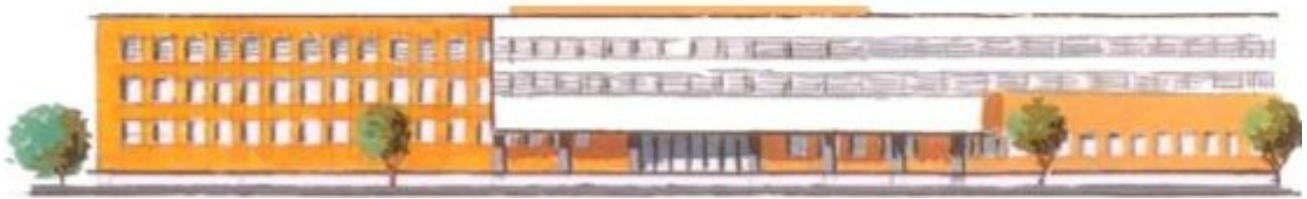


DESARROLLO E IMPLEMENTACIÓN DE UNA PLATAFORMA WEB CON ARQUITECTURA DE MICROSERVICIOS

Plataforma Web para una pizzería (Redpizza)

IES Ramón del Valle-Inclán



Autor: Felipe Alonso Fernández

Ciclo Formativo de Grado Superior en Desarrollo de Aplicaciones
Multiplataforma (DAM)

IES Ramón del Valle Inclán

Curso 2024 / 2025

Tutor del Proyecto: Rafael Reina Ramírez

Entorno tecnológico	4
Despliegue del aplicativo	5
Análisis del sistema	7
Historias de Usuario	7
Casos de Uso	10
Diagrama de Clases	16
Tipo X	17
Atributos del Tipo X	19
Atributos de Producto	19
Matriz de Rastreabilidad	20
Mapa de Navegabilidad	22
Desarrollo e implementación de los microservicios	23
1. Propósito de los microservicios	23
2. Tecnologías utilizadas	23
3. Estructura de los microservicios	23
4. Modelos de datos	24
Productos	24
5. Endpoints de la API	25
6. Script iniciación contenedores	26
Estructura general del docker-compose	26
Primera versión	26
Segunda Versión	28
Explicación del script de iniciación de la base de datos y de las tablas	30
7. Pruebas y validación	31
Desarrollo e implementación del frontend	32
1. App.jsx, index.html y UserContext.jsx	32
2. Productos.Service	35
3. Componentes	37
3.1 Navbar	37
Diseño responsive	37
3.2 Footer	38
Diseño responsive	38
3.3 Principal	39
Diseño responsive	39
3.4 Carta	40
Diseño responsive	40
3.5 Login, NuevoProducto y EditarProducto	41
Login	41
NuevoProducto y EditarProducto	41
Diseño responsive	41
3.6 Localizacion	42

Diseño responsive	42
3.7 Contacto	43
Diseño responsive	43
Desarrollo y explicación del despliegue en Hostinger	44
1. Contratación y configuración inicial del VPS	44
2. Preparación del servidor VPS	45
3. Puesta en marcha del backend	45
4. Configuración del entorno frontend	45
5. Dominio, seguridad y acceso web	46
6. Configuración del proxy con nginx	47
7. Despliegue de producción	49

Entorno tecnológico

- **Frontend** desarrollado con **React**:
 - Interfaz de cliente para:
 - Visualizar carta de productos y ofertas.
 - Ver ubicación del local y zona de reparto.
 - Interfaz de administrador para:
 - Modificar productos.
 - Cambiar precios.
- **Backend** basado en **microservicios con FastAPI (Python)**:
 - Microservicio de **Productos**
 - El microservicio utiliza una base de datos **PostgreSQL**.

Funcionalidades web disponibles

- **Login para administrador.**
- **Visualización de la carta de productos.**
- **Visualización de ofertas.**
- **Consulta de ubicación del local y zona de reparto.**
- **Página de contacto con el restaurante**
- **Gestión y modificación de contenidos desde interfaz de administrador.**

Despliegue del aplicativo

- **Frontend y dominio principal:**
 - Se desplegará en un **servidor de hosting VPS** contratado a través de **Hostinger**.
 - El dominio apunta a este servidor donde estará alojada la aplicación React.
- **Microservicio (Productos):**
 - También desplegados en el **VPS de Hostinger**, funcionando como contenedor independiente.
 - El microservicio será accesible desde el frontend a través de sus respectivas APIs REST.
 - Se utilizará **Docker** para contenerizar cada microservicio.
- **Base de datos:**
 - PostgreSQL se instalará en el mismo VPS o en un contenedor independiente.
- **Gestión del VPS:**
 - Se utilizará acceso SSH para configurar el entorno.
 - Se instalará y configurará **Docker, Docker Compose**.
 - Ejecución de **docker compose up -d** para mantener los servicios activos
 - Configuración para exponer correctamente los puertos y acceso a la API.
 - Instalación de Node.js y npm para el frontend.
 - Uso de PM2 para ejecutar npm run dev y mantener el frontend siempre activo durante el desarrollo.
 - Compra y configuración del dominio en Hostinger, apuntando el registro A a la IP del VPS.
 - Instalación y configuración de nginx como proxy inverso para servir frontend y backend.

- Instalación y configuración de certificado SSL gratuito con Let's Encrypt para HTTPS.
- Ajuste de CORS en FastAPI
- Generación del build de producción del frontend con npm run build.
- Copia del contenido de dist a /var/www/redpizza.org y configuración de nginx para servir archivos estáticos.

Análisis del sistema

Historias de Usuario

HU-01	Consultar la carta de la pizzería
Descripción	Como cliente de la pizzería, quiero consultar la carta en la página web para ver los productos disponibles.
Requisito Funcional-01	Caso de Uso: Consultar Carta.
Requisito de Información-01	Como cliente de la pizzería, quiero que el sistema me muestre la siguiente información de cada producto: <ul style="list-style-type: none">• Nombre del producto• Descripción• Imagen del producto

HU-02	Consultar ofertas disponibles en la pizzería
Descripción	Como cliente de la pizzería, quiero ver imágenes de las ofertas para ayudarme a ahorrar en mi pedido.
Requisito Funcional-02	Caso de Uso: Consultar ofertas.
Requisito de Información-02	Como cliente de la pizzería, quiero que el sistema me muestre los siguientes datos de las ofertas: <ul style="list-style-type: none">• Nombre de la oferta• Descripción• Precio o Descuento

HU-03	Consultar ubicación y zona de reparto
Descripción	Como cliente de la pizzería, quiero ver la ubicación del local y la zona de reparto para saber si pueden entregar un pedido a mi domicilio.
Requisito Funcional-03	Caso de Uso: Consultar ubicación y zona de reparto.
Requisito de Información-03	Como cliente de la pizzería, quiero que el sistema me muestre: <ul style="list-style-type: none"> • Dirección del local • Mapa con la ubicación • Información sobre zonas de entrega

HU-04	Login de administrador
Descripción	Como administrador de la página web, quiero acceder con un usuario y contraseña para gestionar los productos y ofertas.
Requisito Funcional-04	Caso de Uso: Iniciar sesión como administrador.
Requisito de Información-04	Como administrador, quiero que el sistema valide mis credenciales y me permita acceder solo si los datos son correctos.

