Universidad de Cantabria

Facultad de Ciencias

APLICACIÓN PARA LA GESTIÓN DE ALMACENES E INVENTARIOS

(Application for warehouse and inventory management)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Adrián San Luis Torre

Director: Pablo Sánchez Barreiro

Julio - 2025

Índice

Resumen 3
Abstract 3
1 Introducción, Objetivos y Metodología 4
1.1 Introducción 4
1.2 Objetivos 4
1.3 Metodología de Desarrollo e Infraestructura 4
2 Ingeniería de Requisitos 7
2.1 Consideraciones previas 7
2.2 Captura de requisitos 7
2.3 Requisitos funcionales 7
2.4 Requisitos no funcionales 10
3 Arquitectura y Diseño 11
3.1 Arquitectura 11
3.2 Modelo de Dominio 12
3.3 Diseño de la base de datos 15
3.4 Diseño de API REST 15
4 Implementación 17
4.1 Router Angular 17
4.2 Añadir nuevo producto 19
4.2.1 Interfaz de usuario 19
4.2.2 Controlador 22
4.3 Historial de pedidos entrantes 25
4.3.1 Interfaz de usuario 25
4.3.2 Controlador 28
5 Pruebas 30
6 Conclusiones 35
Referencias 37

Resumen

El objetivo de este trabajo es desarrollar una aplicación web para la gestión de almacenes e inventarios. La finalidad del proyecto es adquirir experiencia práctica en la creación de una solución informática completa, abarcando tanto el frontend, desarrollado con Angular, como el backend, implementado con Node.js y MySQL. La aplicación permite registrar productos, controlar su entrada y salida, gestionar pedidos y mantener un historial de operaciones, ofreciendo una herramienta funcional para la administración del inventario.

Abstract

The objective of this project is to develop a web application for warehouse and inventory management. The purpose of the project is to gain practical experience in building a complete software solution, encompassing both the frontend, developed with Angular, and the backend, implemented with Node.js and MySQL. The application allows for product registration, tracking of incoming and outgoing stock, order management, and maintenance of an operation history, providing a functional tool for inventory administration.

1 Introducción, Objetivos y Metodología

1.1 Introducción

Este Trabajo Fin de Grado tiene como objetivo el desarrollo de un sistema informático web para la gestión de inventarios de empresas y comercios. Se trata de un sistema ficticio, cuyos requisitos fueron acordados con mi tutor e ideado con la finalidad, principalmente, de adquirir experiencia práctica en la construcción de una aplicación web completa, abarcando tanto la parte del cliente como la del servidor.

En un principio, iba a hacer el TFG con la empresa en la que hice las prácticas extracurriculares, pero por un tema de protección de datos no pude, así que opté por realizar este trabajo.

1.2 Objetivos

El objetivo de este proyecto es desarrollar una pequeña aplicación para la gestión del inventario del almacén de una empresa. Concretamente, la aplicación debe ser capaz de realizar las siguientes funcionalidades:

- Dar de alta nuevos productos.
- Controlar la salida de productos.
- Registrar la llegada de nuevas unidades de productos ya existentes.
- Realizar búsquedas de productos en base a distintos criterios.
- Listar productos con stock bajo o cercano al umbral mínimo.
- Registrar la llegada de pedidos.
- Tener un historial de los pedidos.
- Realizar pedidos automáticamente cuando un producto se encuentre por debajo de su umbral.

1.3 Metodología de Desarrollo e Infraestructura

Para llevar a cabo este objetivo y cumplir con las funcionalidades requeridas, hemos utilizado la herramienta *Trello*[1], con la que hemos llevado un seguimiento de las tareas, asignándoles distintos estados según su fase de desarrollo y moviéndolas progresivamente a medida que se completaban, permitiendo así un control del avance del proyecto. La figura 1 muestra el estado del tablero *Trello* en un momento concreto del trabajo. Este tablero está formado por cuatro columnas, que representan los diferentes estados de las

[1] https://trello.com

tareas. *Lista de tareas* contiene las tareas que aún no se están desarrollando, *En proceso* incluye las tareas en las que se está trabajando, *Revisar* agrupa las tareas que ya han sido desarrolladas y que están esperando a ser revisadas de forma conjunta con mi tutor, y, por último, *Hecho* muestra las tareas completadas.

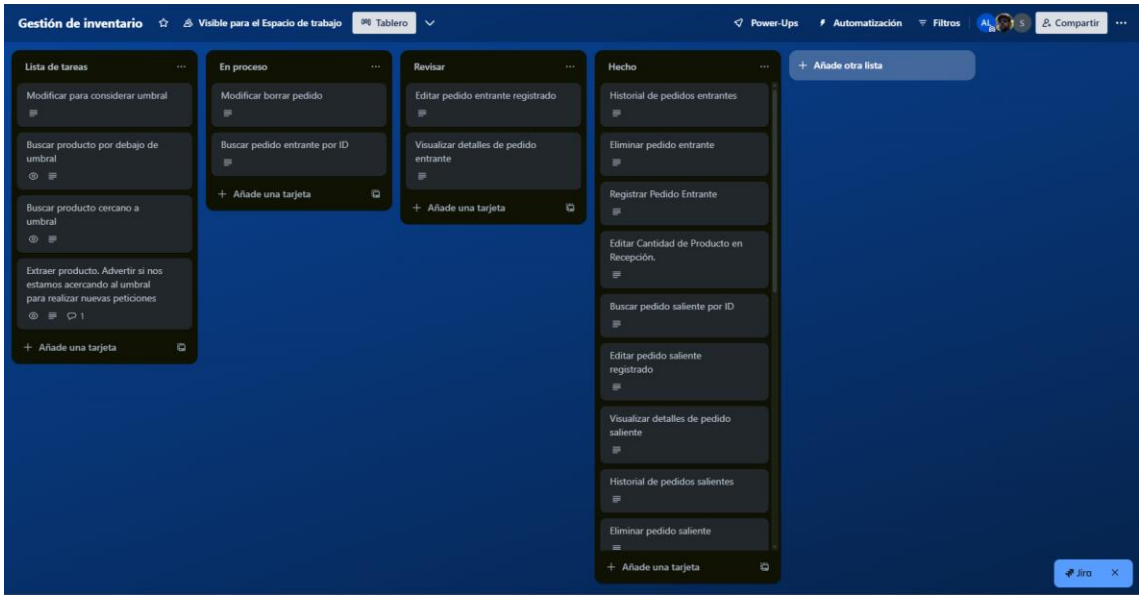


Figura 1. Tablero Trello

Hemos mantenido un seguimiento semanal del tablero, a través de reuniones con mi tutor, Pablo, donde hemos revisado el progreso de las tareas, resuelto dudas y ajustado la planificación en función de los avances y posibles imprevistos.

Las tareas de este tablero se han especificado como *historias de usuario* [1] e incluyen, además de una descripción, las pruebas de aceptación para dicha tarea. La figura 2 muestra un ejemplo de historia de usuario.

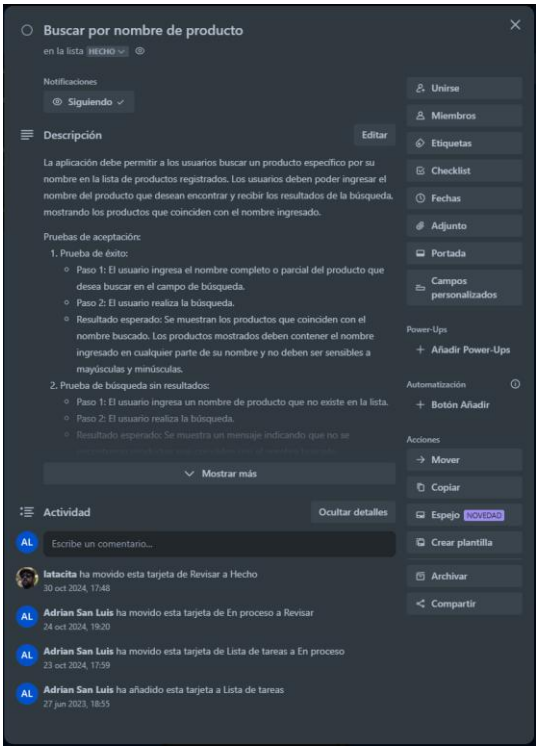


Figura 2. Historia de usuario

Para hablar de la tecnología utilizada, podemos dividir el proyecto en tres partes:

- Para el desarrollo del frontend se ha empleado Angular [2]. No tenía grandes conocimientos sobre este framework, así que este trabajo me ha servido para ampliar mis capacidades de desarrollo en este aspecto. Además, se han realizado pruebas unitarias utilizando Karma [3] junto con Jasmine [4].

- Para el desarrollo del backend he decidido usar Node.js [5]. Una de las principales razones para elegir Node es su entorno de ejecución basado en JavaScript, el mismo lenguaje utilizado en el desarrollo del frontend. En este caso, las pruebas se han llevado a cabo utilizando Jest [6].

- Para almacenar los productos y pedidos, se ha creado una base de datos SQL. Se ha utilizado MySQL [7] como gestor de base de datos. Este ha sido implementado a través de XAMPP [8], un entorno de desarrollo local.

- A lo largo del desarrollo se ha utilizado Git [9] para el control de versiones[2], permitiendo un seguimiento organizado de los cambios realizados y facilitando la gestión del proyecto de forma estructurada, realizando los distintos desarrollos en ramas separadas.

[2] El código completo de este proyecto está disponible en los siguientes repositorios. https://github.com/adriansanluis2000/Front y https://github.com/adriansanluis2000/Back

2 Ingeniería de Requisitos

2.1 Consideraciones previas

La aplicación contempla dos tipos de pedidos: entrantes y salientes. Los pedidos entrantes son aquellos que un comprador nos hace. Por el contrario, los pedidos salientes son los que hacemos nosotros para reponer stock en nuestro almacén.

