



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

Desarrollo de una aplicación web para la gestión centralizada de múltiples marketplaces

Documento:

Memoria

Autor/Autora:

Aleix Ribas Torras

Director/Directora - Codirector/Codirectora:

Francisco José Múgica Alvarez

Maria Angela Nebot Castells

Titulación:

Grado en Ingeniería en Tecnologías Aeroespaciales

Convocatoria:

Primavera, 2025

TRABAJO FIN DE ESTUDIOS

Sumario

1. Introducción	1
2. Marco Teórico	2
2.1. Comercio electrónico y canales de venta en línea	2
2.1.1. Plataformas <i>E-commerce</i>	3
2.1.2. <i>Marketplaces</i>	4
2.2. Modelos de distribución de software SaaS	6
2.3. Arquitectura y tecnologías de una aplicación web	7
2.3.1. <i>Frontend</i>	9
2.3.2. <i>Backend</i>	10
2.3.3. Base de datos	12
2.3.4. API	15
3. Diseño y Desarrollo de la plataforma	18
3.1. Proceso inicial de desarrollo de la plataforma	18
3.1.1. Separación de tecnologías <i>frontend</i> y <i>backend</i>	18
3.2. Diseño de la base de datos	19

3.2.1. Bloques de funcionalidades	20
3.3. Desarrollo del <i>backend</i>	33
3.3.1. Estructuración del proyecto	36
3.3.2. Definición de los modelos	39
3.3.3. Definición de los <i>serializers</i>	41
3.3.4. Definición de las vistas y URLs	47
3.3.5. Implementaciones relevantes	53
3.4. Desarrollo del <i>frontend</i>	55
3.4.1. Estructuración del proyecto	56
3.4.2. Desarrollo de vistas y componentes	58
4. Conclusiones	65

Índice de figuras

2.1. Modelos de distribución de software. Fuente: [3]	7
2.2. Arquitectura de una aplicación web.	8
2.3. Diagrama de flujo del patrón de diseño MVC.	12
2.4. Tablas de ejemplo de una base de datos para guardar pedidos.	14
2.5. Ejemplo de consulta SQL y su equivalente en ORM para la obtención de dos campos de un registro de una tabla.	14
2.6. Ejemplo de una petición a una API REST con la correspondiente respuesta en formato JSON.	16
2.7. Ejemplo de una respuesta a una petición sin y con <i>query parameters</i>	17
3.1. Esquema de flujo de creación de un producto y su vinculación con un canal de venta.	28
3.2. Diagrama de la base de datos	34
3.3. Estructura de un proyecto de Django.	37
3.4. Estructura del proyecto.	38
3.5. Estructura de los modelos del proyecto.	41
3.6. Funcionamiento de los <i>serializers</i>	42

3.7. Estructura de los <i>serializers</i> del proyecto.	47
3.8. Estructura de las vistas del proyecto.	51
3.9. Enrutamiento de todas las URLs del <i>backend</i> con sus correspondientes métodos de los <i>ViewSet</i> s.	52
3.10. Estructura de archivos del <i>frontend</i> de la plataforma.	58
3.11. Menú lateral de navegación de la plataforma.	59
3.12. Vista general de pedidos con algunos pedidos de ejemplo.	61
3.13. Vista detallada de un pedido concreto.	62
3.15. Modal de edición de productos de un pedido.	63
3.14. Campos editables de un pedido dependiendo de su origen.	63
3.16. Proceso de búsqueda de productos con <i>debounce</i>	64

Acrónimos

API Application Programming Interface.

CRM Customer Relationship Management.

SaaS Software as a Service.

Capítulo 1

Introducción

Capítulo 2

Marco Teórico

En este capítulo se presenta el marco teórico que sustenta el desarrollo de la aplicación web para la gestión centralizada de múltiples *marketplaces*. Se abordan conceptos clave relacionados con el comercio en línea, sus dificultades y sus múltiples vertientes. Además, se exploran las tecnologías y herramientas utilizadas en el desarrollo de la aplicación, así como las metodologías de trabajo adoptadas durante el proceso.

2.1. Comercio electrónico y canales de venta en línea

En los últimos años, el comercio electrónico se ha convertido en una parte fundamental de la economía global. La capacidad de comprar y vender productos y servicios a través de internet ha transformado la forma en que las empresas interactúan con sus clientes. Este fenómeno ha dado lugar a la aparición de nuevos canales de venta, los llamados canales de venta en línea, que permiten a las empresas llegar a un público más amplio y diversificado, donde antes dependían de tiendas físicas o distribuidores locales.

No obstante, el comercio electrónico no es tan fácil como puede parecer en primera instancia. Existen tres grandes desafíos que las empresas deben enfrentar: la competencia, la infraestructura tecnológica y logística.

En internet todo el mundo juega con las mismas reglas; las facilidades que ofrece este medio

son iguales para todos. El factor de proximidad al cliente ya no es el diferencial, sino la capacidad de ofrecer un producto o servicio que se diferencie del resto, tanto en calidad, precio o experiencia de compra. Esto genera que la competencia sea feroz, y las empresas deben encontrar formas innovadoras de destacar entre la multitud, tal como podrían ser las promociones, el marketing digital o la experiencia de usuario.

Por otro lado, el comercio electrónico requiere de una infraestructura tecnológica que permita listar productos, gestionar pedidos y pagos, y mantener una comunicación fluida con los clientes. Esto implica no solo contar con un sitio web atractivo y funcional, sino también con sistemas de gestión de inventario, plataformas de pago seguras y herramientas de análisis de datos que permitan tomar decisiones informadas. Todo esto puede resultar costoso y complicado de implementar, especialmente para pequeñas y medianas empresas que no cuentan con los recursos necesarios, ni en términos de personal, ni de dinero.

Por último, la logística es otro de los grandes retos del comercio electrónico. En el comercio tradicional, los productos se entregan directamente al cliente en la tienda. En el comercio electrónico, las empresas deben gestionar el almacenamiento, el envío y la entrega de productos a los clientes, lo que puede resultar complicado y costoso. La gestión de inventarios, la selección de proveedores de transporte y la coordinación de envíos son solo algunos de los aspectos logísticos que las empresas deben tener en cuenta para garantizar una experiencia de compra satisfactoria que cumpla con las expectativas de los clientes.

Estos tres factores son solo algunos de los muchos desafíos que enfrentan las empresas en el comercio electrónico. Por este mismo motivo, diferentes soluciones han surgido para ayudar a las empresas a superar estos obstáculos y aprovechar al máximo las oportunidades que ofrece el comercio en línea. Entre estas soluciones se encuentran dos que destacan por encima de las demás: las plataformas *e-commerce* y los *marketplaces*. Ambas ofrecen a las empresas la posibilidad de vender sus productos y servicios en línea, pero lo hacen de maneras diferentes.

2.1.1. Plataformas *E-commerce*

Diseñar y programar una tienda en línea desde cero puede ser un proceso largo y costoso. Una tienda en línea no es una simple página web, sino un sistema complejo que debe gestionar

una gran cantidad de información, como productos, precios, inventarios, pedidos y clientes.

Muchas empresas optan por utilizar las llamadas plataformas *e-commerce*. Una plataforma *e-commerce* es una aplicación que permite a las empresas crear y gestionar su propia tienda en línea en relativamente pocos pasos. Existen muchas plataformas *e-commerce*, pero todas se caracterizan por ofrecer una serie de herramientas y funcionalidades por defecto de manera que las empresas puedan crear su tienda en línea sin necesidad de tener conocimientos técnicos avanzados. Estas plataformas suelen incluir plantillas de diseño, sistemas de gestión de inventario, herramientas de marketing y análisis, y opciones de pago seguras. Además, muchas de ellas ofrecen integraciones con otros servicios y funcionalidades adicionales, todo bajo los llamados *plugins* [1].

Todo este conjunto de facilidades hacen que el uso de plataformas *e-commerce* sea una opción atractiva para muchas empresas, especialmente para aquellas que están comenzando en el comercio electrónico o que no cuentan con los recursos necesarios para desarrollar su propia tienda en línea desde cero. Sin embargo, también existen desventajas asociadas al uso de estas plataformas. Por ejemplo, las empresas pueden tener menos control sobre el diseño y la funcionalidad de su tienda en línea, y pueden estar sujetas a las políticas y tarifas de la plataforma, entre muchas otras cosas. Además, algunas plataformas pueden no ser escalables o flexibles lo suficiente como para adaptarse a las necesidades cambiantes de una empresa en crecimiento.

Existe una amplia variedad de plataformas *e-commerce* en el mercado, cada una con sus propias características y funcionalidades. Algunas de las más populares son Shopify, WooCommerce (*plugin* de WordPress), Magento y PrestaShop. Cada una de estas plataformas tiene sus propias ventajas y desventajas, y la elección de la plataforma adecuada dependerá de las necesidades, objetivos específicos y las dimensiones de cada empresa.

2.1.2. *Marketplaces*

Los *marketplaces* son plataformas en línea que permiten a las empresas vender sus productos y servicios a través de un canal de venta compartido. A diferencia de las plataformas *e-commerce*, donde las empresas crean y gestionan su propia tienda en línea, los *marketplaces*

permiten a las empresas listar sus productos y servicios junto con los de otras empresas en una única plataforma. Esto significa que las empresas pueden aprovechar la audiencia y el tráfico del *marketplace* para llegar a nuevos clientes sin necesidad de invertir en marketing o publicidad.

Aquí radica realmente la ventaja de los *marketplaces*: la posibilidad de llegar a un público más amplio y diversificado sin necesidad de invertir grandes cantidades de dinero en marketing o publicidad. Tanto en los *e-commerce* tradicionales (tiendas en línea creadas desde cero) como en las plataformas *e-commerce*, las empresas deben invertir considerables cantidades de dinero para atraer tráfico a su tienda en línea, mientras que en los *marketplaces* el tráfico ya está allí, lo que significa que las empresas pueden aprovecharlo para aumentar sus ventas y llegar a nuevos clientes.

No obstante, este tipo de plataformas también tienen sus desventajas. En primer lugar, todos los productos acostumbran a estar bajo una comisión de manera que la empresa o bien debe subir el precio de su producto o asumir la pérdida de margen. Está comisión puede rondar entre el 10 % y el 20 %. En segundo lugar, los *marketplaces* pueden ser muy competitivos, pues un mismo producto puede ser vendido por distintas empresas, dando lugar a una guerra de precios que puede afectar la rentabilidad de las empresas. Por último, en un *marketplace* la empresa no tiene ningún tipo de control sobre la experiencia de compra del cliente, lo que puede afectar la percepción de la marca y la lealtad del cliente, además de que debe someterse a la política de la plataforma, que muchas veces puede no ser beneficiosa para la empresa y puede llegar a afectar sus operaciones.

Existen múltiples *marketplaces*, tanto de productos físicos, como Amazon, eBay o AliExpress, como de productos digitales o servicios, como Udemy, Uber o Glovo. Cada uno de estos tiene sus propias características y funcionalidades, y la elección del *marketplace* adecuado dependerá de las necesidades y objetivos específicos de cada empresa. Es importante destacar que el uso de una plataforma *e-commerce* no excluye la posibilidad de listar productos o servicios en un *marketplace*. De hecho, muchas empresas optan por combinar ambas estrategias para maximizar su alcance y diversificar sus canales de venta, aprovechando las ventajas que ofrece cada una de estas opciones [2].

2.2. Modelos de distribución de software SaaS

Hay una gran variedad de modelos de distribución de software, tal como pueden ser el modelo *On-Premise*, el modelo *Infrastructure as a Service* (IaaS) o el modelo *Platform as a Service* (PaaS). Sin embargo, el modelo que más se utiliza en la actualidad es el modelo *Software as a Service* (SaaS).

Años atrás, el software se distribuía principalmente a través de licencias perpetuas, donde los usuarios compraban una licencia para utilizar el software en sus propios servidores o computadoras. Este modelo requería que los usuarios gestionaran la infraestructura y el mantenimiento del software, lo que podía ser costoso y complicado. Esto significaba que el proveedor del software simplemente facilitaba el producto y el usuario debía hacerse cargo de la instalación, configuración y mantenimiento del mismo. Esto podía resultar complicado y costoso, especialmente para pequeñas y medianas empresas que no contaban con los recursos necesarios para gestionar su propia infraestructura. Esto es conocido como una infraestructura *On-Premise*.

Con la llegada de internet y la nube, surgieron nuevos modelos de distribución de software que permitieron a las empresas ofrecer sus productos y servicios de manera más eficiente y escalable. El modelo SaaS es uno de los más populares y se basa en la idea de que el software se aloja en la nube y se accede a través de internet. Esto significa que los usuarios no necesitan instalar ni gestionar el software en sus propios servidores u ordenadores, sino que pueden acceder a él a través de un navegador web.

Sin embargo, la dependencia de la conexión a internet, la falta de control sobre la infraestructura y la seguridad de los datos son puntos críticos en el modelo. Además, los proveedores de SaaS suelen cobrar tarifas mensuales o anuales por el uso del software, pues el mantenimiento de la infraestructura y el soporte técnico son responsabilidad del proveedor.

Existen también otras alternativas, como pueden ser los modelos IaaS y PaaS. Cada uno de estos modelos tiene sus propias ventajas y desventajas, y dependiendo del tipo de negocio y las necesidades del cliente, uno puede ser más adecuado que otro. En la figura 2.1 se pueden observar los diferentes modelos de distribución de software y sus características.

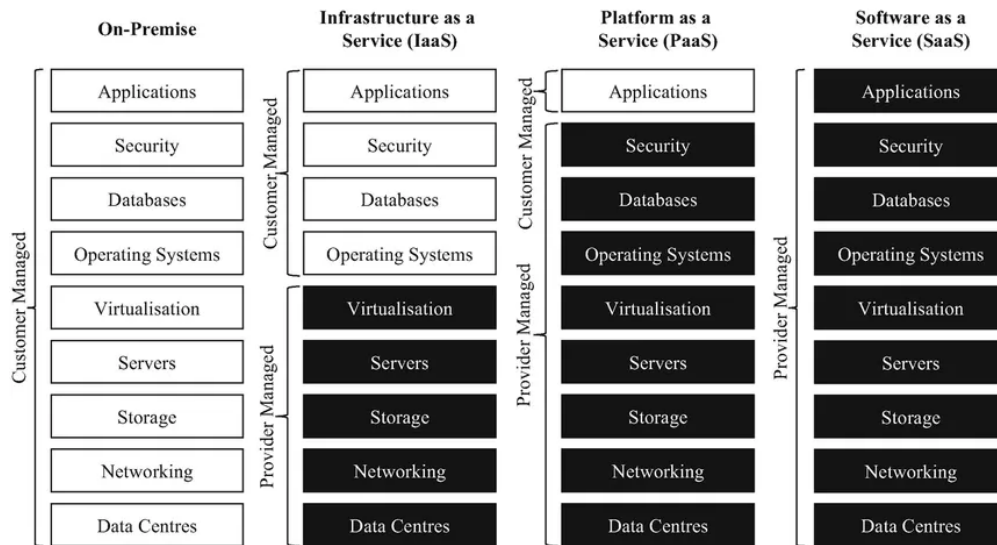


Figura 2.1: Modelos de distribución de software. Fuente: [3]

2.3. Arquitectura y tecnologías de una aplicación web

En el mundo del software existen dos tipos de aplicaciones: las aplicaciones de escritorio y las aplicaciones web. Las aplicaciones de escritorio son aquellas que se instalan en un ordenador y se ejecutan de forma local, mientras que las aplicaciones web son aquellas que se ejecutan en un servidor y se acceden a través de un navegador web.

Desde los inicios del desarrollo de software, las aplicaciones de escritorio han sido la norma. Sin embargo, en los últimos años ha habido un cambio significativo hacia el desarrollo de aplicaciones web, pues el avance de las distintas tecnologías web y la conectividad a internet han mitigado considerablemente las desventajas que anteriormente presentaban [4].

Las aplicaciones de escritorio no requieren de un servidor externo para funcionar, toda la lógica se encuentra en el ordenador del usuario y es este mismo quien lo ejecuta. Esto hace que la aplicación sea más rápida y eficiente, ya que no hay necesidad de enviar datos a través de internet. No obstante, esto también significa que el usuario debe instalar la aplicación en su ordenador y mantenerla actualizada, lo que puede ser un inconveniente [5].

Por otro lado, las aplicaciones web sí que requieren de un servidor externo para funcionar. Esto significa que el usuario no necesita instalar nada en su ordenador, ya que la aplicación se ejecuta en el servidor y se accede a través de un navegador web. Este paradigma de desarrollo permite que la aplicación sea más accesible, pues se puede acceder a la aplicación

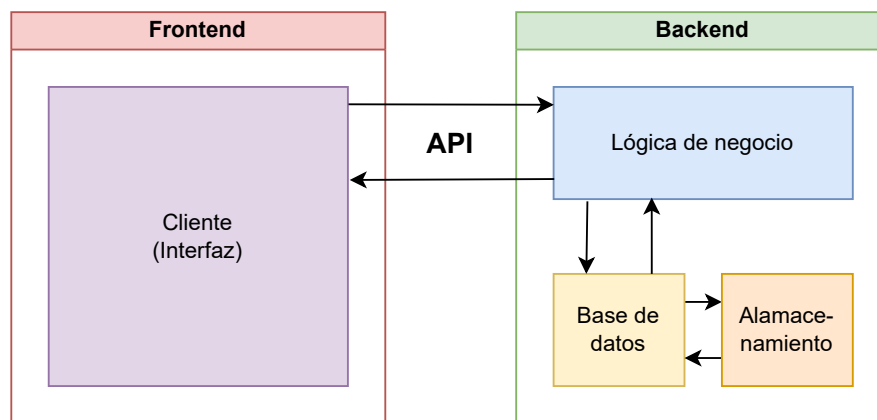


Figura 2.2: Arquitectura de una aplicación web.

desde cualquier dispositivo y lugar con conexión a internet. Además, como todo se encuentra en el servidor y no en el usuario, las actualizaciones son mucho más inmediatas. Sin embargo, esto también significa que la aplicación puede ser más lenta y menos eficiente debido a que la lógica y los datos no se encuentran en el ordenador del usuario, sino en el servidor [5].

La mejor conectividad a internet y el avance de las tecnologías web han permitido que las aplicaciones web sean cada vez más rápidas y eficientes. Esto ha llevado a un aumento en la popularidad de las aplicaciones web, y muchas empresas están optando por desarrollar aplicaciones web en lugar de aplicaciones de escritorio.

Con todo esto, se ha formado lo que se conoce como arquitectura web, que hace referencia a la forma en que se organiza y estructura el software. Esto incluye la forma en que se comunican los diferentes componentes de la aplicación, así como la forma en que se almacenan y gestionan los datos. Todas las aplicaciones web están compuestas por tres componentes principales: el cliente, la lógica de negocio y la base de datos.

