GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Desarrollo de una APP con integración de tecnologías inteligentes para gestionar la búsqueda de aparcamiento en la ciudad

Autor: Jesús Rueda Montes

Director: María Dolores Carnicero García

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Desarrollo de una APP con integración de tecnologías inteligentes para gestionar la búsqueda de aparcamiento en la ciudad

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2024/2025 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Jesús Rueda Montes Fecha: 02/07/2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado digitalmente por CARNICERO GARCIA MARIA DOLORES - 27479985E
Fecha: 2025.07.03 07:32:07 +02'00'

Fdo.: María Dolores Carnicero García Fecha: 02/ 07/ 2025

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Desarrollo de una APP con integración de tecnologías inteligentes para gestionar la búsqueda de aparcamiento en la ciudad

Autor: Jesús Rueda Montes

Director: María Dolores Carnicero García

Madrid

Agradecimientos

A todos aquellos maestros que, en algún momento, creyeron en mí;

a los amigos con los que he compartido y con los que sigo compartiendo camino;

y a mi familia, que en tantas ocasiones me ha levantado.

A ellos les debo todo.

DESARROLLO DE UNA APP CON INTEGRACIÓN DE TECNOLOGÍAS INTELIGENTES PARA GESTIONAR LA BÚSQUEDA DE APARCAMIENTO EN LA CIUDAD

Autor: Rueda Montes, Jesús.
Director: Carnicero García, María Dolores.
Entidad Colaboradora: ACLA INGENIERIA DE INSTALACIONES SL.

RESUMEN DEL PROYECTO

En este proyecto se ha dado una solución eficiente y escalable al problema de la difícil búsqueda de aparcamiento en la ciudad. Para ello, se ha desarrollado una aplicación móvil multiplataforma que permite al usuario consultar la disponibilidad de los parkings en tiempo real. Además, dicha aplicación integra tecnologías inteligentes como sensores de presencia y algoritmos de tratamiento de imágenes de cámaras de videovigilancia.

Palabras clave: Aparcamiento, Aplicación móvil, Sensor, Tratamiento de imágenes, Disponibilidad en tiempo real, Ciudades inteligentes, IoT

1. Introducción

Madrid es una gran ciudad para vivir. Sin embargo, al igual que otras muchas ciudades importantes del mundo, peca de ser una ciudad excesivamente masificada. Realizar tareas tan rutinarias como encontrar aparcamiento se ha convertido en un reto muchas veces imposible. El aumento del tráfico urbano acaba repercutiendo lentamente en la calidad de vida de los vecinos así como de los visitantes. Además, el consumo de combustible generado en estos desplazamientos innecesarios, no solo repercute en el bolsillo de los conductores, sino que además incrementa la emisión de gases contaminantes en un aire que respiran millones de personas. Con este panorama sobre la mesa y sabiendo que, al igual que Madrid, existen otras muchas ciudades masificadas a lo largo y ancho del globo, la búsqueda de una solución accesible, eficiente, sostenible y escalable se convierte en una prioridad.

2. Definición del proyecto

Ante este preocupante problema, se pretende dar una solución moderna y tecnológica que consiste en el desarrollo de una aplicación móvil multiplataforma accesible e intuitiva que facilite al usuario la búsqueda de aparcamiento libre en tiempo real. Para ello, la app está capacitada para dar soporte al uso colaborativo de datos que los propios usuarios señalizan en tiempo real. De forma complementaria, el sistema ofrece soporte para integrar los datos extraídos de diversas tecnologías inteligentes como son sensores de presencia o el tratamiento en tiempo real de imágenes con algoritmos de visión artificial.

Este proyecto no solo consiste en desarrollar una app móvil, sino que también busca crear un impacto positivo en la ciudad al cumplir los siguientes objetivos:

- Optimizar el tiempo que los ciudadanos dedican a buscar aparcamiento.
- Disminuir el impacto medioambiental nocivo derivado de esta actividad.

- Mejorar la calidad de vida de los ciudadanos y reducir su estrés.
- Ofrecer una solución eficiente y escalable a muchas otras ciudades del país y del mundo.

3. Descripción del sistema

La arquitectura general del sistema es la que a continuación se muestra:

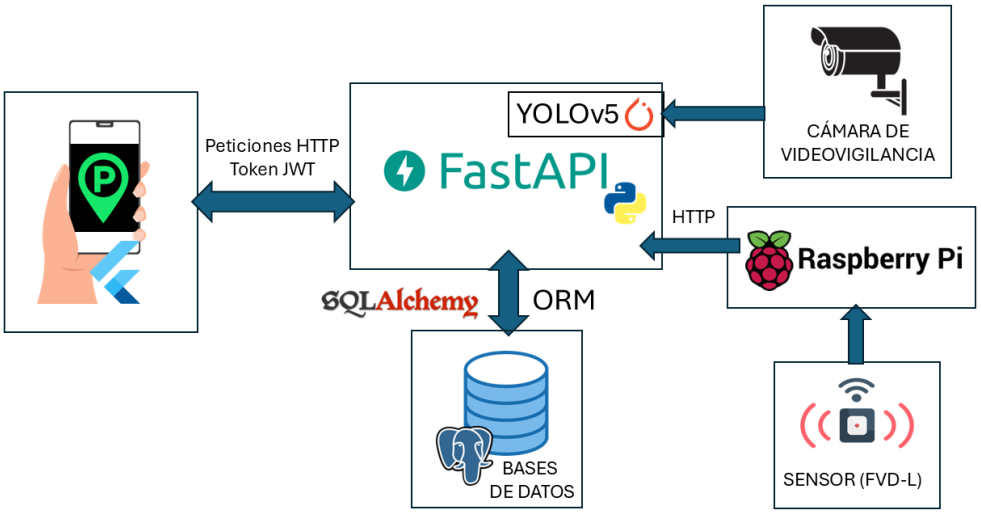

Ilustración 1: Arquitectura general del sistema

El sistema sigue el modelo de una arquitectura cliente-servidor en el que el servidor FastAPI [1] es el módulo más importante. En él convergen todos los datos del sistema y se ofrecen directamente al usuario (frontend) mediante peticiones HTTP. Además, para mejorar la experiencia de usuario, se mantiene el proceso de autenticación mediante tokens JWT. Los datos persistentes se almacenan en bases de datos PostgreSQL [2] y en la comunicación entre ambos bloques participa la herramienta SQLAlchemy ORM [3].

A su vez, la información extraída por los sensores de presencia (FVD-L) instalados físciamente en las plazas de parking, es directamente procesada por un dispositivo Raspberry Pi [4] y enviada al backend mediante peticiones HTTP. Por otro lado, las cámaras de videovigilancia de los aparcamientos envían su pipeline al servidor FastAPI, y el módulo interno lo procesa usando algoritmos de visión artificial (YOLOv5 [5]).

Como se puede apreciar, la aplicación funcional no es más que la punta del iceberg que el usuario ve de un sistema inteligente que integra a la perfección herramientas dispares y tecnologías modernas.