Por lo demás, el dominio de la aplicación es bastante intuitivo y no tiene ninguna característica especial que deba ser explicada.

2.2 Captura de requisitos

Al ser un producto ficticio, el proceso de captura de requisitos no ha existido, ya que no disponíamos de fuentes reales y accesibles de las cuales extraer requisitos. Por tanto, los requisitos fueron simplemente consensuados entre mi director del Trabajo Fin de Grado y yo.

2.3 Requisitos funcionales

En cuanto a los requisitos funcionales, se han especificado como *historias de usuario*. En total, el proyecto ha contado con 28 historias de usuario[3].

A continuación, explicaremos brevemente un par de historias de usuario para ilustrar su contenido.

[3] https://trello.com/b/c39uPf2r/gestion-de-inventario

Registrar Pedido Entrante

La aplicación debe permitir a los usuarios registrar los productos enviados a clientes mediante pedidos entrantes. Al registrar un pedido entrante, las unidades correspondientes deben descontarse automáticamente del inventario. Cada pedido debe quedar registrado en la aplicación para su trazabilidad.

Pruebas de aceptación:

1. Prueba de Éxito: Registrar Pedido Entrante

- **Escenario**: El usuario registra un pedido entrante con éxito.
  a. **Acción**: El usuario selecciona "Pedidos → Registrar pedido entrante".
     - **Resultado Esperado**: El sistema abre el formulario o ventana de recepción de un nuevo pedido.
  b. **Acción**: El usuario busca un producto en el desplegable.
     - **Resultado Esperado**: El sistema muestra el listado de productos disponibles en el desplegable y permite seleccionar uno.
  c. **Acción**: El usuario indica la cantidad recibida del producto.
     - **Resultado Esperado**: El sistema permite al usuario ingresar una cantidad válida (número entero positivo menor o igual al stock disponible).
  d. **Acción**: El usuario añade el producto al pedido.
     - **Resultado Esperado**: El sistema añade el producto y la cantidad al pedido actual, verifica el stock disponible, y permite continuar con el registro.
  e. **Repetir los pasos 2-4 para cada producto que se desee añadir al pedido**.
  f. **Acción**: El usuario finaliza el pedido.
     - **Resultado Esperado**: El sistema guarda el pedido completo en la base de datos y actualiza las cantidades de inventario de cada producto en el pedido.

2. Prueba de Error por Cantidad Negativa

- **Escenario**: El usuario intenta registrar una cantidad negativa para un producto.
- **Acción**: El usuario selecciona un producto y especifica una cantidad negativa.
- **Resultado Esperado**: El sistema muestra un mensaje de error indicando que la cantidad no puede ser negativa y evita que el producto sea añadido al pedido hasta que se ingrese una cantidad válida.

3. Prueba de Error por Stock Insuficiente

- **Escenario:** El usuario intenta registrar una cantidad mayor al stock disponible.
- **Acción:** El usuario selecciona un producto y especifica una cantidad mayor al inventario disponible.
- **Resultado Esperado:** El sistema muestra un mensaje de error indicando que la cantidad solicitada supera el stock disponible y no permite añadir el producto al pedido hasta que se corrija.

4. Prueba de Error por Cantidad No Numérica

- **Escenario**: El usuario intenta ingresar un valor no numérico en el campo de cantidad.
- **Acción**: El usuario selecciona un producto y escribe texto u otro carácter no numérico en la cantidad.
- **Resultado Esperado**: El sistema muestra un mensaje de error indicando que la cantidad debe ser un número, y no permite añadir el producto al pedido hasta que se corrija el valor.

5. Prueba de Error por Fallo de Conexión a Internet

- **Escenario**: Se interrumpe la conexión a Internet durante el proceso de registro del pedido.
- **Acción**: El usuario selecciona o intenta añadir productos al pedido mientras no hay conexión.
- **Resultado Esperado**: El sistema notifica al usuario sobre la falta de conexión y guarda el progreso del pedido localmente. Una vez que se restablece la conexión, el sistema intenta completar el registro del pedido en la base de datos.

6. Prueba de Error por Fallo en la Base de Datos

- **Escenario**: Ocurre un error en la base de datos al intentar guardar el pedido.
- **Acción**: El usuario finaliza el pedido, pero el sistema no puede registrar los datos en la base de datos.
- **Resultado Esperado**: El sistema muestra un mensaje de error al usuario, indicando el problema con la base de datos, y permite reintentar la operación o almacenar temporalmente el pedido para intentarlo más tarde.

Listado 1. Registrar Pedido Entrante

Durante el proceso de registro de pedidos entrantes (listado 1), el sistema debe permitir al usuario seleccionar productos desde un menú desplegable, indicar una cantidad válida, y añadirlos al pedido. Una vez completado, el pedido se almacena en la base de datos y se

actualiza el inventario descontando las unidades correspondientes. En caso de que el usuario introduzca una cantidad negativa, el sistema debe mostrar un mensaje de error e impedir que se añada el producto hasta que se corrija el valor. Si se introduce una cantidad superior al stock disponible, el sistema también debe advertir al usuario y no permitir el registro hasta que se indique una cantidad válida. Asimismo, si el usuario introduce un valor no numérico en el campo de cantidad, el sistema debe rechazarlo con un mensaje de error pidiendo un número, aunque esto no es posible ya que el campo ha sido definido con *type="number"*. Finalmente, si durante el proceso se pierde la conexión a internet u ocurre un fallo al guardar el pedido en base de datos, la aplicación debe informar al usuario, guardar el progreso localmente y reintentar el guardado una vez restablecida la conexión.

Editar Cantidad de Producto en Recepción.

Este caso de uso permite al usuario editar la cantidad de un producto añadido previamente en el pedido antes de finalizarlo. Al cambiar la cantidad, el sistema actualiza el pedido automáticamente. Si la cantidad editada es menor que 1, el producto se elimina del pedido.

Pruebas de aceptación:

1. Prueba de Éxito: Editar Cantidad de Producto en el Pedido

- **Escenario**: El usuario modifica la cantidad de un producto en el pedido con éxito.
- **Acción**: El usuario selecciona un campo de cantidad para un producto añadido y cambia el valor a un número positivo válido.
- **Resultado Esperado**: El sistema actualiza la cantidad del producto en el pedido en tiempo real y refleja el cambio en el total del pedido.

2. Prueba de Error por Cantidad Menor a 1

- **Escenario**: El usuario intenta ingresar una cantidad menor a 1 para un producto en el pedido.
- **Acción**: El usuario ingresa 0 o un valor negativo en el campo de cantidad.
- **Resultado Esperado**: El sistema llama al método quitarProducto() y elimina el producto del pedido. La lista de productos en el pedido se actualiza y el total refleja el cambio.

3. Prueba de Error por Cantidad No Numérica

- **Escenario**: El usuario intenta ingresar un valor no numérico en el campo de cantidad.
- **Acción**: El usuario ingresa texto o caracteres no numéricos en el campo de cantidad.
- **Resultado Esperado**: El sistema impide el cambio, muestra un mensaje de advertencia o evita que se ingrese el valor no numérico, manteniendo la cantidad anterior válida en el pedido.

4. Prueba de Error por Cantidad con Decimales

- **Escenario**: El usuario ingresa un valor decimal para la cantidad.
- **Acción**: El usuario intenta ingresar un número decimal (ej., 1.5) en el campo de cantidad.
- **Resultado Esperado**: El sistema redondea el número a un entero o muestra un mensaje de error indicando que solo se permiten cantidades enteras.

5. Prueba de error por exceder stock disponible de un producto

- **Escenario**: El usuario ingresa un valor superior al stock.
- **Acción**: El usuario intenta ingresar una cantidad para un producto que supera el stock disponible.
- **Resultado Esperado**: El sistema muestra un mensaje de error indicando que la cantidad solicitada supera el stock disponible y reduce la cantidad al máximo disponible.

Listado 2. Editar Cantidad de Producto en Recepción

Durante el proceso de registro de un pedido, el sistema permite al usuario modificar la cantidad de cualquier producto previamente añadido (listado 2). Si el usuario introduce un número entero positivo válido, el sistema actualiza automáticamente la cantidad del

producto y recalcula el total del pedido. En caso de que ingrese una cantidad menor a 1, ya sea cero o negativa, el sistema pregunta al usuario si quiere eliminar el producto del pedido y, en caso afirmativo, actualiza la lista y el total correspondiente. Si el usuario intenta ingresar un valor no numérico, como texto o símbolos, el sistema impide la modificación, mantiene la cantidad anterior válida y muestra una advertencia. Cuando se introduce un valor decimal, el sistema debe mostrar un mensaje de error o redondear el número a un valor entero, según la configuración establecida. Por último, si la cantidad editada supera el stock disponible del producto, el sistema debe notificar al usuario mediante un mensaje de error y ajustar automáticamente la cantidad al máximo permitido según el inventario actual.

2.4 Requisitos no funcionales

En cuanto a los requisitos no funcionales, se revisó la norma ISO 25010 [10], llegándose a la conclusión de que la aplicación no tenía que satisfacer ningún requisito no funcional de manera especial. Es decir, el sistema debía tener buen rendimiento, usabilidad o mantenibilidad en la misma medida que cualquier otra aplicación informática adecuadamente diseñada, pero sin que existiese ninguna característica o restricción concreta que la aplicación tuviese que satisfacer en especial.

