UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

TRABAJO FIN DE ESTUDIOS

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

Sumario

Índice de figuras

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.

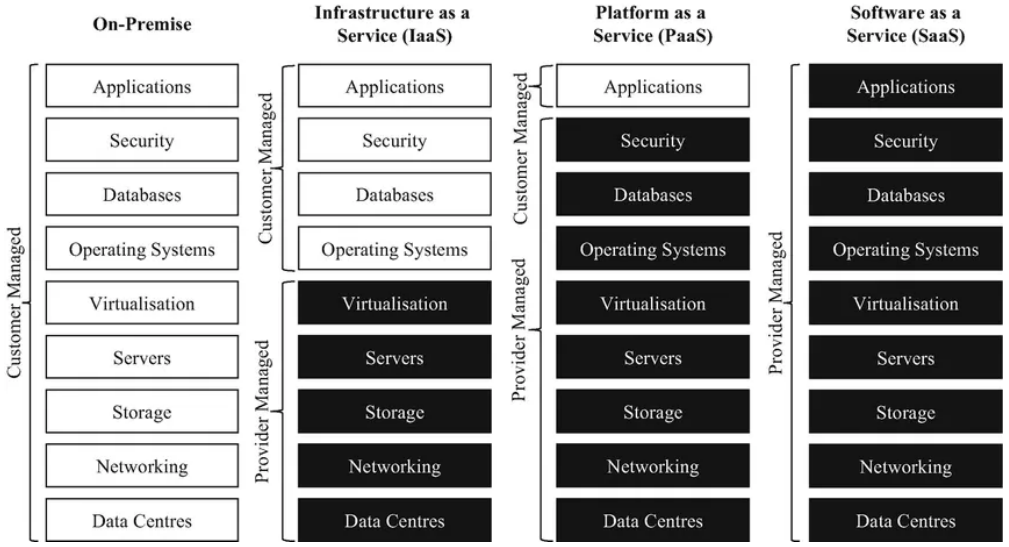

Figura 2.1: Modelos de distribución de software. Fuente: [3]

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

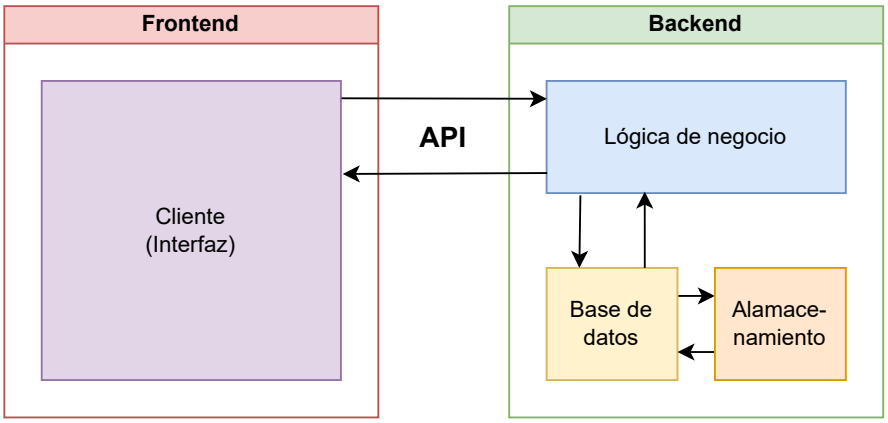

Figura 2.2: Arquitectura de una aplicación web.

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

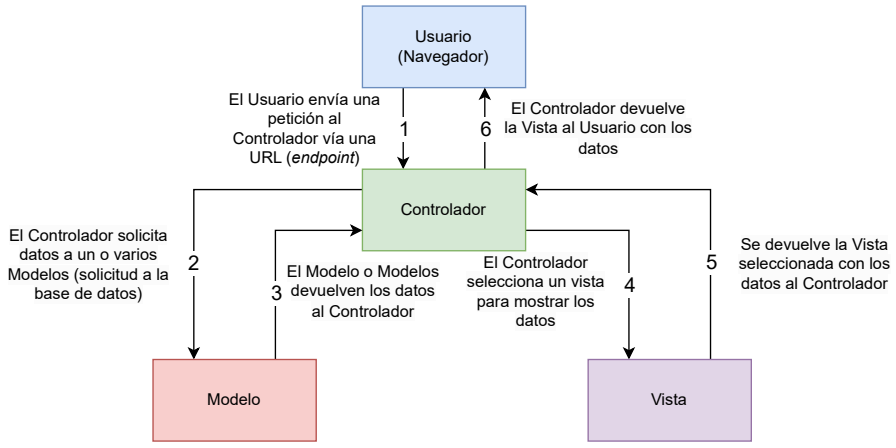

Figura 2.3: Diagrama de flujo del patrón de diseño MVC.