4. Resultados

La aplicación desarrollada, *ParkiNeo*, ofrece una interfaz moderna y accesible. Haz clic aquí. para ver un vídeo de uso de la aplicación.

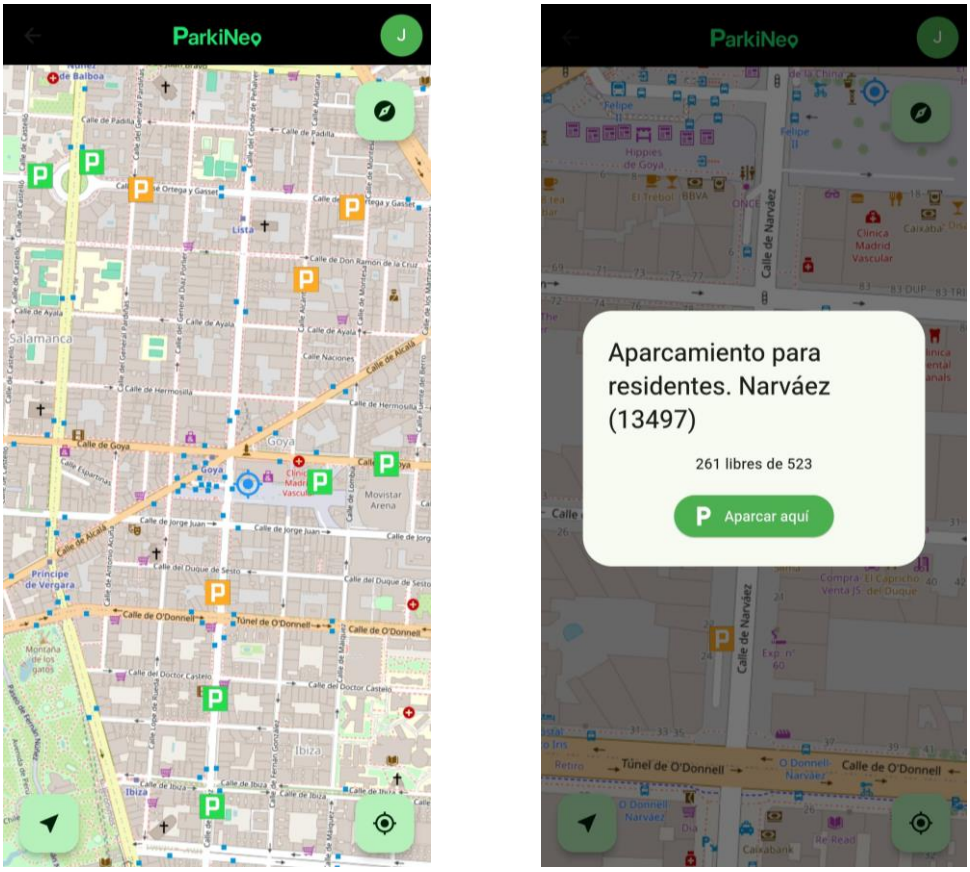

Ilustración 2: Capturas de mapa de ParkiNeo y tarjeta de parking

5. Conclusiones

El proyecto supone un avance significativo tras agilizar el proceso de búsqueda de aparcamiento. La integración de sensores de presencia y algoritmos de procesado de imagen garantiza una fiabilidad y precisión de los datos nunca vista en soluciones tecnológicas similares. Además, el uso colaborativo de datos entre usuarios ayuda a que la app siga siendo funcional y eficaz en todos aquellos aparcamientos que no tienen instalados sensores o cámaras de videovigilancia. Por último, cabe destacar que, tanto la interfaz de usuario como la experiencia de usuario son sencillas e intuitivas.

6. Referencias

[1] S. Rahman and H. Anwar, "FastAPI: Modern Web Framework for Building APIs with Python 3.7+," *FastAPI*, [Online]. Available: https://fastapi.tiangolo.com/

[2] PostgreSQL Global Development Group, "PostgreSQL," *PostgreSQL Documentation*, [Online]. Available: https://www.postgresql.org/

[3] SQLAlchemy, "SQLAlchemy ORM," *SQLAlchemy Documentation*, [Online]. Available: https://docs.sqlalchemy.org/

[4] Raspberry Pi Foundation, "Raspberry Pi," [Online]. Available: https://www.raspberrypi.com/

[5] G. Jocher et al., "YOLOv5 by Ultralytics," *GitHub Repository*, 2020. [Online]. Available: https://github.com/ultralytics/yolov5

DEVELOPMENT OF AN APP WITH INTEGRATION OF INTELLIGENT TECHNOLOGIES TO MANAGE PARKING SEARCH IN THE CITY

Author: Rueda Montes, Jesús.
Supervisor: Carnicero García, María Dolores.
Collaborating Entity: ACLA INGENIERIA DE INSTALACIONES SL.

ABSTRACT

This project provides an efficient and scalable solution to the challenge of finding parking in the city. A cross-platform mobile application has been developed that allows users to check real-time parking availability. Moreover, the application integrates smart technologies such as presence sensors and image processing algorithms using surveillance cameras.

Keywords: Parking, Mobile application, Sensor, Image processing, Real-time availability, Smart-cities, IoT

1. Introduction

Madrid is a great city to live in. However, like many other major cities around the world, it suffers from being excessively overcrowded. Carrying out such routine tasks as finding a parking spot has become a challenge, often an impossible one. The increase in urban traffic gradually ends up affecting the quality of life of both residents and visitors. Moreover, the fuel consumption generated by these unnecessary trips not only impacts drivers' wallets, but also increases the emission of polluting gases into the air breathed by millions of people. With this situation on the table, and knowing that, like Madrid, there are many other overcrowded cities around the globe, the search for an accessible, efficient, sustainable, and scalable solution becomes a priority.

2. Project definition

In light of this concerning issue, the aim is to provide a modern and technological solution consisting of the development of an accessible and intuitive cross-platform mobile application that helps users find available parking spots in real time. To achieve this, the app is designed to support the collaborative use of data reported by users themselves in real time. Additionally, the system is equipped to integrate data obtained from various smart technologies, such as presence sensors or real-time image processing using computer vision algorithms.

This project is not only about developing a mobile app, but also seeks to create a positive impact on the city by fulfilling the following objectives:

- Optimize the time citizens spend searching for parking.
- Reduce the harmful environmental impact resulting from this activity.
- Improve citizens’ quality of life and reduce their stress.
- Provide an efficient and scalable solution for many other cities across the country and around the world.