3 Arquitectura y Diseño

3.1 Arquitectura

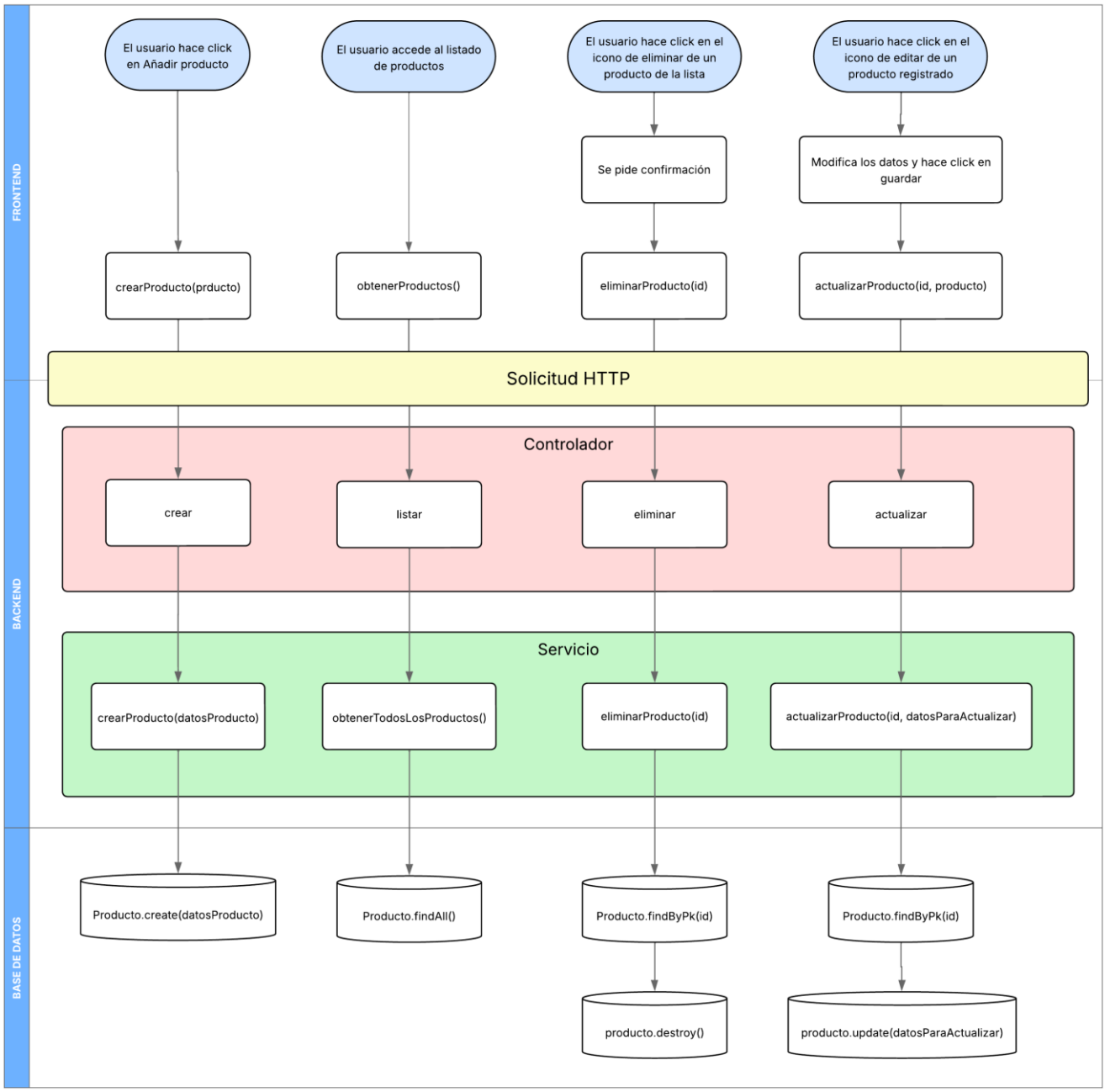


Figura 3. Esquema de Procesos de Producto

La figura 3 ilustra la arquitectura de la aplicación. Para hacerlo nos centraremos únicamente en los procesos de Producto, ya que lo que nos interesa es mostrar la comunicación entre el frontend, el backend y la base de datos.

La aplicación sigue una arquitectura en capas que separa la lógica del cliente, implementado en Angular, la lógica del servidor, implementado en Node.js, y el acceso a la base de datos, realizado mediante Sequelize [11].

El flujo comienza cuando el usuario selecciona *Añadir producto*. Esto desencadena la ejecución del método `crearProducto` del servicio *ProductoService*, que realiza una petición HTTP al servidor.

En el backend, dicha solicitud es gestionada por el controlador, que se encarga de recibir los datos y delegar la operación al servicio de *Producto*, donde reside la lógica de negocio. El servicio valida y procesa la información antes de interactuar con la base de datos.

Para asegurar la persistencia de los datos, se utiliza Sequelize como ORM (Object-Relational Mapper). Desde el servicio, se llama a los métodos del elemento de dominio Producto para consultar, crear, actualizar o eliminar información en la base de datos relacional.

Una vez completada la operación, la respuesta viaja en orden inverso hasta llegar al frontend, donde Angular se encarga de reflejar los cambios en la interfaz de usuario.

3.2 Modelo de Dominio

La figura 4 muestra el modelo de dominio de la aplicación. La aplicación se estructura en torno a tres entidades fundamentales: Producto, Pedido y Solicitud.

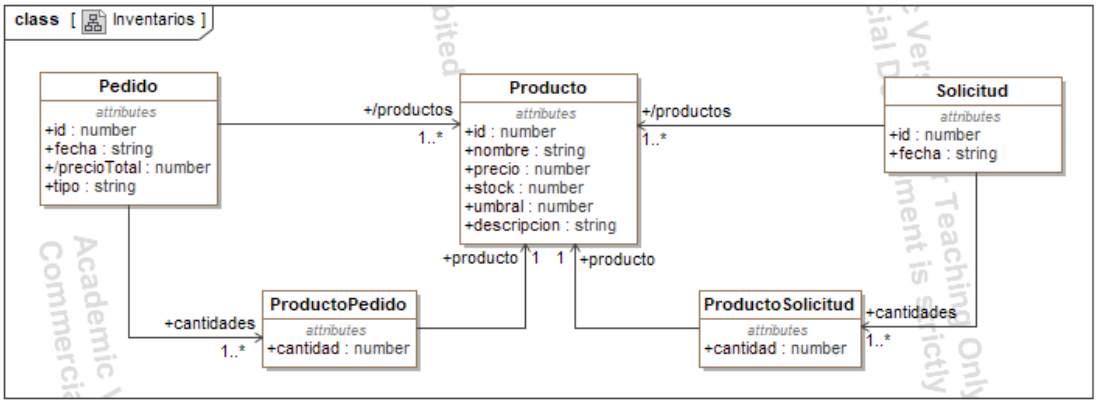


Figura 4. Modelo de Dominio

Producto

El elemento *Producto* representa los artículos disponibles en el inventario. Cada producto contiene información clave para su gestión y control de stock. Posee los siguientes campos:

- **id:** Identificador único del producto. Es un entero que se autoincrementa.
- **nombre:** Nombre del producto. Es una cadena única y obligatoria.
- **precio:** Precio unitario del producto, representado como un número decimal con valor mínimo de 0,01.

- **stock:** Cantidad disponible del producto en el almacén. Es un número entero con valor mínimo 0.
- **umbral:** Cantidad mínima del producto que puede haber en el almacén. Se usa como referencia para mostrar alertas o realizar solicitudes. También es un entero mayor o igual que 0.
- **descripción:** Campo de texto opcional con una descripción detallada del producto.

Este modelo es la base para la gestión de inventario, ya que cada pedido o solicitud está compuesto por uno o más productos.

Pedido

El elemento *Pedido* representa una petición de productos, que puede ser de tipo *entrante* (venta o salida de stock) o *saliente* (ingreso de stock). Posee los siguientes campos:

- **id:** Identificador único del pedido.
- **fecha:** Fecha y hora en la que ha sido realizado el pedido. Por defecto, se registra la fecha actual.
- **precioTotal:** Suma del precio total de los productos incluidos en el pedido.
- **tipo:** Indica si el pedido es *entrante* o *saliente*.

ProductoPedido es una clase intermedia que permite especificar la cantidad de un producto dentro de un pedido determinado.

Solicitud

La entidad *Solicitud* representa un pedido generado cuando el stock de uno o varios productos cae por debajo del umbral mínimo definido. Su finalidad es reabastecer los productos con stock bajo para evitar quedarse sin existencias. Posee los siguientes campos:

- **id:** Identificador único de la solicitud.
- **fecha:** Fecha y hora en la que se ha generado la solicitud. Por defecto, se asigna la fecha actual.

ProductoSolicitud es una clase que permite especificar la cantidad de cada producto en cada solicitud.

El listado 3 muestra la definición de la entidad de dominio Producto utilizando Sequelize, que permite interactuar con bases de datos relacionales mediante objetos JavaScript. En este modelo, se especifican los campos de un producto y se incluyen validaciones que

garantizan la integridad de los datos. Además, se imponen restricciones de tipo de dato y cantidad mínima a los valores numéricos, y se establecen ciertos campos como obligatorios.

```
1   const Producto = sequelize.define('Producto', {
2     id: {
3       type: DataTypes.INTEGER,
4       primaryKey: true,
5       autoIncrement: true
6     },
7     nombre: {
8       type: DataTypes.STRING,
9       allowNull: false,
10       validate: {
11         notNull: { msg: 'El nombre del producto es obligatorio' },
12         notEmpty: { msg: 'El nombre del producto no puede estar vacío'},
13       },
14       unique: true
15     },
16     precio: {
17       type: DataTypes.DECIMAL(10, 2),
18       allowNull: false,
19       validate: {
20         notNull: { msg: 'El precio del producto es obligatorio' },
21         isFloat: { msg: 'El precio debe ser un número válido' },
22         min: {
23           args: [0.01],
24           msg: 'El precio mínimo debe ser al menos 0.01'
25         }
26       }
27     },
     ...
28   });
29
30   module.exports = Producto;
```

Listado 3. Entidad Producto

En el listado 3 se muestran varios campos de un *Producto*. El id (líneas 2-6) es un identificador único de cada producto y se genera automáticamente, el nombre (líneas 7-15) es obligatorio, debe ser único y no puede estar vacío y el precio (líneas 16-27) debe ser un número decimal válido, con un mínimo de 0,01.