Estos tres componentes se dividen en dos bloques: el *frontend* y el *backend*. El *frontend* es la parte de la aplicación que interactúa con el usuario, es decir, el cliente, mientras que el *backend* es la parte de la aplicación que se encarga de gestionar los datos y la lógica de negocio. Ambos bloques se comunican entre sí a través de una API (Interfaz de Programación de Aplicaciones), que es un conjunto de reglas y protocolos que permiten que diferentes componentes de software se comuniquen entre sí. Todo esto se puede ver en la figura 2.2.

2.3.1. *Frontend*

El *frontend* es la parte de la aplicación que interactúa con el usuario. Esto incluye la interfaz de usuario, que es la parte de la aplicación que el usuario ve y con la que interactúa, así como la lógica de presentación, que es la parte de la aplicación que se encarga de mostrar los datos al usuario. El *frontend* se desarrolla principalmente utilizando HTML, CSS y JavaScript, tecnologías que se ejecutan de manera nativa en los navegadores.

El HTML (*Hypertext Markup Language*) es el lenguaje de marcado utilizado para estructurar el contenido de una página web. El CSS (*Cascading Style Sheets*) es el lenguaje utilizado para dar estilo a una página web, es decir, para definir cómo se verá el contenido estructurado por el HTML. Por último, JavaScript es un lenguaje de programación que se utiliza para añadir interactividad a una página web, es decir, para permitir que el usuario interactúe con la aplicación.

Con todo esto, el servidor envía todo este contenido al usuario de manera que su navegador lo pueda representar, en caso del HTML y el CSS, y ejecutar, en caso del JavaScript.

Desarrollar aplicaciones de manera nativa, es decir, utilizando HTML, CSS y JavaScript sin el apoyo de ninguna librería o herramienta adicional, puede ser complicado, tedioso y poco eficiente. Para facilitar esta tarea, existen distintos *frameworks* que proporcionan un conjunto de herramientas y funcionalidades pensadas para abstraer la complejidad del desarrollo. De esta manera, los *frameworks* cumplen principalmente dos propósitos:

- **Ahorrar tiempo:** Los *frameworks* permiten al desarrollador ahorrar tiempo ofreciendo funcionalidades predefinidas a problemas comunes o recurrentes. En el caso de una aplicación web, esto puede incluir la gestión de rutas, la gestión del estado de la aplicación, la gestión de formularios, entre otros [6].
- **Garantizar buenas prácticas de desarrollo:** Los *frameworks* suelen seguir patrones de diseño y buenas prácticas de desarrollo que ayudan a los desarrolladores a escribir código limpio, mantenible y escalable. Adicionalmente ofrecen características de seguridad por defecto, de manera que el desarrollador no tiene que implementar sus propias medidas de seguridad que pueden resultar ser vulnerables. Un ejemplo es la

autenticación de usuarios, donde el *framework* se encarga de gestionar la creación y validación de los *tokens* de acceso, así como la gestión de sesiones [6].

Algunos de los *frameworks* más populares para el desarrollo de *frontend* son React, Angular y Vue.js. En el caso de este proyecto, se ha optado por utilizar React, un *framework* desarrollado por Facebook que en los últimos años se ha convertido en un estándar de la industria. React es un *framework* basado en componentes, lo que significa que la interfaz de usuario se divide en unidades independientes y reutilizables, facilitando así la creación de aplicaciones más complejas y escalables.

React se desarrolla utilizando JavaScript, un lenguaje de programación de tipado dinámico, lo que significa que no requiere de declarar el tipo de las variables al momento de crearlas. Esta característica, si bien ofrece flexibilidad, puede provocar errores difíciles de detectar. Para solventarlo, existe TypeScript, un lenguaje que extiende JavaScript añadiendo tipado estático [7]. Con TypeScript, el desarrollador puede especificar el tipo de las variables, permitiendo que los errores de tipado se detecten en tiempo de interpretación. Aunque los navegadores solo interpretan JavaScript, el código en TypeScript se transpila automáticamente a JavaScript. Por este motivo, en este proyecto se ha decidido utilizar TypeScript para mejorar la calidad y la robustez del código [8].

Finalmente, otro motivo relevante para la elección de React es su gran comunidad de desarrolladores, así como la amplia disponibilidad de librerías y herramientas que extienden sus funcionalidades básicas. La elección de React y el detalle de su funcionamiento se explican en profundidad en la sección (Añadir sección parte desarrollo frontend).

2.3.2. *Backend*

El *backend* es la parte de la aplicación que administra la funcionalidad general de la aplicación. Cuando el usuario interactúa con el *frontend*, la interacción envía una solicitud al *backend* para que la procese y devuelva una respuesta [9]. De este modo, el *backend* se encarga de gestionar la lógica de negocio, es decir, es la parte de la aplicación que se encarga de procesar los datos y realizar las operaciones necesarias para realizar las distintas funcionalidades de ésta. Adicionalmente contiene la base de datos, que es donde se almacenan todos

los datos de la aplicación.

A diferencia del *frontend*, el *backend* no se ejecuta en el navegador del usuario, sino en un servidor. Esto significa que el *backend* puede utilizar lenguajes de programación y tecnologías que no son compatibles con los navegadores, como Java, Python o Ruby. No obstante, un factor que si tiene en común con el *frontend* es la existencia de *frameworks*.

Para el desarrollo de este proyecto se ha decidido utilizar Django, un *framework* escrito en Python. Django es un *framework* un tanto especial, ya que permite desarrollar tanto el *frontend* como el *backend* de una aplicación web. Sin embargo, en este proyecto se ha optado por utilizar Django únicamente para el desarrollo del *backend*, dejando el *frontend* a cargo de React, pues se considera que es la mejor opción para el desarrollo de aplicaciones web modernas. En cuanto al aspecto más técnico, Django está basado en el modelo MVC (Modelo-Vista-Controlador), un patrón de diseño que separa la lógica en tres componentes principales: el modelo, que se encarga de gestionar los datos; la vista, que se encarga de mostrar los datos al usuario; y el controlador, que se encarga de gestionar la interacción entre el modelo y la vista [10].

El funcionamiento del modelo MVC se representa en la figura 2.3. Para entender mejor este flujo, se puede tomar como ejemplo una acción concreta dentro del proyecto: acceder a la sección de productos de la aplicación. A continuación, se explica paso a paso lo que ocurre en ese proceso:

1. **Solicitud del usuario** El usuario accede a la sección de productos desde el *frontend*, ya sea a través de un menú o directamente introduciendo una URL. Esta acción genera una solicitud HTTP que se envía a un *endpoint* del *backend* (paso 1 en la figura).
2. **Recepción por parte del controlador:** El *endpoint* está vinculado a una función del controlador, que es el encargado de procesar la solicitud. En este caso, el controlador interpreta que se necesita acceder a los datos de los productos y actúa en consecuencia (paso 2).
3. **Consulta al modelo:** El controlador solicita la información al modelo correspondiente, es decir, a los productos. El modelo representa la estructura de datos y contiene la lógica necesaria para interactuar con la base de datos de manera más sencilla y segura

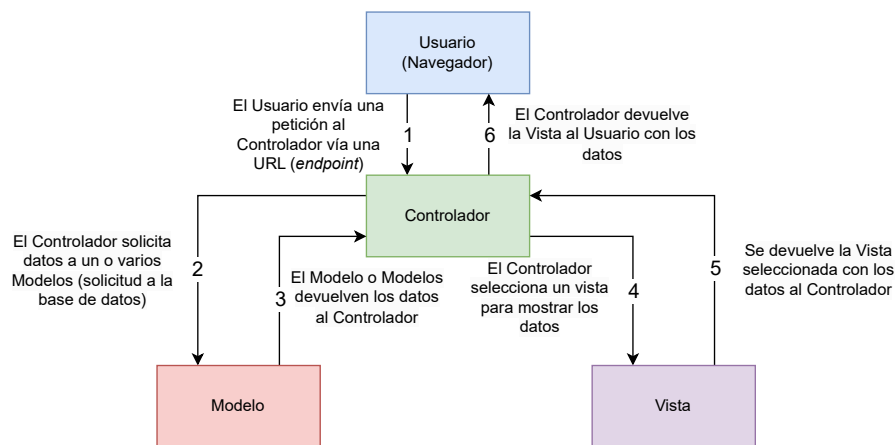


Figura 2.3: Diagrama de flujo del patrón de diseño MVC.

(paso 3).

4. **Respuesta del modelo:** El modelo realiza la consulta a la base de datos y devuelve al controlador los datos solicitados, en este caso, la lista de productos (paso 4).
5. **Selección de la vista:** Con los datos recibidos, el controlador selecciona la vista adecuada para estructurar la respuesta. Esta vista se encarga de preparar los datos en un formato que el *frontend* pueda interpretar, normalmente JSON (*JavaScript Object Notation*) (paso 5).
6. **Respuesta al usuario:** La vista estructurada en formato JSON se devuelve al controlador, que finalmente la envía como respuesta al usuario. El *frontend* recibe estos datos y se encarga de representarlos en la interfaz gráfica, mostrando al usuario la información solicitada: los productos (paso 6).

Una mayor profundidad sobre el funcionamiento de los modelos y la base de datos se puede encontrar en la sección 2.3.3 y sobre la comunicación entre el *frontend* y el *backend* en la sección 2.3.4.

2.3.3. Base de datos

La base de datos es una colección de datos electrónicamente almacenados y organizados de manera que se puedan acceder, gestionar y actualizar fácilmente. En el caso de una

aplicación web, la base de datos se utiliza para almacenar toda la información necesaria para el funcionamiento de ésta [11].

Existen distintos tipos de bases de datos, pero las más comunes son las bases de datos relacionales y las bases de datos no relacionales. Las bases de datos relacionales almacenan los datos en tablas, donde cada fila representa un registro y cada columna representa un campo. Este tipo de base de datos es ideal para aplicaciones que requieren una estructura de datos rígida y bien definida. Por otro lado, las bases de datos no relacionales almacenan los datos en formatos más flexibles, como documentos o pares clave-valor. Este tipo de base de datos es ideal para aplicaciones que requieren una estructura de datos más flexible y escalable [12].

En este proyecto se ha optado por utilizar una base de datos relacional, pues la mayoría de los datos que se gestionan son estructurados y requieren una relación entre ellos. Así pues, se ha decidido usar PostgreSQL, un sistema de gestión de bases de datos relacional de código abierto que funciona de manera nativa con Django.

Con el tipo de base de datos definido, es fundamental comprender el funcionamiento básico de una base de datos relacional. Este tipo de base de datos organiza la información en tablas, cada una con un nombre específico. Las tablas están compuestas por un conjunto fijo de columnas, que representan los campos o atributos de los datos, y un conjunto variable de filas, donde cada una corresponde a un registro u objeto dentro de la tabla. Cada fila, es decir, cada registro, tiene un identificador único conocido como clave primaria, que permite distinguirlo de los demás registros de la tabla y relacionarlo con otras tablas a partir de claves foráneas. Las claves foráneas son columnas que establecen una relación entre dos o más tablas, permitiendo que los datos de una tabla se vinculen con los datos de otras. Esto es especialmente útil para representar relaciones entre diferentes entidades dentro de la base de datos, como por ejemplo, la relación entre un pedido, el cliente que lo ha realizado y sus productos asociados, tal como se muestra en la figura 2.4.

Para crear, modificar, eliminar y consultar los datos de una base de datos relacional, se utiliza SQL (*Structured Query Language*), un lenguaje de programación diseñado específicamente para gestionar bases de datos. SQL permite realizar operaciones como la creación de tablas, la inserción de datos, la actualización de registros y la consulta de información

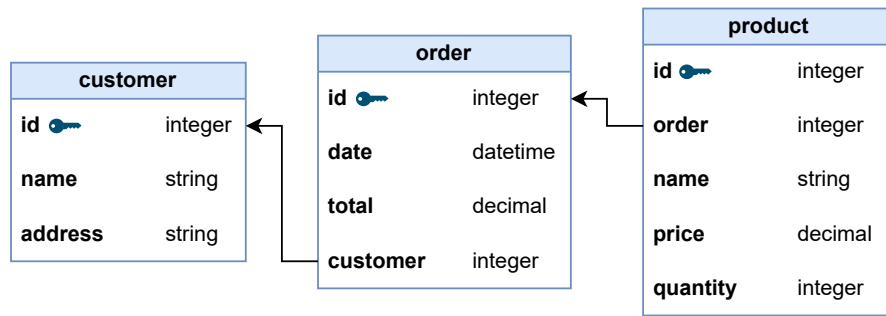


Figura 2.4: Tablas de ejemplo de una base de datos para guardar pedidos.

[13]. No obstante, realizar consultas SQL directamente puede ser complicado y propenso a errores, especialmente en aplicaciones más complejas. **Explicar fallos seguridad, queries mal optimizadas...** Por este motivo, Django ofrece una capa de abstracción llamado sistema ORM (*Object-Relational Mapping*), que permite interactuar con la base de datos utilizando objetos y clases de Python en lugar de escribir consultas SQL directamente. De esta manera, con ORM cada tabla de la base de datos es representada por lo que se llama un modelo, que es una clase de Python que define la estructura de la tabla y sus relaciones con otras tablas. Cada instancia de un modelo representa una fila en la tabla correspondiente, y los atributos de la clase representan las columnas de la tabla. Este enfoque se puede ver ejemplificado en la figura 2.5, donde se muestra una consulta SQL y su equivalente en ORM para obtener los campos `total` y `date` del registro que tiene `customer = 7` de la tabla `order`.

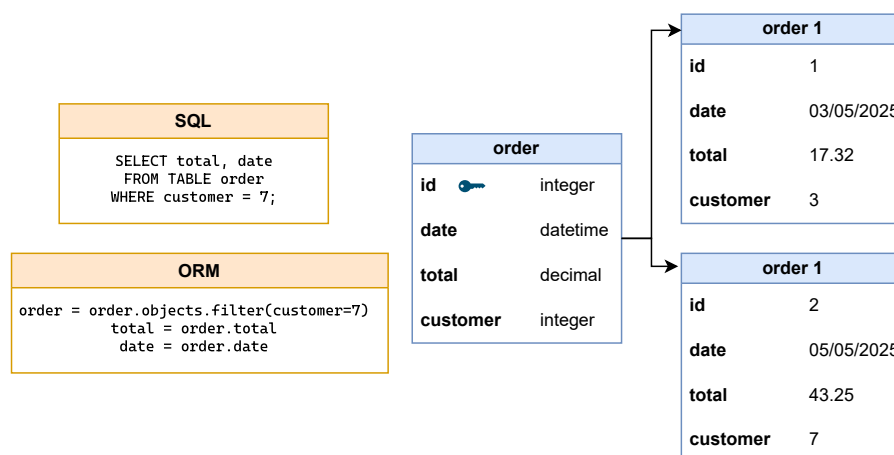


Figura 2.5: Ejemplo de consulta SQL y su equivalente en ORM para la obtención de dos campos de un registro de una tabla.

2.3.4. API

La API (*Application Programming Interface*) es un conjunto de mecanismos que permiten la comunicación entre diferentes componentes de software mediante unas definiciones y unos protocolos. En el caso de una aplicación web, la API es la interfaz que permite al *frontend* comunicarse con el *backend* y viceversa. Esto significa que cuando el usuario interactúa con el *frontend* se envían solicitudes a la API (al *backend*), que las procesa y devuelve una respuesta. No obstante, las API no solo permiten la comunicación entre el *frontend* y el *backend*, sino que también permiten la comunicación entre diferentes aplicaciones. Un ejemplo que aplica a este proyecto es la comunicación entre el *backend* de la aplicación y un *marketplace*, donde se emplea una API. Para obtener un pedido de un *marketplace*, el *backend* debe enviar una solicitud a la API del *marketplace* y recibe una respuesta con la información del pedido [14].

La arquitectura de una API se entiende en términos de cliente y servidor, de manera que la aplicación que envía la solicitud se llama cliente y la aplicación que recibe y procesa la solicitud se llama servidor. En el ejemplo anterior, el *backend* actúa como cliente y el *marketplace* actúa como servidor.

Sin embargo, las API pueden ser de distintos tipos, dependiendo de cómo se estructuren y cómo se comuniquen. En el caso de este proyecto, se ha optado por utilizar una API REST (*Representational State Transfer*), que son las más comunes en la actualidad. Una API REST es un tipo de API que utiliza el protocolo HTTP para la comunicación entre el cliente y el servidor, lo que significa que las solicitudes y respuestas se envían a través de HTTP, utilizando principalmente los siguientes métodos para realizar operaciones sobre los recursos:

- **GET:** Se utiliza para obtener información de un recurso. Por ejemplo, si se quiere obtener la lista de productos de la aplicación, se enviaría una solicitud GET a la API con la URL correspondiente.
- **POST:** Se utiliza para crear un nuevo recurso. Por ejemplo, si se quiere crear un nuevo producto, se enviaría una solicitud POST a la API con la información del producto en el cuerpo de la solicitud.
- **PUT:** Se utiliza para actualizar un recurso existente. Por ejemplo, si se quiere actua-

lizar la información de un producto, se enviaría una solicitud PUT a la API con la información actualizada en el cuerpo de la solicitud.

- **PATCH:** Se utiliza para actualizar parcialmente un recurso existente. Por ejemplo, si se quiere actualizar solo el precio de un producto, se enviaría una solicitud PATCH a la API con la información actualizada en el cuerpo de la solicitud.
- **DELETE:** Se utiliza para eliminar un recurso. Por ejemplo, si se quiere eliminar un producto, se enviaría una solicitud DELETE a la API con la URL del producto a eliminar.

Tal como se puede observar en los ejemplos de cada uno de los métodos, la API REST utiliza URLs para identificar los recursos. Cada recurso tiene una URL única que se utiliza para acceder a él. Por ejemplo, la URL para acceder a la lista de productos podría ser `https://api.ejemplo.com/products`, mientras que la URL para acceder a un producto específico podría ser `https://api.ejemplo.com/products/1`, donde el número 1 representa el identificador único del producto. Algunos métodos, como pueden ser el POST y el PATCH incluyen un *body*, que es el cuerpo de la solicitud, donde se encuentran los datos que se envían en la petición.

Tanto el *body* como la respuesta de la API acostumbran a estar en formato JSON, un formato basado en texto estructurado de manera que permite representar datos de manera sencilla y legible. En la figura 2.6 se puede ver un ejemplo de una petición a una API REST y la respuesta en formato JSON. En este caso, se está realizando una petición GET a la API para obtener el producto que tiene `id = 1`, y la respuesta es un objeto JSON que contiene la información de dicho producto.