HU-05	Gestión de productos
Descripción	Como administrador de la página web, quiero poder añadir, eliminar, modificar y editar productos para mantener la información actualizada.
Requisito Funcional-05	Caso de Uso: Gestión de productos
Requisito de Información-05	Como administrador, quiero que el sistema almacene y me permita modificar la siguiente información: <ul style="list-style-type: none"> • Nombre del producto • Descripción • Precio • Imagen del producto

HU-06	Dominio personalizado para la página web
Descripción	Como propietario de la pizzería, quiero que la página web esté alojada en un dominio con un nombre relacionado con la pizzería para que los clientes la encuentren fácilmente.
Requisito Funcional-06	Caso de Uso: Configurar dominio personalizado.
Requisito de Información-06	Como propietario, quiero que el sistema utilice un dominio con un nombre adecuado y representativo de la pizzería.

Casos de Uso

Caso de Uso	
Nombre	Consultar la carta de la pizzería
Id	CU-01
Descripción	El cliente podrá ver la carta de la pizzería en la página web, con todos los productos disponibles junto con su descripción, precio e imagen.
Actores Primarios	Cliente
Actores Secundarios	Sistema
Precondiciones	<ul style="list-style-type: none">- La página web debe estar disponible.- Los productos deben estar registrados en el sistema.
Curso Normal	<ol style="list-style-type: none">1. El cliente accede a la sección de la carta en la web.2. El sistema muestra los productos con su información.3. El cliente visualiza los productos y sus detalles.
Postcondiciones	<ul style="list-style-type: none">- El cliente puede conocer los productos disponibles.
Alternativas	<ol style="list-style-type: none">1. Si la página web no carga, el cliente no podrá ver la carta.2. Si no hay productos registrados, la carta aparecerá vacía.

Caso de Uso	
Nombre	Consultar ofertas
Id	CU-02
Descripción	El cliente podrá ver las ofertas disponibles en la página web con su descripción y descuento
Actores Primarios	Cliente
Actores Secundarios	Sistema
Precondiciones	<ul style="list-style-type: none">- La página web debe estar disponible.- Las ofertas deben estar registradas en el sistema.
Curso Normal	<ol style="list-style-type: none">1. El cliente accede a la sección de ofertas.2. El sistema muestra las ofertas con su información.3. El cliente revisa las ofertas disponibles.
Postcondiciones	<ul style="list-style-type: none">- El cliente puede conocer las ofertas actuales.
Alternativas	<ol style="list-style-type: none">1. Si la página web no carga, el cliente no podrá ver las ofertas.2. Si no hay ofertas registradas, la sección aparecerá vacía.

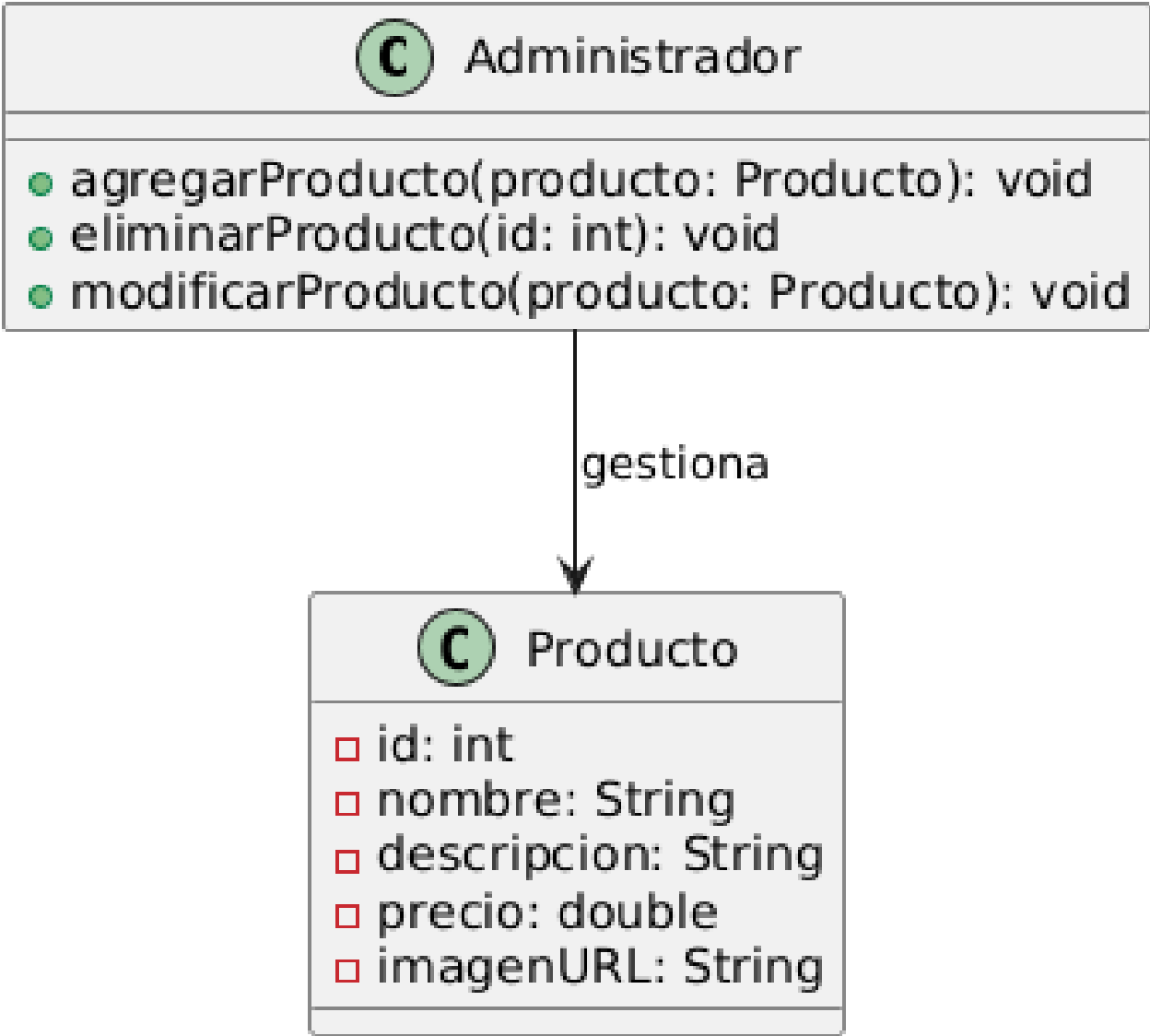
Caso de Uso	
Nombre	Consultar ubicación y zona de reparto
Id	CU-03
Descripción	El cliente podrá ver la dirección del local y la zona de reparto en la página web.
Actores Primarios	Cliente
Actores Secundarios	Sistema
Precondiciones	<ul style="list-style-type: none">- La página web debe estar disponible.- La información de la ubicación y zona de reparto debe estar registrada.
Curso Normal	<ol style="list-style-type: none">1. El cliente accede a la sección de ubicación2. El sistema muestra la dirección, el mapa y la información de la zona de reparto.
Postcondiciones	<ul style="list-style-type: none">- El cliente puede conocer la ubicación y la zona de reparto.
Alternativas	<ol style="list-style-type: none">1. Si no hay conexión, la información no podrá mostrarse.