Para guardar un nuevo producto en la base de datos, se utiliza el método `create()` que ofrece Sequelize. Este método genera automáticamente una consulta *INSERT* para añadir el producto. Además, Sequelize aplica las validaciones definidas en el modelo. También podemos obtener información almacenada de forma sencilla con los métodos `findAll()`, que obtiene todos los productos registrados, o `findByPk(id)`, que busca un producto por su clave primaria (id).

3.3 Diseño de la base de datos

Para la gestión y almacenamiento de los productos y pedidos se ha utilizado una base de datos SQL, con un gestor MySQL. La estructura de la base de datos se ha generado de forma automática mediante el uso de Sequelize, un ORM para Node.js que permite definir los modelos de datos directamente en el código JavaScript.

A través de Sequelize, se define cada modelo, por ejemplo, *Producto*, con sus respectivas propiedades, tipos de datos, validaciones y restricciones. Posteriormente, Sequelize se encarga de sincronizar estos modelos con la base de datos y autogenerar las tablas correspondientes. Esto facilita el mantenimiento y la evolución del esquema sin necesidad de escribir sentencias SQL manuales.

3.4 Diseño de API REST

El diseño de la API REST del proyecto sigue una estructura que separa los recursos principales en rutas independientes para productos, pedidos y solicitudes.

El diseño de las rutas se ha realizado utilizando las peticiones HTTP adecuadas (GET, POST, PUT, DELETE) para cada tipo de operación.

Las rutas están agrupadas de la siguiente forma:

/api/productos	→ Gestión de productos
/api/pedidos	→ Gestión de pedidos entrantes y salientes
/api/solicitudes	→ Gestión de solicitudes de reposición
/verificar-nombre	→ Verificación única de nombre de producto

A continuación, se muestra un ejemplo de funcionamiento de dos endpoints.

Endpoint 1: Crear un producto

- **Ruta:** POST /api/productos
- **Descripción:** Añade un nuevo producto al inventario
- **Cuerpo de la petición (JSON):**

```
{
  "nombre": "Producto1",
  "precio": 9.99,
  "stock": 50,
  "umbral": 10,
  "descripcion": "Descripción del producto"
}
```

- **Respuesta exitosa (201):**

```
{
  "id": 1,
  "nombre": "Producto1",
  "precio": 9.99,
  "stock": 50,
  "umbral": 10,
  "descripcion": "Descripción del producto"
}
```

- **Errores comunes: 400:** La solicitud enviada al servidor es incorrecta o malformada.

Endpoint 2: Obtener historial de pedidos entrantes

- **Ruta:** GET /api/pedidos?tipo=entrante
- **Descripción:** Devuelve una lista de pedidos filtrados por tipo
- **Parámetros de consulta:** Tipo (opcional)
- **Respuesta exitosa (201):**

```
[
    {
        "id": 7,
        "fecha": "2025-03-12T16:36:50.000Z",
        "precioTotal": "300.00",
        "tipo": "entrante",
        "Productos": [
            {
                "id": 2,
                "nombre": "Producto2",
                "precio": "50.00",
                "stock": 13,
                "umbral": 20,
                "descripcion": "",
                "ProductoPedido": {
                    "cantidad": 6
                }
            }
        ]
    },
  ...
]
```

- **Errores comunes: 500:** Error interno del servidor.

La tabla 1 muestra los distintos endpoints que conforman la aplicación.

Recurso	Método	Código de respuesta
/productos	POST	201, 400
/productos	GET	200, 404
/productos/{id}	GET	200, 404
/productos/{id}	PUT	200, 404
/productos/{id}	DELETE	200, 404
/pedidos	POST	201, 500
/pedidos	GET	200, 500
/pedidos/{id}	GET	200, 404, 500
/pedidos/{id}	DELETE	200, 404, 500
/pedidos/{id}	PUT	200, 404, 500
/pedidos/devolver-stock/{id}	POST	200, 500
/solicitudes	POST	201, 400
/solicitudes	GET	200, 500
/solicitudes/{id}	DELETE	200, 400
/solicitudes/{id}	PUT	200, 400
/verificar-nombre	GET	200, 500

Tabla 1. Endpoints de la aplicación

4 Implementación

En este apartado describiremos los diferentes elementos que conforman la implementación de nuestra aplicación.

4.1 Router Angular

El archivo app.routes.ts es una parte esencial de la configuración de enrutamiento en una aplicación Angular. Su objetivo principal es definir las rutas que el usuario puede visitar dentro de la aplicación y asociar cada una de ellas con el componente correspondiente que debe visualizarse en esa ruta. Este archivo cobra especial importancia en el desarrollo de un SPA (*Single Page Application*) [12], ya que permite cargar dinámicamente los componentes sin necesidad de recargar toda la página, lo que mejora la experiencia de usuario.

El listado 4 muestra el fichero de rutas utilizado en este proyecto. Se muestra únicamente una parte del contenido, ya que el resto de rutas son similares.

```
 1   export const routes: Routes = [
 2   {
 3      path: '',
 4      loadComponent: () =>
 5         import('./components/home/home.component').then(m => m.HomeComponent )
 6   },
 7   {
 8      path: 'productos',
 9      loadComponent: () =>
10         import('./components/lista-productos/lista-productos.component')
              .then(m => m.ListaProductosComponent)
11   },
12   {
13      path: 'agregar-producto',
14      loadComponent: () =>
15         import('./components/agregar-producto/agregar-producto.component')
              .then(m => m.AgregarProductoComponent)
16   },
17   {
18      path: 'registrar-pedido-entrante/:pedidoId',
19      loadComponent: () =>
20         import('./components/registrar-pedido-entrante/registrar-pedido-
             entrante.component').then(m => m.RegistrarPedidoEntranteComponent)
21   },
22   {
23      path: 'historial-pedidos-entrantes',
24      loadComponent: () =>
25         import('./components/historial-pedidos-entrantes/historial-pedidos-
             entrantes.component').then(m => m.HistorialPedidosEntrantesComponent)
26   },
27   {
28      path: 'solicitudes-pendientes',
29      loadComponent: () =>
30         import('./components/solicitudes-pendientes/solicitudes-
              pendientes.component').then(m=> m.SolicitudesPendientesComponent)
31   }
32   ];
```

Listado 4. Fichero de rutas de la aplicación

Este archivo define una colección de objetos de tipo *Route*, donde cada objeto especifica:

- **path:** la URL relativa que se asigna a la ruta. Por ejemplo, *productos* o *registrar-pedido-entrante/:pedidoId*.
- **loadComponent:** importa dinámicamente el componente correspondiente. Esta técnica se llama *lazy loading* y permite que los componentes se carguen bajo demanda, mejorando así el rendimiento general de la aplicación.

El uso de :pedidoId indica que la ruta acepta un parámetro dinámico. Esto es útil para editar o visualizar un pedido específico basándose en su identificador.

Las rutas cubren las distintas secciones de la aplicación:

- Página de inicio (‘’). (líneas 2-6)
- Listado de productos (*productos*). (líneas 7-11)
- Formulario para añadir nuevos productos (*agregar-producto*). (líneas 12-16)

- Formularios para editar un pedido entrante existente (*registrar-pedido-entrante/:pedidoId*). (líneas 17-21)
- Historial de pedidos entrantes (*historial-pedidos-entrantes*). (líneas 22-26)
- Sección de solicitudes pendientes (*solicitudes-pendientes*) (líneas 27-31)

A continuación, describiremos la implementación de dos funcionalidades completas del sistema, una de inserción y otra de lectura.

4.2 Añadir nuevo producto

4.2.1 Interfaz de usuario

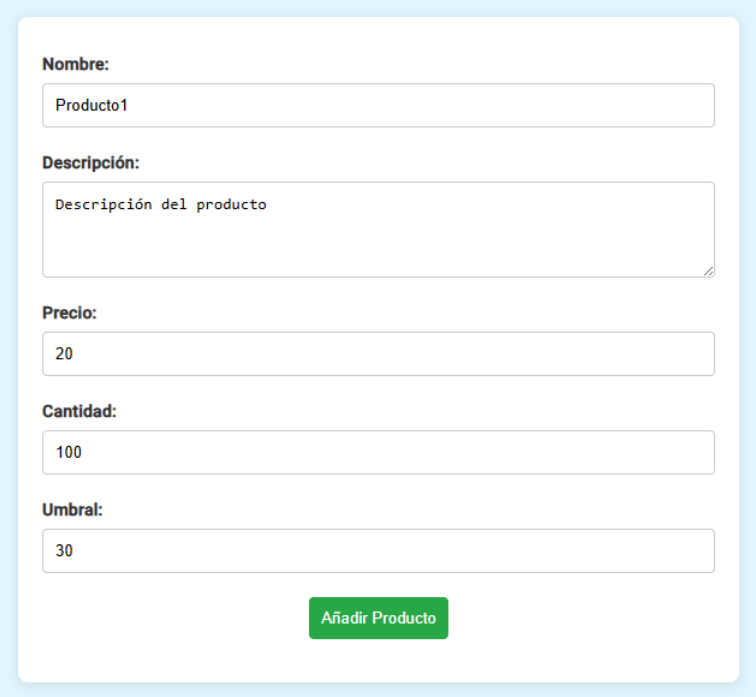


Figura 5. Formulario para añadir un nuevo producto

En la aplicación se ha implementado un formulario para la creación de nuevos productos, el cual permite introducir los datos necesarios para registrar un artículo en el sistema. Este formulario está construido utilizando Angular Forms y se muestra en la figura 5.