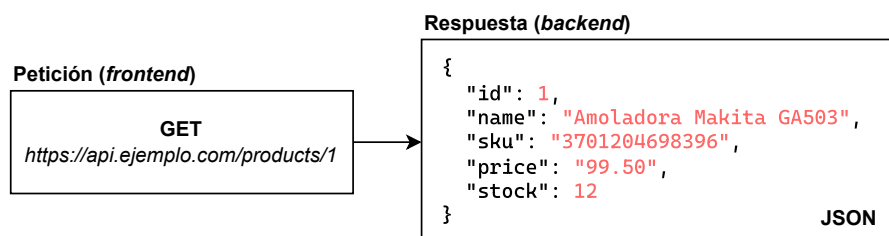


Figura 2.6: Ejemplo de una petición a una API REST con la correspondiente respuesta en formato JSON.

Por último, es importante destacar los llamados *query parameters*, que son parámetros que se pueden añadir a la URL para filtrar o modificar la respuesta de la API. Por ejemplo, si se quiere obtener solo los productos que tienen stock, se podría añadir un parámetro a la URL como `?stock=true`. Esto permite que la API devuelva solo los productos que cumplen con ese criterio, lo que puede ser útil para optimizar las consultas y reducir la cantidad de datos transferidos. De esta manera, para obtener los productos que están en oferta, la URL de la API podría ser `https://api.ejemplo.com/products?stock=true`. Esto puede verse en la figura 2.7, donde se muestra un ejemplo de una respuesta a una petición sin y con *query parameters*. En este caso, la primera respuesta devuelve todos los productos, mientras que la segunda respuesta devuelve solo los productos que tienen un stock superior a 0.

GET <code>https://api.ejemplo.com/products</code>	GET <code>https://api.ejemplo.com/products?stock=true</code>
<pre>[{ "id": 1, "name": "Amoladora Makita GA503", "sku": "3701204698396", "price": "99.50", "stock": 12 }, { "id": 2, "name": "Taladro Bosch GSB 13 RE", "sku": "3165140840430", "price": "79.90", "stock": 0 }, { "id": 3, "name": "Sierra Circular DeWalt DWE560", "sku": "5035048290184", "price": "129.00", "stock": 5 }]</pre> <p>JSON</p>	<pre>[{ "id": 1, "name": "Amoladora Makita GA503", "sku": "3701204698396", "price": "99.50", "stock": 12 }, { "id": 3, "name": "Sierra Circular DeWalt DWE560", "sku": "5035048290184", "price": "129.00", "stock": 5 }]</pre> <p>JSON</p>

Figura 2.7: Ejemplo de una respuesta a una petición sin y con *query parameters*.

Capítulo 3

Diseño y Desarrollo de la plataforma

En este capítulo se explicará todo el proceso de diseño y desarrollo de la plataforma, dando especial énfasis a la justificación de las decisiones tomadas y a la explicación de los distintos problemas que se han ido encontrando a lo largo del proceso. En concreto, se detallará el diseño de la base de datos, el desarrollo del *backend* y el desarrollo del *frontend*. No obstante, antes de entrar en detalle en cada una de estas secciones, se explicará el proceso inicial de desarrollo de la plataforma y se justificarán las tecnologías elegidas, cumplimentando así la sección 2.3 del capítulo 2.

3.1. Proceso inicial de desarrollo de la plataforma

El desarrollo de la aplicación web no surge de simplemente decidir qué tecnologías se van a utilizar y empezar a programar. Antes de comenzar a desarrollar la plataforma se ha llevado a cabo un proceso de diseño que ha permitido definir la arquitectura del sistema, las tecnologías a utilizar y el flujo de trabajo.

3.1.1. Separación de tecnologías *frontend* y *backend*

El primer paso que se ha realizado y una vez ya definido el objetivo de la aplicación y las funcionalidades que se querían implementar, se ha llevado a cabo un análisis de como

estructurar la plataforma. Como ya se ha comentado en la sección 2.3, se ha optado por una arquitectura dividida en dos partes: el *backend*, incluyendo la base de datos, y el *frontend*. No obstante, a pesar de que Django ofrece la posibilidad de crear ambas partes, se ha decidido utilizar React. Esta decisión ha supuesto un reto, ya que ha significado realizar un *frontend* entero además de preparar una API en el *backend* para que ambos se puedan comunicar. Sin embargo, esta decisión ha permitido crear una aplicación más escalable, flexible y, sobre todo, dinámica.

Django es un *framework* que funciona del lado del servidor, lo que significa que cada vez que se quiere mostrar una página distinta, el servidor tiene que procesar la petición y devolver la página completa. De esta manera, cuando el usuario cambia de página, el servidor carga todos los recursos (HTML, CSS y JavaScript) y los rellena con los datos necesarios, sirviendo una página estática. Por el contrario, React es un *framework* que funciona del lado del cliente, lo que significa que el servidor solo tiene que enviar los datos necesarios y el cliente se encarga de mostrar la información. Con esto, el servidor solo tiene que enviar los datos necesarios y el cliente se encarga de mostrar la información. Esto permite crear aplicaciones más dinámicas y rápidas, ya que no es necesario recargar la página cada vez que se quiere mostrar un nuevo contenido.

Este enfoque, a pesar de ser más complejo, es el estándar en la actualidad y es por este motivo que se ha optado por esta división de tecnologías.

3.2. Diseño de la base de datos

Para estructurar el proyecto, se ha optado por empezar definiendo la base de datos. Diseñar la base de datos inicialmente permite tener una visión general de como se va a estructurar el proyecto y como sus distintas partes se van a relacionar entre sí. Al fin y al cabo, la base de datos es el núcleo de la aplicación, ya que de ella dependen todas las funcionalidades.

En concreto, para el proyecto se ha decidido hacer uso del sistema de gestión de bases de datos PostgreSQL. Esta elección es resultado de la experiencia previa que he tenido con este sistema, ya que he trabajado con él en proyectos anteriores y me he familiarizado con su funcionamiento. Además, PostgreSQL es un sistema de código abierto, lo que significa que

es gratuito y se puede utilizar sin restricciones, además de ser muy robusto y escalable, lo que lo hace ideal para aplicaciones de gran tamaño, como podría ser esta en un futuro. Por último, es el sistema más recomendado por Django, lo que facilita la integración entre ambos.

3.2.1. Bloques de funcionalidades

Antes de empezar a diseñar la base de datos, se deben definir los bloques claves de la aplicación para así poder estructurar los datos de manera que se puedan implementar de la mejor manera posible. En este caso, los bloques claves son los siguientes:

- **Bloque de canales:** La herramienta debe permitir la gestión de los distintos canales de venta en línea. Aunque el usuario no crea ni edita de manera directa los canales de venta, interactúa con ellos, de manera que su información debe estar almacenada en la base de datos.
- **Bloque de pedidos:** La herramienta debe centralizar todos los pedidos de los distintos canales de venta en línea y permitir la gestión de los mismos. Esto incluye la posibilidad de crear, editar y eliminar pedidos, así como la posibilidad de marcar un pedido como enviado o entregado.
- **Bloque de productos:** La herramienta debe permitir la gestión de los productos disponibles en los distintos canales de venta en línea. Esto incluye la posibilidad de crear, editar y eliminar productos de los distintos canales, así como la posibilidad de editar sus atributos, tales como el precio, la descripción y la imagen, entre muchos otros.
- **Bloque de usuarios y autenticación:** La herramienta debe permitir la gestión de los usuarios que pueden acceder a la aplicación. Esto incluye la posibilidad de crear, editar y eliminar usuarios, así como la posibilidad de asignarles distintos permisos y roles dentro de la aplicación.

Con los bloques clave definidos, se puede concluir que la base de datos debe contener cuatro tablas principales: una para los pedidos, otra para los productos, otra para los canales y una

última para los usuarios. A partir de aquí, se pueden definir las distintas tablas que van a complementar las principales.

Adicionalmente, dado que para este proyecto la aplicación se presenta como una fase inicial de una desarrollo mucho más grande, que en un futuro podría convertirse en una solución comercializable, es indispensable tener en cuenta la escalabilidad de toda la aplicación. Por este motivo, los cuatro bloques que a continuación se detallarán han sido diseñados de manera que se puedan ampliar en un futuro sin necesidad de realizar cambios significativos en la base de datos.

Bloque de canales de venta

El bloque de canales de venta es el conjunto de tablas y relaciones que almacenan toda la información correspondiente a los distintos canales de venta en línea. El usuario no interactúa directamente con los canales de venta, es decir, no los crea ni los edita, pero sí que interactúa con ellos al sincronizarlos con sus pedidos y productos.

Este bloque es el más claro ejemplo de la escalabilidad de la aplicación y de la base de datos, ya que trata a cada canal de venta como una entidad independiente, lo que permite añadir nuevos canales sin necesidad de realizar cambios a la estructura de la base de datos.

Este bloque está formado únicamente por una tabla, que es la siguiente:

- **Canal de venta [marketplace]:** Esta tabla almacena la información de los distintos canales de venta en línea. Los campos que contiene son los siguientes:
 - **id:** Identificador único del canal de venta. *Clave primaria (entero).*
 - **name:** Nombre del canal de venta. *Cadena de caracteres.*
 - **logo_url:** URL del logo del canal de venta. *Cadena de caracteres.*
 - **color:** Color del canal de venta. *Cadena de caracteres.*
 - **country:** País del canal de venta. *Entero.*

Con la tabla definida, se puede describir el funcionamiento del bloque. El desarrollador de la aplicación puede añadir nuevos canales de venta añadiendo nuevas filas a la tabla.

Este enfoque permite que nuevos canales de venta puedan ser añadidos sin necesidad de hacer grandes cambios en la aplicación, pues cada canal de venta es tratado como una entidad independiente. Al añadir un nuevo canal de venta el usuario verá el nuevo canal en la aplicación tal como si fuera uno ya existente, pudiendo sincronizar sus pedidos y productos con él.

Bloque de pedidos

El bloque de pedidos es el conjunto de tablas y relaciones que almacenan toda la información correspondiente a los pedidos. En cada pedido es importante almacenar la información de éste, como el estado, la fecha, el método de pago, el canal de venta, entre otros. Además, para saber donde se debe enviar el pedido, es importante almacenar la información del cliente, como su nombre, dirección y teléfono. Por último, también es importante almacenar la información de los productos que componen el pedido, como su nombre, precio y cantidad solicitada.

Conociendo la información que se debe almacenar, se pueden definir las siguientes tablas:

- **Pedido [order]:** Esta tabla almacena la información general de cada pedido. Los campos que contiene son los siguientes:
 - **id:** Identificador único del pedido. *Clave primaria (entero).*
 - **order_id:** Identificador del pedido en el canal de venta. *Cadena de caracteres.*
 - **status:** Estado del pedido (pendiente, enviado, entregado, cancelado). *Entero.*
 - **order_date:** Fecha en la que se realizó el pedido. *Fecha y hora.*
 - **total_price:** Precio total del pedido. *Decimal.*
 - **ticket:** Número de ticket del pedido. *Cadena de caracteres.*
 - **ticket_refund:** Número de ticket de la devolución del pedido. *Cadena de caracteres.*
 - **pay_method:** Método de pago del pedido (tarjeta, transferencia, efectivo). *Entero.*
 - **package_quantity:** Cantidad de bultos (paquetes) del pedido. *Entero.*

- **weight:** Peso del pedido. *Decimal*.
 - **notes:** Notas del pedido. *Cadena de caracteres*.
 - **origin:** Origen del pedido (creado automáticamente, importado, manual). *Entero*.
 - **updated_at:** Fecha de la última actualización del pedido. *Fecha y hora*.
 - **carrier_id:** Identificador del transportista del pedido. *Entero y relación N:1 con la tabla carrier*.
 - **customer_id:** Identificador del cliente del pedido. *Entero y relación N:1 con la tabla customer*.
 - **marketplace_id:** Identificador del canal de venta del pedido. *Entero y relación N:1 con la tabla marketplace*.
- **Cliente [customer]:** Esta tabla almacena la información del cliente. Se divide entre información de facturación e información de envío, ya que es habitual que los canales de venta permitan añadir ambos tipos de información. Los campos que contiene son los siguientes:
- **id:** Identificador único del cliente. *Clave primaria (entero)*.
 - **bill_phone:** Teléfono de facturación. *Cadena de caracteres*.
 - **bill_email:** Correo electrónico de facturación. *Cadena de caracteres*.
 - **bill_firstname:** Nombre del cliente para la facturación. *Cadena de caracteres*.
 - **bill_lastname:** Apellido del cliente para la facturación. *Cadena de caracteres*.
 - **bill_company:** Empresa del cliente para la facturación. *Cadena de caracteres*.
 - **bill_address:** Dirección de facturación. *Cadena de caracteres*.
 - **bill_city:** Ciudad de facturación. *Cadena de caracteres*.
 - **bill_zipcode:** Código postal de facturación. *Cadena de caracteres*.
 - **bill_country:** País de facturación. *Entero*.
 - **ship_phone:** Teléfono de envío. *Cadena de caracteres*.
 - **ship_email:** Correo electrónico de envío. *Cadena de caracteres*.
 - **ship_firstname:** Nombre del cliente para el envío. *Cadena de caracteres*.

- `ship_lastname`: Apellido del cliente para el envío. *Cadena de caracteres.*
 - `ship_company`: Empresa del cliente para el envío. *Cadena de caracteres.*
 - `ship_address`: Dirección de envío. *Cadena de caracteres.*
 - `ship_city`: Ciudad de envío. *Cadena de caracteres.*
 - `ship_zipcode`: Código postal de envío. *Cadena de caracteres.*
 - `ship_country`: País de envío. *Entero pequeño.*
- **Artículo** [`orderitem`]: Esta tabla almacena la información de los productos que componen el pedido. Los campos que contiene son los siguientes:
- `id`: Identificador único del artículo. *Clave primaria (entero).*
 - `order_id`: Identificador del pedido al que pertenece el artículo. *Entero y relación N:1 con la tabla `order`.*
 - `marketplace_product_id`: Identificador del producto del artículo. *Entero y relación N:1 con la tabla `marketplace_product`.*
 - `purchase_price`: Precio del producto en el momento que se adquirió. *Decimal.*
 - `quantity`: Cantidad solicitada del producto del artículo. *Entero.*
- **Transportista** [`carrier`]: Esta tabla almacena la información de los transportistas. Los campos que contiene son los siguientes:
- `id`: Identificador único del transportista. *Clave primaria (entero).*
 - `name`: Nombre del transportista. *Cadena de caracteres.*

Con esto, se puede observar que el bloque está formado por una tabla principal, la de pedidos `order`, que almacena la información general de cada pedido, y tres tablas complementarias: la de clientes `customer`, que almacena la información del cliente, la de artículos `orderitem`, que almacena la información de los productos que componen el pedido, y la de transportistas `carrier`, que almacena la información de los transportistas.

Si bien es cierto que la tabla de cliente podría parecer ser redundante, pues toda la información del cliente `customer` podría almacenarse directamente en la tabla de pedidos, se ha optado por crear una tabla independiente para poder reutilizar la información del cliente en

otros pedidos. De esta manera, si un cliente realiza varios pedidos, su información solo se almacena una vez, lo que permite reducir el espacio de almacenamiento y mejorar la eficiencia de la base de datos, además de permitir hacer filtros, búsquedas y análisis de datos más eficientes; opciones que podrían ser muy útiles en un futuro si se quiere implementar una funcionalidad de análisis de datos.

Por otro lado, la tabla de artículos **orderitem** no almacena la información del producto, sino que relaciona el pedido con el producto del canal de venta. Esta práctica permite, al igual que con la tabla de clientes, reutilizar la información del producto en otros pedidos, lo que reduce el espacio de almacenamiento y mejora la eficiencia de la base de datos. Además, este enfoque permite que en un futuro se puedan hacer herramientas de análisis de datos, ya que se puede relacionar el pedido con el producto del canal de venta y obtener información sobre las ventas de cada producto. El único campo que almacena información del producto es el precio de compra, ya que es importante conocer el precio al que se adquirió el producto en el momento de la compra al ser un dato esencial muy cambiante.

Por último, la tabla de transportista **carrier** tiene un funcionamiento similar al de la tabla de canales de venta, ya que cada transportista es tratado como una entidad independiente. Esto permite al desarrollador añadir nuevos transportistas sin necesidad de hacer grandes cambios en la aplicación, y al usuario ver los nuevos transportistas en la aplicación tal como si fueran uno ya existente, pudiendo asignarlos a los pedidos.

De esta manera, al generarse un nuevo pedido se añadiría una nueva fila a la tabla de pedidos **order**, una nueva fila a la tabla de clientes **customer** (si el cliente no existe ya) y tantas filas a la tabla de artículos **orderitem** como tipos de productos que el cliente haya adquirido.

Bloque de productos

El bloque de productos es el conjunto de tablas y relaciones que conforman y almacenan toda la información correspondiente a los productos. Es sin duda el bloque más complejo de la aplicación, ya que cada canal de venta puede tener sus propios productos y cada producto puede tener sus propios atributos o propiedades.

Así pues, primero de todo se debe entender como se puede relacionar un producto con un

canal de venta y como se pueden gestionar los distintos atributos de cada uno de ellos para que la aplicación pueda ser lo más escalable y flexible posible. Para ello, se ha optado por crear las tablas siguientes:

- **Producto [product]:** Esta tabla almacena la información general de cada producto.

Los campos que contiene son los siguientes:

- **id:** Identificador único del producto. *Clave primaria (entero).*
- **name:** Nombre del producto. *Cadena de caracteres.*
- **sku:** Código SKU del producto. *Cadena de caracteres.*
- **reference:** Referencia del producto. *Cadena de caracteres.*
- **price:** Precio del producto. *Decimal.*
- **stock:** Cantidad de stock del producto. *Entero.*
- **parent_id:** Identificador del producto padre, en caso de que el producto sea una variante de otro producto. Si el producto es padre o no tiene hijos, el campo está vacío. *Entero y relación N:1 con la misma tabla product.*
- **image:** URL de la imagen del producto. *Cadena de caracteres.*

Como se puede observar, el bloque de productos se puede dividir en dos capas: una primera capa (capa **product**) que almacena la información del producto, sin entrar a una vinculación con ningún canal de venta, y una segunda capa (capa **marketplaceproduct**) que relaciona el producto con el canal de venta. Ambas capas son bastante similares, ya que tienen una tabla principal que almacena el producto en sí, una tabla complementaria que almacena los tipos de atributos que se pueden asignar a los productos y otra tabla adicional que almacena los valores de los atributos de cada producto. *De esta manera, a partir de ahora, los productos vinculados a un canal de venta van a ser llamados **marketplace products** y los productos sin vinculación a ningún canal de venta van a ser llamados **products**.*

Un producto **product** puede tener distintos tipos de atributos, como el color, la talla, el peso, etc. Por este motivo, se ha optado por crear una tabla de tipos de atributos **productattributetype** que almacena los distintos tipos de atributos que se pueden asignar

a los productos. Con esto, el desarrollador puede añadir nuevos tipos de atributos sin necesidad de hacer cambios en la lógica de la aplicación. No obstante, una vez se ha definido el tipo de atributo, se debe asignar un valor a este tipo de atributo para cada producto. Por este motivo, se ha creado una tabla de valores de atributos `productattribute` que almacena los valores de los atributos de cada producto. Así pues, para cada capa habrá un total de tres tablas: una para los productos, otra para los tipos de atributos y una última para los valores de los atributos.