3. System description

The general architecture of the system is as shown below:

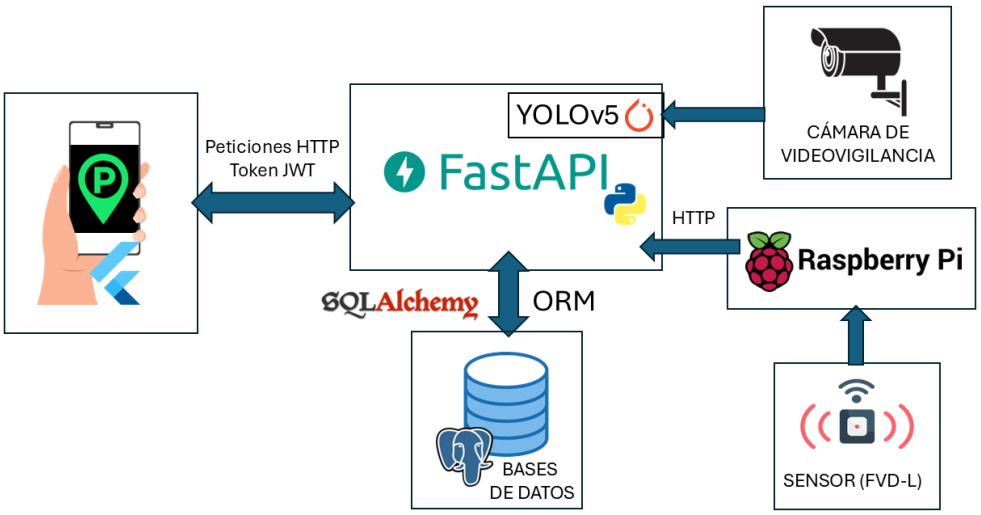

Illustration 1: Global System Architecture

The system follows a client-server architecture model in which the FastAPI [1] server is the most important module. It acts as the central hub where all system data converge and are delivered directly to the user (frontend) through HTTP requests. Additionally, to enhance the user experience, the authentication process is maintained using JWT tokens. Persistent data is stored in PostgreSQL [2] databases, and communication between these components is handled by the SQLAlchemy ORM tool [3].

At the same time, the information collected by the presence sensors (FVD-L) physically installed in the parking spaces is processed directly by a Raspberry Pi [4] device and sent to the backend via HTTP requests. On the other hand, parking surveillance cameras send their video streams to the FastAPI server, where an internal module processes them using computer vision algorithms (YOLOv5 [5]).

As can be seen, the working application is just the tip of the iceberg that users interact with—part of an intelligent system that seamlessly integrates diverse tools and modern technologies.

4. Results

The developed application, *ParkiNeo*, offers a modern and accessible interface. Click here to watch. a video demonstrating the app in use.

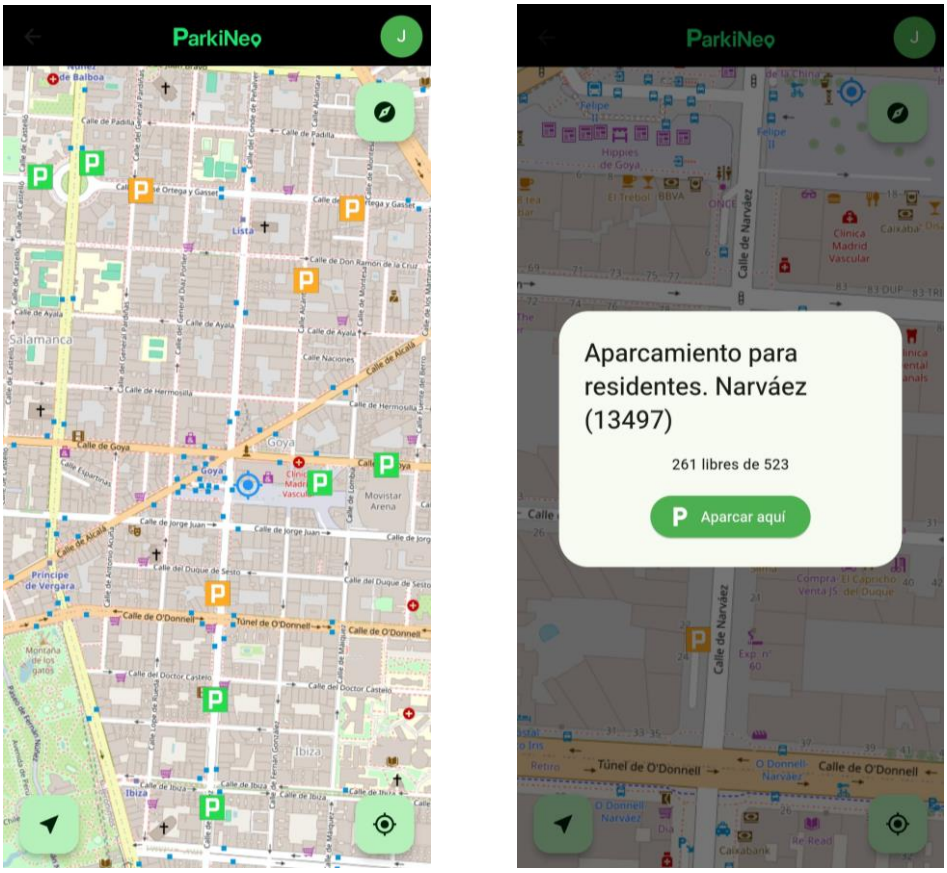

Illustration 2: Screenshots of the ParkiNeo map and parking card

5. Conclusions

The project represents a significant advancement by streamlining the parking search process. The integration of presence sensors and image processing algorithms ensures a level of data reliability and accuracy never before seen in similar technological solutions. Additionally, the collaborative use of data among users helps keep the app functional and effective in all parking areas that lack installed sensors or surveillance cameras. Finally, it is worth highlighting that both the user interface and the user experience are simple and intuitive.

6. Referencias

[1] S. Rahman and H. Anwar, “FastAPI: Modern Web Framework for Building APIs with Python 3.7+,” FastAPI, [Online]. Available: https://fastapi.tiangolo.com/

[2] PostgreSQL Global Development Group, “PostgreSQL,” PostgreSQL Documentation, [Online]. Available: https://www.postgresql.org/

[3] SQLAlchemy, “SQLAlchemy ORM,” SQLAlchemy Documentation, [Online]. Available: https://docs.sqlalchemy.org/

[4] Raspberry Pi Foundation, “Raspberry Pi,” [Online]. Available: https://www.raspberrypi.com/

[5] G. Jocher et al., “YOLOv5 by Ultralytics,” GitHub Repository, 2020. [Online]. Available: https://github.com/ultralytics/yolov5

Índice de la memoria

Índice de figuras

Índice de tablas

Capítulo 1. INTRODUCCIÓN

Madrid es una de las mejores ciudades del mundo. Es culturalmente enriquecedora, goza de una excelente vida nocturna y su gente es cercana, generosa y abierta. Sin embargo, al igual que otras muchas ciudades importantes del mundo, Madrid peca de ser una ciudad excesivamente masificada. Fijándose en los datos de 2023, el censo de la ciudad ascendió a los 3,3 millones de habitantes y, ese mismo año, la metrópoli fue visitada por 10,6 millones de turistas. A parte de esto, hay que tener en cuenta la gran cantidad de gente que vive en los municipios colindantes o "ciudades dormitorio" (Leganés, Getafe, Fuenlabrada, Móstoles y Alcorcón, entre otros) y que trabaja diariamente en Madrid. Con esta gran cantidad de afluencia, algunos recursos urbanos, como las plazas de aparcamiento, se ven altamente comprometidos.