El formulario cuenta con validaciones a nivel de plantilla que garantizan la introducción de datos válidos antes de permitir el envío del formulario. Estas validaciones incluyen:

- **Campos obligatorios**: se utiliza el atributo *required* para evitar que se envíen datos incompletos.
- **Restricción de valores mínimos**: los campos numéricos (precio, stock, umbral) deben ser mayores que 0.

- **Mensajes de error personalizados**: cuando un campo no cumple las validaciones y ha sido modificado por el usuario, se muestra un mensaje de error específico que ayuda al usuario a corregir la entrada.

Además, el botón de envío permanece deshabilitado hasta que todos los campos obligatorios son válidos.

Este sistema de validaciones contribuye a una experiencia de usuario más robusta y reduce significativamente el riesgo de introducir datos incorrectos en el sistema.

La figura 6 muestra un caso en el que el formulario no cumple las validaciones definidas y se muestra un mensaje de error al usuario, además de deshabilitar el botón de *Añadir Producto*.

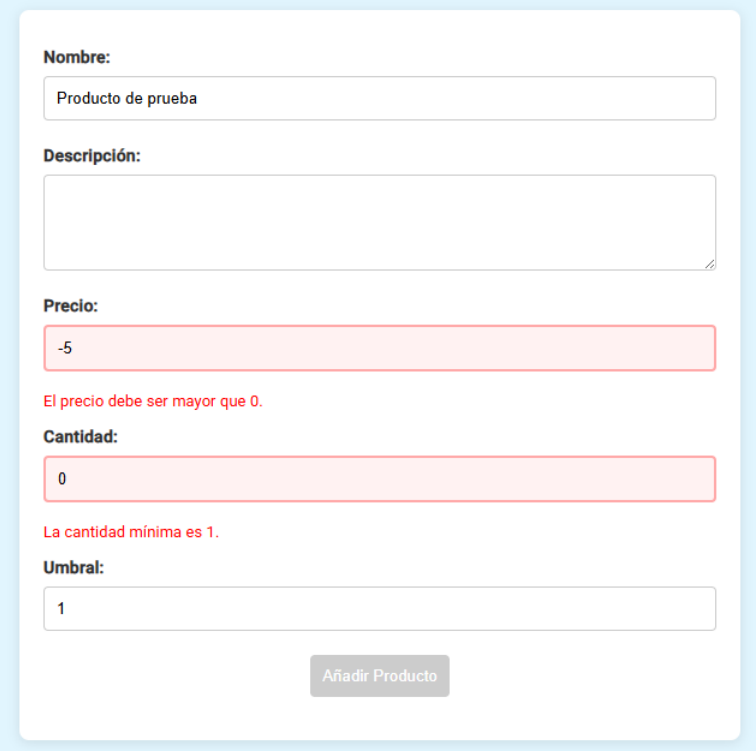


Figura 6. Ejemplo de formulario no válido

Al pulsar el botón *Añadir Producto* se llama al método `agregarProducto()`, el cual se ilustra parcialmente en el listado 5.

```
 1   agregarProducto(form: NgForm) {
 2     this.errorMessage = '';
 3
 4     if (!form.valid) {
 5       Object.keys(form.controls).forEach((field) => {
 6         const control = form.controls[field];
 7         control.markAsTouched({ onlySelf: true });
 8       });
 9     }
10
11    if (this.producto.precio <= 0) {
12      this.errorMessage = 'El precio debe ser mayor que 0.';
13      return;
14    }
15
16    -- Más validaciones --
17
18    this.productoService
19      .verificarNombreProducto(this.producto.nombre)
20      .subscribe({
21        next: (exists) => {
22          if (exists) {
23            this.errorMessage =
24              'El nombre del producto ya existe. Por favor, elige otro nombre.';
25            return;
26          }
27
28          if (form.valid) {
29            this.productoService.crearProducto(this.producto).subscribe({
30              next: (data) => {
31                console.log('Producto añadido:', data);
32                this.resetForm(form);
33              },
34              error: (e) => {
35                if (e.status === 0) {
36                  this.errorMessage =
                     'Error de conexión. Verifica tu conexión a internet y vuelve a intentarlo.';
37                } else {
38                  this.errorMessage =
                     'Error al agregar el producto, inténtalo de nuevo.';
39                }
40              },
41            });
42          }
43        },
44        error: (e) => {
45          this.errorMessage =
              'Error al verificar el nombre del producto. Inténtalo de nuevo.';
46          console.error(e);
47        },
48      });
49  }
```

Listado 5. Método del backend utilizado para añadir un producto a la base de datos

Este método se encarga de gestionar el proceso completo de validación y creación de un nuevo producto. A continuación, se detalla su funcionamiento paso a paso:

- Primero, se limpia cualquier mensaje de error anterior. Esto asegura que posibles errores previos no se mantengan visibles (línea 2).
- Si el formulario no es válido, se recorren todos los campos y se marcan como *touched* (líneas 4-9). Esto fuerza a Angular a mostrar los mensajes de validación en la interfaz, ayudando al usuario a corregir errores.
- A continuación, antes de realizar llamadas al backend, se aplican validaciones relacionadas con las restricciones sobre los datos, como, por ejemplo, que el precio sea mayor que 0.
- Se realiza una petición a través del servicio `ProductoService` para verificar si ya existe otro producto con el mismo nombre en la base de datos (líneas 18-19). Esto es crucial para evitar duplicados y garantizar la integridad del catálogo. Este servicio aparece en el listado 6.

```
verificarNombreProducto(nombre: string): Observable<boolean> {
    return this.http.get<boolean>(`http://localhost:3000/verificar-nombre?nombre=${nombre}`);
}
```

Listado 6. Llamada desde el front para comprobar si un producto existe

- Si el nombre ya existe, se informa al usuario mediante un mensaje de error, si no, se continúa con la creación del producto utilizando el método `crearProducto` del servicio, que envía el producto al backend.
- Finalmente, en caso de éxito, se informa por consola y se resetea el formulario, dejando el componente listo para una nueva entrada. En caso de error, se muestra el mensaje correspondiente al usuario.

4.2.2 Controlador

Una vez que el método `agregarProducto`() en Angular ha verificado que el formulario es válido, ha pasado las validaciones personalizadas y ha confirmado que no existe un producto con el mismo nombre, envía los datos del nuevo producto al backend mediante una petición HTTP POST. Esta petición es recibida por el método crear del controlador productoController.js. Este método se ilustra en el listado 7.

```
 1  exports.crear = async (req, res) => {
 2    try {
 3      const { nombre, precio, stock, umbral } = req.body;
 4
 5      if (!nombre || typeof nombre !== 'string') {
 6        return res.status(400).json({
 7         message: "El nombre es requerido y debe ser una cadena de texto válida"});
 8      }
 9
10      if (!precio || typeof precio !== 'number' || precio <= 0) {
11        return res.status(400).json({
12          message: "El precio es requerido y debe ser un número positivo" });
13      }
14
15      if (!stock || typeof stock !== 'number' || stock <= 0) {
16        return res.status(400).json({
17          message: "La cantidad es requerida y debe ser un número positivo" });
18      }
19
20      if (umbral > stock) {
21        return res.status(400).json({
22          message: "El umbral no puede ser mayor que el stock disponible" });
23      }
24
25      const producto = await productoService.crearProducto(req.body);
26      res.status(201).json(producto);
27
28    } catch (error) {
29      res.status(404).json({
30        message: "Error al procesar la solicitud",
31        error: error.message
32      });
33    }
34  };
```

Listado 7. Método del controlador que valida los datos y llama al servicio correspondiente

El controlador accede a los datos enviados desde el frontend mediante `req.body`. En este caso, se extraen los campos `nombre`, `precio`, `stock` y `umbral` (línea 3).

Aunque en el frontend ya se realizaron validaciones, el backend las replica como medida de seguridad, ya que se podrían crear peticiones HTTP desde fuera de la interfaz del sistema. Cada validación devuelve un error 400 con un mensaje descriptivo si no se cumple (líneas 6, 11, 16 y 21).

Si todos los datos son válidos, se llama al método `crearProducto()` definido en `productoService`, que es el encargado de insertar el producto en la base de datos usando Sequelize (línea 25). Si la operación es exitosa, se devuelve el objeto del producto creado con código de estado 201 Created, indicando que el recurso fue creado correctamente (línea 26).

Cualquier error inesperado, como problemas con la base de datos, fallos de conexión, etc.) se captura con un bloque `try-catch`, devolviendo un mensaje genérico y el detalle del error en la propiedad `error` (líneas 28-33).

Por último, una vez que el controlador ha validado y autorizado la petición de creación de producto, se ejecuta el método `crearProducto(datosProducto)`, el cual aparece en el listado 8.

```
1  async crearProducto(datosProducto) {
2      try {
3          const producto = await Producto.create(datosProducto);
4          return producto;
5      } catch (error) {
6        throw new Error('Error al crear el producto: ' + error.message);
7      }
8  }
```

Listado 8. Método del servicio que crea el producto en base de datos

Para la persistencia, `crearProducto` utiliza la entidad `Producto`, descrita anteriormente, para insertar el nuevo producto. El método `create` inserta el nuevo producto en la base de datos, registrando todos sus campos, y devuelve el objeto completo que incluye el identificador único asignado automáticamente, así como las marcas de tiempo correspondientes a la creación y posible actualización del registro. Si ocurre un error durante la creación, el bloque *catch* lo captura y lanza un nuevo error personalizado que será manejado en la capa del controlador.