Para vincular el producto con el canal de venta, se ha creado una tabla de los productos del canal de venta `marketplaceproduct`. Esta tabla tiene una relación N:1 con la tabla de productos `product`, lo que significa que un producto puede estar vinculado a varios productos de canales de venta, pero un producto de canal de venta solo puede tener un único producto. De esta manera, se permite que un producto pueda ser vendido en varios canales de venta sin necesidad de duplicar la información del producto. Adicionalmente y de manera análoga a la primera capa, como cada canal de venta puede tener sus propios atributos, se ha creado una tabla de tipos de atributos del canal de venta `marketplaceproductattributetype` que almacena los distintos tipos de atributos que se pueden asignar a los productos del canal de venta. Consecuentemente, también se ha creado una tabla de valores de atributos del canal de venta `marketplaceproductattribute` que almacena los valores de los atributos de cada producto del canal de venta.

De esta manera, tanto los atributos de los productos como los atributos de los productos del canal de venta forman con sus respectivas tablas de productos una relación N:M, lo que significa que un producto puede tener varios atributos y un atributo puede pertenecer a varios productos, sin la posibilidad de dos tipos de atributos iguales puedan existir en un mismo producto.

Adicionalmente, cabe remarcar el motivo por el que se separan los atributos de los productos y los atributos de los productos del canal de venta. Cada canal de venta define sus propios atributos para los productos, por lo que es necesario diferenciarlos. Por ejemplo, un canal de venta puede tener un atributo de color y otro de talla, mientras que otro canal de venta puede tener un atributo de peso y otro de dimensiones. Además, cada canal de venta puede tener sus propios valores para los mismos atributos, como por ejemplo el color rojo o el color

azul. Por este motivo, se ha optado por crear dos capas diferenciadas para los productos y los productos del canal de venta.

Con esta estructura discutida, en el esquema 3.1 se puede observar un caso de uso para lograr un mayor entendimiento de como funcionaría el flujo de creación de un producto y su vinculación con un canal de venta. En concreto, el esquema muestra como se crea un producto y se vincula a un canal de venta, añadiendo los distintos tipos de atributos.

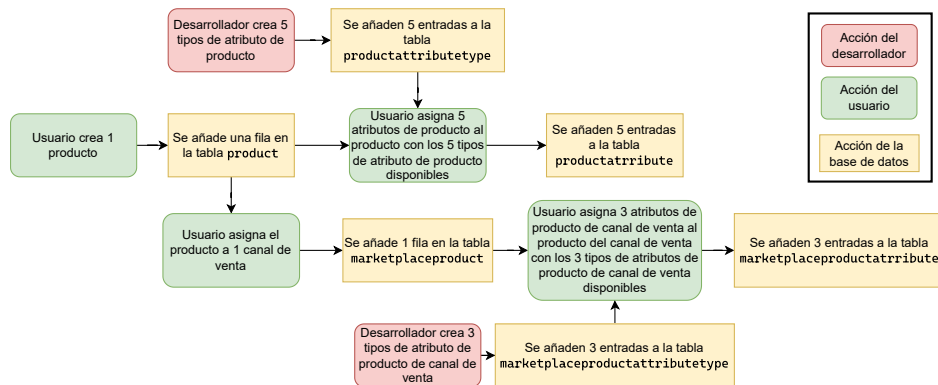


Figura 3.1: Esquema de flujo de creación de un producto y su vinculación con un canal de venta.

Por último, en las tablas de atributos, tanto de productos como de productos del canal de venta, se ha decidido hacer una columna para cada tipo de valor, es decir, una columna por si el valor es un número, una cadena de caracteres, una fecha, etc. Dichas columnas van acompañadas otra columna, llamada **data_type** que indica el tipo de valor, de manera que se puede saber que columna contiene el valor del atributo. El tipo de atributo también tiene una columna **data_type**, de manera que deben ser ambos coincidentes. Así pues, a modo de ejemplo, si el tipo de atributo es "Talla", el **data_type** será "Número" su entrada tendrá todas las columnas vacías exceptuando la columna **data_int**.

Bloque de usuarios y autenticación

El bloque de usuarios y autenticación es el conjunto de tablas y relaciones que almacenan toda la información correspondiente a los usuarios que pueden acceder a la aplicación. Este bloque es esencial para la gestión de la aplicación, ya que permite controlar quién puede acceder a la aplicación y qué permisos tiene cada usuario. Un usuario no debe poder ver ni

interactuar con los pedidos y productos de otros usuarios, por lo que es necesario implementar un sistema de autenticación y autorización que permita controlar el acceso a la aplicación y mostrar únicamente la información que le corresponde a cada uno.

Hacer un buen sistema de usuarios o autenticación es normalmente una tarea muy compleja, ya que se deben tener en cuenta muchos aspectos, como la seguridad, la privacidad y los permisos, entre muchas otras cosas. Por este motivo, es una práctica bastante común y recomendable utilizar un sistema de gestión de usuarios ya existente e integrarlo y adaptarlo a las necesidades de la aplicación. Al considerarse una buena práctica, al menos para aplicaciones de tamaño pequeño o mediano, existen múltiples sistemas, como pueden ser Auth0, Firebase Authentication o Amazon Cognito. Sin embargo, como en este proyecto se ha optado por utilizar Django, se ha decidido utilizar su propio sistema de gestión de usuarios que este ofrece nativamente, ya que es un sistema robusto, seguro y fácil de integrar. Si bien es cierto que no es un sistema tan flexible como los mencionados anteriormente, es más que suficiente para las necesidades de la aplicación, al menos en un principio, y permite una integración rápida y sencilla con el *backend*.

Las funcionalidades que ofrece el sistema de autenticación de Django son las siguientes:

- Registro de usuarios: Permite a los usuarios registrarse en la aplicación, creando una cuenta con un nombre de usuario y una contraseña.
- Inicio de sesión y cierre de sesión: Permite a los usuarios iniciar sesión en la aplicación con su nombre de usuario y contraseña, y cerrar sesión cuando lo deseen.
- Cambios y recuperación de contraseñas: Permite a los usuarios cambiar su contraseña y recuperar su cuenta en caso de que la hayan olvidado.
- Gestión de permisos y grupos de usuarios: Permite asignar permisos y roles a los usuarios, de manera que se pueda controlar qué funcionalidades pueden utilizar y qué datos pueden ver. Esto es especialmente útil para aplicaciones que tienen distintos tipos de usuarios, como administradores, clientes o empleados.
- Control de acceso: Permite controlar el acceso a las distintas funcionalidades de la aplicación, de manera que solo los usuarios autorizados puedan acceder a ellas.

Con las funcionalidades definidas, se pueden estudiar las tablas que Django va a utilizar para almacenar toda la información que permita realizar estas funcionalidades. En concreto, el sistema de autenticación hace uso las siguientes tablas:

- **Usuario [auth_user]:** Esta tabla almacena la información de los usuarios. Los campos que contiene son los siguientes:
 - **id:** Identificador único del usuario. *Clave primaria (entero).*
 - **username:** Nombre de usuario del usuario. *Cadena de caracteres.*
 - **password:** Contraseña del usuario. *Cadena de caracteres.*
 - **email:** Correo electrónico del usuario. *Cadena de caracteres.*
 - **first_name:** Nombre del usuario. *Cadena de caracteres.*
 - **last_name:** Apellido del usuario. *Cadena de caracteres.*
 - **is_active:** Indica si el usuario está activo o no. *Booleano.*
 - **is_staff:** Indica si el usuario es un miembro del personal o no. Esto permite que el usuario pueda acceder al panel de administración de Django. *Booleano.*
 - **is_superuser:** Indica si el usuario es un superusuario o no. Esto permite que el usuario tenga todos los permisos y pueda acceder a todas las funcionalidades de la aplicación. *Booleano.*
 - **last_login:** Fecha y hora del último inicio de sesión del usuario. Este campo es útil para saber cuándo fue la última vez que el usuario accedió a la aplicación. Si el usuario nunca ha iniciado sesión, este campo estará vacío. *Fecha y hora.*
 - **date_joined:** Fecha y hora en la que el usuario se registró en la aplicación. Este campo es útil para saber cuándo se creó la cuenta del usuario. *Fecha y hora.*
- **Grupo [auth_group]:** Esta tabla almacena los grupos de usuarios. Los campos que contiene son los siguientes:
 - **id:** Identificador único del grupo. *Clave primaria (entero).*
 - **name:** Nombre del grupo. *Cadena de caracteres.*
- **Permiso [auth_permission]:** Esta tabla almacena los permisos que pueden ser asignados a los usuarios y grupos. Los campos que contiene son los siguientes:

- **id**: Identificador único del permiso. *Clave primaria (entero)*.
 - **name**: Nombre del permiso. *Cadena de caracteres*.
 - **codename**: Código del permiso. Este campo es único y se utiliza para identificar el permiso en la aplicación. *Cadena de caracteres*.
- **Permiso de grupo [auth_group_permissions]**: Esta tabla almacena la relación entre los grupos y los permisos que tienen asignados. Así pues, esta tabla relaciona las tablas `auth_group` y `auth_permission`. Los campos que contiene son los siguientes:
- **id**: Identificador único de la relación. *Clave primaria (entero)*.
 - **group_id**: Identificador del grupo al que pertenece el permiso. Este campo es una relación N:1 con la tabla `auth_group`. *Entero*.
 - **permission_id**: Identificador del permiso asignado al grupo. Este campo es una relación N:1 con la tabla `auth_permission`. *Entero*.
- **Permiso de usuario [auth_user_user_permissions]**: Esta tabla almacena la relación entre los usuarios y los permisos que tienen asignados. Consecuentemente, la tabla actúa como tabla intermedia entre las tablas `auth_user` y `auth_permission`. Los campos que contiene son los siguientes:
- **id**: Identificador único de la relación. *Clave primaria (entero)*.
 - **user_id**: Identificador del usuario al que pertenece el permiso. Este campo es una relación N:1 con la tabla `auth_user`. *Entero*.
 - **permission_id**: Identificador del permiso asignado al usuario. Este campo es una relación N:1 con la tabla `auth_permission`. *Entero*.
- **Usuario de grupo [auth_user_groups]**: Esta tabla almacena la relación entre los usuarios y los grupos a los que pertenecen. De esta manera, actúa como tabla intermedia entre las tablas `auth_user` y `auth_group`. Los campos que contiene son los siguientes:
- **id**: Identificador único de la relación. *Clave primaria (entero)*.
 - **user_id**: Identificador del usuario al que pertenece el grupo. Este campo es una relación N:1 con la tabla `auth_user`. *Entero*.

- **group_id**: Identificador del grupo al que pertenece el usuario. Este campo es una relación N:1 con la tabla **auth_group**. *Entero*.
- **Sesión [django_session]**: Esta tabla almacena las sesiones de los usuarios. Los campos que contiene son los siguientes:
 - **session_key**: Clave única de la sesión. Este campo es único y se utiliza para identificar la sesión en la aplicación. *Cadena de caracteres*.
 - **session_data**: Datos de la sesión. Este campo almacena los datos de la sesión en formato JSON. *Cadena de caracteres*.
 - **expire_date**: Fecha de expiración de la sesión. Este campo indica cuándo caduca la sesión y se elimina automáticamente. *Fecha y hora*.

Con las siete tablas definidas, es necesario explicar el funcionamiento de cada una de ellas y como se relacionan entre sí. En primer lugar, todo parte de la tabla de usuarios **auth_user**, que almacena la información de estos, y de la tabla de permisos **auth_permission**, que almacena los permisos que pueden ser asignados. A partir de aquí, existen dos maneras de asignar permisos a los usuarios:

- **Asignación directa**: Se pueden asignar permisos directamente a los usuarios, lo que significa que el usuario tendrá acceso a las funcionalidades y datos correspondientes a esos permisos. Este enfoque es útil para asignar permisos a usuarios individuales, lo que puede resultar necesario en algunos casos.
- **Asignación a través de grupos**: Se pueden crear grupos de usuarios y asignarles permisos, lo que significa que todos los usuarios del grupo tendrán acceso a las funcionalidades y datos correspondientes a esos permisos. Este segundo enfoque es útil para asignar permisos a un grupo de usuarios, lo que puede resultar más eficiente y fácil de gestionar.

Para la asignación directa, se utiliza la tabla **auth_user_user_permissions**, que relaciona los usuarios con los permisos que tienen asignados. De esta manera, la tabla relaciona las tablas **auth_user** y **auth_permission**. Cada permiso que se le asigna a un usuario representará

una instancia de la tabla. Gracias a esta tabla, se pueden asignar permisos a los usuarios de manera individual, lo que permite un control más granular sobre los permisos de cada usuario.

Por otro lado, para la asignación a través de grupos, se utilizan las tablas `auth_group` y `auth_group_permissions`. La primera almacena los grupos de usuarios y la segunda relaciona los grupos con los permisos que tienen asignados, mediante la relación entre las tablas `auth_group` y `auth_permission`. Cada permiso que se le asigna a un grupo representará una instancia de la tabla. De esta manera, se pueden crear grupos de usuarios y asignarles permisos, lo que permite un control más eficiente sobre los permisos de los usuarios, pues es mucho más óptimo asignar permisos a un grupo que asignar permisos usuario por usuario.

No obstante, para asignar los usuarios a los grupos, se utiliza la tabla `auth_user_groups`, que relaciona los usuarios con los grupos a los que pertenecen mediante la relación entre las tablas `auth_user` y `auth_group`. De esta manera, se pueden asignar usuarios a grupos y, por tanto, asignarles los permisos correspondientes a esos grupos.

Por último, es importante mencionar como se guarda la contraseña de los usuarios en la base de datos. Idealmente, cualquier persona que pueda llegar a tener acceso a la base de datos, ya sea un desarrollador, un administrador o un atacante, no debería poder acceder a las contraseñas de los usuarios. Por este motivo, las contraseñas no se guardan en texto plano, sino que se almacenan de manera segura utilizando un algoritmo de encriptación. La encriptación de contraseñas es una de aquellas prácticas donde uno no deber reinventar la rueda, ya que existen múltiples algoritmos de encriptación seguros y probados. En este caso, Django utiliza el algoritmo PBKDF2 para realizar la encriptación, guardando la contraseña en el formato siguiente:

`<algoritmo>${<número de iteraciones>}${<salto>}${<hash>}`

3.3. Desarrollo del *backend*

Una vez se ha definido la base de datos, uno ya puede tener una idea general de como puede funcionar la lógica de la aplicación. Al fin y al cabo, se debe entender el *backend* como el

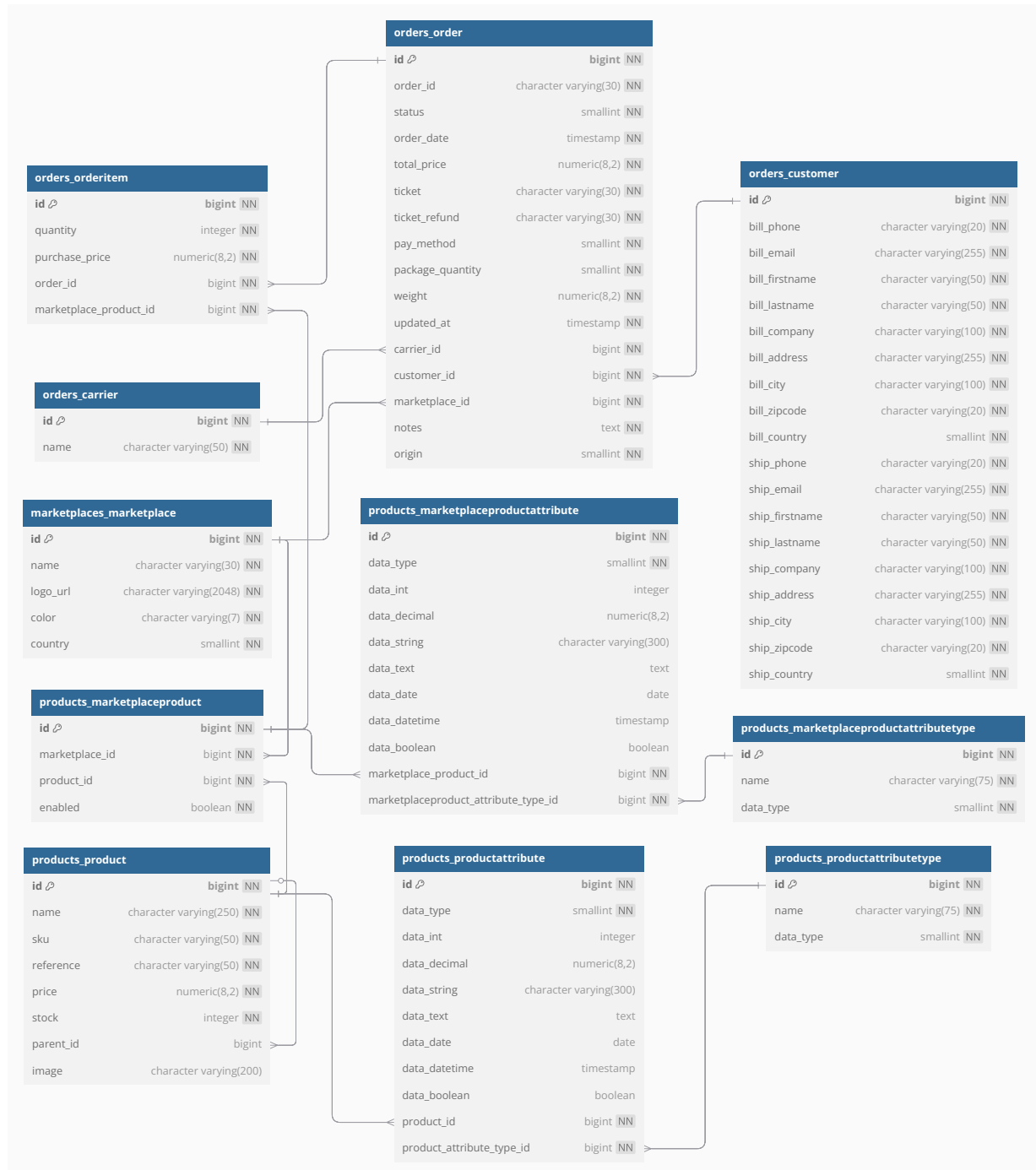


Figura 3.2: Diagrama de la base de datos

elemento que permite a la aplicación gestionar los datos y aplicarles cierta lógica para que esta realice las operaciones que el usuario ordena desde el *frontend*.