La búsqueda de una plaza de aparcamiento público en el centro de la ciudad, por parte tanto de residentes como de visitantes, se ha convertido en un reto arduo, frustrante y, muchas veces, imposible. La ingente cantidad de tiempo invertido en la búsqueda de estacionamiento, así como el aumento del tráfico urbano, deteriora lentamente la calidad de vida de los conductores, pasajeros y transeúntes que ven cómo van aumentando los niveles de estrés en su día a día. Además, el consumo de combustible generado en estos desplazamientos innecesarios, aparte de repercutir en el bolsillo de los conductores, incrementa la emisión de gases contaminantes a un aire que respiran millones de ciudadanos, lo que termina afectando negativamente a su salud. Otros factores que tampoco ayudan a mejorar esta situación son los prohibitivos precios de las plazas de aparcamiento privadas y los largos tiempos de espera y la alta saturación en sus horas punta del transporte público.

Con este panorama sobre la mesa y sabiendo que, al igual que Madrid, existen otras muchas ciudades masificadas a lo largo y ancho del globo, la búsqueda de una solución accesible, eficiente, sostenible y escalable se convierte en una prioridad.

1.1 MOTIVACIÓN DEL PROYECTO

La difícil búsqueda de estacionamiento en el centro de una ciudad masificada como Madrid se ha convertido en un problema que está disminuyendo la calidad de vida, tanto de los conductores como de los ciudadanos en general. Los conductores pierden una ingente cantidad de tiempo buscando aparcamiento, y esto les puede llegar a generar una gran cantidad de estrés. Además, el innecesario gasto de combustible empleado en esta tarea repercute negativamente en sus bolsillos y también afecta negativamente a la calidad del aire que respiran. Todos los ciudadanos acaban viendo más deteriorada su salud.

A pesar de que existen algunas propuestas tecnológicas para tratar de solucionar este problema, no existe ninguna solución tan eficiente, accesible, sostenible y escalable como la que se propone en el presente proyecto. En este proyecto se pretende desarrollar una aplicación móvil accesible e intuitiva que facilite al usuario la búsqueda de aparcamiento libre en tiempo real. Para ello se van a integrar diversas tecnologías inteligentes que se complementan las unas a las otras (sensores físicos instalados, tratamiento de imágenes, acceso a recursos públicos, etc) y el uso colaborativo de datos proporcionados por los usuarios. Esta última opción no solo mejorará la precisión de los resultados, sino que, además, creará una comunidad activa que contribuirá a la promoción de la app.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

La solución tecnológica propuesta para este problema se basa en el desarrollo de una aplicación móvil. Dicha app sigue una arquitectura de cliente-servidor ya que la aplicación móvil se comunica con el servidor FastAPI mediante peticiones HTTP siguiendo el protocolo de API-REST. Las tecnologías, protocolos, bibliotecas y otras herramientas utilizadas en este proyecto se pueden clasificar en aquellas cuya funcionalidad está relacionada con el lado del cliente (frontend) o con aquellas que está relacionadas con el lado del servidor (backend).

2.1 HERRAMIENTAS FRONTEND

El frontend de un proyecto software abarca conceptos como la interfaz del usuario o la presentación de los datos al usuario. Se utilizan las tecnologías que a continuación se desarrollan:

2.1.1 FLUTTER

Flutter es un framework de interfaces gráficas desarrollado por la empresa Google y lanzado oficialmente en 2017 [1]. Su principal ventaja frente a otros frameworks similares es que permite el desarrollo multiplataforma; tanto en Android, como en iOS como en el desarrollo web o de aplicaciones de escritorio. Utiliza el lenguaje de programación Dart, que es ligeramente diferente en comparación con los lenguajes tradicionales de Kotlin (Android) o Swift (iOS).

En Dart, cada elemento de la interfaz es un widget, desde un simple botón hasta la propia pantalla completa, y la app se va desarrollando al combinar dichos widgets. Además, su enfoque es declarativo y permite la fácil integración con otras librerías. A parte de esto, Flutter tiene su propio motor de renderizado lo que permite que se puedan hacer pequeños cambios en tiempo real sin tener que reiniciar toda la app (hot reload).

Estos son algunos de los principales motivos por los que se ha optado por utilizar Flutter [1] como principal herramienta de desarrollo de este proyecto.

2.1.2 DEPENDENCIAS DE FLUTTER

2.1.2.1 Flutter_map y Latlong2

El mapa al que tiene acceso el usuario final es uno de los elementos más importantes de este proyecto. A través de él, el individuo es capaz de consultar la ubicación y la disponibilidad en tiempo real de los diferentes parkings. Para realizar esta tarea, se han usado las dependencias de flutter_map y de latlong2. La primera permite al usuario explorar el mapa, moviéndose por él y ampliando distintas zonas de interés. Además, flutter_map permite la opción de superponer el mapa con otros elementos como marcadores. En toda esta labor, se utiliza OpenStreetMap [2]; un dataset de datos geográficos de código abierto y gratuito (ideal para proyectos académicos o de bajo presupuesto). A su vez, latlong2 es imprescindible para trabajar con coordenadas (latitud y longitud) lo que permite mostrar elementos en el mapa de forma precisa y eficaz, como por ejemplo, los marcadores de los parkings.

2.1.2.2 Geolocator

Esta extensión de Flutter accede al GPS del dispositivo móvil después de que el usuario haya aceptado los permisos de uso del mismo. En este proyecto, este plugin es necesario para visualizar la ubicación en tiempo real del conductor en el mapa.

2.1.2.3 Http

Este paquete brinda la infraestructura necesaria para establecer la relación entre los datos almacenados en el servidor y la presentación de estos al usuario final. La comunicación entre ambas partes se realiza mediante peticiones HTTP y esta dependencia soporta todos los métodos típicos del modelo CRUD (Create, Read, Update & Delete). Es por ello que mediante este paquete se puede acceder a toda la información de los parkings registrados en la ciudad, se pueden añadir cámaras y sensores para que se automaticen los datos de disponibilidad en tiempo real y, entre otras cosas, los usuarios se puedan autenticar. Este

último proceso se realiza utilizando encabezados Bearer que envían el token JWT (alto grado de seguridad).

2.1.2.4 Flutter_secure_storage

Este plugin es necesario para almacenar eficientemente y de forma segura el token de autenticación del usuario que realiza un inicio de sesión. Gracias a esta funcionalidad, el usuario no tiene que introducir sus credenciales de inicio de sesión cada vez que accede a la aplicación, especialmente tras un breve tiempo de inactividad.

2.2 HERRAMIENTAS BACKEND

El backend de un programa informático hace referencia a todas aquellas herramientas y metodologías utilizadas para interconectar y guardar los datos del cliente y de la aplicación en un servidor.