Caso de Uso	
Nombre	Iniciar sesión como administrador
Id	CU-04
Descripción	El administrador podrá iniciar sesión con su usuario y contraseña para gestionar productos y ofertas.
Actores Primarios	Administrador
Actores Secundarios	Sistema
Precondiciones	<ul style="list-style-type: none">- El administrador debe tener credenciales válidas.- La página web debe estar funcionando.
Curso Normal	<ol style="list-style-type: none">1. El administrador accede a la página de login.2. Ingresa sus credenciales.3. El sistema valida la información.4. Si es correcta, el administrador entra al panel de gestión.
Postcondiciones	<ul style="list-style-type: none">- El administrador puede gestionar los productos.
Alternativas	<ol style="list-style-type: none">1. Si las credenciales son incorrectas, se muestra un error.

Caso de Uso	
Nombre	Gestión de productos
Id	CU-05
Descripción	El administrador podrá añadir, eliminar, modificar y editar productos en la carta.
Actores Primarios	Administrador
Actores Secundarios	Sistema
Precondiciones	- El administrador debe haber iniciado sesión.
Curso Normal	<ol style="list-style-type: none">1. El administrador accede al panel de gestión.2. Puede añadir, eliminar o modificar productos.3. Los cambios se reflejan en la carta de la web.
Postcondiciones	- La información de los productos queda actualizada.
Alternativas	<ol style="list-style-type: none">1. Si hay un error, los cambios no se guardarán correctamente.

Caso de Uso	
Nombre	Configurar dominio personalizado
Id	CU-06
Descripción	La página web estará alojada en un dominio con un nombre relacionado con la pizzería.
Actores Primarios	Administrador
Actores Secundarios	Proveedor de hosting
Precondiciones	<ul style="list-style-type: none">- Debe haberse adquirido un dominio.
Curso Normal	<ol style="list-style-type: none">1. El administrador adquiere un dominio.2. Se configura el hosting con el dominio adquirido.3. La página web queda accesible con el dominio personalizado.
Postcondiciones	<ul style="list-style-type: none">- La página web es accesible desde el dominio configurado.
Alternativas	<ol style="list-style-type: none">1. Si el dominio no está bien configurado, la página puede no ser accesible.

Diagrama de Clases



Tipo X

Tipo-001	Administrador		
Requisitos asociados	RI-04: Iniciar sesión como administrador. RI-05: Gestión de productos		
Descripción	Este subsistema representa los usuarios que son administradores		
Supertipos	Usuario		
Subtipos	N/A		
Componentes			
	Nombre	Tipo OCL	Mult.
	N/A		
Comentarios	N/A		

Tipo-002	Producto		
Requisitos asociados	RI-01: Consultar la carta de la pizzería RI-05: Gestión de productos		
Descripción	Este subsistema almacena y permite la gestión de los productos, registrando el id, el nombre, la descripción, el precio y la url de una imagen de un producto en específico.		
Supertipos	N/A		
Subtipos	N/A		
Componentes			
	Nombre	Tipo OCL	Mult.
	id	int	
	nombre	String	
	descripción	String	
	precio	double	
	imagenURL	String	
Comentarios	N/A		

Atributos del Tipo X

Atributos de Producto

Atributo-001	Producto::id
Descripción	Identificador único del producto.
Tipo	int
Comentarios	Debe ser único para cada producto.

Atributo-002	Producto::nombre
Descripción	Nombre del producto.
Tipo	String
Comentarios	No debe estar vacío ni superar los 100 caracteres.

Atributo-003	Producto::descripcion
Descripción	Descripción detallada del producto.
Tipo	String
Comentarios	Puede contener detalles sobre ingredientes o características.

Atributo-004	Producto::precio
Descripción	Precio del producto.
Tipo	double
Comentarios	Debe ser un valor positivo.

Atributo-005	Producto::imagenURL
Descripción	URL de la imagen del producto.
Tipo	String
Comentarios	Debe contener un enlace válido a una imagen.

05																			
RF-06						X													X
RF-07							X												
RI-01	X							X											
RI-02		X							X										
RI-03			X							X									
RI-04				X							X								
RI-05					X							X							
RI-06						X							X						

Mapa de Navegabilidad



Desarrollo e implementación de los microservicios

1. Propósito de los microservicios

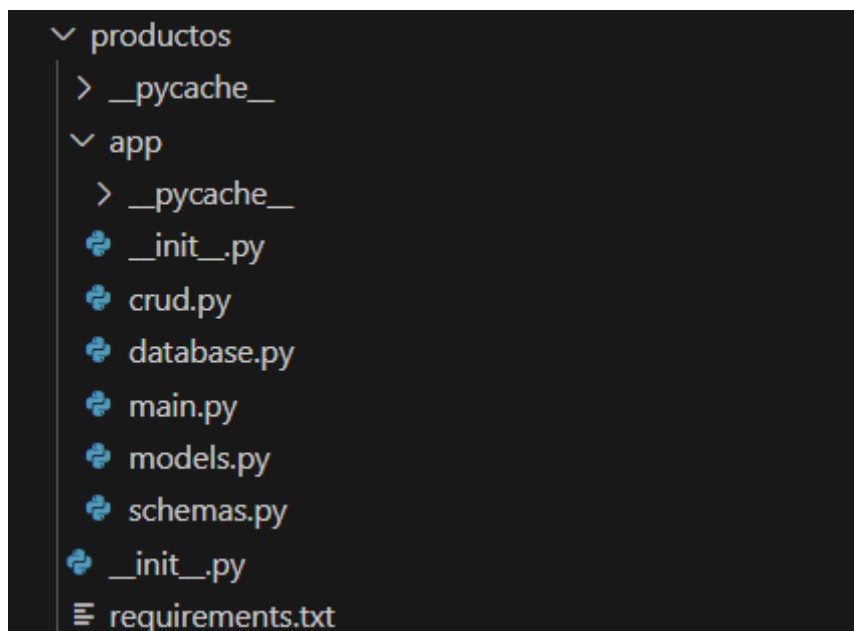
El propósito del microservicio es gestionar la información de productos para la aplicación web. Este microservicio permite almacenar, consultar, modificar y gestionar productos en una base de datos PostgreSQL. Además, se integran con una interfaz de administrador en la página web, donde se puede realizar la administración de productos de manera fácil y eficiente.

2. Tecnologías utilizadas

Los microservicios están desarrollados utilizando las siguientes tecnologías:

- **FastAPI:** Framework para la construcción de APIs rápidas y eficientes.
- **SQLAlchemy:** ORM para interactuar con la base de datos PostgreSQL de manera sencilla.
- **PostgreSQL:** Sistema de gestión de bases de datos relacional utilizado para almacenar los productos y ofertas.
- **Docker y Docker Compose:** Utilizados para contenerizar los microservicios y la base de datos, facilitando la gestión de los contenedores y su despliegue.
- **Pydantic:** Para la validación y definición de modelos de datos en las solicitudes y respuestas de la API.

3. Estructura de los microservicios



__init__.py: Encontramos 2 dentro del microservicio, el primero dentro del directorio raíz del microservicio (productos) y el segundo dentro del directorio app. Sirve para indicar a python que el directorio actuará como paquete y en el caso de querer hacer alguna importación esta pueda hacerse.

crud.py: Aquí es donde defines las operaciones que se van a hacer sobre los modelos de datos. Por ejemplo: crear un producto en la base de datos, obtener un producto por ID, obtener todos los productos, etc...

database.py: Aquí es donde se define como se realizará la conexión a la base de datos PostgreSQL, para ello se usa **SQLAlchemy**

main.py: Este es el archivo principal donde se inicializa y ejecuta la aplicación FastAPI. Aquí se definen las rutas (endpoints) y se manejan las solicitudes HTTP como GET, POST...

models.py: En este archivo defines los modelos de datos utilizando SQLAlchemy también. Aquí es donde se definen las tablas de la base de datos.

schemas.py: Este archivo contiene los esquemas de Pydantic para las solicitudes y respuestas de la API. Los esquemas se utilizan para **validar** y **serializar** los datos. Pydantic es una biblioteca que valida automáticamente los datos que se reciben en las solicitudes HTTP.

requirements.txt: Este archivo contiene las dependencias necesarias para tu microservicio. Es un archivo de texto donde se especifican todas las librerías que deben instalarse para que el microservicio funcione correctamente.