A continuación, a modo de resumen, se presenta el flujo completo del proceso de creación de productos:

1. **Formulario en Angular (agregarProducto)**
    - Se valida que todos los campos estén completos.
    - Se comprueba que el nombre no exista previamente.
    - Se validan valores específicos (precio, stock, umbral).
    - Se envía la petición HTTP al backend con los datos del producto.

2. **Controlador en Node.js (productoController)**
    - Se valida nuevamente el contenido de la petición (nombre, precio, stock, umbral).
    - Se llama al método del servicio crearProducto().

3. **Servicio de productos (productoService.crearProducto)**
    - Se crea el producto en la base de datos usando la entidad Producto.

- Se devuelve el producto creado.

4. **Respuesta al frontend**
   - Si todo es correcto, el backend responde con un código 201 y el nuevo producto.
   - Angular resetea el formulario y muestra una confirmación.
   - Si hay errores, se muestra un mensaje apropiado según el tipo de fallo.

4.3 Historial de pedidos entrantes

4.3.1 Interfaz de usuario

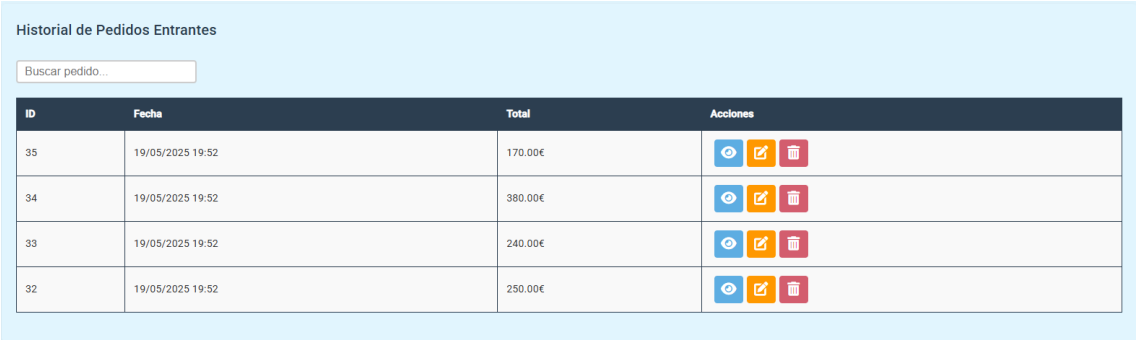


Figura 7. Historial de pedidos

Esta interfaz, que se muestra en la figura 7, permite al usuario consultar de forma estructurada todos los pedidos entrantes registrados en el sistema, entendiendo estos como los pedidos que los clientes hacen a la empresa. Es decir, representan los productos que salen del almacén y, por tanto, reducen el stock disponible.

La funcionalidad principal de esta vista es facilitar el seguimiento cronológico de todas las salidas de productos, ayudando al usuario a gestionar y supervisar el flujo de ventas o entregas realizadas.

La interfaz del componente está formada por los siguientes elementos:

- **Barra de búsqueda:** situada en la parte superior izquierda, permite al usuario buscar pedidos por identificador. Esta funcionalidad resulta especialmente útil cuando el historial contiene un volumen elevado de registros.
- **Tabla de pedidos:** cada fila de la tabla representa un pedido y muestra:
  - **ID:** identificador único del pedido.
  - **Fecha:** momento en el que se registró la salida del producto.
  - **Total:** importe total en euros.
  - **Acciones:** botones que permiten visualizar, editar o eliminar el pedido.

- **Botones de acción:**
  - **Ver:** muestra el contenido detallado del pedido, tal como se ilustra en la figura 8.

Figura 8. Detalles del pedido

  - **Editar:** Este botón redirige al componente *Registrar Pedido Entrante*, en el que se carga automáticamente la información del pedido seleccionado. Esto permite al usuario modificar directamente el pedido como si lo estuviera registrando por primera vez, tal como se muestra en la figura 9, manteniendo la coherencia de la interfaz.

Figura 9. Listado de productos en el pedido

  - **Eliminar:** elimina el pedido del sistema, previa confirmación del usuario. En algunos casos, esta operación puede implicar también la reversión del stock descontado inicialmente, asegurando así la integridad del inventario.

Esta interfaz es un componente desarrollado en Angular y emplea una tabla dinámica con estilos personalizados. Se ha utilizado una paleta de colores coherente con el resto de la aplicación y se han integrado iconos representativos para tratar de mejorar la usabilidad.

Al cargarse el componente, se llama al método `obtenerHistorial()`, el cual se muestra en el listado 9.

```
 1    obtenerHistorial(): void {
 2      this.pedidoService.obtenerHistorialPedidos('entrante').subscribe({
 3        next: (data: Pedido[]) => {
 4          if (data.length === 0) {
 5            this.errorMessage = 'No se encontraron pedidos.';
 6          } else {
 7            this.errorMessage = '';
 8            this.pedidosOriginales = data;
 9            this.pedidos = data.sort((a, b) => {
10              return new Date(b.fecha).getTime() - new Date(a.fecha).getTime();
11            });
12          }
13        },
14        error: (e) => {
15          if (e.status === 0) {
16            this.errorMessage = 'Error de conexión.
17                   Verifica tu conexión a internet y vuelve a intentarlo.';
18          } else {
19            this.errorMessage = 'Error al obtener el historial de pedidos.
20                   Por favor, inténtalo de nuevo más tarde.';
21          }
22        },
23      });
24    }
```

Listado 9. Método del frontend que devuelve la lista con todos los pedidos entrantes

Este método se encarga de obtener y organizar la lista de pedidos entrantes registrados en la base de datos.

A continuación, se detalla el funcionamiento paso a paso. En primer lugar, se realiza una petición al servicio `pedido.service`, solicitando únicamente los pedidos de tipo *entrante* (línea 2). Esto permite reutilizar el mismo servicio tanto para pedidos entrantes como salientes, especificando el tipo como parámetro.

Cuando la respuesta del servidor es exitosa, el método evalúa el contenido recibido. Si no hay pedidos (data.length === 0), se asigna un mensaje al atributo `errorMessage` para informar al usuario de que no se encontraron pedidos (líneas 4-5). Si hay pedidos, se limpia cualquier mensaje de error anterior, se almacena la lista original en *pedidosOriginales*, por si más adelante se desea aplicar búsquedas sin alterar los datos iniciales, y se ordenan los pedidos por fecha, del más reciente al más antiguo, usando el método `sort()` con objetos `Date` (líneas 7-11). Esta organización garantiza que el usuario vea primero los pedidos más actuales.

Si ocurre un error durante la petición, se maneja de forma diferenciada según el tipo de fallo. Si es un error de red (status 0), esto suele indicar una desconexión de internet. Por tanto, se muestra un mensaje específico al usuario. En caso de otros errores, se informa al usuario con un mensaje genérico.

Como se mencionaba anteriormente, el método `obtenerHistorial()` hace uso de `PedidoService` para enviar una solicitud al servidor, y espera como respuesta un array de objetos de tipo `Pedido`.

```
1   obtenerHistorialPedidos(tipo: string): Observable<any> {
2       let params = new HttpParams();
3       if (tipo) {
4         params = params.append('tipo', tipo);
5       }
6       return this.http.get(this.apiUrl, { params });
7   }
```

Listado 10. Llamada al servidor para obtener el historial de pedidos

El método ilustrado en el listado 10 encapsula la lógica para realizar una petición HTTP GET al backend, con el fin de recuperar los pedidos filtrados por tipo. A continuación, se explican sus pasos:

- Este método, en primer lugar, crea una instancia de `HttpParams`, que permite especificar los parámetros de la URL (*query string*) que se enviarán al servidor (línea 2).
- Si se ha especificado un tipo de pedido (en este caso, *entrante*), se añade a los parámetros de la petición (líneas 3-6). Esto se traduce en una URL como `GET /api/pedidos?tipo=entrante`.
- Finalmente, se lanza la petición HTTP utilizando `HttpClient`, especificando la URL del *endpoint* y los parámetros definidos. El resultado es un *Observable* que devuelve la lista de productos filtrada. Un Observable es una fuente de datos que emite valores de forma asíncrona, a la que otros pueden suscribirse para recibir esos valores.

4.3.2 Controlador

La petición anterior es recibida por el método `obtenerPedidos(req, res)` del controlador `pedidoController.js`, que se muestra en el listado 11.

```
1    async obtenerPedidos(req, res) {
2     try {
3       const tipo = req.query.tipo;
4       const pedidos = await pedidoService.obtenerPedidos(tipo);
5       res.status(200).json(pedidos);
6     } catch (error) {
7       res.status(500).json({ mensaje: 'Error al obtener los pedidos', error });
8     }
9   }
```

Listado 11. Método obtenerPedidos del controlador

Este método, en primer lugar, lee el parámetro `tipo` desde la query string (línea 3). Este valor llega desde el frontend en una URL y permite filtrar los pedidos por su naturaleza: *entrante* (ventas) o *saliente* (reposición de stock).

A continuación, se delega en el método `obtenerPedidos(tipo)` del servicio. Si todo ha ido bien, se devuelven los pedidos con código 200 OK (línea 5). En caso de que ocurra algún error, se responde con código 500 Internal Server Error y un mensaje explicativo (línea 7).