2.2.1 FASTAPI

FastAPI es un framework web que en este software es utilizado para crear APIs con el lenguaje de programación de Python [3]. Está pensada para trabajar de forma asíncrona y sigue el estándar OpenAPI. Además, da soporte para garantizar la seguridad de los datos permitiendo el uso del protocolo OAuth2.

En este proyecto configura el núcleo central del backend ya que gestiona los datos de diferentes bloques importantes como lo son el acceso estructurado a la base de datos PostgreSQL, los datos extraídos por los sensores y la información recopilada del tratamiento de imágenes mediante modelos como YOLOv5. Aunque sería técnicamente posible realizar estas tareas por separado (por ejemplo, con scripts independientes en Python), FastAPI [3] permite unificarlas de forma escalable, segura y eficiente.

2.2.2 POSTGRESQL Y PGADMIN

PostgreSQL es uno de los sistemas de gestión de bases de datos más populares y utilizados en el desarrollo de aplicaciones y de software de arquitectura cliente-servidor [6]. Es una

herramienta de código abierto que destaca por su robustez y escalabilidad. Por dichas razones se ha elegido como principal gestor de bases de datos de este proyecto.

Para gestionar los datos y trabajar con una interfaz gráfica más intuitiva y flexible se ha elegido la herramienta auxiliar PgAdmin. De esta manera, visualizar, editar y extraer datos de las diferentes bases de datos han sido tareas más cómodas de realizar.

2.2.3 SQLALCHEMY Y PYDANTIC

Una de las herramientas de mapeo objeto-relacional (ORM en inglés) más flexible hasta la fecha es SQLAlchemy. Esta herramienta es la encargada de trabajar con clases y objetos en lugar de con consultas SQL directamente. Esto facilita el trabajo del desarrollador y hace que el código sea más limpio y legible.

A su vez, Pydantic es una herramienta de validación de datos que es capaz de integrarse de forma nativa con FastAPI. Resulta muy útil para comprobar que los datos enviados y recibidos por el cliente cumplen el formato especificado y así se consiguen evitar muchos errores de inconsistencia.

En resumen, ambas herramientas trabajan de forma conjunta: SQLAlchemy facilita la gestión de la interacción de los datos almacenados y Pydantic valida que dichos datos cumplen el formato preestablecido.

2.2.4 AUTENTICACIÓN Y SEGURIDAD: JWT + BCRYPT + OAUTH2

Para el proceso de autenticación, registro e inicio de sesión se han utilizado distintas herramientas que, aparte de conseguir satisfactoriamente que se pueda cumplir con la funcionalidad prevista, garantizan la seguridad en todas las partes del proceso.

Por un lado, JWT (JSON Web Token) es la tecnología capaz de generar los tokens de sesión y hacer que sea más eficiente el proceso de autenticación posterior. Por otro lado, bcrypt es el algoritmo encargado de hashear las contraseñas del usuario para su posterior

almacenamiento y evitar la pérdida de información del cliente en caso de robo de bases de datos. Por último, OAuth2 es un protocolo que actúa integrado en FastAPI para gestionar, de forma segura, la comparación de contraseñas en el procedimiento de la autenticación.

2.2.5 UVICORN

Es el servidor ASGI (Asynchronous Server Gateway Interface) que ejecuta la aplicación FastAPI en este trabajo. Levanta el servidor y permite que la API reciba peticiones HTTP. Es el motor que pone en marcha el backend. Es muy popular por su rapidez y ligereza, lo que lo hace ideal para aplicaciones orientadas a APIs REST o a microsevicios.

2.3 HERRAMIENTAS DE TECNOLOGÍAS INTELIGENTES

A continuación se expone un análisis de las herramientas y librerías que gestionan la información extraída por las tecnologías inteligentes del tratamiento de imágenes y la implementación de los sensores magnetorresistivos.

2.3.1 YOLOv5

Esta moderna librería (de Pytorch) es el corazón del tratamiento de imágenes en el proyecto al ser capaz de detectar, de forma inteligente, los automóviles en las imágenes de las cámaras de videovigilancia [4]. Se trata de un modelo de visión artificial preentrenado con millones de imágenes y capaz de detectar cientos de objetos (no solo coches). Utiliza un enfoque de detección de una sola pasada (one-stage detection) que hace que esta tecnología ofrezca un rendimiento excelente en tareas en tiempo real.

2.3.2 OPENCV

Esta es una de las liberías más famosas y utilizadas en el campo de la visión artificial por su versatilidad y sencillez [5]. Permite realizar todo tipo de operaciones con imágenes (desde las más sencillas hasta las más complejas). En este estudio se ha realizado para extraer fotogramas en tiempo real a partir de una URL asociada a una cámara de videovigilancia para su posterior análisis con YOLOv5.

2.3.3 GPIOZERO

Se trata de una biblioteca de Python ampliamente utilizado en el ámbito de la electrónica y la automatización de la mano del dispositivo Raspberry Pi. Interactúa con los pines GPIO (General Purpose Input/Output) para elevar el nivel de programación del manejo en los sensores.

En este proyecto, esta librería se ha utilizado concretamente para simular el funcionamiento de una Raspberry Pi conectada a un sensor magnetorresistivo (de tipo FVD-L). La Raspberry Pi actúa como dispositivo intermedio entre el sensor y el almacenamiento de datos al leer el estado del sensor y trasmitirlo, en tiempo real, al backend a través de una petición HTTP.

2.4 HERRAMIENTAS AUXILIARES

Finalmente, se analizan otras herramientas utilizadas y que han sido de vital importancia para la consecución del éxito en este proyecto:

2.4.1 POSTMAN

Se trata de una herramienta de desarrollo y trabajo de APIs. Prueba, valida y depura peticiones HTTP sin necesidad de ejecutar la interfaz del cliente final. Permite simular llamadas al servidor FastAPI, visualizar sus respuestas y comprobar que los endpoints están funcionando correctamente.

En este proyecto se ha utilizado para verificar el correcto funcionamiento de los procesos de autenticación, manipulación de datos de los parkings, incorporación de cámaras y simulación de sensores.

2.4.2 VISUAL STUDIO CODE

Quizás el editor de código fuente más popular y utilizado en todo el mundo por programadores y desarrolladores de software [7]. Se ha elegido en este proyecto por ser muy potente, versátil y flexible para trabajar con todas las herramientas expuestas anteriormente (tanto del lado del frontend como del lado del backend).

Capítulo 3. ESTADO DE LA CUESTIÓN

Antes de embarcarse en la aventura de diseñar y desarrollar una aplicación móvil que sirva para solucionar el problema de la difícil búsqueda de aparcamiento en la ciudad, conviene estudiar las posibles soluciones existentes a este problema. De esta forma, no solo se reconoce el estado del arte en esta materia, sino que, además, se detectan todas aquellas carencias existentes en las soluciones propuestas y se tratan de subsanar en la etapa de diseño del software.