4. Modelos de datos

Productos

```
class Producto(Base):
    __tablename__ = "productos"

    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String, index=True)
    descripcion = Column(String)
    precio = Column(Float)
    disponible = Column(Boolean, default=True)
    imagen_url = Column(String, nullable=True)
```


5. Endpoints de la API

El microservicio de **Productos** tiene los siguientes endpoints:

- **GET /productos/**: Obtiene todos los productos disponibles.
- **GET /productos/{producto_id}**: Obtiene los detalles de un producto específico por su ID.
- **POST /productos/nuevo_producto**: Crea un nuevo producto en la base de datos.
- **GET /productos/tipo/{tipo}**: Obtiene todos los productos que pertenecen a un tipo específico.
- **PUT /productos/{producto_id}**: Modifica un producto existente por su ID.
- **DELETE /productos/{producto_id}**: Elimina un producto específico de la base de datos.

Esta información también se puede ver en la documentación de Swagger del microservicio

6. Script iniciación contenedores

Estructura general del docker-compose

Para iniciar el contenedor del microservicio (productos) junto con el contenedor de la base de datos PostgreSQL, se ha utilizado un archivo docker-compose.yml que define y coordina los dos servicios.

El comportamiento es el siguiente:

1. Se lanza primero el contenedor db (PostgreSQL).
2. Una vez verificado que PostgreSQL está disponible (gracias a un healthcheck), se inicia el contenedor del microservicio.
3. Esto asegura que el servicio no intente conectarse a la base de datos antes de que esté lista.

Primera versión

```
version: "3.9"

services:
  productos:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: productos
    ports:
      - "8000:8000"
    volumes:
      - ./productos:/app/productos
    depends_on:
      db:
        condition: service_healthy
    environment:
      - DATABASE_URL=postgresql://redpizza:1234@db:5432/productos_db
    networks:
      - backend
    restart: always
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/productos"]
      interval: 30s
```

```
    timeout: 10s
    retries: 5
    start_period: 30s

db:
  image: postgres:13
  container_name: db_postgres_tfg
  environment:
    POSTGRES_USER: redpizza
    POSTGRES_PASSWORD: 1234
    POSTGRES_MULTIPLE_DATABASES: productos_db,ofertas_db
  volumes:
    - postgres_data:/var/lib/postgresql/data
    -
./init-multiple-databases.sh:/docker-entrypoint-initdb.d/init-multiple-
databases.sh
  networks:
    - backend
  ports:
    - "5432:5432"
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U redpizza"]
    interval: 5s
    timeout: 5s
    retries: 5
    start_period: 10s

networks:
  backend:
    driver: bridge

volumes:
  postgres_data:
```

Segunda Versión

En esta segunda versión, ya tenemos la imagen del microservicio subida a docker, el procedimiento al ejecutar el docker-compose es igual que en la primera versión, primero se inicia el contenedor de postgres y cuando se verifica que este está iniciado, se inicia el contenedor del microservicio.

```
version: "3.9"

services:
  productos:
    image: falofer0405/productos:latest
    container_name: productos
    ports:
      - "8000:8000"
    volumes:
      - ../productos:/app/productos
    depends_on:
      db:
        condition: service_healthy
    environment:
      - DATABASE_URL=postgresql://redpizza:1234@db:5432/productos_db
    networks:
      - backend
    restart: always
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/productos"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s

  db:
    image: postgres:13
    container_name: db
    environment:
      POSTGRES_USER: redpizza
      POSTGRES_PASSWORD: 1234
      POSTGRES_MULTIPLE_DATABASES: productos_db,ofertas_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
```

```
-  
../init-multiple-databases.sh:/docker-entrypoint-initdb.d/init-multiple  
-databases.sh  
  networks:  
    - backend  
  ports:  
    - "5432:5432"  
  healthcheck:  
    test: ["CMD-SHELL", "pg_isready -U redpizza"]  
    interval: 10s  
    timeout: 5s  
    retries: 5  
    start_period: 30s  
  
networks:  
  backend:  
    driver: bridge  
  
volumes:  
  postgres_data:
```

Explicación del script de iniciación de la base de datos y de las tablas

Se utiliza un script llamado **init-multiple-databases.sh**, que se monta automáticamente dentro del contenedor **db** al iniciarse por primera vez. Este script se encarga de crear múltiples bases de datos en el servidor **PostgreSQL**, utilizando la variable de entorno **POSTGRES_MULTIPLE_DATABASES**.

```
#!/bin/bash

set -e
set -u

function create_user_and_database() {
    local database=$1
    echo "Creating database '$database'"
    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
        CREATE DATABASE $database;
        GRANT ALL PRIVILEGES ON DATABASE $database TO $POSTGRES_USER;
EOSQL
}

if [ -n "$POSTGRES_MULTIPLE_DATABASES" ]; then
    echo "Multiple database creation requested:
$POSTGRES_MULTIPLE_DATABASES"
    for db in $(echo $POSTGRES_MULTIPLE_DATABASES | tr ',' ' '); do
        create_user_and_database $db
    done
    echo "Multiple databases created"
fi
```

- El script se ejecuta automáticamente al arrancar el contenedor por primera vez, gracias a que se encuentra en la ruta **/docker-entrypoint-initdb.d/**.
- Utiliza **psql** (herramienta oficial de línea de comandos de PostgreSQL) para crear las bases de datos listadas en la variable **POSTGRES_MULTIPLE_DATABASES**, separadas por comas.
- Cada base de datos recibe todos los privilegios para el usuario definido por **POSTGRES_USER**.
- Está protegido contra errores gracias a **set -e** (detiene en errores) y **set -u** (detiene en variables no definidas).

La **base de datos** (**productos_db**) se crea al iniciar el contenedor de **PostgreSQL** por primera vez.

La tabla se crea automáticamente cuando se inicia el microservicio, gracias a la lógica definida en su backend usando **SQLAlchemy** (es un **ORM** para **Python** que permite interactuar con bases de datos relacionales).

En este proyecto, SQLAlchemy se encarga de:

- Definir las estructuras de las tablas (models)
- Establecer la conexión con la base de datos
- Crear las tablas automáticamente con la instrucción:
 - *Base.metadata.create_all(bind=engine)*

7. Pruebas y validación

Para probar los microservicios, se ha utilizado **Postman**. Con Postman, se realizaron pruebas para verificar que los endpoints de ambos microservicios de productos y ofertas respondieran correctamente, devolviendo los datos esperados y realizando las operaciones CRUD.