Por último, el método del servicio `obtenerPedidos(tipo)`, contenido en el listado 12, contiene la lógica de acceso a la base de datos. Es responsable de aplicar filtros según el tipo de pedido y devolver los pedidos con sus productos asociados.

```
1    async obtenerPedidos(tipo) {
2        try {
3            const filtros = tipo ? { tipo: { [Op.eq]: tipo } } : {};
4            return await Pedido.findAll({
5                where: filtros,
6                include: {
7                    model: Producto,
8                    through: {
9                        attributes: ['cantidad']
10                   }
11               }
12           });
13       } catch (error) {
14           throw new Error('Error al obtener los pedidos: ' + error.message);
15       }
16   }
```

Listado 12. Método obtenerPedidos del servicio

El método, en primer lugar, construye dinámicamente el filtro para los pedidos (línea 3). Si se recibe un tipo, por ejemplo, *entrante*, se filtran los pedidos por ese tipo, si no se especifica, se devuelven todos los pedidos.

A continuación, se realiza la consulta a la base de datos por medio de Sequelize. Esta se hace sobre la clase Producto, utilizando el filtro previamente construido (línea 3) y se incluye la relación con el modelo Producto. Además, se accede a los datos de la clase ProductoPedido para recuperar la cantidad de cada producto del pedido (líneas 4-12).

Finalmente, si ocurre algún fallo en la consulta, se lanza un error personalizado que será capturado por el controlador.

Por último, a modo de cierre, se resume el flujo completo de la obtención del historial de pedidos:

1. **Componente en Angular (`obtenerHistorial`)**
    - Se invoca al cargar el componente.
    - Se llama al método `obtenerHistorialPedidos('entrante')` del servicio.
    - Se gestiona la respuesta:

- Si hay datos, se ordenan del más reciente al más antiguo.
- Si no hay pedidos, se muestra un mensaje informativo.
- Si ocurre un error, se informa con un mensaje según el tipo de fallo.

2. **Servicio en Angular (`pedido.service.obtenerHistorialPedidos`)**
   - Se construye una petición HTTP GET al backend.
   - Se añade el parámetro `tipo=entrante` a la query string.
   - Se envía la petición al *endpoint* correspondiente.

3. **Controlador en Node.js (`pedidoController.obtenerPedidos`)**
   - Se recibe la petición con el parámetro `tipo`.
   - Se delega la operación al servicio del *backend* llamando a `obtenerPedidos()`.
   - Si se recuperan los pedidos correctamente, se envían con código 200 OK.
   - Si ocurre un error, se responde con un código 500 y un mensaje explicativo.

4. **Servicio en Node.js (pedidoService.obtenerPedidos)**
   - Se construye dinámicamente un filtro por tipo (si se especifica).
   - Se consulta la base de datos usando Sequelize.
   - Se incluyen los productos asociados a cada pedido, junto con la cantidad desde la clase intermedia.
   - Se devuelven los resultados al controlador.

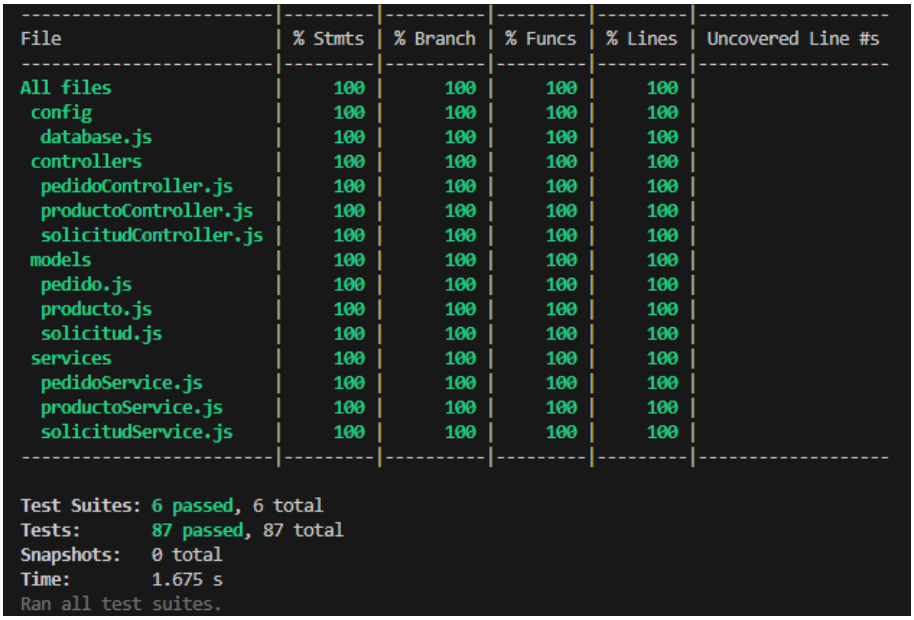
5. **Respuesta al frontend**
   - El componente Angular recibe los pedidos y los muestra en la tabla.
   - Si hay errores, se notifica al usuario mediante un mensaje claro.

5 Pruebas

Durante el desarrollo del proyecto se ha llevado a cabo una estrategia de pruebas unitarias exhaustiva, con el objetivo de garantizar la estabilidad y fiabilidad de la aplicación. Estas pruebas permiten verificar que cada unidad de código (funciones o métodos individuales) se comporta como se espera en distintos escenarios, incluyendo tanto los casos exitosos como los casos de error.

Para las pruebas unitarias del backend se ha utilizado Jest, un framework de pruebas muy popular en entornos JavaScript, que permite simular dependencias, controlar flujos asíncronos y validar los resultados de manera precisa.

La figura 10 muestra un resumen de los resultados obtenidos tras ejecutar todos los tests implementados en el backend.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
config	100	100	100	100	
database.js	100	100	100	100	
controllers	100	100	100	100	
pedidoController.js	100	100	100	100	
productoController.js	100	100	100	100	
solicitudController.js	100	100	100	100	
models	100	100	100	100	
pedido.js	100	100	100	100	
producto.js	100	100	100	100	
solicitud.js	100	100	100	100	
services	100	100	100	100	
pedidoService.js	100	100	100	100	
productoService.js	100	100	100	100	
solicitudService.js	100	100	100	100	

Test Suites: 6 passed, 6 total
Tests: 87 passed, 87 total
Snapshots: 0 total
Time: 1.675 s
Ran all test suites.

% Stmts (Sentencias ejecutadas) • % Branch (Ramas condicionales)
% Funcs (Funciones probadas) • % Lines (Líneas cubiertas)
Figura 10. Resultado de las pruebas unitarias del backend

En total, se han definido 87 pruebas unitarias distribuidas en 6 tests suites. Se ha obtenido un 100% de cobertura, lo que asegura que el comportamiento de cada componente del backend, incluidos controladores, modelos y servicios, ha sido verificado bajo distintas condiciones, anticipando posibles errores y garantizando un funcionamiento consistente.

El listado 13 detalla con más profundidad un par de pruebas unitarias desarrolladas para el servicio `ProductoService`.

```
1  it('debería eliminar un producto y devolver un mensaje de éxito', async () => {
2      const idMock = 1;
3
4      Producto.findByPk.mockResolvedValue({
5          destroy: jest.fn().mockReturnThis(),
6      });
7
8      const resultado = await productoService.eliminarProducto(idMock);
9
10     expect(resultado).toEqual({ message: 'Producto eliminado con éxito' });
11     expect(Producto.findByPk).toHaveBeenCalledWith(idMock);
12 });
13
14 it('debería lanzar un error si el producto no se encuentra', async () => {
15     const idMock = 1;
16     Producto.findByPk.mockResolvedValue(null);
17
18     await expect(productoService.actualizarProducto(idMock, {}))
                   .rejects.toThrow('Producto no encontrado');
19   });
```

Listado 13. Ejemplo de pruebas unitarias de ProductoService en el backend

Para ilustrar el proceso de diseño de las pruebas nos centraremos en las pruebas de ProductoService.

El servicio `ProductoService` contiene la lógica principal para interactuar con productos. Las funciones más relevantes probadas en este servicio son:

- **crearProducto**: Se prueba tanto el caso exitoso en el que se crea un producto correctamente, como los escenarios en los que la creación falla por errores internos. Se asegura que la función `Producto.create()` se llame con los datos adecuados y que los errores se manejen correctamente lanzando excepciones personalizadas.

- **obtenerTodosLosProductos**: Esta función se encarga de recuperar todos los productos existentes. Las pruebas validan que se devuelvan los productos esperados cuando la operación es exitosa, y que se lancen errores si la llamada a la base de datos falla.

- **obtenerProductoPorId**: Se prueban tres escenarios clave: cuando se obtiene un producto correctamente por su ID, cuando no se encuentra ningún producto con ese ID, lo que debe lanzar un error (listado 13, líneas 14-19), y cuando ocurre un fallo inesperado en la consulta.

- **actualizarProducto**: En este caso, las pruebas simulan la existencia del producto a actualizar mediante `Producto.findByPk`, y comprueban que se llamen los métodos de actualización con los datos adecuados. También se prueban los errores tanto por producto inexistente como por fallos durante la actualización.

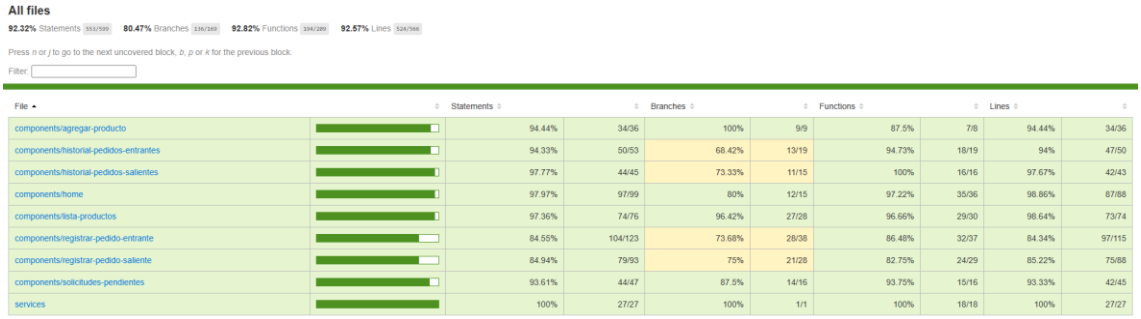
- **eliminarProducto**: Se valida que un producto pueda eliminarse correctamente usando `destroy`, y que se devuelva un mensaje adecuado (listado 13, líneas 1-12). Asimismo, se contemplan errores como la ausencia del producto o fallos al intentar eliminarlo.

