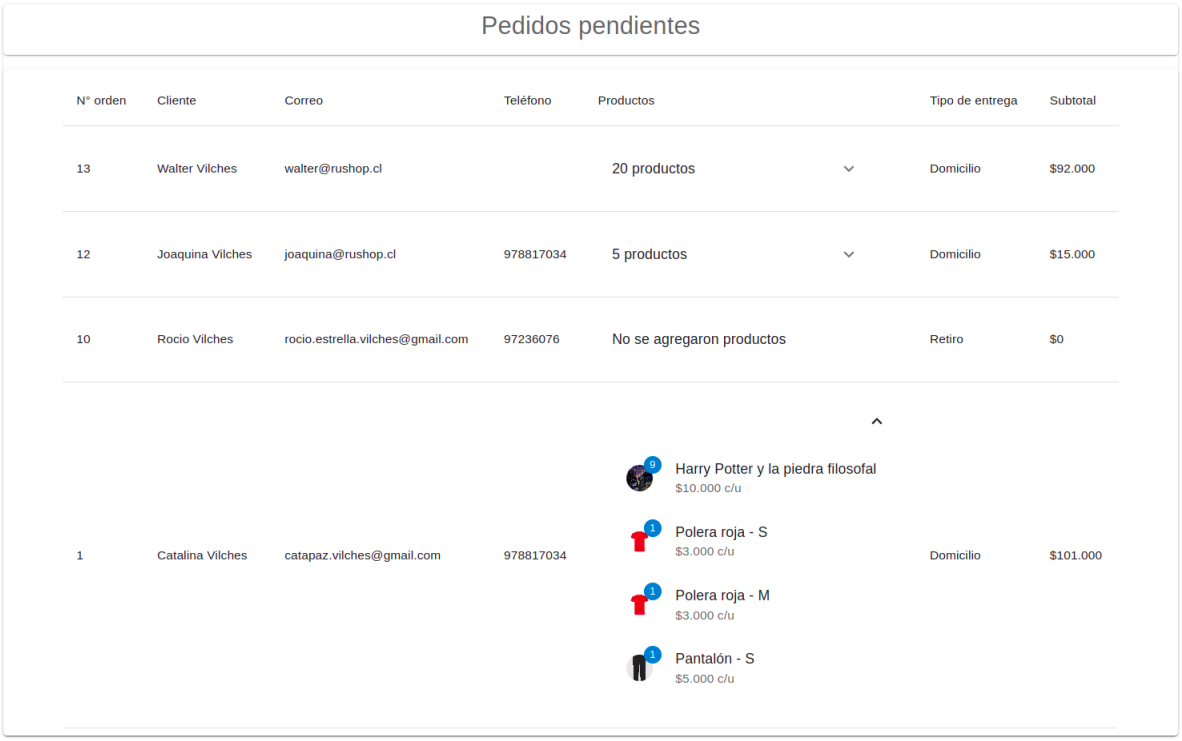

Figura 6.11: Lista de pedidos pendientes

	Subsección 6.1.1	Subsección 6.1.2	Subsección 6.1.3	Subsección 6.1.4	Subsección 6.1.5	Subsección 6.1.6
1	✓					
2	✓					
3	✓					
4		✓				
5			✓			
6			✓			
7				✓		
8				✓		
9		✓				
10						
11					✓	
12					✓	
13					✓	
14						✓
15					✓	
16					✓	
17						
18						

Tabla 6.1: Validación funcional de los requisitos enumerados en la sección 3.1.2 que se cumplen con la solución implementada.

6.2. Validación funcional

En la tabla 6.1 cada columna es una parte de la plataforma que fue descrita en la sección 6.1. Las filas son los requisitos listados en la sección 3.1.2. Con esta tabla se puede ilustrar cuáles fueron los requisitos levantados en la etapa de análisis del proyecto que se lograron cumplir con el producto implementado.

Se puede ver que se cumplieron 15 de los 18 requisitos propuestos, lo cual es bastante positivo ya que se alcanzó a implementar la mayoría de las funcionalidades que se planificaron para este trabajo de título.

Uno de los requisitos que no se cumplió fue el 10, al final del capítulo anterior se explicó porque se decidió postergar esta funcionalidad. Los requisitos 17 y 18 no alcanzaron a desarrollarse ya que se decidió priorizar las funcionalidades relacionadas con el flujo de compra de los clientes.

Capítulo 7

Conclusión

Con la solución implementada descrita en esta memoria, se cumplió el objetivo de desarrollar una plataforma que permita a micro empresarios sin conocimientos en programación, administrar una tienda virtual. Además, potenciales clientes pueden realizar pedidos a través de la plataforma.

Los objetivos específicos que se cumplieron fueron primero el de investigar y evaluar las soluciones existentes (se muestra esta investigación en la sección 2.2), luego se diseñó un prototipo que permita a micro empresarios sin conocimientos en programación, crear y administrar una tienda virtual. El prototipo diseñado que se muestra en la sección 3.2.3 también abordó la experiencia de usuario para que los clientes puedan comprar productos a través de las tiendas virtuales creadas por los administradores. Por último, se hizo una validación funcional de la solución en la sección 6.2.