Desarrollo e implementación del frontend

1. App.jsx, index.html y UserContext.jsx

```
import { useState } from 'react'
import './App.css'
import Navbar from './components/Navbar';
import Footer from './components/Footer';
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import "bootstrap/dist/js/bootstrap.bundle.min";
import Principal from './components/Principal';
import Carta from './components/Carta';
import NuevoProducto from './components/NuevoProducto';
import EditarProducto from './components/EditarProducto';
import Localizacion from './components/Localizacion';
import Contacto from './components/Contacto';
import Login from './components/Login';
import { UserProvider } from './context/UserContext';

function App() {

  return (
    <UserProvider>
      <Router>
        <Navbar />
        <Routes>
          <Route path="/" element={<Navigate to={"/inicio"} />} />
          <Route path="/inicio" element={<Principal />} />
          <Route path="/carta" element={<Carta />} />
          <Route path="/productos/nuevo_producto"
element={<NuevoProducto />} />
          <Route path="/productos/:id" element={<EditarProducto />} />
          <Route path="/localizacion" element={<Localizacion />} />
        </Routes>
      </Router>
    </UserProvider>
  )
}
```



```
    <Route path="/contacto" element={<Contacto />} />
    <Route path="/login-panel-admin" element={<Login />} />
    <Route path="*" element={<Navigate to="/" />} />
  </Routes>
  <Footer />
</Router>
</AuthProvider>
)
}

export default App
```

En el archivo **App.jsx** organizamos la navegación principal de la aplicación. Utilizamos la librería **react-router-dom** para definir las rutas que permiten a los usuarios moverse entre las distintas secciones sin necesidad de recargar la página. Desde aquí se accede a vistas como la carta de productos, la localización, el contacto o el panel de administración. También incluimos componentes comunes como la barra de navegación (**Navbar**) y el pie de página (**Footer**), que se mantienen visibles en todo momento.

Además, integramos un contexto de usuario, definido en el archivo **UserContext.jsx**. Este contexto nos permite saber si el usuario ha iniciado sesión como administrador. Mediante el hook **useState**, controlamos una variable booleana llamada **isAuthenticated**. Cuando el usuario accede correctamente al sistema, actualizamos esta variable a **true**, y cuando cierra sesión, la devolvemos a **false**. Esto nos permite mostrar u ocultar ciertas funcionalidades de la web de forma sencilla y centralizada.

El **AuthProvider** actúa como proveedor global de este contexto, envolviendo a todos los componentes de la aplicación. Así, cualquier componente puede acceder fácilmente a esta información mediante el hook **useUser**, sin necesidad de pasar datos manualmente entre componentes.

```
import React, { createContext, useContext, useState } from 'react';

const UserContext = createContext(null);

export const AuthProvider = ({ children }) => {

  const [isAuthenticated, setIsAuthenticated] = useState(false);

  const login = () => setIsAuthenticated(true);

  const logout = () => setIsAuthenticated(false);

  return (
```

```
<UserContext.Provider value={{ isAuthenticated, login, logout }}>

  {children}

</UserContext.Provider>

);

};

export const useUser = () => useContext(UserContext);
```

Por último, el archivo **index.html** actúa como la base estructural del sitio. Contiene el contenedor principal (**<div id="root">**) donde React inserta todos los componentes, y configuramos también el título de la página junto con el icono que aparece en el navegador.

```
<!doctype html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <link rel="icon" type="image/svg+xml"
href="src/assets/logo_redpizza_jpg.jpg" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Redpizza | Somos la pizza!</title>

  </head>

  <body>

    <div id="root"></div>

    <script type="module" src="/src/main.jsx"></script>

  </body>

</html>
```

2. Productos.Service

```
import axios from "axios";

export default axios.create({
  baseURL: "/",
  headers: {
    "Content-type": "application/json",
  },
});
```

Para conectar nuestra aplicación frontend con el microservicio de productos, utilizamos la librería **axios**, que nos permite realizar fácilmente peticiones HTTP. Esta conexión la configuramos en el archivo **http-common-productos.js**, donde indicamos la dirección base de las llamadas a la API.

En nuestro caso, dejamos la base URL como '/' para que las peticiones se realicen directamente desde el mismo servidor donde se aloja el frontend. Esta estrategia nos permite evitar problemas relacionados con **CORS** (política de seguridad del navegador) y garantiza que las llamadas se hagan por el mismo protocolo (**HTTPS**), lo que es importante para mantener la seguridad de la aplicación una vez desplegada.

En el archivo **ProductoDataService.js** agrupamos todas las funciones necesarias para trabajar con los productos: crear uno nuevo, obtener uno específico, listar productos por tipo, actualizarlos o eliminarlos. De esta forma, centralizamos todas las operaciones relacionadas con los productos en un único servicio, lo que hace que el código sea más limpio, organizado y fácil de mantener.

Gracias a esta estructura, cualquier componente del frontend puede importar este servicio y utilizar sus métodos para comunicarse con el backend de forma sencilla y coherente.

```
import http from "../../http-common-productos.js";

class ProductoDataService {

  createProducto(data) {

    return http.post("/productos/nuevo_producto", data);

  }

  getProducto(id) {

    return http.get(`/productos/${id}`);

  }

  getProductosByTipo(tipo) {

    return http.get(`/productos/tipo/${tipo}`);

  }

  updateProducto(id, data) {

    return http.put(`/productos/${id}`, data);

  }

  deleteProducto(id) {

    return http.delete(`/productos/${id}`);

  }

}

export default new ProductoDataService();
```


3. Componentes

En este apartado explicamos los principales componentes que forman la interfaz del usuario en nuestra aplicación. Cada componente representa una parte específica de la página web y cumple una función concreta dentro de la navegación o la interacción. A lo largo de las siguientes secciones, detallamos su funcionalidad y cómo los hemos adaptado para ofrecer una experiencia visual correcta en distintos dispositivos gracias al diseño responsive.

3.1 Navbar

El componente **Navbar** representa la barra de navegación principal de la página. Desde aquí permitimos a los usuarios moverse entre las distintas secciones del sitio: inicio, carta, localización y contacto. Para ello, utilizamos el componente **Link** de **react-router-dom**, que nos permite cambiar de página sin recargar la web, ofreciendo una experiencia más fluida.

Hemos dividido la barra en tres bloques: a la izquierda se muestran los enlaces principales (INICIO y CARTA), en el centro aparece el logotipo de la pizzería, y a la derecha los accesos a LOCALIZACIÓN y CONTACTO. Esta estructura nos permite mantener el equilibrio visual y facilitar la navegación del usuario.

Además, incorporamos un menú desplegable adaptado para dispositivos móviles. Al hacer clic en el icono , se muestra una versión simplificada del menú con los mismos enlaces, pero centrados verticalmente.

Diseño responsive

Para que el **Navbar** se adapte correctamente a distintos tamaños de pantalla, hemos implementado varias reglas CSS usando media queries. A partir de un ancho de 872 píxeles o menos, ocultamos los bloques laterales de navegación (**navbar-left** y **navbar-right**) y mostramos el menú móvil. También reducimos el tamaño del logotipo y aumentamos el tamaño del texto en los botones del menú móvil para facilitar su lectura en pantallas pequeñas.

3.2 Footer

El componente **Footer** representa el pie de página de la aplicación. Lo usamos para mostrar un mensaje legal con el nombre de la pizzería y el año actual, actualizándose automáticamente gracias a **new Date().getFullYear()**. Este tipo de información aporta seriedad al proyecto y contribuye a completar visualmente la estructura de la web.

Este componente aparece en todas las secciones del sitio, ya que lo incluimos de forma global dentro de la estructura del **App.jsx**. Al estar centrado y ocupar toda la parte inferior de la pantalla, ayuda a equilibrar el diseño general.