De este modo, como ya se ha comentado en la sección 2.3.2, el *backend* se ha desarrollado utilizando el *framework* Django. Esta decisión es resultado de la experiencia previa con este *framework*, de la experiencia en programación con Python y del hecho de que Django es uno de los *frameworks* más utilizados para el desarrollo *backend*. Cabe mencionar que además es un *framework* categorizado como *batteries included*, lo que significa que incluye muchas funcionalidades ya implementadas. Algunas de estas funcionalidades son la gestión de usuarios y seguridad, detallada en la sección 3.2.1, y la gestión de la base de datos mediante un ORM, detallada en la sección 2.3.3. Hacer uso de estas funcionalidades es una bastante buena práctica en fases iniciales de desarrollo, ya que permite centrarse en la lógica de la aplicación y no en la implementación de funcionalidades que ya están disponibles, además de que estas funcionalidades considerablemente complejas, pero que son muy comunes en todas las aplicaciones. Adicionalmente, son funcionalidades seguras y que normalmente ofrecen buenos patrones de diseño, a pesar de que a no son tan flexibles y escalables como si se implementaran desde cero.

Por otro lado, Django tiene la opción de crear páginas mediante plantillas, lo que permite crear una aplicación web completa con un *frontend* y un *backend* en el mismo proyecto, es decir, sin salir de la propia infraestructura del *framework*. Sin embargo, siguiendo lo comentado en la sección 3.1.1, esta práctica es poco usada en la actualidad, ya que no permite crear aplicaciones dinámicas, pues por cada pequeño cambio de la página el servidor tiene que procesar la petición y devolver la página completa. Por poner un ejemplo, si se quiere mostrar una lista de elementos y se quiere añadir un nuevo elemento, el servidor tiene que procesar la petición, generar la página completa con la lista de elementos actualizada y devolverla al cliente, lo que supone una recarga de la página. Esto, a pesar de ser funcional y ser la norma algunos años atrás, ofrece una experiencia de usuario bastante pobre, lo que hoy en día no es aceptable. Por estos mismos motivos, se ha optado por crear un *backend* que sirva una API RESTful, que es la norma en la actualidad, y un *frontend* que consuma dicha API. Para ello, se ha utilizado Django REST Framework, una extensión de Django que permite crear APIs de tipo REST ofreciendo funcionalidades como la serialización de datos, la autenticación y la autorización, entre otras.

Con todo esto definido, para dejar descrito todo el *backend* de la aplicación, se ha seguido un proceso de desarrollo que se detalla a continuación. Este proceso se ha dividido en varias secciones, cada una de las cuales se centra en un aspecto concreto del desarrollo del *backend*. Así pues, se ha comenzado por la estructuración del proyecto, que es la base sobre la que se construirá el resto del *backend*. A continuación, se han definido los modelos, que son la representación de los datos en la base de datos. Después, se han establecido los *serializers*, que son los encargados de convertir los modelos en datos JSON y viceversa para así formalizar la API REST. Finalmente, se han desarrollado las vistas y las URLs, que son las encargadas de gestionar las peticiones y respuestas de la API.

3.3.1. Estructuración del proyecto

La estructuración del proyecto es un paso fundamental en el desarrollo de cualquier parte de la aplicación, ya sea el *backend* como el *frontend*. Estructurar correctamente permite organizar el código de manera que sea fácil de entender y mantener para que en un futuro se puedan añadir funcionalidades sin alterar la estructura ya existente. Cada *framework* tiene su propio patrón de organización, el cual se recomienda para que el desarrollo sea lo más óptimo posible y en un futuro permita su escalabilidad.

Un proyecto de Django se compone de una serie de aplicaciones, cada una de las cuales se encarga de ofrecer una funcionalidad concreta. Dentro de cada aplicación, la estructura es la misma, de manera que cada aplicación tiene su propio directorio con los siguientes principales ficheros:

- `models.py`: Es donde se definen los modelos de la aplicación.
- `serializers.py`: Es donde se definen los *serializers* de la aplicación.
- `views.py`: Es donde se definen las vistas de la aplicación.
- `urls.py`: Es donde se definen las URLs de la aplicación.
- `admin.py`: Es donde se definen las configuraciones del panel de administración de Django.

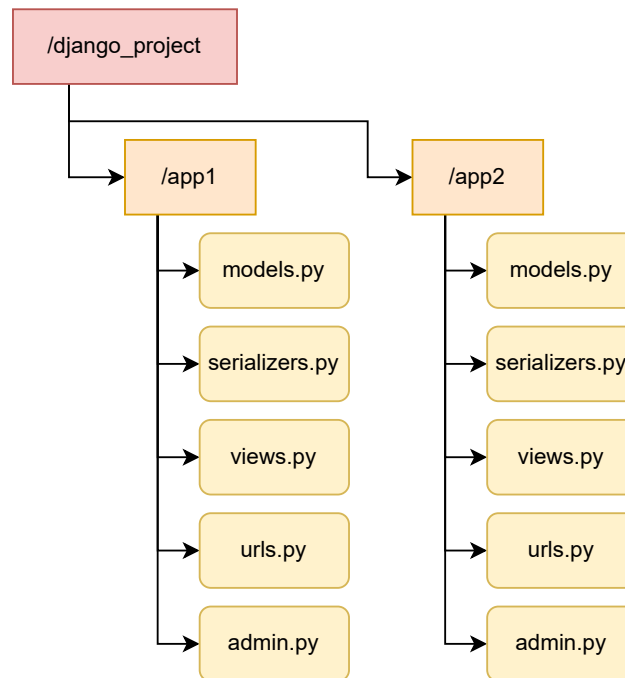


Figura 3.3: Estructura de un proyecto de Django.

Cada uno de estos ficheros van a ser detallados en las siguientes secciones, pues su importancia es fundamental para que el *backend* funcione correctamente. En la figura 3.3 se puede ver un ejemplo de la estructura de un proyecto de Django, donde se pueden observar las aplicaciones y los ficheros que componen cada una de ellas.

En este proyecto, la decisión sobre cómo estructurar las funcionalidades no ha sido trivial. Si bien es cierto que se pueden identificar dos usos principales, la gestión de pedidos y la gestión de productos, ninguno de ellos es completamente independiente del otro. De hecho, ambos están estrechamente relacionados en distintos puntos del flujo de trabajo: los pedidos tienen productos asociados.

Además, la gestión de los *marketplaces*, aunque actualmente no concentra una lógica compleja, constituye un pilar fundamental de la aplicación. No solo es un eje que articula tanto productos como pedidos, sino que también representa un área con un gran potencial de crecimiento. En el futuro, es probable que esta parte del proyecto incorpore nuevas funcionalidades relacionadas con integraciones, sincronización de datos o reglas específicas por canal de venta, lo que justifica su separación en una aplicación propia desde el inicio.

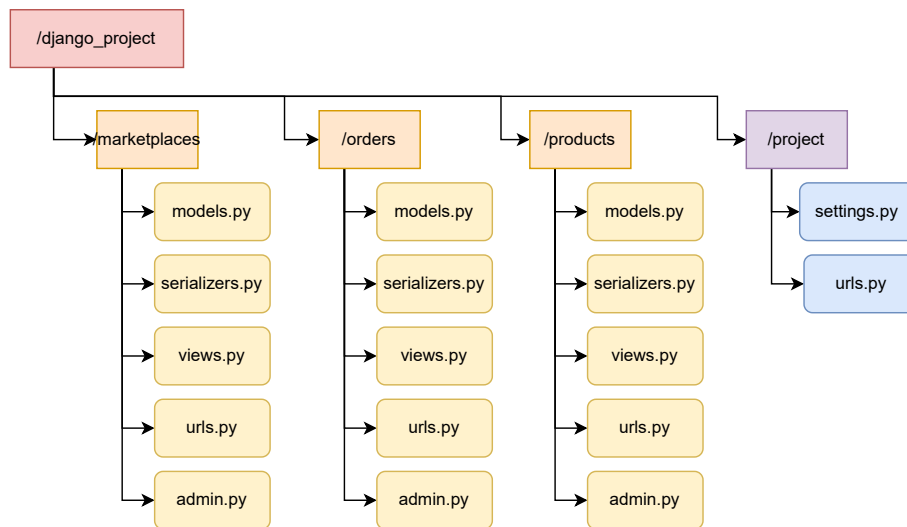


Figura 3.4: Estructura del proyecto.

Precisamente por estas razones, se ha optado por separar la aplicación en tres módulos/aplicaciones independientes: orders, products y marketplaces. Esta división permite una mejor organización del código, una separación clara de responsabilidades y una mayor facilidad para escalar y mantener el proyecto a medida que crece. Aunque existen relaciones entre los distintos dominios, mantenerlos como apps independientes mejora la cohesión interna de cada uno y facilita enormemente que en un futuro se puedan añadir nuevas funcionalidades o modificar las existentes sin afectar al resto del sistema.

En conclusión, el proyecto quedará estructurado en un total de 3 aplicaciones, es decir, tres directorios, cada uno de los cuales contendrá los ficheros mencionados anteriormente. Dicha estructura se puede ver en la figura 3.4.

Sin embargo, y ya para terminar, cabe mencionar un directorio adicional que se puede observar en la anterior figura 3.4, el directorio project. Este directorio no es una aplicación, sino que es el directorio principal del proyecto. En este se encuentran los ficheros de configuración del proyecto, como el fichero `settings.py`, que contiene la configuración del proyecto como podría ser la configuración de la base de datos, y el fichero `urls.py`, que contiene las URLs del proyecto. Al primero no se le dará mucha relevancia en esta memoria, ya que, a pesar de ser muy importante, sale un poco del alcance. Sin embargo, el segundo sí que se va a detallar, ya que es donde se definen las URLs que apuntan a cada aplicación. En la sección 3.3.4 se explicará cómo se definen dichas URLs y cómo se conectan con las vistas, que son

las encargadas de gestionar las peticiones y respuestas de la API.

3.3.2. Definición de los modelos

Con la estructura del proyecto definida, el siguiente paso es definir los modelos. Como se explicó en la sección 2.3.3, Django incluye de forma nativa un ORM que permite definir las tablas de la base de datos mediante clases de Python, y sus registros mediante objetos de esas clases. Con esto, se puede definir la base de datos y hacer consultas a la misma de una manera mucho más sencilla y rápida, ya que no es necesario escribir consultas SQL, sino que se pueden utilizar los métodos del ORM para realizar las operaciones necesarias.

Cada tabla de la base de datos es llamada modelo, y cada modelo se define mediante una clase de Python. Cada atributo de la clase representa una columna de la tabla, y cada instancia de la clase representa un registro de la tabla. Además, Django ofrece una serie de tipos de datos que se pueden utilizar para definir los atributos de las clases, como `CharField` para cadenas de texto, `IntegerField` para enteros, `DateTimeField` para fechas y horas, entre otros.

De esta manera, habiendo diseñado ya la estructura de la base de datos en la sección 2.3.3, definir los modelos es un proceso relativamente sencillo. Volviendo a la división en bloques de funcionalidades hecha en la sección 3.2.1, uno puede observar que, exceptuando el bloque de usuarios y autenticación, los tres bloques restantes corresponden a las tres aplicaciones que se han definido en la sección 3.3.1. Por lo tanto, las tablas de cada uno de los bloques se encontrarán definidas en el archivo `models.py` de cada una de las aplicaciones.

Dentro de cada uno de los ficheros `models.py` se definen las clases que representan los modelos de la aplicación. Cada clase hereda de `models.Model`, que es la clase base de Django para todos los modelos. A continuación, se definen los atributos de la clase, que son los campos de la tabla. Cada campo se define como un atributo de la clase y se le asigna un tipo de dato, como `CharField`, `IntegerField`, `DateTimeField`, entre otro. Además, se pueden definir relaciones entre modelos utilizando los campos `ForeignKey`, `ManyToManyField` y `OneToOneField`, dependiendo de la relación entre tablas que se quiera establecer.

```
class Product(models.Model):
    parent = models.ForeignKey(
        "self",
        on_delete=models.CASCADE,
        null=True,
        blank=True,
        related_name="children",
    )
    name = models.CharField(
        max_length=250,
    )
    sku = models.CharField(
        max_length=50,
    )
    reference = models.CharField(
        max_length=50,
    )
    price = models.DecimalField(
        max_digits=8,
        decimal_places=2,
    )
    stock = models.PositiveIntegerField()
    image = models.URLField(
        null=True,
    )
```

Fragmento 3.1: Definición del modelo `Product` para definir la tabla `product`.

En el fragmento de código 3.1 se puede ver un ejemplo de cómo se ha definido el modelo `Product`. Cada uno de los atributos de la clase representa una columna de la tabla `product` de la base de datos, tal como pudo ser observado en la sección 2.3.3, y cada uno de los tipos de datos utilizados corresponde a un tipo de dato de la base de datos. Por ejemplo, el atributo `price` es un campo de tipo `DecimalField`, que corresponde a una columna de tipo decimal llamada `price` de la table `product` de la base de datos. Además, cada atributo tiene una serie de parámetros que permiten definir características adicionales del campo. Siguiendo el ejemplo, el atributo `price` tiene los parámetros `max_digits` y `decimal_places`,

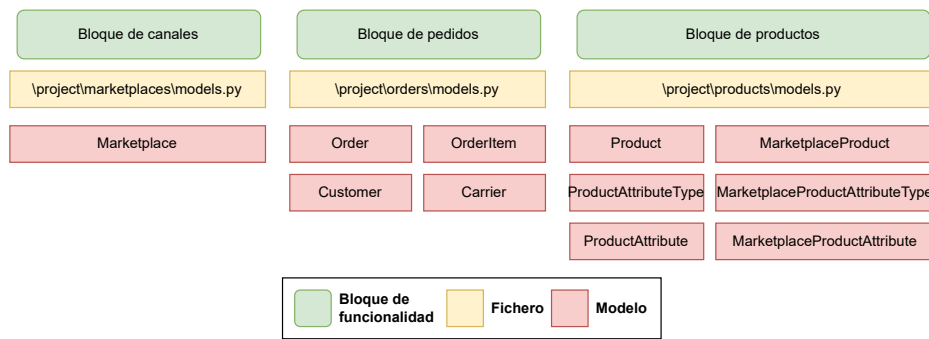


Figura 3.5: Estructura de los modelos del proyecto.

que permiten definir el número máximo de dígitos y el número de decimales que se pueden almacenar en el campo, respectivamente. Estos parámetros son muy útiles para garantizar la integridad de los datos y evitar errores al almacenar información en la base de datos.

Así pues, siguiendo cada uno de los bloques de funcionalidades definidos 3.2.1 y la estructura entre aplicaciones establecida en la sección 3.3.1, se han definido los modelos siguiendo la figura 3.5. Como es observable, cada fichero `models.py` almacena los modelos del bloque de funcionalidades correspondiente a cada aplicación, de manera que en caso de que se quiera añadir un nuevo modelo, este se puede añadir en el fichero correspondiente sin alterar la estructura del proyecto.

3.3.3. Definición de los *serializers*

Una vez definidos los modelos, el siguiente paso es crear los *serializers*. Aunque en la práctica la definición de *serializers* y vistas suele realizarse de forma conjunta, ya que las vistas necesitan los *serializers* para enviar y recibir datos desde el *frontend*, y estos carecen de utilidad sin una vista que los utilice, desde el punto de vista didáctico resulta más natural presentar primero la estructura de los *serializers* y, a continuación, su uso dentro de las vistas. Por esta razón, en lugar de seguir el orden habitual, se explicarán primero los *serializers* y después las vistas, en la correspondiente sección 3.3.4.

Con esta previa aclaración hecha, se puede empezar a definir que son los *serializers* y cuál es su función dentro del *backend*. Volviendo a la sección 2.3.4, se ha comentado que la respuesta de una API REST acostumbra ser en un formato estructurado llamado JSON. Este formato

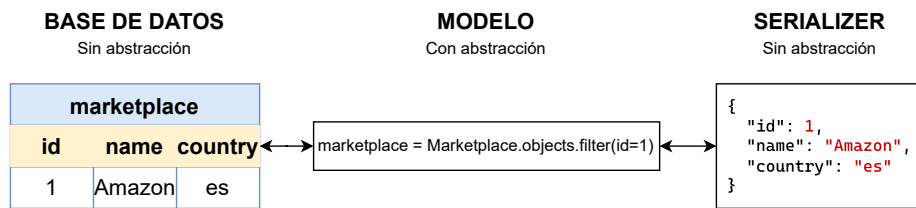


Figura 3.6: Funcionamiento de los *serializers*.

es fácil de leer y entender, tanto para humanos como para máquinas, y es el estándar de facto para las APIs REST. Sin embargo, los datos que se manejan en el *backend* suelen ser más complejos, ya que están representados por los modelos, que son objetos de Python. Por lo tanto, para poder enviar y recibir datos entre el *frontend* y el *backend*, es necesario convertir estos objetos en un formato más sencillo y manejable, como JSON. De esta manera, en términos generales, los *serializers* son componentes que permiten convertir datos complejos, como los modelos, en formatos más simples y manejables. Un *serializer* toma un modelo o un conjunto de datos y los convierte en formato JSON para que puedan ser enviados al *frontend* y que este los pueda interpretar. De esta manera, tal como puede observarse en la figura 3.6, los *serializers* eliminan el nivel de abstracción de los modelos convirtiéndolos en un formato que no deja de ser nada más que un conjunto de texto estructurado.

No obstante, también tienen una función inversa, que es la de convertir datos en formato JSON en objetos de Python, lo que permite recibir datos desde el *frontend* y transformarlos en objetos que se puedan manipular en el *backend*. Todo esto, acompañado de una validación de los datos, ya que los *serializers* también se encargan de validar que los datos recibidos cumplen con las reglas definidas en los modelos. Por ejemplo, si se recibe un dato que no es del tipo correcto o que no cumple con las restricciones definidas en el modelo, el *serializer* devolverá un error indicando qué dato es incorrecto y por qué. Esto es muy útil para evitar errores en el *backend* y garantizar que los datos que este trata son válidos y cumplen con las reglas definidas.