(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 [13]. No

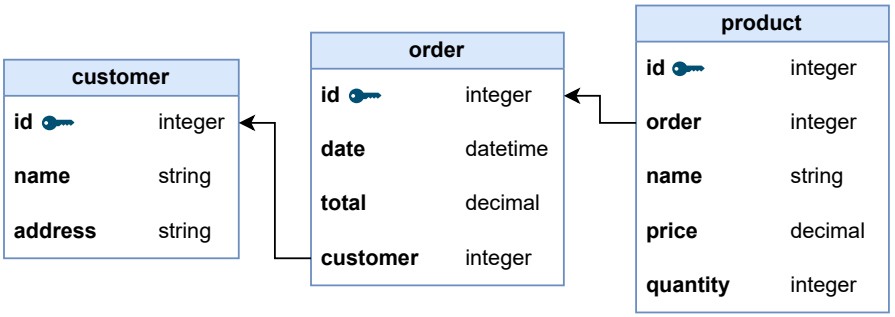

Figura 2.4: Tablas de ejemplo de una base de datos para guardar pedidos.

obstante, realizar consultas SQL directamente puede ser complicado y propenso a errores, especialmente en aplicaciones más complejas. 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`.

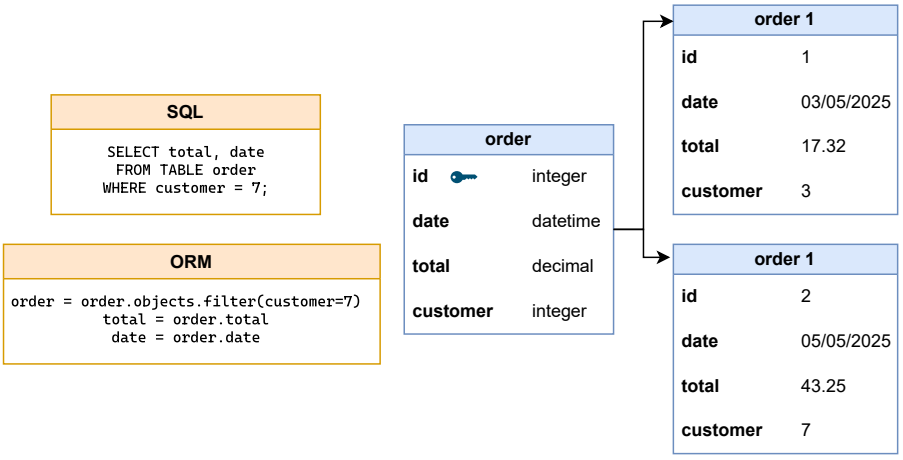

Figura 2.5: Ejemplo de consulta SQL y su equivalente en ORM para la obtención de dos campos de un registro de una tabla.

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* actua como cliente y el *marketplace* actua 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.

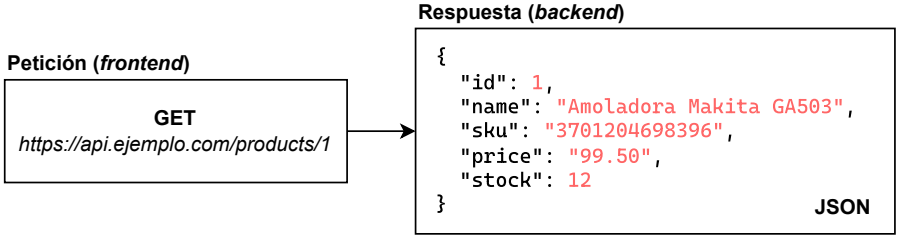

Figura 2.6: Ejemplo de una petición a una API REST con la correspondiente respuesta en formato JSON.

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.

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

Capítulo 3

Análisis Comercial y de Competencia

Capítulo 4

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.

4.1. 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.

4.1.1. 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.

4.2. 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.

4.2.1. 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:** 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 4.1 se puede observar un caso de uso para lograr un mayor en 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.

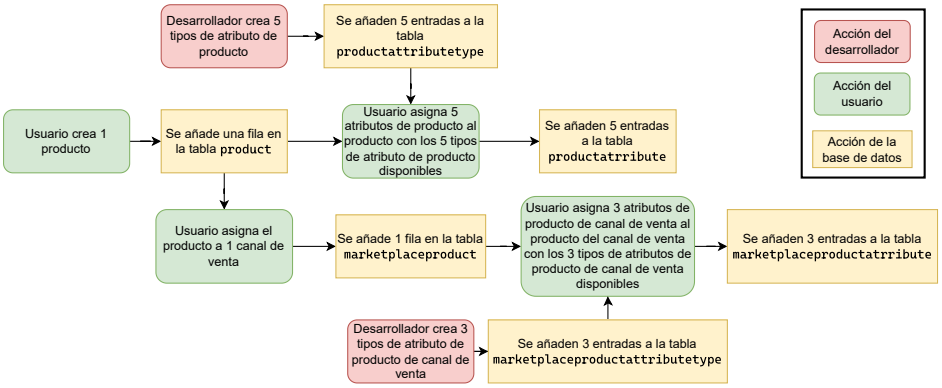

Figura 4.1: Esquema de flujo de creación de un producto y su vinculación con un canal de venta.