Diseño responsive

Para asegurar que el **Footer** se vea correctamente en todos los dispositivos, aplicamos un estilo adaptable. Usamos márgenes automáticos y un ancho máximo para centrar el contenido. Además, mediante una media query para pantallas de menos de 450 píxeles, reducimos el tamaño del texto y los márgenes para mejorar la visualización en móviles. Así conseguimos que el pie de página siga siendo legible y visualmente adecuado en cualquier tamaño de pantalla.

3.3 Principal

El componente **Principal** actúa como la página de inicio de la plataforma. Es una de las partes más visuales e importantes del proyecto, ya que está diseñada para captar la atención del visitante desde el primer momento.

Comenzamos con una sección tipo *hero* en la parte superior, donde mostramos el logotipo de la pizzería sobre una imagen de fondo difuminada, acompañada del eslogan de la pizzería (“¡somos la pizza!”) y las dos formas de realizar pedidos: a través del servicio Just Eat o por teléfono.

Debajo, mostramos dos imágenes destacadas: una representa la carta y otra las ofertas. Al hacer clic sobre cualquiera de ellas, se abre una vista ampliada en forma de modal, mejorando la experiencia visual sin salir de la página.

A continuación, mostramos una sección con los productos más populares. Cada producto se presenta con una imagen circular y una pequeña descripción. Por último, incluimos una sección de valoraciones reales de clientes, donde se muestran opiniones positivas junto con la puntuación en forma de estrellas y la fecha de la reseña.

Diseño responsive

Para garantizar que esta página principal se vea correctamente en todo tipo de dispositivos, hemos aplicado un diseño responsive:

- En pantallas pequeñas, las imágenes de la carta y ofertas se reorganizan en una sola columna para facilitar la visualización.
- Reducimos progresivamente el tamaño del logotipo, las tipografías y las imágenes según el ancho de pantalla.
- Las secciones de productos destacados y valoraciones pasan de una cuadrícula de varias columnas a una sola columna en móviles, asegurando que todos los elementos se mantengan visibles sin hacer zoom.
- En la sección *hero*, centramos todos los elementos y ajustamos márgenes

3.4 Carta

El componente **Carta** es uno de los ejes centrales del proyecto, ya que muestra el catálogo completo de productos disponibles en la pizzería, organizados por categorías como pizzas, burgers, baguettes, entrantes, bebidas o helados.

Integramos una barra de navegación horizontal donde se pueden seleccionar las distintas categorías. Al hacer clic en una de ellas, el sistema actualiza el estado local (**activeCategory**) y ejecuta una nueva llamada a la API para recuperar los productos de esa categoría. Esta lógica la gestionamos con el hook **useEffect**, que está atento a cualquier cambio en la categoría activa o en el estado de autenticación del usuario. Cada vez que alguno de estos elementos cambia, se ejecuta automáticamente una función asincrónica que consulta los productos en el backend mediante **ProductoDataService.getProductosByTipo()**.

Una vez recibimos los datos, realizamos un filtrado adicional: si el usuario no está autenticado, eliminamos del resultado aquellos productos que no están marcados como disponibles (**producto.disponible === false**). Esto asegura que los clientes solo vean los productos realmente disponibles para la venta.

Los productos se almacenan en el estado local (**productos**) y, a partir de ahí, utilizamos un bucle **.map()** para recorrer ese listado y generar dinámicamente una tarjeta (**producto-card**) por cada uno. Cada tarjeta muestra una imagen, el nombre del producto y su descripción. Si el usuario es administrador, también puede seleccionar una tarjeta para realizar acciones de edición o eliminación, y la interfaz lo destaca visualmente cambiando su estilo.

Cuando se detecta que el usuario tiene permisos de administración, aparecen botones adicionales para **añadir**, **editar** o **eliminar** productos. Estos botones solo se habilitan si previamente se ha seleccionado un producto, lo que evita errores de gestión o acciones sin contexto.

En la vista de cliente, al pulsar sobre un producto se abre un modal que muestra una versión ampliada de su imagen junto con su descripción completa. Este sistema mejora la experiencia de navegación, ya que evita redirecciones innecesarias y mantiene al usuario centrado en la página de la carta.

Diseño responsive

Para asegurar que la carta se vea correctamente en dispositivos de cualquier tamaño, hemos adaptado el diseño con varias medidas:

- En resoluciones bajas, la barra de categorías se vuelve desplazable lateralmente, lo que permite navegar entre secciones sin ocupar demasiado espacio.
- La cuadrícula de productos se reorganiza automáticamente: en pantallas grandes se muestran varias columnas, mientras que en móviles se reduce a una o dos columnas.

- El texto de los botones y las descripciones se ajusta al ancho de pantalla, y en dispositivos muy pequeños ocultamos algunos textos secundarios para priorizar la imagen y el nombre del producto.
- El modal de visualización también se adapta en tamaño y disposición, reduciendo el alto de las imágenes

3.5 Login, NuevoProducto y EditarProducto

Estos tres componentes están directamente relacionados con la gestión interna del sitio, y solo están disponibles para el usuario administrador. Los componentes **NuevoProducto** y **EditarProducto** comparten archivo css (**EditarProducto.css**) ya que son muy parecidos.

Login

El componente **Login** permite al administrador iniciar sesión introduciendo un usuario y contraseña. Para validar el acceso, comparamos los datos ingresados con valores predefinidos y aplicamos un **cifrado SHA-256** a la contraseña antes de verificarla, añadiendo una capa de seguridad básica. Si las credenciales son correctas, actualizamos el contexto (**UserContext**) para reflejar ese estado. A partir de ese momento, el usuario puede acceder a funciones exclusivas como la gestión de productos.

En caso de error, mostramos un mensaje informativo justo encima del formulario

NuevoProducto y EditarProducto

Estos dos componentes permiten al administrador añadir nuevos productos o modificar los ya existentes. Ambos utilizan un formulario con campos como nombre, descripción, precio, imagen, tipo de producto y disponibilidad. Al guardar los cambios, realizamos una llamada al backend usando el servicio **ProductoDataService** para actualizar o insertar los datos en la base de datos.

En el caso de edición, usamos **useParams** para obtener el ID del producto desde la URL y cargamos sus datos en el formulario mediante **useEffect**. Así, el formulario se rellena automáticamente con la información existente.

Diseño responsive

Todos estos componentes están diseñados para adaptarse correctamente a diferentes dispositivos:

- Centramos los formularios en pantalla y establecemos un ancho máximo para que se vean bien en monitores grandes, pero también ajustamos el padding y el tamaño de fuente en pantallas pequeñas.
- El contenedor principal (**edit-producto-container**) y la tarjeta de login se adaptan en altura y márgenes, conservando una estructura equilibrada incluso en móviles.

3.6 Localizacion

El componente **Localizacion** ofrece a los usuarios toda la información necesaria para realizar pedidos y conocer dónde se encuentra el restaurante.

Comenzamos con el **hero**, donde mostramos el logotipo de la pizzería sobre una imagen exterior del local, junto con el título de la sección. A continuación, mostramos dos columnas: una con la información de contacto y otra con el mapa.

En la parte izquierda incluimos las formas de realizar pedidos: un botón con el logotipo de **Just Eat** que redirige directamente al perfil del restaurante y un número de teléfono destacado. También añadimos el horario de apertura organizado por el tramo horario de tarde y el de noche.