La implementación del prototipo diseñado se cumplió parcialmente ya que si bien se cumplió la mayoría de los requisitos, para la experiencia de usuario de los administradores se utiliza el módulo de Django en vez de las interfaces diseñadas en el prototipo de la solución. Además los clientes sólo pueden realizar pedidos desde la plataforma y no pagar con un método de pago que permita realizar el flujo completo de una compra en línea.

Para implementar la plataforma desarrollada se dividió el semestre en sprints de dos semanas, donde al inicio de cada sprint se planificó cuáles tareas se desarrollarían, ordenándolas por prioridad. Esta metodología permitió tener fluidez en el trabajo realizado, manteniendo un ritmo constante donde al final de cada sprint siempre se tuvo funcionalidades nuevas en la aplicación.

Las tecnologías utilizadas cumplen con los requisitos necesarios para desarrollar una plataforma de esta envergadura. Por un lado, se tiene el repositorio del backend desarrollado con el framework Django, el cual permite crear una API fácilmente una vez que se entienden y aprenden a utilizar los distintos conceptos y recursos que utiliza el framework.

Por el lado del frontend, el uso de React permite visualizar y desarrollar la experiencia de usuario de un sistema, de forma intuitiva y coherente. El uso de esta herramienta junto a las librerías externas que se utilizaron, permitieron que implementar funcionalidades com-

plejas, como el uso de pasos en una interfaz y un carrusel para mostrar imágenes, pudieran desarrollarse en poco tiempo.

El uso de la plataforma Heroku para el despliegue de la aplicación permitió obtener la solución disponible desde cualquier computador que tenga acceso a internet, sin tener que gastar dinero en ello. Además Heroku permite configurar la base de datos con el gestor Postgresql, lo cual le entrega robustez al software al mantener los datos del sistema en un servidor que esté funcionando las 24 horas del día. Esto facilitará realizar demostraciones a clientes reales, pudiendo mostrar el potencial que tiene la aplicación y los beneficios que podrían obtener si deciden utilizarla para su negocio.

A modo personal, la memorista logró aplicar exitosamente en este trabajo de título los conocimientos adquiridos en los distintos cursos de la carrera Ingeniería Civil en Computación. La principal motivación para realizar este trabajo era obtener un software funcional realizado en su totalidad por la memorista, el cual podría mejorar en el futuro y utilizarlo en las instancias que considere pertinentes. La realización de este trabajo de título logró cumplir con las expectativas de la memorista.

Como trabajo futuro quedan varias tareas que se tienen planificadas en la plataforma Jira, entre ellas se tiene: realizar tests unitarios y de integración, para mejorar la calidad del software y poder aplicar integración continua; configurar los permisos de los distintos usuarios de la aplicación; implementar la funcionalidad para que los clientes puedan pagar en línea; seguir desarrollando el módulo de administración diseñado en el prototipo de la solución; agregar un registro e inicio de sesión de cualquier usuario para que clientes que compren en una tienda más de una vez no tengan que volver a ingresar sus datos de contacto en el sistema; utilizar un servicio de hosting más robusto que Heroku como AWS y por último queda implementar una gama amplia de funcionalidades que puede tener esta plataforma como permitir a los usuarios elegir un método de reparto que esté integrado en la aplicación (como Chilexpress, Starken, Blue, etc) o un módulo de mensajería instantánea entre los administradores y clientes.

Bibliografía

[1] Balsamiq: *Sitio web de Balsamiq*. https://balsamiq.com/, visitado el 22 de Agosto de 2022.

[2] Claire Drumond: *¿Qué es scrum?* https://www.atlassian.com/es/agile/scrum, visitado el 22 de Agosto de 2022.

[3] Django Software Foundation: *Sitio web de Django*. https://www.djangoproject.com/, visitado el 18 de Julio de 2022.

[4] Encode: *Sitio web de Django Rest Framework*. https://www.django-rest-framework.org/, visitado el 18 de Julio de 2022.

[5] Gitlab: *Sitio web de Gitlab*. https://about.gitlab.com/, visitado el 22 de Agosto de 2022.

[6] Jumpseller: *Sitio web de Jumpseller*. https://www.jumpseller.com/, visitado el 18 de Julio de 2022.

[7] Material UI SAS: *Sitio web de Material UI*. https://mui.com/, visitado el 18 de Julio de 2022.

[8] Mercado Libre: *Sitio web de Mercado Libre*. https://www.mercadolibre.cl/, visitado el 22 de Agosto de 2022.

[9] Meta Open Source: *Sitio web de React*. https://reactjs.org/, visitado el 18 de Julio de 2022.

[10] OECD: *OECD Economic Surveys: Chile 2021*. OECD Publishing, 2021. https://www.oecd-ilibrary.org/content/publication/79b39420-en.

[11] Salesforce: *Sitio web de Heroku*. https://www.heroku.com/, visitado el 18 de Julio de 2022.

[12] Shopify: *Sitio web de Shopify*. https://www.shopify.com/, visitado el 18 de Julio de 2022.

[13] Stephanie Chevalier: *E-commerce in Latin America - statistics & facts*. https://www.statista.com/topics/2453/e-commerce-in-latin-america/#topicHeader__wrapper, visitado el 22 de Agosto de 2022.

[14] WooCommerce: *Sitio web de WooCommerce*. `https://woocommerce.com/`, visitado el 18 de Julio de 2022.

[15] Yapo.cl: *Sitio web de Yapo.cl*. `https://new.yapo.cl/`, visitado el 22 de Agosto de 2022.