Con el concepto de *serializer* ya definido, se puede empezar a ver como estos se estructuran dentro de Django. Como ya pudo ser observado en la figura 3.4, cada aplicación tiene su propio fichero `serializers.py`, donde se definen los *serializers* del bloque de funcionalidades correspondiente, de manera análoga a los modelos. Cada *serializer* se define como una clase que hereda de `serializers.ModelSerializer`, que es la clase base de Django REST

Framework para los *serializers* basados en modelos. Al estar basada en modelos, esta clase permite definir los campos del *serializer* de forma automática a partir de los modelos definidos en el fichero `models.py`. Esto significa que no es necesario definir manualmente cada campo del *serializer*, sino que se pueden utilizar los campos del modelo directamente, lo que simplifica mucho su definición y, en caso de actualizar un modelo, no es necesario actualizar el *serializer* manualmente, ya que este se actualizará automáticamente. Existe la posibilidad de hacer *serializers* sin modelos, pero en este caso, donde el interés de la API es exponer los modelos de la base de datos, no tiene sentido hacer *serializers* sin modelos. Adicionalmente, los *serializers* basados en modelos también ofrecen la posibilidad de definir campos adicionales que no están presentes en el modelo, lo que permite añadir información extra al *serializer* sin necesidad de modificar el modelo. Esto es muy útil para añadir información adicional que no es necesaria en la base de datos, pero que sí es útil para el *frontend*. Por último, también existe la posibilidad de anidar *serializers* dentro de otros, de manera que se pueden crear *serializers* más complejos que representen relaciones entre modelos. Esto es muy útil para representar relaciones de tipo uno a muchos o muchos a muchos, ya que permite incluir información de los modelos relacionados dentro del mismo *serializer*.

```
class CustomerSerializer(serializers.ModelSerializer):
    class Meta:
        model = Customer
        fields = "__all__"

class OrderItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = OrderItem
        fields = "__all__"

class OrderSerializer(serializers.ModelSerializer):
    customer = CustomerSerializer()
    order_items = OrderItemSerializer(many=True)

    class Meta:
        model = Order
        fields = "__all__"
```

Fragmento 3.2: Definición de los *serializers* `Customer`, `OrderItem` y `Order`.

En el fragmento de código 3.2 se puede ver un ejemplo de cómo se han definido los *serializers* `Customer`, `OrderItem` y `Order`. Como se puede observar, el *serializer* de `Order` incluye los *serializers* de `Customer` y `OrderItem` como campos anidados, lo que permite incluir información de los modelos relacionados dentro del mismo *serializer*. Con este se consigue que en el momento de serializar un pedido, los campos del cliente y los productos del pedido se incluyan dentro del mismo *serializer*, lo que permite que con una sola petición se obtenga toda la información necesaria para mostrar un pedido completo. Dicha serialización se puede observar en el fragmento JSON 3.3, donde se muestra el resultado del *serializer* del modelo `Order` del fragmento 3.2. Sin embargo, anidar campos no siempre es la mejor opción, pues muchas veces el *frontend* no requiere de toda la información de los modelos relacionados, sino que solo necesita una parte y, como mayor sea la cantidad de información que se envíe, mayor será el tamaño de la respuesta y más lenta será la carga de la página. Por lo tanto, es importante tener en cuenta qué información es necesaria y cuál no, y anidar los campos solo cuando sea necesario.

```
{
  "id": 4,
  "order_id": "M123213",
  "status": 1,
  "origin": 0,
  "order_date": "2025-01-13T16:03:0
    9.07000Z",
  "total_price": "91.84",
  "ticket": "152535",
  "ticket_refund": "",
  "pay_method": 0,
  "package_quantity": 4,
  "weight": "7.14",
  "notes": "testgttehetjet",
  "updated_at": "2025-05-05T16:33:1
    9.570528Z",
  "marketplace": 1,
  "carrier": 1,
  "customer": {
    "id": 3,
    "bill_phone": "333333334",
    "bill_email": "charlie.brown.
      verybrown@example.com",
    "bill_firstname": "Charlie",
    "bill_lastname": "Brown",
    "bill_company": "Peanuts Inc.",
    "bill_address": "555 Peanut
      Street",
    "bill_city": "Santa Rosa",
    "bill_zipcode": "95404",

    "bill_country": 0,
    "ship_phone": "333333333",
    "ship_email": "charlie.
      brown@example.com",
    "ship_firstname": "Charlie",
    "ship_lastname": "Brown",
    "ship_company": "Peanuts Inc.",
    "ship_address": "555 Peanut
      Street",
    "ship_city": "Santa Rosa",
    "ship_zipcode": "95404",
    "ship_country": 0
  },
  "order_items": [
    {
      "id": 6,
      "marketplace_product": 1,
      "quantity": 2,
      "purchase_price": "99.50",
      "order": 4
    },
    {
      "id": 7,
      "marketplace_product": 4,
      "quantity": 1,
      "purchase_price": "0.50",
      "order": 4
    }
  ]
}
```

Fragmento 3.3: Serialización de un pedido mediante el *serializer* `Order` y los correspondientes *serializers* `Customer` y `OrderItem` anidados.

Precisamente por este último punto, la definición de los *serializers* ha sido una de las tareas más complejas no solo del desarrollo del *backend*, sino de toda la aplicación. A la hora de

tomar decisiones sobre su estructura, se han tenido en cuenta tres aspectos clave:

1. Reutilización y generalización del código:

En el desarrollo de software, es habitual intentar que el código sea lo más genérico posible para favorecer su reutilización en diferentes partes de la aplicación. Esta práctica, aunque muy extendida, puede tener inconvenientes si se lleva al extremo, ya que un código excesivamente genérico puede resultar difícil de entender, mantener y, en algunos casos, poco eficiente.

2. Serializadores excesivamente completos:

Una posible estrategia consiste en crear un único *serializer* que incluya todos los campos del modelo, además de anidar los *serializers* de todos los modelos relacionados. Este enfoque puede ser útil en situaciones concretas, como al mostrar una vista detallada de un único pedido. No obstante, se vuelve ineficiente y poco adecuado si el objetivo es, por ejemplo, listar múltiples pedidos, ya que se estarían transmitiendo muchos datos innecesarios.

3. Serializadores demasiado específicos:

En el extremo opuesto, definir un *serializer* diferente para cada uso concreto puede derivar en una alta duplicación de código y una mayor dificultad para mantenerlo. Asimismo, limitar un *serializer* a los campos de un único modelo sin contemplar relaciones relevantes obliga al *frontend* a realizar múltiples peticiones para obtener la información completa, lo que afecta negativamente al rendimiento y a la experiencia de usuario.

Teniendo en cuenta estos factores, se ha optado por un enfoque híbrido que combina serializadores diseñados por funcionalidad (según las necesidades de cada vista o componente del *frontend*) con otros más centrados en la estructura del modelo. Esta solución busca un equilibrio entre reutilización, claridad, eficiencia y facilidad de mantenimiento.

Con lo anterior especificado, en la figura 3.7 se presenta la estructura general de los *serializers* del proyecto. En términos generales, existe un *serializer* asociado a los modelos más relevantes, aunque también se incluyen *serializers* que combinan varios modelos para cubrir

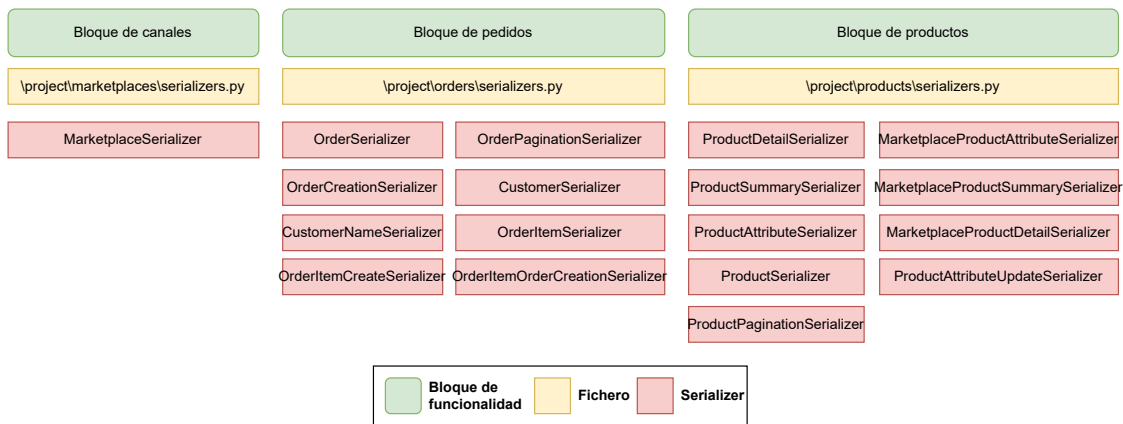


Figura 3.7: Estructura de los *serializers* del proyecto.

casos de uso más complejos. Asimismo, hay *serializers* reducidos que contienen únicamente los campos necesarios para vistas específicas, optimizando así la eficiencia y claridad.

En el apartado 3.3.4 se explicará cómo se emplean estos *serializers* dentro de las vistas y cómo se enlazan con las URLs correspondientes. Finalmente, en la sección 3.4 se mostrará la aplicación práctica de cada *serializer* en las distintas páginas del *frontend*.

3.3.4. Definición de las vistas y URLs

Para finalizar el desarrollo del *backend*, se deben definir las vistas y las URLs que llaman a dichas vistas. Las vistas son el componente encargado de gestionar las peticiones y respuestas de manera que cuando se acceda a una URL concreta, se ejecute la lógica necesaria para procesar la petición y devolver la respuesta adecuada. En el caso de una API REST, las vistas se encargan de recibir las peticiones HTTP, procesar los datos y devolver una respuesta en formato JSON.

En Django, las vistas se definen en el archivo `views.py` de cada aplicación, siguiendo una estructura coherente con la de los modelos y los *serializers*. En el contexto de Django REST Framework, existen diversas formas de definir vistas, pero una de las más habituales consiste en utilizar vistas basadas en modelos junto con los denominados *ViewSet*.

Estas vistas, al estar construidas sobre los modelos, permiten interactuar directamente con ellos, de forma análoga a como lo hacen los *serializers*. El uso de *ViewSet* facilita la de-

finición de operaciones CRUD (Crear, Leer, Actualizar y Eliminar), ya que Django REST Framework proporciona métodos predefinidos para cada una de estas acciones. Gracias a esto, no es necesario implementar manualmente cada uno de los métodos básicos de la API que comparten todos los modelos. Dichos métodos que se implementan de forma automática son prácticamente equivalentes a los métodos del protocolo HTTP, descritos en la sección [2.3.4](#), y son los siguientes:

- **list:** Permite obtener una lista de todos los registros del modelo. Hace uso del método GET sin parámetros en la URL.
- **retrieve:** Permite obtener un registro concreto del modelo. Hace uso del método GET con un parámetro en la URL que indica el identificador del registro.
- **create:** Permite crear un nuevo registro del modelo. Hace uso del método POST con los datos del nuevo registro en el cuerpo de la petición.
- **update:** Permite actualizar un registro concreto del modelo. Hace uso del método PUT con el identificador del registro en la URL y los datos actualizados en el cuerpo de la petición.
- **partial_update:** Permite actualizar parcialmente un registro concreto del modelo. Hace uso del método PATCH con el identificador del registro en la URL y los datos actualizados en el cuerpo de la petición.
- **destroy:** Permite eliminar un registro concreto del modelo. Hace uso del método DELETE con el identificador del registro en la URL.

No obstante, cuando se requiere una lógica más específica o se necesita implementar funcionalidades que no se cubren con los métodos estándar, es posible definir operaciones personalizadas dentro del propio *ViewSet*.

```
class CustomerViewSet(ModelViewSet):
    queryset = Customer.objects.all()
    serializer_class = CustomerSerializer

    @action(detail=False, methods=["get"], url_path="search")
    def search(self, request):
        search_query = request.query_params.get("q", "")
        if not search_query:
            return Response({"error": "Search query is
required"}, status=status.HTTP_400_BAD_REQUEST)

        queryset = self.get_queryset().filter(
            Q(bill_email__icontains=search_query)
        )

        serializer = self.get_serializer(queryset, many=
True)
        return Response(serializer.data, status=status.
HTTP_200_OK)
```

Fragmento 3.4: Definición del *ViewSet* para el modelo *Customer*.

En el fragmento de código 3.4 se puede observar un ejemplo de cómo se ha definido un *ViewSet* para el modelo *Customer*. Este *ViewSet* hereda de *viewsets.ModelViewSet*, lo que le otorga automáticamente los seis métodos mencionados anteriormente. Además de estos métodos, se ha añadido un método personalizado llamado *search* que a partir de un *queryparam* permite buscar clientes por su dirección de correo electrónico. Por último, se ha definido un *serializer_class* que indica qué *serializer* se debe utilizar para serializar los datos del modelo. En este caso, se ha utilizado el *serializer CustomerSerializer*, que es el encargado de convertir los objetos del modelo *Customer* en formato JSON y viceversa. De esta manera, a esta vista se le pueden hacer las siguientes peticiones a través de las URLs:

- GET */customers/*: Devuelve una lista de todos los clientes.
- GET */customers/{id}/*: Devuelve un cliente concreto por su identificador.

- **POST** `/customers/`: Crea un nuevo cliente. En el cuerpo de la petición se deben enviar los datos del nuevo cliente en formato JSON.
- **PUT** `/customers/{id}/`: Actualiza un cliente concreto por su identificador. En el cuerpo de la petición se deben enviar los datos actualizados del cliente en formato JSON.
- **PATCH** `/customers/{id}/`: Actualiza parcialmente un cliente concreto por su identificador. En el cuerpo de la petición se deben enviar los datos actualizados del cliente en formato JSON.
- **DELETE** `/customers/{id}/`: Elimina un cliente concreto por su identificador.
- **GET** `/customers/?q={email}`: Busca clientes por su dirección de correo electrónico.

A cada uno de estos métodos con su correspondiente URL se les denomina *endpoint*, y son los puntos de acceso a la API. Cada *endpoint* corresponde a una operación concreta que se puede realizar sobre el modelo, y cada uno de ellos tiene una URL única que permite acceder a él. De esta manera, se puede interactuar con la API de forma sencilla y estructurada.

Habiendo escogido el enfoque de utilizar *ViewSets* y habiendo mostrado un ejemplo de como estos funcionan, se ha definido la metodología a seguir para estructurar las vistas del proyecto. En primer lugar, se ha optado por crear un *ViewSet* por cada modelo relevante, lo que permite agrupar las operaciones relacionadas con cada modelo en un único lugar. Esto facilita la organización del código y mejora la legibilidad, ya que cada *ViewSet* se encarga de gestionar las operaciones de un único modelo y, en caso de requerir nuevas funcionalidades, se pueden añadir directamente en el *ViewSet* correspondiente sin afectar al resto del proyecto.

La figura 3.8 muestra la estructura general de las vistas del proyecto. Cada vista implementa los seis métodos predeterminados que proporcionan los *ViewSets* y, además, incorpora los métodos personalizados para dar respuesta a necesidades específicas de la aplicación. Cada módulo cuenta con su propio archivo `views.py`, donde se definen los *ViewSets* correspondientes a los modelos que forman parte de dicha aplicación.

Finalmente, una vez definidos los *ViewSets*, es necesario establecer las URLs que apuntan a cada uno de ellos. Esto se realiza en el fichero `urls.py` de cada aplicación, donde se definen las rutas que corresponden a cada *ViewSet*. Análogamente, para enrutar cada una de las

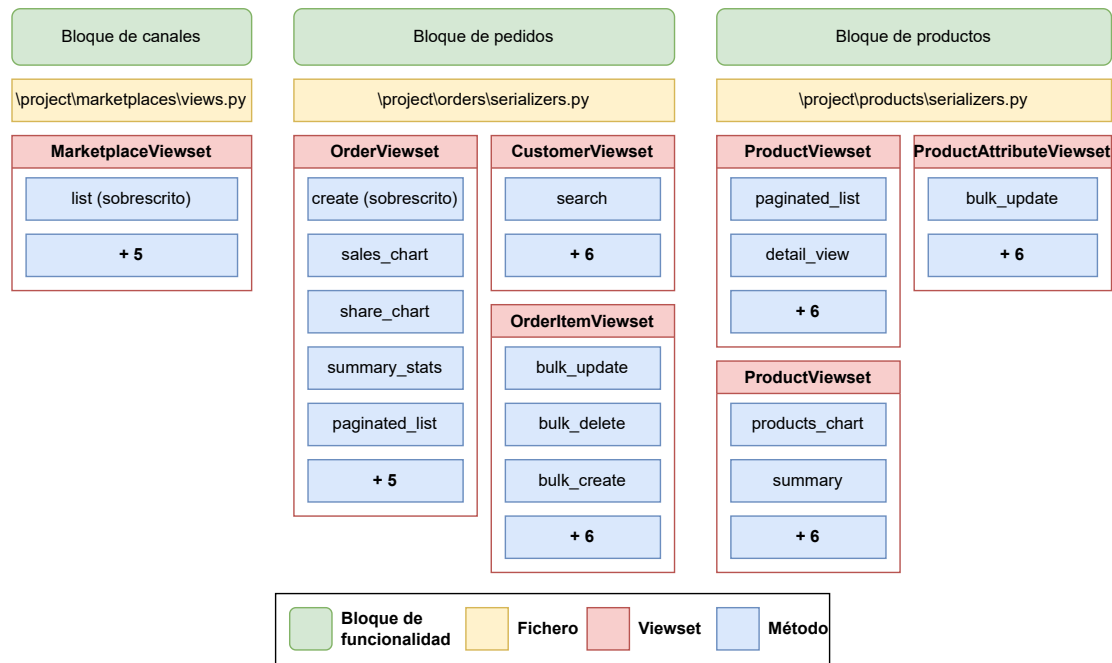


Figura 3.8: Estructura de las vistas del proyecto.

aplicaciones, se hace uso del fichero `urls.py` del directorio `project`, tal como se había ya descrito en la sección 3.3.1. De esta manera, se define una URL base para cada aplicación y, a partir de esta, se definen las URLs que apuntan a cada uno de los *ViewSets*.

En la figura 3.9 se puede observar la estructura general de todas las URLs y los correspondientes métodos de los *ViewSets* que se pueden solicitar al *backend*. Cada URL corresponde a un *endpoint* de la API, y cada uno de ellos tiene un método asociado que indica qué operación se puede realizar sobre el modelo. Por ejemplo, la URL `orders/customers/{id}/` corresponde al método `retrieve` del *ViewSet* de `Customer` de la aplicación `orders`, lo que permite obtener un cliente concreto por su identificador.