En la parte derecha integramos un mapa interactivo de **Google Maps** que muestra la ubicación exacta de RedPizza. Esto se hace mediante un iframe que nos proporciona Google Maps.

Diseño responsive

Para que esta sección se adapte bien a todo tipo de pantallas, hemos aplicado varias medidas responsive:

- En pantallas pequeñas (menos de 768px), la distribución en columnas se convierte en un formato vertical. Las secciones de contacto y mapa se apilan una sobre otra para mejorar la legibilidad.
- Reducimos progresivamente el tamaño del logotipo, los textos y los iconos de Just Eat en función del ancho de pantalla.
- El mapa se adapta completamente al ancho del dispositivo, conservando su visibilidad y funcionalidad

3.7 Contacto

El componente **Contacto** permite a los usuarios enviar mensajes directamente al equipo del restaurante a través de un formulario de contacto. Este apartado está pensado para recibir consultas, sugerencias o cualquier tipo de comentario por parte de los clientes.

Mostramos una sección **hero** con el logotipo de RedPizza sobre un fondo exterior, seguido de un formulario central. El formulario contiene campos para nombre, email, teléfono, asunto y mensaje, así como un checkbox obligatorio para aceptar la política de privacidad. Esto asegura que cumplimos con los requisitos legales en cuanto a tratamiento de datos personales.

Utilizamos la librería **emailjs** para enviar los mensajes sin necesidad de configurar un servidor propio. Cuando el usuario pulsa "Enviar", los datos se empaquetan y se mandan directamente a una cuenta de correo electrónico configurada previamente en la plataforma de **EmailJS**. Si el mensaje se envía correctamente, mostramos una alerta de éxito y limpiamos los campos del formulario. En caso de error, mostramos una alerta informativa al usuario.

Diseño responsive

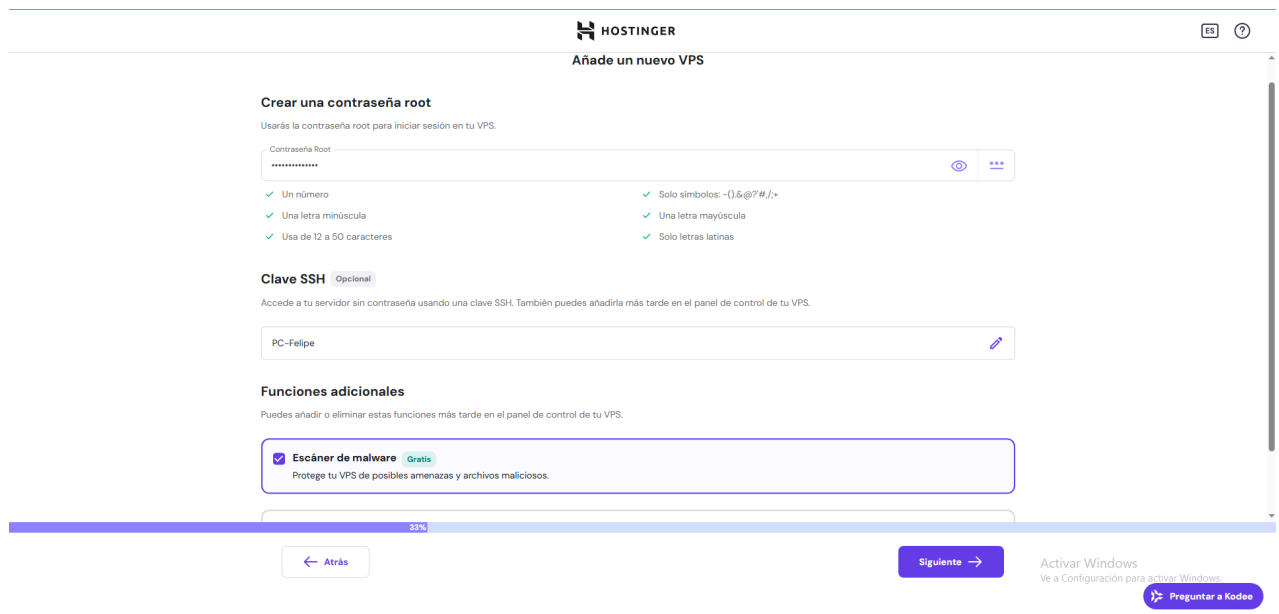
El diseño del formulario está totalmente adaptado para funcionar correctamente en todo tipo de dispositivos:

- En móviles, reducimos el tamaño del contenedor, los títulos y los campos, garantizando una visualización clara sin necesidad de hacer zoom.
- Reorganizamos el contenido para que ocupe el ancho completo en pantallas pequeñas, con especial atención a los campos de texto y al botón de envío.
- El checkbox de aceptación se adapta con un diseño más compacto, sin sacrificar la accesibilidad ni la claridad.

Desarrollo y explicación del despliegue en Hostinger

1. Contratación y configuración inicial del VPS

Antes de comenzar con el despliegue, contraté un servidor VPS a través de **Hostinger**, el cual incluía acceso vía **SSH** y seleccione como sistema operativo **Ubuntu** sin interfaz gráfica. Desde el panel de Hostinger seleccione la contraseña que quería como root y configure la clave SSH con mi ordenador personal para poder conectarme.



Una vez que se configuró e inició el VPS, accedí desde mi ordenador con el siguiente comando:

```
ssh root@IP_DEL_SERVIDOR
```

Una vez dentro, actualicé el sistema para poder descargar lo necesario para el despliegue con:

```
sudo apt update && sudo apt upgrade -y
```

Con esto, el VPS quedó listo para comenzar el despliegue del proyecto.

2. Preparación del servidor VPS

A continuación, instalé las herramientas necesarias para usar Docker, Docker Compose y Git en el VPS. Instalé **Docker**, **Docker Compose** y **git** con:

```
sudo apt install docker.io docker-compose git -y
```

Después, generé las claves SSH para autenticarme con GitHub:

```
ssh-keygen -t ed25519 -C "felipeinstitutoservet@gmail.com"
```

Copié la clave pública (`~/.ssh/id_ed25519.pub`) y la añadí a mi cuenta de GitHub. Luego cloné el repositorio del proyecto al servidor:

```
git clone git@github.com:FelipeAlonsoFernandez/TFG.git
```

3. Puesta en marcha del backend

Con el repositorio ya en el VPS, accedí al directorio donde se encontraba el archivo **docker-compose.yml** y ejecuté:

```
docker compose up -d
```

Esto levantó los contenedores de la base de datos y del microservicio de productos en segundo plano. Además, edité el archivo **docker-compose.yml** para exponer correctamente los puertos (`8000:8000`) y permitir la comunicación entre frontend y backend.

4. Configuración del entorno frontend

Instalé **Node.js** y **npm** con:

```
sudo apt install nodejs npm -y
```

También instalé y utilicé **pm2** para mantener el frontend ejecutándose continuamente, aunque el servidor se reiniciará en algún momento y que así la página web estuviese activa siempre:

```
sudo npm install -g pm2  
pm2 start npm --name "redpizza-frontend" -- run dev -- --host
```

-- host: este argumento adicional **se pasa directamente a Vite**, para que sirva el frontend en la IP pública del servidor, no solo en **localhost**.

5. Dominio, seguridad y acceso web

Adquirí un dominio personalizado desde **Hostinger** con el nombre **redpizza.org**. Desde su panel de control, configuré el registro **tipo A** para que apuntara a la IP pública de mi VPS.