Estas pruebas se han diseñado utilizando mocks del modelo `Producto`, lo que permite simular comportamientos sin depender de una base de datos real. Esta técnica garantiza un entorno de pruebas rápido, controlado y predecible.

En conjunto, estas pruebas aseguran que el servicio cumple con los requisitos funcionales definidos y se comporta de manera robusta frente a distintos escenarios, lo que contribuye significativamente a la fiabilidad de la aplicación antes de su despliegue final.

Para la parte frontend de la aplicación también se han desarrollado pruebas unitarias utilizando Karma como entorno de ejecución y Jasmine como framework de pruebas. Esta combinación es la más habitual en proyectos Angular, ya que Angular CLI la integra de forma predeterminada, facilitando su configuración y ejecución.

El objetivo principal de estas pruebas ha sido verificar que los distintos componentes, servicios y funcionalidades del frontend se comportan correctamente de forma aislada. Esto incluye, por ejemplo, comprobar que los formularios validan los datos como se espera, que los servicios realizan correctamente las llamadas al backend y que los métodos de los componentes manejan adecuadamente la lógica de la interfaz de usuario.



All files

92.32% Statements 553/599 **80.47%** Branches 236/293 **92.82%** Functions 194/209 **92.57%** Lines 524/566

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
components/agregar-producto		94.44%	34/36	100%	9/9	87.5%	7/8	94.44%	34/36
components/historial-pedidos-entrantes		94.33%	50/53	68.42%	13/19	94.73%	18/19	94%	47/50
components/historial-pedidos-salientes		97.77%	44/45	73.33%	11/15	100%	16/16	97.67%	42/43
components/home		97.97%	97/99	80%	12/15	97.22%	35/36	98.86%	87/88
components/lista-productos		97.36%	74/76	96.42%	27/28	96.66%	29/30	98.64%	73/74
components/registrar-pedido-entrante		84.55%	104/123	73.68%	28/38	86.48%	32/37	84.34%	97/115
components/registrar-pedido-saliente		84.94%	79/93	75%	21/28	82.75%	24/29	85.22%	75/88
components/solicitudes-pendientes		93.61%	44/47	87.5%	14/16	93.75%	15/16	93.33%	42/45
services		100%	27/27	100%	1/1	100%	18/18	100%	27/27

Figura 11. Resultado de las pruebas unitarias del frontend

La figura 11 muestra el informe de cobertura de pruebas unitarias del frontend, generado tras ejecutar todos los test con Karma y Jasmine. En ella se puede observar el nivel de cobertura alcanzado por cada componente de la aplicación en cuatro métricas fundamentales:

- **Statements** (sentencias ejecutadas)
- **Branches** (ramas condicionales)
- **Functions** (funciones probadas)
- **Lines** (líneas cubiertas)

En total, se ha alcanzado una cobertura global del 92.32 % en sentencias, 80.47 % en ramas, 92.82 % en funciones y 92.57 % en líneas de código, lo que refleja un trabajo sólido de verificación del comportamiento de la interfaz de usuario.

De manera complementaria al backend, el listado 14 muestra, a modo de ejemplo, un par de pruebas definidas para *ProductoService*.

```
 1  it('debería crear un producto', () => {
 2    service.crearProducto(mockProducto).subscribe((response) => {
 3      expect(response).toEqual({ success: true });
 4    });
 5
 6    const req = httpMock.expectOne(apiUrl);
 7    expect(req.request.method).toBe('POST');
 8    expect(req.request.body).toEqual(mockProducto);
 9
10    req.flush({ success: true });
11  });
12
13  it('debería eliminar un producto por ID', () => {
14    const id = 1;
15
16    service.eliminarProducto(id).subscribe((response) => {
17      expect(response).toEqual({ success: true });
18    });
19
20    const req = httpMock.expectOne(`${apiUrl}/${id}`);
21    expect(req.request.method).toBe('DELETE');
22
23    req.flush({ success: true });
24  });
```

Listado 14. Ejemplo de pruebas unitarias de ProductoService en el frontend

En este conjunto de pruebas se ha verificado el correcto funcionamiento de los métodos principales del servicio ProductoService, encargado de gestionar las operaciones relacionadas con los productos desde el frontend. Para ello se ha utilizado HttpClientTestingModule, que permite simular peticiones HTTP sin necesidad de acceder a un servidor real, y HttpTestingController, que controla y verifica las solicitudes realizadas.

A continuación, se describen las pruebas implementadas:

1. **Creación del servicio**: Verifica que el servicio *ProductoService* se inicializa correctamente mediante *Angular's TestBed*.
2. **`crearProducto`**: Comprueba que se envía al endpoint correcto (/api/productos), que el método HTTP utilizado es POST, y que el cuerpo de la petición coincide con el objeto del producto. Finalmente, simula una respuesta `{ success: true }` y verifica que el observable emite ese resultado (listado 14, líneas 1-11).
3. **`obtenerProductos`**: Comprueba que se realiza una solicitud GET al endpoint */api/productos*, y que el servicio devuelve correctamente un array de productos en la respuesta.

4. **`eliminarProducto`**: Testea que se realiza una solicitud DELETE al endpoint */api/productos/:id* y que la respuesta del servidor es interpretada correctamente. Se simula la eliminación de un producto con ID 1 (listado 14, líneas 13-24).

5. **`actualizarProducto`**: Se valida que se envía una petición PUT al endpoint */api/productos/:id* con el objeto del producto actualizado. Se comprueba tanto la URL como el cuerpo de la petición y que la respuesta es manejada correctamente.

6. **`verificarNombreProducto`**: Comprueba si ya existe un producto con ese nombre. Verifica que se construya correctamente la URL con el parámetro de consulta y que se reciba el valor booleano esperado como respuesta.

Estas pruebas cubren los métodos clave del servicio y aseguran que la lógica de interacción con la API REST se ejecuta correctamente. Además, permiten detectar cambios no deseados si se modifican las rutas o los datos enviados, lo cual es especialmente útil en entornos en los que el frontend y el backend evolucionan de forma independiente.

6 Conclusiones

A lo largo del desarrollo de este proyecto, he podido aplicar y consolidar los conocimientos adquiridos durante la carrera, pero, sobre todo, me ha permitido adentrarme de forma práctica en el desarrollo full stack, un ámbito que me interesaba especialmente.

Desde la construcción del backend con Node.js y Sequelize, hasta la implementación del frontend con Angular, pasando por la creación y diseño de la base de datos, he tenido la oportunidad de abordar todas las capas de una aplicación web completa. Esto me ha ayudado a entender mejor cómo se relacionan entre sí los distintos componentes de un sistema y a desarrollar buenas prácticas tanto en el lado del cliente como en el del servidor.

Además, he aprendido a trabajar con herramientas y tecnologías ampliamente utilizadas en el sector, como Jasmine y Karma para pruebas unitarias del frontend, y Jest para el backend, lo que me ha permitido incorporar hábitos de desarrollo profesional como el testing automatizado, la validación de datos o el manejo de errores.

En definitiva, este proyecto me ha servido para consolidar mis habilidades técnicas, reforzar mi perfil como desarrollador full stack y adquirir una experiencia muy completa que me será de gran utilidad en el ámbito profesional.

En cuanto al futuro de la aplicación, podría utilizarse para gestionar el inventario y llevar un registro de los pedidos de, por ejemplo, una tienda de ropa, aunque no es su finalidad.

Para presentar el contenido de manera más clara y organizada, he utilizado herramientas como ChatGPT, que me han ayudado a mejorar la redacción de descripciones, generar explicaciones técnicas más precisas y dar un formato más pulido a ciertas secciones del documento. Aunque todo el desarrollo y las decisiones técnicas han sido propias, esta asistencia me ha permitido expresar mejor el trabajo realizado.

Referencias

[1] M. Cohn, "User Stories Applied", Addison-Wesley Professional, Marzo 2004.

[2] A. Bampakos, "Learning Angular", 5ª ed., Packt Publishing, Enero 2025.

[3] Karma, "Karma Test Runner," [Online]. Available: https://karma-runner.github.io. [Accessed 02/ 07/ 2025].

[4] Jasmine, "Jasmine Behavior-Driven Development Framework," [Online]. Available: https://jasmine.github.io. [Accessed 02/ 07/ 2025].

[5] D. Durante, "Supercharging Node.js Applications with Sequelize", Pack Publishing, Octubre 2022.

[6] M. O. Source, "Jest," [Online]. Available: https://jestjs.io. [Accessed 02/ 07/ 2025].

[7] O. Corporation, "MySQL," [Online]. Available: https://www.mysql.com. [Accessed 02/ 07/ 2025].

[8] A. Friends, "XAMPP," [Online]. Available: https://www.apachefriends.org. [Accessed 02/ 07/ 2025].

[9] B. S. Scott Chacon, "Pro Git", 2ª ed., Apress, Noviembre 2014.

[10] Joint Technical Committee ISO/IEC JTC 1, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model.", International Standard Organization (ISO) e International Electrotechnical Commission (IEC), ISO/IEC 25010:2023, Noviembre 2023.

[11] S. Contributors, ""Sequelize – A promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server."," [Online]. Available: https://sequelize.org. [Accessed 02/ 07/ 2025].

[12] M. M. y. J. Powell, "Single Page Web Application", Manning, Septiembre 2013.