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 "Númeroz su entrada tendrá todas las columnas vacías exceptuando la columna `data_int`.

4.3. Desarrollo del *backend*

4.4. Desarrollo del *frontend*

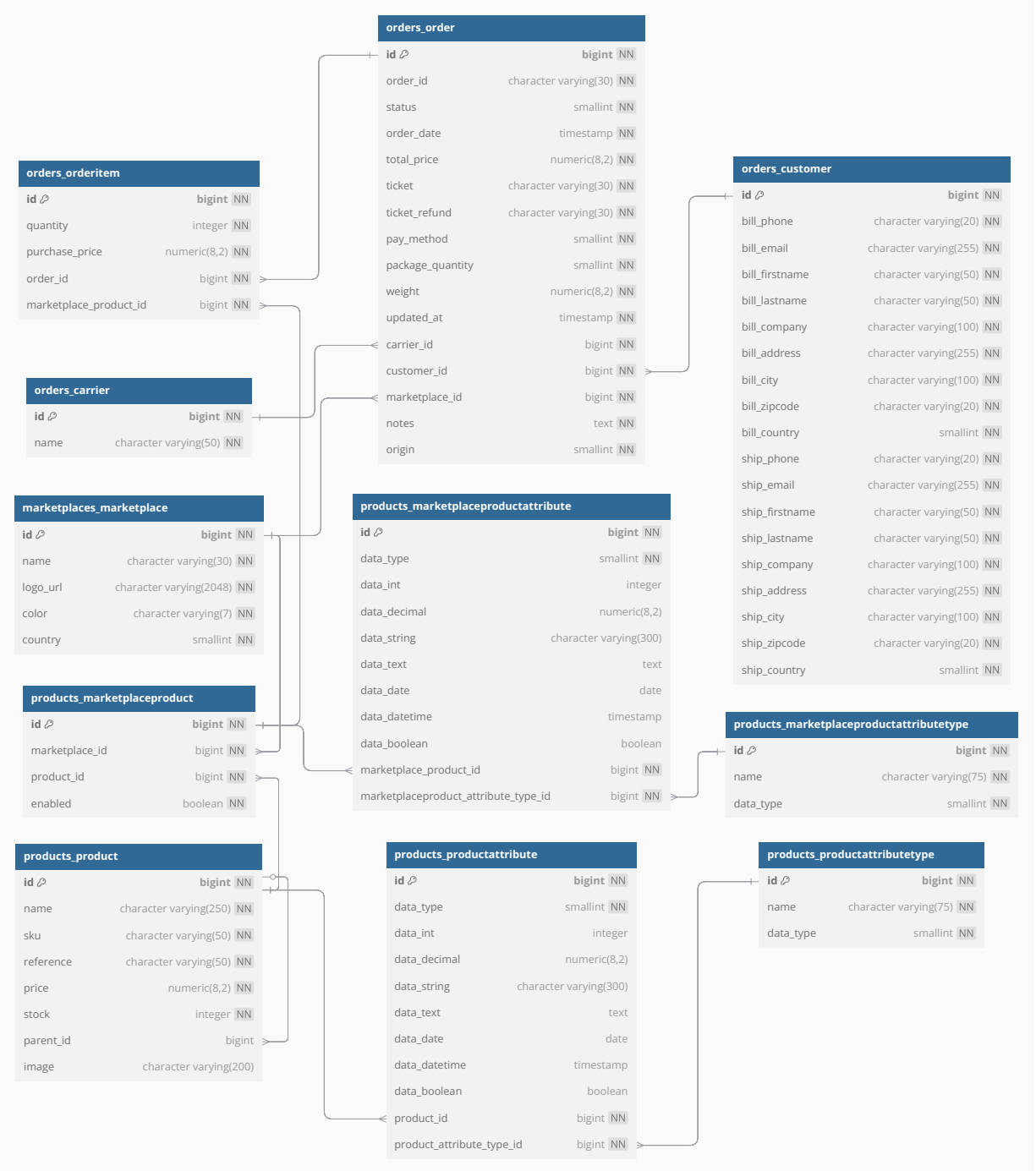

Figura 4.2: Diagrama de la base de datos

Capítulo 5

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.
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.