3.1 APLICACIONES DE NAVEGACIÓN Y TRÁFICO GENERAL

0otros. Estas aplicaciones brindan al usuario todo tipo de información necesaria para la correcta circulación de mismo, como la ruta óptima para llegar a un destino, la velocidad máxima permitida en la vía o el tiempo estimado del trayecto. Además, también son capaces de brindar información en tiempo real como el estado del tráfico de determinadas vías o la existencia de radares o controles de circulación por parte de las autoridades. También pueden resultar excelentes opciones a la hora de buscar parkings públicos o privados para estacionar, tanto en la ciudad como en las zonas interurbanas. No obstante, estas aplicaciones ofrecen muy poca información en lo referido a la disponibilidad en tiempo real de dichas plazas de aparcamiento. Otras carencias que tienen estas aplicaciones son las referidas a informar sobre las restricciones específicas existentes acerca del estacionamiento disponible o las diferentes normativas aplicables a cada calle o vía.

3.2 APLICACIONES DE APARCAMIENTO

Algunos servicios informáticos como *Parclick*, *Parkopedia* o, en menor medida, *ElParking*, [8] se centran en la búsqueda de aparcamientos privados o de pago. Desde estas aplicaciones los usuarios finales son capaces de reservar con antelación un estacionamiento, comparar precios y realizar pagos a través de la app. Esto puede llegar a mejorar la experiencia del

usuario en ciertos contextos. No obstante, estos programas software tienen como inconveniente que solo se centran en aparcamientos de índole privada y eso repercute negativamente en su alcance y escalabilidad, que se ven altamente limitados por depender de operadores de aparcamientos externos. Además, no incorporan tecnologías inteligentes (como, por ejemplo, el tratamiento de imágenes o la integración de datos extraídos de sensores) ni el uso colaborativo de datos entre usuarios para actualizar la disponibilidad de los parkings en tiempo real.

3.3 SISTEMAS BASADOS EN SENSORES

En algunas ciudades como San Francisco (*SFpark*) o Barcelona (*SMOU*) [9] se ha optado por la instalación de sensores físicos en las plazas de aparcamiento. Estos sensores suelen estar instalados en el pavimento de las plazas de estacionamiento y envían su información a través de una API para que sea accesible por el usuario final a través de una app. Esta tecnología inteligente da resultados precisos, fiables y en tiempo real. No obstante, presenta algunos inconvenientes como son los altos costes de instalación y mantenimiento y la limitación en su escalabilidad. En ciudades como Madrid, donde la cobertura abarca múltiples distritos, esta solución puede resultar muy costosa e inviable, al menos, sin un plan municipal específico.

3.4 USO DE DATOS ABIERTOS Y FUENTES MUNICIPALES

Administraciones de ciertas ciudades, como la de Nueva York, publican portales de movilidad urbana [10]. En estas plataformas se puede consultar información acerca de parquímetros, normativas y otra información útil para la búsqueda de aparcamiento. A pesar de que estas herramientas resultan muy útiles para programadores, quienes pueden incluir esta información en el desarrollo de sus programas software, estos portales no suelen estar diseñados para ser interpretados directamente por los usuarios finales. No constan de herramientas para interpretar automáticamente la información y, además, los datos pueden no ser del todo fiables o pueden estar desactualizados.

3.5 TRATAMIENTO DE IMÁGENES Y VISIÓN ARTIFICIAL

Con el creciente desarrollo de las tecnologías de inteligencia artificial (deep learning y machine learning), se ha ido explorando el uso de algoritmos de visión artificial para automatizar y dar solución al problema del aparcamiento en la ciudad. A través de la captura de imágenes en tiempo real, se podrían detectar plazas libres en tiempo real. A pesar de que está técnica ha dado buenos resultados en entornos controlados con aparcamientos cerrados, su despliegue en entorno urbanos abiertos sigue siendo un reto a día de hoy. Hay una serie de factores técnicos que influyen crucialmente en el éxito de esta tarea como son: la variabilidad de la iluminación y de las condiciones meteorológicas y el alto coste que supone la necesidad de tener imágenes de alta calidad, entre otros.

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

Vivir en una gran ciudad ofrece muchas ventajas como, por ejemplo, tener acceso a mayores zonas comerciales o tener una mayor oferta de actividades de ocio, cultura y deporte. No obstante, la alta masificación de la ciudad hace que se pierda mucho tiempo en actividades tan rutinarias como lo es buscar aparcamiento. Lo más preocupante de todo es que esto acaba afectando negativamente a la calidad de vida de los ciudadanos.

Desde hace aproximadamente quince años, con la aparición de los primeros smartphones y con el auge que ha experimentado el desarrollo de aplicaciones móviles comerciales, se ha tratado de dar una solución tecnológica en forma de software de aplicación móvil. Muchas de estas soluciones son eficaces y cuentan con una interfaz de usuario fácil e intuitiva. No obstante, en los últimos años se ha podido experimentar una auténtica revolución en el campo de la inteligencia artificial. No solo se han producido avances en el área de la inteligencia artificial generativa, sino que también en otros muchos como en el del tratamiento de imágenes y la visión artificial.

La solución propuesta en este proyecto no consiste en otra aplicación genérica más, que trata dar una solución suficiente a este problema. La solución que se propone en este proyecto integra las tecnologías más punteras e innovadoras de los últimos meses, con otras tecnologías que han demostrado ser efectivas, para brindar al ciudadano la mejor experiencia de uso con los resultados más eficaces hasta la fecha. De esta manera, la calidad de su vida y la de sus generaciones venideras se verá altamente recompensada.

4.2 OBJETIVOS

El presente proyecto tiene los siguientes objetivos que a continuación se describen:

4.2.1 DESARROLLAR UNA APP MÓVIL

Desarrollar una aplicación móvil para la búsqueda de aparcamiento en tiempo real. Dicha app debe facilitar a los usuarios la localización de aparcamiento en tiempo real. Para ello, se dará cobertura al intercambio de datos entre usuarios para verificar y reportar acciones de aparcamiento y de abandono de parking. Además, integra las últimas tecnologías inteligentes como son los sensores de posición y los algoritmos de procesado de imágenes obtenidas de cámaras de videovigilancia. De esta manera, el sistema se adapta a los cambios en tiempo real y mejora la precisión de los resultados. La aplicación es gratuita y cuenta con una interfaz de usuario sencilla e intuitiva.

4.2.2 OPTIMIZAR EL TIEMPO DE BÚSQUEDA DE APARCAMIENTO

Optimizar el tiempo que los conductores dedican a buscar aparcamiento. Lograr que buscar estacionamiento en una gran ciudad sea una tarea sencilla y accesible a cualquier ciudadano residente o visitante. De esta manera, se busca reducir el tráfico urbano y los largos tiempos de espera de circulación en la metrópoli.

4.2.3 DISMINUIR EL IMPACTO MEDIOAMBIENTAL

Disminuir el impacto medioambiental derivado de la lenta búsqueda de aparcamiento en las ciudades masificadas. Al agilizar el proceso de búsqueda de aparcamiento, se consigue disminuir el consumo de combustible y las emisiones de gases de efecto invernadero derivadas de dicha actividad. De esta manera, se logra tener una ciudad más sostenible siendo más limpio y sano el aire que respiran millones de ciudadanos.