Luego instalé **nginx**, que es un servidor web que se encarga de gestionar las peticiones que llegan al dominio. En mi caso, nginx actúa como **proxy inverso**, es decir, recibe las solicitudes del usuario y las redirige según el tipo de contenido: si es un enlace nuevo dentro del dominio, entrega los archivos del frontend de ese enlace; si es una petición a la API, la envía al backend. Esto me permite tener tanto el frontend como el backend bajo un mismo dominio.

```
sudo apt install nginx -y
```

Después instalé **Certbot** para nginx:

```
sudo apt install certbot python3-certbot-nginx -y
```

Con esta herramienta instale un certificado SSL gratuito a través de Let's Encrypt, es una Autoridad de Certificación gratuita y automatizada que emite certificados SSL/TLS. El certificado SSL permite que la web funcione con HTTPS, lo que cifra todas las comunicaciones entre el usuario y el servidor y mejora la seguridad del sitio. La instalación la realicé con el siguiente comando:

```
sudo certbot --nginx -d redpizza.org -d www.redpizza.org
```

-d redpizza.org -d www.redpizza.org: especifica los **dominios** para los que se quiere generar el certificado. En este caso, se incluyen tanto el dominio principal como su versión con "www"

Gracias a este paso, todos los datos que pasan entre el navegador del usuario y el servidor están protegidos.

6. Configuración del proxy con nginx

Para que tanto el frontend como el backend fueran accesibles desde el mismo dominio, creé un archivo de configuración para nginx en la ruta:

```
/etc/nginx/sites-available/redpizza.org
```

El contenido del archivo es el siguiente:

```
server {
    listen 80;
    server_name redpizza.org www.redpizza.org;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name redpizza.org www.redpizza.org;

    ssl_certificate /etc/letsencrypt/live/redpizza.org/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/redpizza.org/privkey.pem;

    location / {
        root /var/www/redpizza.org;
        try_files $uri $uri/ /index.html;
    }

    location /productos/ {
        proxy_pass http://localhost:8000/productos/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

```
server {  
    listen 80;  
    server_name redpizza.org www.redpizza.org;  
    return 301 https://$host$request_uri;  
}
```

Este **primer bloque** indica que si alguien intenta acceder a la web usando HTTP (puerto 80), nginx redirige automáticamente a la versión segura con HTTPS. Esto lo quería hacer para que no saliese el mensaje de web no segura cuando estabas en la pagina

```
server {  
    listen 443 ssl;  
    server_name redpizza.org www.redpizza.org;  
  
    ssl_certificate /etc/letsencrypt/live/redpizza.org/fullchain.pem;  
    ssl_certificate_key /etc/letsencrypt/live/redpizza.org/privkey.pem;
```

En este **segundo bloque**, nginx escucha en el puerto 443 (el puerto estándar para HTTPS). Especifico el nombre del dominio y le indico dónde se encuentran los certificados SSL generados previamente con Certbot. Esto activa la navegación segura con HTTPS.

```
location / {  
    root /var/www/redpizza.org;  
    try_files $uri $uri/ /index.html;  
}
```

Aquí se configura nginx para servir el frontend. Todo lo que llegue a / se buscará en la carpeta /var/www/redpizza.org, donde se encuentra el build generado con npm run build. La directiva try_files permite que React maneje las rutas internas correctamente, ya que si no se encuentra el archivo solicitado, se redirige a index.html.

```
location /productos/ {  
    proxy_pass http://localhost:8000/productos/;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-Forwarded-Proto $scheme;  
}  
}
```

Proxy inverso: es un servidor que se coloca frente a un servidor web y que actúa como un intermediario entre el cliente (navegador web) y el servidor de origen

Este bloque define un **proxy inverso** para las peticiones al backend. Todas las solicitudes que empiecen por `/productos/` se redirigen al microservicio que corre en `localhost:8000`. Las cabeceras añadidas aseguran que el backend reciba correctamente la IP del cliente y la información de la conexión original.

Una vez creado el archivo de configuración, lo habilité con el siguiente comando:

```
sudo ln -s /etc/nginx/sites-available/redpizza.org
/etc/nginx/sites-enabled/
```

Este comando crea un **enlace** entre **sites-available** y **sites-enabled**, qué es la forma en la que nginx activa sitios web.

Antes de aplicar los cambios, comprobé que no hubiera errores en la configuración con:

```
sudo nginx -t
```

Y finalmente recargué nginx para que leyera los nuevos archivos de configuración:

```
sudo systemctl reload nginx
```

Con esto, el dominio **redpizza.org** quedó totalmente funcional: el frontend sirve como sitio estático y las peticiones al backend se redirigen correctamente, todo bajo HTTPS.

7. Despliegue de producción

Finalmente generé el build de producción con:

```
npm run build
```

Y lo copié a la carpeta donde nginx sirve los archivos:

```
sudo cp -r dist/* /var/www/redpizza.org
```

A partir de ahí, cada vez que actualizo el frontend, simplemente hago:

```
git pull
npm run build
sudo cp -r dist/* /var/www/redpizza.org
```

Con estos pasos logré tener la web completamente funcional y con un sistema de despliegue simple y efectivo.

Conclusiones

Este proyecto no ha sido simplemente un trabajo de clase: lo he tratado como si fuera un encargo profesional real, porque así lo es. Desde el principio supe que esta web no iba a quedarse en una carpeta o en un repositorio abandonado, sino que iba a ponerse en marcha de verdad, y eso me obligó a cuidar cada detalle al máximo.

Todo lo que he hecho durante el desarrollo ha tenido un objetivo claro: que el resultado final sea útil, funcional y fiable, tanto para los clientes que la van a visitar como para quien la va a gestionar, que en este caso es mi padre. Por eso, enfoqué el proyecto como si estuviera trabajando para un cliente real, buscando no solo que funcionara, sino que quedara bien, se viera profesional y pudiera mantenerse en el tiempo.

A lo largo de este proyecto he podido aplicar de forma práctica muchos de los conocimientos adquiridos durante el grado superior, tanto en el desarrollo de aplicaciones web como en la gestión y despliegue de sistemas.

Durante el desarrollo del backend aprendí a organizar bien los microservicios y a trabajar con FastAPI y PostgreSQL de una forma clara y eficiente. Me resultó muy útil usar Docker para tenerlo todo separado y bien controlado, y así evitar problemas al desplegarlo.

En el frontend me centré mucho en el diseño visual y en que la experiencia fuera sencilla tanto en ordenador como en cualquier otro dispositivo. Me esforcé por hacer una interfaz llamativa, con todo bien estructurado y que se pudiera usar sin complicaciones.

Una de las partes más completas fue el despliegue. Me encargué de contratar el VPS, montar todo desde cero, configurar el dominio, el certificado HTTPS, nginx y dejar la web funcionando de forma pública. Aunque al principio me costó entender algunos pasos, al final conseguí tenerlo todo activo, seguro y con posibilidad de actualizar fácilmente.

En general, ha sido un trabajo donde he aprendido mucho, sobre todo porque he tenido que buscar soluciones reales a problemas reales. He tocado muchas áreas: desarrollo, diseño, bases de datos, seguridad y despliegue. Me voy con la sensación de haber hecho algo completo, útil. Saber que lo va a usar mi padre en su negocio ha sido una motivación extra para dar lo mejor de mí y no conformarme con algo “que funcione”, sino con algo que esté realmente bien hecho.