Llegar a esta estructura de URLs ha sido un proceso iterativo, pues se ha tenido que cambiar varias veces a medida que la aplicación ha ido evolucionando y el *frontend* ha ido requiriendo nuevos *endpoints*. Estructurar el enrutamiento de tal cantidad de *endpoints* es complejo y supone un reto si se quiere hacer correctamente de manera que en un futuro se puedan añadir funcionalidades si alterar mucho lo ya existente. Sin embargo, se ha tratado de mantener una estructura coherente y lógica, de manera que solamente leyendo las URLs se pueda entender qué operaciones se pueden realizar sobre cada modelo y cómo se pueden acceder a ellas.

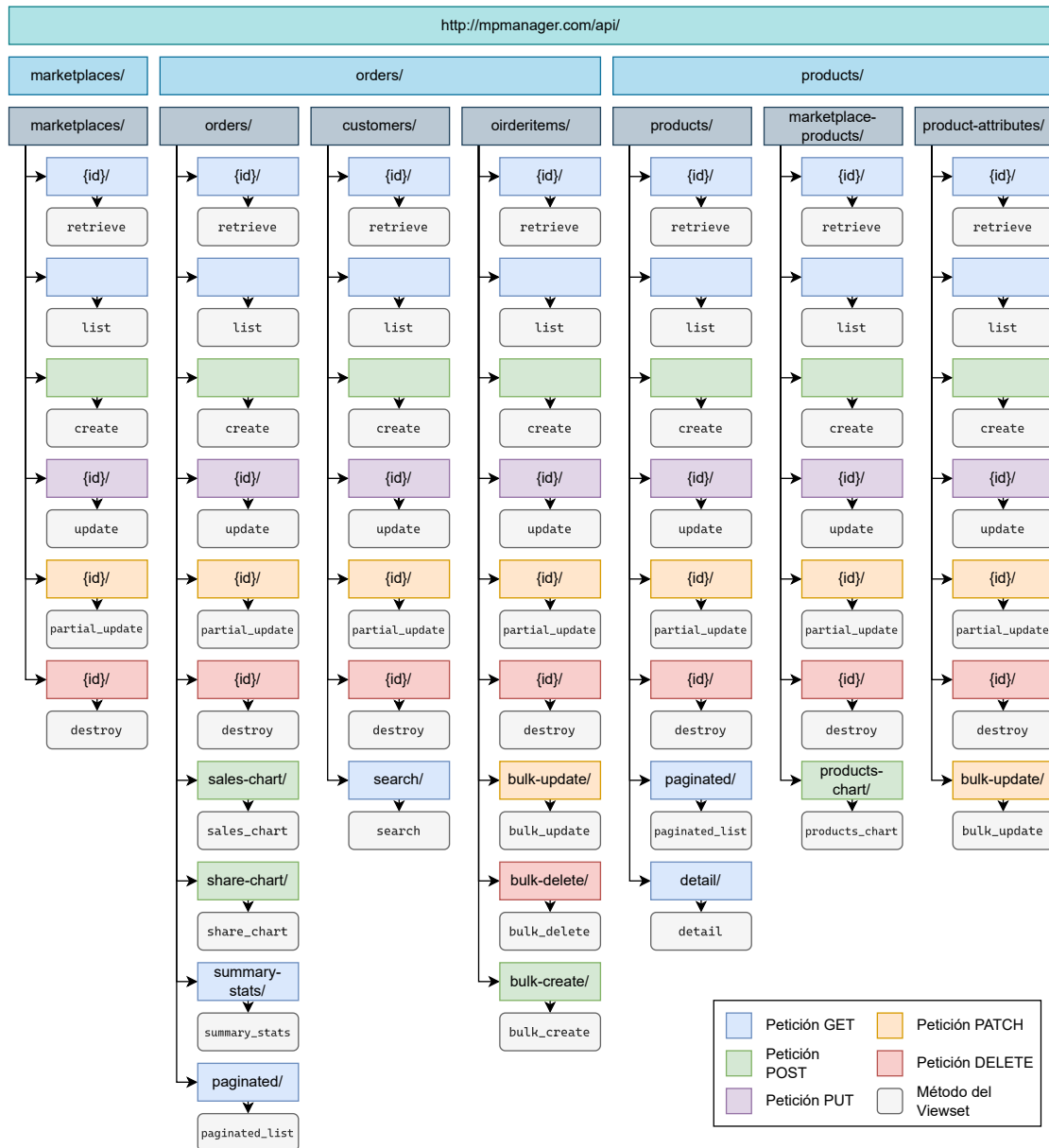


Figura 3.9: Enrutamiento de todas las URLs del *backend* con sus correspondientes métodos de los *ViewSets*.

3.3.5. Implementaciones relevantes

Para cerrar la sección de desarrollo del *backend*, es importante destacar algunas implementaciones relevantes que han requerido una atención especial por su complejidad o impacto funcional. Entre ellas, se encuentra la incorporación de un sistema de paginación para optimizar la entrega de datos en la API, así como la incorporación de distintos métodos para la actualización, creación y eliminación de entidades de manera masiva.

Paginación de *endpoints*

Cuando se manejan grandes volúmenes de datos, la paginación se convierte en una herramienta fundamental para garantizar una experiencia de usuario fluida y eficiente. Como se ha mencionado anteriormente, a mayor cantidad de datos transmitidos, mayor será el tamaño de la respuesta y, en consecuencia, más lenta será la carga de la página. Por este motivo, en aquellas vistas donde se espera un listado extenso de elementos, resulta imprescindible implementar un sistema de paginación que permita dividir los resultados en páginas más pequeñas y manejables.

Un *endpoint* con paginación permite que, mediante el uso de *query parameters*, el cliente especifique tanto el número de resultados por página como la página concreta que desea consultar. En la mayoría de los casos, al usuario no le interesa obtener todos los resultados de forma simultánea, sino acceder progresivamente a los datos en función de su necesidad concreta.

Adicionalmente a la paginación, en muchos casos también es útil poder filtrar los resultados según ciertos criterios o realizar búsquedas específicas. Por ejemplo, en un listado de productos, el usuario podría querer ver solo aquellos que cumplen con ciertas características o que pertenecen a una categoría específica. Para ello, se pueden añadir *query parameters* adicionales que permitan filtrar los resultados según los criterios deseados.

Volviendo a la figura 3.9, se puede observar que existen dos *endpoints* que incorporan paginación: uno para listar los pedidos y otro para listar los productos. Estos dos *endpoints* van a ser esenciales en sus correspondientes páginas del *frontend*, de manera que hacer que sus

respuestas sean lo más eficientes posible es crucial para garantizar una buena experiencia de usuario.

Para detallar de manera más práctica la implementación de la paginación, se va a tomar como ejemplo el *endpoint* de los pedidos. En este caso, la URL es la siguiente:

GET

`/orders/paginated/?page={n}&limit={m}&marketplace_ids={ids}&search={query}`

En esta URL, se pueden observar los siguientes *query parameters*:

- **page**: Indica el número de página que se quiere consultar. Por defecto, si no se especifica, se toma el valor 1.
- **limit**: Indica el número de resultados por página. Por defecto, si no se especifica, se toma el valor 10.
- **marketplace_ids**: Permite filtrar los pedidos por los identificadores de los *marketplaces* asociados. Si no se especifica, se obtienen todos los pedidos.
- **search**: Permite buscar pedidos por un término concreto. Si no se especifica, no se realiza ninguna búsqueda.

Realizando una petición a esta URL, se obtiene una respuesta en formato JSON que contiene los pedidos correspondientes a la página solicitada, así como información adicional sobre la paginación. Esta información incluye el número total de páginas, el número total de resultados y el número de resultados por página. De esta manera, el cliente puede saber cuántas páginas hay en total y cuántos resultados hay en cada página, lo que le permite navegar por los resultados de manera más eficiente.

Actualización, creación y eliminación masiva

En muchas ocasiones, es necesario realizar operaciones masivas sobre los datos, como actualizar, crear o eliminar múltiples registros de una sola vez. Hacer tantas peticiones como

registros a modificar puede resultar ineficiente y generar una carga innecesaria en el servidor, especialmente si se trata de un gran número de registros. Por este motivo, se ha implementado la posibilidad de realizar estas operaciones de forma masiva a través de un único *endpoint* para determinados modelos, tal como se puede observar en la figura 3.9 bajo los métodos que incluyen la palabra *bulk*.

En concreto, el principal modelo al que se ha aplicado esta funcionalidad es el de `OrderItem`, que representa los productos de un pedido. La necesidad de implementar operaciones masivas en este modelo se podrá observar en la debida sección del *frontend*; sin embargo, es fácilmente comprensible que, al gestionar pedidos con múltiples productos, la posibilidad de actualizar, crear o eliminar varios productos de un pedido de forma simultánea resulta prácticamente indispensable.

En concreto, se han implementado tres operaciones masivas para el modelo `OrderItem`:

- **bulk_update**: Haciendo uso del método PATCH, permite actualizar múltiples productos de un pedido de forma simultánea. La petición debe incluir en el cuerpo un listado de productos con sus respectivos identificadores y los campos a actualizar.
- **bulk_create**: Haciendo uso del método POST, permite crear múltiples productos de un pedido de forma simultánea. La petición debe incluir en el cuerpo un listado de productos con los datos necesarios para su creación.
- **bulk_delete**: Haciendo uso del método DELETE, permite eliminar múltiples productos de un pedido de forma simultánea. La petición debe incluir en el cuerpo un listado de identificadores de los productos a eliminar.

3.4. Desarrollo del *frontend*

Definida la base de datos y desarrollado el *backend*, el *frontend* es la última parte de la plataforma que resta por ser definido y desarrollado. El *frontend* es la parte de la plataforma con la que el usuario interactúa directamente, y es la responsable de mostrar la información al usuario para que este pueda realizar las acciones que desee.

Existen múltiples tecnologías que permiten desarrollar el *frontend* de una plataforma web, y en este caso se ha optado por utilizar React, un *framework* de JavaScript que hoy en día es uno de los más utilizados para el desarrollo de aplicaciones web. Precisamente del hecho de que React sea un estándar de facto en el desarrollo de aplicaciones web, se deriva la decisión de haber utilizado esta tecnología. Es un *framework* muy potente, respaldado por una gran cantidad de documentación y recursos, además de una enorme empresa como es Meta (anteriormente conocida como Facebook) que lo respalda y mantiene. Esto asegura que React se mantenga actualizado y sea compatible con las últimas tecnologías web.

A pesar de que React funciona con JavaScript, siguiendo lo mencionado en la sección 2.3.1, se ha optado por utilizar TypeScript. TypeScript es un superconjunto de JavaScript que añade tipado estático y otras características que facilitan el desarrollo y mantenimiento del código. Esto permite detectar errores en tiempo de compilación, lo que mejora la calidad del código y reduce la probabilidad de errores en tiempo de ejecución. Estos aspectos son especialmente importantes en un proyecto de gran envergadura, donde la complejidad del código puede aumentar rápidamente. Como se pretende que la plataforma siga evolucionando y creciendo, el uso de TypeScript facilita la escalabilidad del proyecto y evita problemas a largo plazo.

Con estos puntos claros, se ha procedido a desarrollar el *frontend* de la plataforma. Este proceso se ha dividido en principalmente dos fases: la estructuración del proyecto y el desarrollo de las diferentes vistas y componentes que componen la plataforma.

3.4.1. Estructuración del proyecto

Las estructuras de los proyectos desarrollados con React pueden variar dependiendo de las necesidades del proyecto y de las preferencias del equipo de desarrollo. Sin embargo, existen algunas convenciones y buenas prácticas que se suelen seguir para mantener el código organizado y fácil de mantener.

En este caso, se ha optado por una estructura que se divide en principalmente cuatro carpetas principales:

- **src/components:** Esta carpeta reúne todos los componentes reutilizables de la aplica-

ción. En React, los componentes son las unidades básicas que conforman la interfaz: pueden ir desde elementos sencillos como botones hasta estructuras más complejas como formularios completos. Para mantener una buena organización, se divide esta carpeta en subcarpetas: una para los componentes comunes (usados en múltiples partes de la aplicación) y otras para los componentes específicos de cada funcionalidad o sección.

- **src/routes:** En esta carpeta se encuentran definidas todas las rutas de la aplicación utilizando la librería `@tanstack/react-router`. La estructura de rutas sigue una organización jerárquica que refleja la navegación de la aplicación, permitiendo agrupar rutas relacionadas en subcarpetas. Cada archivo de ruta incluye tanto la definición del componente que debe renderizarse como los datos que necesita cargar.
- **src/services:** Esta carpeta contiene los servicios encargados de realizar las peticiones HTTP al *backend* utilizando la librería `axios`. Cada servicio agrupa las funciones relacionadas con un recurso concreto de la aplicación (por ejemplo, pedidos o productos), lo que permite centralizar la lógica de comunicación con el servidor.
- **src/types:** Esta carpeta contiene tanto las definiciones de tipos TypeScript como los esquemas de validación de la aplicación. Los tipos se utilizan para garantizar una correcta tipificación del código en toda la aplicación, mejorando la seguridad y legibilidad. Por su parte, los esquemas, definidos con la librería `zod`, se emplean para validar los datos introducidos por el usuario antes de enviarlos al servidor, asegurando que cumplan con las reglas establecidas (por ejemplo, formatos, campos obligatorios o rangos de valores).

Esta estructura queda plasmada en la figura 3.10, donde se puede observar como se organizan las diferentes carpetas del proyecto. En la siguiente sección se detallará el desarrollo de las vistas y los componentes que componen cada una de ellas, lo que permitirá una mejor comprensión de la figura.

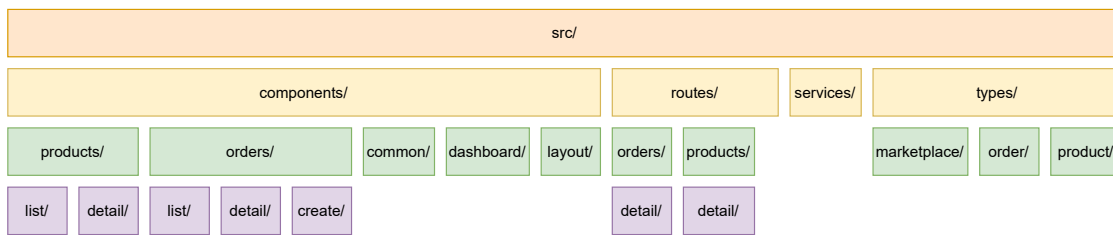


Figura 3.10: Estructura de archivos del *frontend* de la plataforma.

3.4.2. Desarrollo de vistas y componentes

Antes de comenzar con el desarrollo de las vistas, es fundamental tener claro cómo deben ser accesibles las distintas funcionalidades de la plataforma para el usuario final. Por este motivo, resulta útil remitir a la sección 3.2.1, donde se definieron los bloques funcionales principales. A modo de resumen, las funcionalidades clave que debe ofrecer la plataforma son las siguientes:

■ Gestión de pedidos:

- Centralización de los pedidos procedentes de los distintos canales de venta.
- Creación manual de nuevos pedidos.
- Edición de pedidos existentes.

■ Gestión de productos:

- Centralización de los productos provenientes de los distintos canales de venta.
- Asignación de productos a canales de venta específicos.
- Edición de productos ya existentes.

El desarrollo de estas funcionalidades debe enfocarse en ofrecer una experiencia de usuario ágil e intuitiva, permitiendo una gestión eficiente de los canales de venta. El objetivo es que la plataforma se convierta en una herramienta imprescindible para la administración diaria de los distintos canales de venta.

Es importante señalar que los usuarios de la plataforma no disponen de permisos para crear ni editar canales de venta. La incorporación de un nuevo canal es una tarea gestionada exclusivamente por la plataforma, ya que implica un análisis previo, así como la implementación

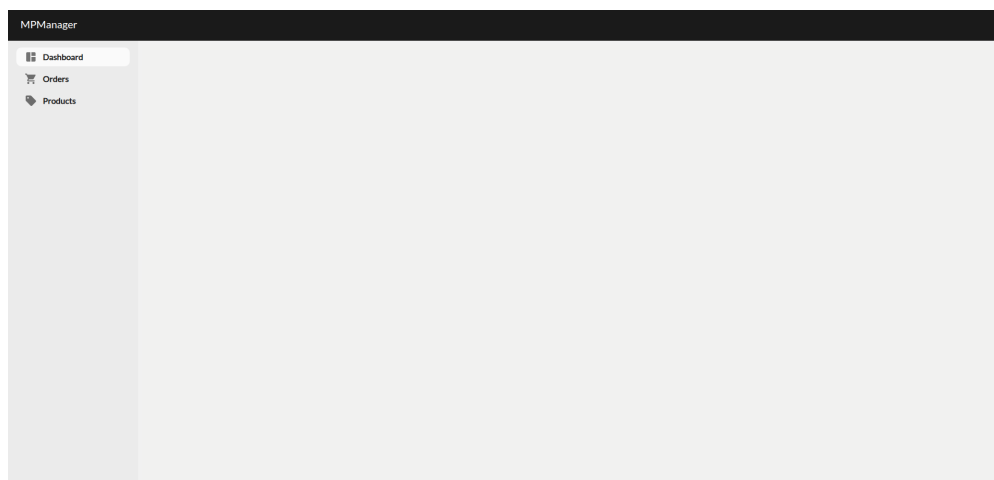


Figura 3.11: Menú lateral de navegación de la plataforma.

y configuración de la API correspondiente, entre otros aspectos técnicos. El usuario final puede asignar productos a los canales ya disponibles, pero no tiene control sobre su creación o configuración.

Anotado este último punto, se puede continuar con el desarrollo de las vistas. Teniendo en cuenta las funcionalidades definidas, la plataforma debe contar con dos vistas principales: una para la gestión de pedidos y otra para la gestión de productos. Como añadido a estas vistas, se ha incluido una vista de inicio que proporciona una visión general de la plataforma y permite observar datos generales del estado de los pedidos y productos.

Así pues, para centralizar las tres vistas principales se ha creado un menú lateral de navegación que permite al usuario acceder fácilmente a cada una de ellas, tal como se muestra en la figura 3.11. Este menú está presente en todas las vistas de la plataforma, permitiendo que el usuario pueda acceder a cada una de las secciones principales de forma rápida. De momento el menú cuenta con tres secciones: *Dashboard*, *Orders* y *Products*. Sin embargo, está diseñado para que, en el futuro, se puedan añadir nuevas secciones fácilmente así permitiendo la expansión de nuevas funcionalidades.