4.2.4 MEJORAR LA CALIDAD DE VIDA DE LOS CIUDADANOS

Mejorar la calidad del aire que respiran conductores y ciudadanos también mejora sus respectivas calidades de vida. Disminuir el consumo de combustible en la ciudad no solo tiene ventajas sostenibles y de salud, sino que también supone un ahorra económico de los conductores. Circular por una ciudad menos congestionada reduce el estrés de los viandantes y contribuye a crear un entorno más habitable y con menor contaminación acústica.

4.2.5 DAR UNA SOLUCIÓN ESCALABLE

Ofrecer una solución tecnológica accesible, eficiente, sostenible y escalable. Se busca encontrar una solución que no responda solo a las necesidades locales de la ciudad de Madrid, sino que también sea escalable y replicable en otras muchas ciudades masificadas a lo largo y ancho del mundo. De este modo, se consigue instaurar un modelo de movilidad urbana más accesible, eficiente y sostenible.

4.3 METODOLOGÍA

Para desarrollar este proyecto software se ha seguido la metodología Agile [11]. Este modelo destaca frente a otras metodologías tradicionales, como el enfoque en cascada, por tratar de conseguir pequeños resultados inmediatos. El desarrollo de software se ve involucrado en una continua mejora al realizarse en pequeños ciclos flexibles e iterativos. Dichos ciclos consisten en una serie de tareas a efectuar y están agrupados en unos bloques llamados sprints (de una o dos semanas de duración). A continuación, se detallan las tareas que se han ido agrupando en distintos sprints a lo largo del desarrollo de este proyecto:

- **Diseño conceptual.** Definición de los requisitos funcionales de la aplicación, así como el diagrama de casos de uso y otros esquemas de interés de la arquitectura software de la app.
- **Obtención de recursos públicos.** Para las primeras pruebas de visión artificial se han obtenido cámaras de parkings de cualquier lugar del mundo y se han usado como parkings objeto de pruebas. En paralelo, para los aparcamientos locales de la ciudad de Madrid, se ha contado con APIs públicas disponibles y accesibles desde la página oficial del Ayuntamiento de Madrid [12]. En dichas APIs se han podido consultar información (como el nombre, la ubicación y la capacidad) de los distintos tipos de aparcamientos existentes en la ciudad (aparcamientos municipales, aparcamientos PAR y aparcamientos para residentes).
- **Visualización del mapa (OpenStreetMap).** Se ha optado por el uso de las librerías open-source y gratuitas de OpenStreetMap [2]. Una vez que el usuario inicia sesión

en la app, este puede ver el mapa de su ciudad y del mundo y navegar por el mismo, cambiando el foco y el zoom, de manera interactiva.

- **Acceso a la ubicación del dispositivo móvil.** Configurando y aceptando los correspondientes permisos del sistema operativo, la ubicación del móvil del usuario es accesible desde la aplicación y se puede mostrar en el mapa.
- **Implementación de la estructura backend (bases de datos de parkings y usuarios).** Utilizando la herramienta de PostgreSQL [6] se han creado las bases de datos con las tablas de parkings y de usuarios. La base de datos que gestiona los parkings tiene como parámetros más reseñables, la id (clave primaria), el nombre del parking, la ubicación del mismo (coordenadas de latitud y longitud) y el número de aparcamientos disponibles y totales de dicho parking. La tabla de usuarios se dedica exclusivamente al inicio de sesión y autenticación.
- **Autenticación, registro e inicio de sesión.** Este proceso incluye la generación y el almacenamiento del token JWT usado en el proceso de autenticación del usuario. Además, toda la información referida al inicio de sesión del conductor se guarda en la tabla de usuarios. Esta tabla tiene como parámetros destacados el nombre del mismo y su contraseña hasheada con el algoritmo de bcrypt.
- **Visualización de los parkings y acceso a su información.** Teniendo acceso a los parámetros de la ubicación en cada parking, se imprime un marcador (una imagen de un cartel de parking) en la exacta localización del mismo. De esta manera, el usuario al pinchar en dicho marcador puede leer la información en una tarjeta informativa y consultar la disponibilidad en tiempo real de dicho parking.
- **Datos colaborativos en tiempo real.** Los usuarios trabajan conjuntamente para dar la información más fiable y precisa en tiempo real. Cuando un conductor estaciona en una plaza de aparcamiento lo comunica al sistema y dicho sistema se encarga de actualizar el backend. Cualquier otro perfil puede leer la información de la disponibilidad actualizada en tiempo real. Lo mismo ocurre cuando un conductor anuncia que abandona el parking en el que ha estado estacionado.
- **Tratamiento de imágenes.** Aparte del uso colaborativo de datos, el sistema tiene la capacidad de incorporar imágenes de videovigilancia de los estacionamientos.

Accediendo a estas imágenes, y mediante el uso de algoritmos de visión artificial basados en YOLO (You Only Look Once) [4], el sistema es capaz de detectar el número exacto de automóviles que se encuentran en el parking en tiempo real.

- **Incorporación de la información de los sensores.** Si las plazas individuales de parkings han sido dotadas de sensores físicos, se puede integrar su información en el backend y mostrar la disponibilidad de los parkings actualizada por estos sensores en tiempo real. Además, los sensores son más fiables que los algoritmos de visión artificial y que las acciones señaladas por los usuarios.
- **Funciones de administrador: asociación de cámaras a parkings.** La figura del administrador es de gran importancia en este proyecto. El administrador es el único ente capaz de asociar los sensores y las cámaras de videovigilancia a los parkings. Puede cambiar el número de plazas totales (y disponibles) de cada aparcamiento y también es el responsable de acceder a los datos municipales de aparcamiento de la ciudad y gestionar dicha información para integrarla en la app.
- **Búsqueda de aparcamiento más cercano.** Para que este proyecto ofrezca una solución más cómoda y sostenible, se ha incorporado la función de que el conductor sepa y pueda acceder al aparcamiento disponible más cercano.
- **Pruebas y depuración de errores.** A medida que se ha ido avanzando en el desarrollo de la app y de forma paralela, se ha ido trabajando en simulaciones de prueba con el modelo backend (usando la herramienta de Postman). En dicho proceso se han solucionado todos los errores que han ido surgiendo.
- **Redacción del informe.** De forma paralela al desarrollo de la app se ha ido redactando el presente informe. Aparte de ser una parte primordial del proyecto, el hecho de que quede documentado todo el proceso facilita los análisis en futuras etapas de validación y evolución de la app.