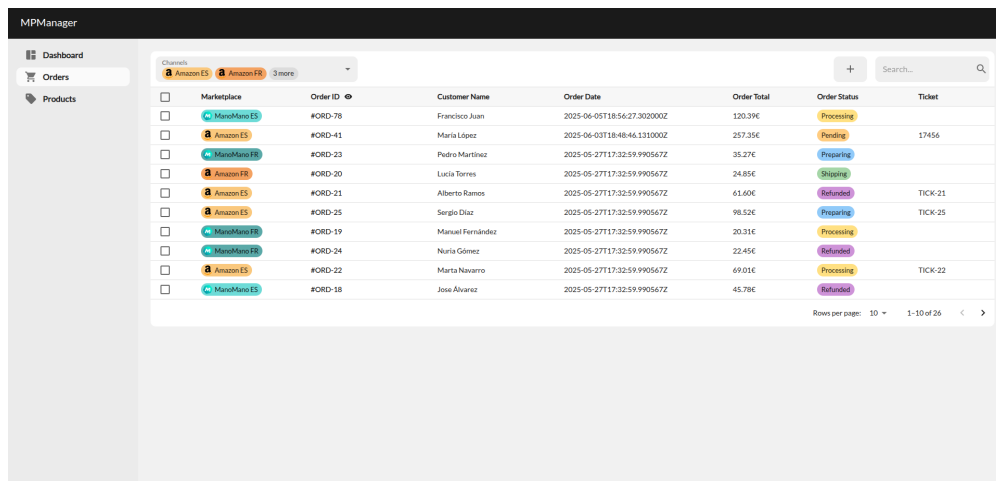
El orden de las secciones del menú lateral sigue una lógica que facilita la navegación. En primer lugar, se encuentra la sección *Dashboard*, que proporciona una visión general del estado de la plataforma, permitiendo al usuario obtener información rápida sobre los pedidos y productos. A continuación, se sitúa la sección *Orders*, que permite gestionar los pedidos de forma centralizada, y finalmente, la sección *Products*, donde se pueden gestionar los productos

disponibles en los distintos canales de venta. No obstante, el orden de desarrollo de las vistas no ha seguido este mismo orden, sino que se ha desarrollado primero la vista de pedidos, seguida de la vista de productos y, por último, la vista de inicio. El orden entre las vistas de pedidos y productos no es relevante, ya que ambas son bastante independientes una de la otra. Sin embargo, la vista de inicio se ha desarrollado al final para poder incluir en ella información relevante sobre el estado de los pedidos y productos, que se obtiene de las vistas anteriores. Por este mismo motivo, a continuación se detallará el desarrollo de las vistas según el orden con el que se han implementado, pues es seguramente el orden más lógico para comprender como todo funciona e interactúa.

Vista de pedidos

La vista de pedidos es una de las más importantes de la plataforma, ya que permite gestionar todos los pedidos provenientes de los distintos canales de venta. Gestionar los pedidos implica poder verlos, editarlos y crear de nuevos. Así pues, dicha vista se ha dividido en dos:

- **Vista general de pedidos:** Esta vista permite al usuario ver todos los pedidos que ha recibido la plataforma, independientemente del canal de venta del que provengan. En esta vista se pueden filtrar los pedidos por el canal de ventas además de poder buscarlos por su identificador o el nombre del cliente.
- **Vista detallada de pedido:** Esta vista permite al usuario ver un pedido concreto, así como editarlo. En esta vista se pueden ver todos los detalles del pedido, incluyendo los productos que lo componen, el estado del pedido y la información del cliente. Además, se pueden realizar acciones como editar el estado del pedido o añadir nuevos productos al mismo.
- **Vista de creación de pedido:** Esta vista permite al usuario crear un nuevo pedido. En esta vista se pueden seleccionar los productos que componen el pedido, así como introducir la información del cliente y el canal de venta al que pertenece el pedido. Una vez creado el pedido, este se añadirá a la lista de pedidos y estará disponible para su gestión.



Marketplace	Order ID	Customer Name	Order Date	Order Total	Order Status	Ticket
ManoMano ES	#ORD-78	Francisco Juan	2025-06-05T18:56:27.302000Z	120.39€	Processing	
Amazon ES	#ORD-41	Maria López	2025-06-03T18:48:46.131000Z	257.35€	Pending	17456
ManoMano FR	#ORD-23	Pedro Martínez	2025-05-27T17:32:59.990567Z	35.27€	Preparing	
Amazon FR	#ORD-20	Lucia Torres	2025-05-27T17:32:59.990567Z	24.85€	Shipping	
Amazon ES	#ORD-21	Alberto Ramos	2025-05-27T17:32:59.990567Z	61.60€	Refunded	TICK-21
Amazon ES	#ORD-25	Sergio Diaz	2025-05-27T17:32:59.990567Z	98.52€	Preparing	TICK-25
ManoMano FR	#ORD-19	Manuel Fernández	2025-05-27T17:32:59.990567Z	20.31€	Processing	
ManoMano FR	#ORD-24	Nuria Gómez	2025-05-27T17:32:59.990567Z	22.45€	Refunded	
Amazon ES	#ORD-22	Marta Navarro	2025-05-27T17:32:59.990567Z	69.01€	Processing	TICK-22
ManoMano ES	#ORD-18	Jose Alvarez	2025-05-27T17:32:59.990567Z	45.78€	Refunded	

Figura 3.12: Vista general de pedidos con algunos pedidos de ejemplo.

La primera de las vistas, la vista general de pedidos, es la que se muestra en la figura 3.12. Esta vista está principalmente compuesta por una tabla que muestra los pedidos recibidos, mostrando el canal de venta del que provienen, el identificador del pedido, el nombre del cliente, la fecha de creación del pedido, el importe total y el estado del mismo. Adicionalmente, se ha añadido un campo de búsqueda que permite filtrar los pedidos por el identificador del pedido o el nombre del cliente; y un filtro por canal de venta que permite ver únicamente los pedidos de los canales seleccionados.

Sin embargo, se puede observar que la tabla no muestra todos los pedidos de la plataforma, sino que muestra únicamente los 10 últimos pedidos recibidos. Como un usuario puede tener cientos o miles de pedidos, es necesario implementar una paginación que permita navegar entre los distintos pedidos. Esta paginación implementada es la discutida en la sección 3.3.5, y permite al usuario cambiar entre las distintas páginas de pedidos, seleccionar cuántos pedidos se quieren ver por página (10, 25, 50 o 100), buscar pedidos concretos y filtrar por canal de venta. De esta forma, para cada cambio de página, de búsqueda o de filtro, se realiza una petición al *backend* para obtener los pedidos correspondientes a la página, búsqueda o filtro seleccionados.

Pulsando sobre un pedido concreto de la tabla, se accede a la vista detallada del pedido, que se muestra en la figura 3.13. En esta vista se pueden ver todos los detalles del pedido, incluyendo los productos que lo componen, la información general del pedido y la información del cliente. Adicionalmente, se puede revisar cómo ha ido evolucionando el estado del pedido a lo largo del tiempo, ya que se muestra un historial de cambios de estado del pedido, y se

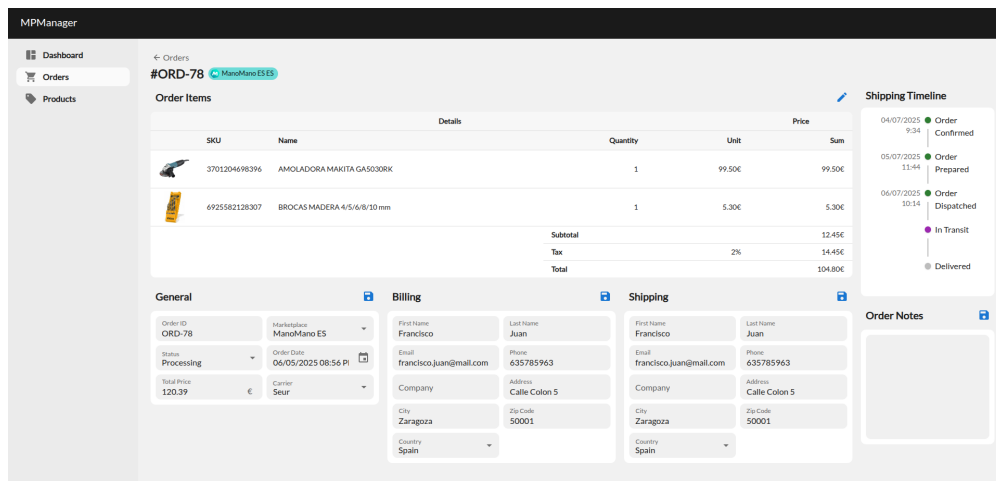
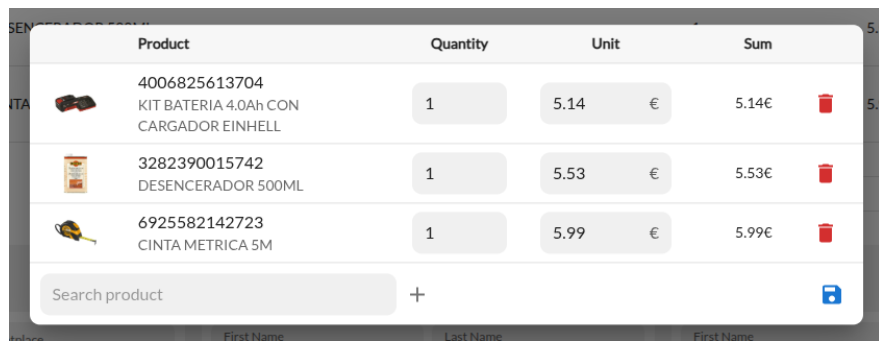





Figura 3.13: Vista detallada de un pedido concreto.

pueden añadir notas para registrar información adicional relevante.

Existe la posibilidad también de editar el pedido, tanto los productos que lo componen como la información general y del cliente. Sin embargo, existe una restricción importante: solo se pueden editar los pedidos que hayan sido creados manualmente por el usuario. Los pedidos que provienen de los canales de venta no pueden ser editados, ya que su información es gestionada directamente por el canal de y no se permite modificarla manualmente. Esto asegura la integridad de los datos y evita posibles inconsistencias en la información. Tal como se muestra en la figura 3.14, los únicos campos que se pueden editar en los pedidos provenientes de los canales de venta son el estado del pedido, el transportista, la dirección de envío y las notas del pedido. Estos campos son los que el usuario puede modificar para actualizar el estado del pedido o rectificar información para que el pedido pueda ser gestionado correctamente.



Product	Quantity	Unit	Sum
 4006825613704 KIT BATERIA 4.0Ah CON CARGADOR EINHELL	1	5.14 €	5.14€
 3282390015742 DESENCERADOR 500ML	1	5.53 €	5.53€
 6925582142723 CINTA METRICA 5M	1	5.99 €	5.99€


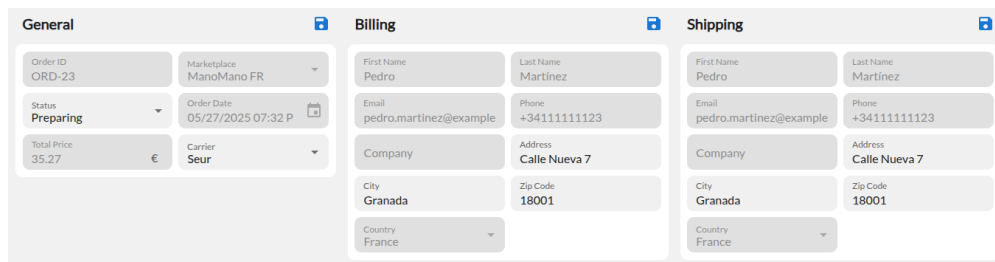
Search product + 

Figura 3.15: Modal de edición de productos de un pedido.



General

Order ID: ORD-23
Marketplace: ManoMano FR
Status: Preparing
Order Date: 05/27/2025 07:32 P
Total Price: 35.27 €
Carrier: Seur

Billing

First Name: Pedro
Last Name: Martinez
Email: pedro.martinez@example.com
Phone: +341111111123
Company:
Address: Calle Nueva 7
City: Granada
Zip Code: 18001
Country: France

Shipping

First Name: Pedro
Last Name: Martinez
Email: pedro.martinez@example.com
Phone: +341111111123
Company:
Address: Calle Nueva 7
City: Granada
Zip Code: 18001
Country: France

(a) Campos de un pedido creado automáticamente por el canal de venta.



General

Order ID: ORD-18
Marketplace: ManoMano ES
Status: Refunded
Order Date: 05/27/2025 07:32 P
Total Price: 45.78 €
Carrier: Seur

Billing

First Name: Jose
Last Name: Álvarez
Email: jose.alvarez@example.com
Phone: +341111111118
Company:
Address: Calle Larios 1
City: Málaga
Zip Code: 29015
Country: France

Shipping

First Name: Ana
Last Name: Álvarez
Email: ana.alvarez@example.com
Phone: +341111111118
Company:
Address: Calle Larios 1
City: Málaga
Zip Code: 29015
Country: France

(b) Campos de un pedido creado manualmente por el usuario.

Figura 3.14: Campos editables de un pedido dependiendo de su origen.

En pedidos creados manualmente, se pueden añadir o quitar productos al pedido, así como editar los productos ya existentes. Para realizar estas acciones se debe pulsar el icono de edición que se encuentra en la parte superior derecha de la tabla de productos. Esto abrirá el modal que se muestra en la figura 3.15, que permite añadir nuevos productos al pedido, así como editar la cantidad y el precio de los ya existentes, o hasta eliminarlos del pedido.

Este modal incluye características bastante complejas, como es la búsqueda de productos dinámicamente. A medida que el usuario escribe en el campo de búsqueda el nombre o SKU del producto, se realiza una petición al *backend* para obtener los productos que coinciden con la búsqueda. Solo los productos que se encuentran en el canal de venta del pedido se mostrarán en la lista de resultados, lo que asegura que solo se puedan añadir productos

válidos al pedido. Además, para evitar que se hagan demasiadas peticiones al *backend*, se ha implementado un *debounce* de 500 milisegundos, lo que significa que la búsqueda solo se realizará si el usuario deja de escribir durante ese tiempo. Esto reduce la carga en el servidor y mejora la experiencia del usuario al evitar peticiones innecesarias.

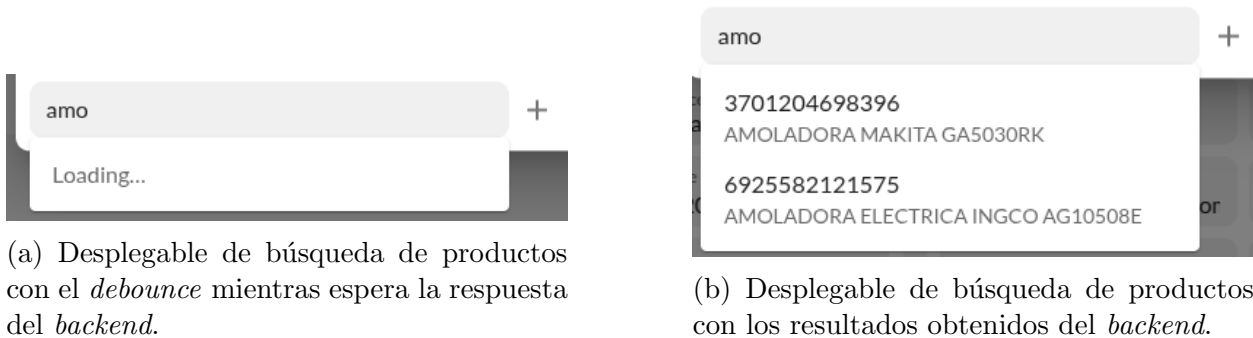


Figura 3.16: Proceso de búsqueda de productos con *debounce*.

Capítulo 4

Conclusiones

Referencias

1. ADOBE. *What is an Ecommerce Platform?* 2021. Disponible también desde: <https://business.adobe.com/blog/basics/ecommerce-platforms>. Visitado: 22/04/2025.
2. SHARETRIBE, Mira Muurinen. *What is a marketplace?* 2024. Disponible también desde: <https://www.sharetribe.com/how-to-build/what-is-a-marketplace/>. Visitado: 25/04/2025.
3. WIKIPEDIA. *Software as a service*. 2025. Disponible también desde: https://es.wikipedia.org/wiki/Software_as_a_service. Visitado: 26/04/2025.
4. PANWAR, Vijay. Web Evolution to Revolution: Navigating the Future of Web Application Development. *International Journal of Computer Trends and Technology*. 2024, vol. 72, págs. 34-40. Disp. desde DOI: [10.14445/22312803/IJCTT-V72I2P107](https://doi.org/10.14445/22312803/IJCTT-V72I2P107).
5. GENDRA, Mariano. *Aplicaciones Web Vs Aplicaciones de Escritorio*. 2021. Disponible también desde: <https://marianogendra.com.ar/Articulos/aplicaciones-web-vs-escritorio>. Visitado: 26/04/2025.
6. MAKE ME A PROGRAMMER, Carlos Schults. *What Is a Programming Framework?* 2022. Disponible también desde: <https://makemeaprogrammer.com/what-is-a-programming-framework/>. Visitado: 28/04/2025.
7. TYPESCRIPT. *TypeScript Documentation*. 2025. Disponible también desde: <https://www.typescriptlang.org/docs/>. Visitado: 28/04/2025.
8. EDTEAM, Alvaro Felipe Chávez. *¿Qué son los lenguajes tipados y no tipados? (Explicación sencilla)*. 2022. Disponible también desde: <https://ed.team/blog/que-son-los-lenguajes-tipados-y-no-tipados-explicacion-sencilla>. Visitado: 28/04/2025.

9. SERVICES, Amazon Web. *¿Cuál es la diferencia entre el front end y back end en el desarrollo de aplicaciones?* [s.f.]. Disponible también desde: <https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend/>. Visitado: 28/04/2025.
10. CÓDIGO FACILITO, Uriel Hernández. *MVC (Model, View, Controller) explicado*. 2015. Disponible también desde: <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado>. Visitado: 28/04/2025.
11. SERVICES, Amazon Web. *What is a database?* [s.f.]. Disponible también desde: <https://aws.amazon.com/what-is/database/>. Visitado: 05/05/2025.
12. SERVICES, Amazon Web. *What's the Difference Between Relational and Non-relational Databases?* [s.f.]. Disponible también desde: <https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/>. Visitado: 05/05/2025.
13. SERVICES, Amazon Web. *¿Qué es SQL (lenguaje de consulta estructurada)?* [s.f.]. Disponible también desde: <https://aws.amazon.com/es/what-is/sql/>. Visitado: 06/05/2025.
14. SERVICES, Amazon Web. *¿Qué es una interfaz de programación de aplicaciones (API)?* [s.f.]. Disponible también desde: <https://aws.amazon.com/es/what-is/api/>. Visitado: 06/05/2025.