4.4 PLANIFICACIÓN

A continuación, se presenta el cronograma del proyecto en forma de diagrama de Gantt. En este, se aprecian las actividades realizadas y la duración con la semana de inicio y de final de las mismas:

Mes	*Noviembre*				*Diciembre*				*Enero*			*Febrero*	
Días del mes	4 - 10	11 - 17	18 - 24	25 - 1	2 - 8	9 - 15	16 - 22	23 - 30	13 - 19	20 - 26	27 - 2	3 - 9	10 - 16
Diseño conceptual													
Obtención de recursos													
Mapa visible													
Ubicación accesible													
Estructura backend													
Autenticación													
Marcadores de los parkings													
Datos colaborativos													

Tabla 1: Planificación del proyecto de noviembre a febrero

Mes	*Febrero*		*Marzo*				*Abril*			*Mayo*			
Días del mes	17 - 23	24 - 2	3 - 9	10 - 16	17 - 23	24 - 30	31 - 6	21 - 27	28 - 4	5 - 11	12 - 18	19 - 25	26 - 1
Integración con sensores													
Tratamiento de imágenes													
Funciones de administrador													
Búsqueda de aparcamiento más cercano													
Testing y depuración de errores													
Redacción del informe													

Tabla 2: Planificación del proyecto de febrero a junio

4.5 ESTIMACIÓN ECONÓMICA

A pesar de que este proyecto se ha realizado principalmente con un fin académico, conviene evaluar el coste económico de una hipotética implementación real y, así, estudiar su

viabilidad económica real. Para ello, se realizan distintas estimaciones de los diferentes costes como son los costes del desarrollo, los costes de la infraestructura, los costes del hardware empleado y los costes de otros servicios utilizados.

4.5.1 COSTES DE DESARROLLO

Este proyecto ha sido desarrollado en un contexto académico. Para estimar los costes que hubiese supuesto desarrollar este trabajo en un entorno profesional, conviene evaluar el tiempo empleado en dicho desarrollo e investigar la tarifa media por tiempo de desarrollo.

Se han empleado un total de unas 300 horas aproximadamente entre tareas de análisis, diseño, implementación y depuración de errores software. Si se estima que el coste de la tarifa media de trabajo de desarrollo es de unos 25 € por hora (que es un precio habitual en los proyectos software freelance de complejidad media), el coste teórico del desarrollo se calcularía así:

$$300\ horas \times 25 \frac{€}{hora} = 7.500\ €$$

En resumen, si este proyecto hubiese sido llevado a cabo por un equipo de desarrolladores en un entorno profesional, se estima que el coste total de la fase de desarrollo hubiese ascendido hasta los 7.500 € aproximadamente.

4.5.2 COSTES DE INFRAESTRUCTURA

Durante las pruebas y la etapa de desarrollo, el servidor FastAPI [3] ha sido lanzado con un equipo informático local (el del desarrollador). No obstante, si esta aplicación se quisiera lanzar comercialmente y para todos los públicos, habría que sumar unos costes anuales derivados de la infraestructura; posiblemente un servidor en la nube que alojase la base de datos y la API REST del proyecto. Se estiman los siguientes costes (anuales):

Concepto	*Servicio estimado*	*Coste anual aproximado*

Hosting de la API REST (FastAPI)	Render, Railway o Heroku	60 – 120 €
Base de datos PostgreSQL [6]	Plan básico cloud (Railway)	60 – 100 €
Certificado SSL y dominio	Namecheap, Cloudflare	20 – 30 €

Tabla 3: Costes de infraestructura

En conclusión, el coste total de infraestructura estimado (anual) rondaría entre los 140 y 250 € aproximadamente, en función del proveedor elegido y la cantidad de uso dado.

4.5.3 COSTES DE HARDWARE

En el funcionamiento real más básico de la aplicación se puede obviar el uso de sensores y de cámaras de video. No obstante, la incorporación de estas herramientas al proyecto supone un salto de calidad notorio que se ve reflejado en un avance en la precisión y la fiabilidad de los datos con los que se trabaja.

Adquirir e instalar una cámara de videovigilancia económica puede costar, aproximadamente, entre unos 40 y 80 €. Teniendo en cuenta que harían falta, como mínimo, dos cámaras de videovigilancia (una que detecte los vehículos que entran y otra que detecte los vehículos que abandonan el parking [4]), el precio aproximado de incluir cámaras de videovigilancia sería de 120 €. El precio podría ascender si se opta por desplegar una red de cámaras que cubre el total de las plazas de un aparcamiento.

No obstante, se podría considerar la opción de reutilizar las cámaras de videovigilancia ya instaladas en los aparcamientos y conseguir ahorrar una gran cantidad de dinero. Sin embargo, cabe recordar que pueden existir impedimentos técnicos y, sobre todo, costosos impedimentos legales (como la obtención de permisos de uso).

A parte de las cámaras de videovigilancia, se pueden instalar, de forma complementaria o sustitutiva, sensores de presencia (son aún más fiables que el tratamiento de imágenes). Después de realizar un estudio de mercado, se ha optado, en esta primera versión, por el uso de sensores magnetorresistivos de tipo FVD-L. Estos sensores son una opción fiable, económica y de bajo consumo. Cada sensor debe ir conectado a un dispositivo electrónico Raspberry Pi [13] para poder procesar los datos extraídos.

A continuación, se presenta el coste estimado de la instalación de un único sensor magnetorresistivo de tipo FVD-L y de otras herramientas necesarias para procesar sus datos:

Componente	*Unidad*	*Coste unitario (€)*
Sensor FVD-L (magnetorresistivo)	Sensor físico	12 €
Raspberry Pi 4 (4 GB RAM)	Microcontrolador	60 €
Fuente de alimentación + cable	Accesorio	10 €
Adaptador GPIO / resistencias	Electrónica	3 €

Tabla 4: Coste derivado de la instalación de un sensor de presencia

En total, la suma de la instalación de un sensor (prototipo) llegaría a los 85 €. No obstante, cada dispositivo Raspberry Pi puede dar cobertura a entre 20 y 30 sensores y la fuente de alimentación puede servir para alimentar a entre 10 y 20 sensores. Esto hace que el coste unitario de la instalación de un sensor sea de unos 16 € aproximadamente.

4.5.4 RESUMEN Y CONCLUSIÓN

Este proyecto ha sido desarrollado sin una inversión económica directa y se ha tratado de utilizar en todo momento herramientas y recursos de libre acceso. No obstante, si está aplicación hubiese sido lanzada por un equipo de desarrollo profesional, el total de los costes sería el siguiente:

- Costes de desarrollo: aproximadamente 7.500 €.
- Costes de infraestructura: aproximadamente entre 140 y 250 € anuales.
- Costes de hardware (opcionales pero recomendables):
    - Cámaras de videovigilancia: Mínimo 120 € aproximadamente.
    - Sensores magnetorresistivos: aproximadamente 16 €/sensor.
- Costes de licencias y software: 0 € (uso exclusivo de programas open source).

En resumen, el coste total estimado del proyecto sería de unos 8.000 € aproximadamente.

A medida que el sistema escalara y alcanzase más ciudades con más aparcamientos y más usuarios dispuestos a utilizar la aplicación, estos costes crecerían, sobre todo en lo relativo al despliegue de cámaras y sensores.

Capítulo 5. SISTEMA DESARROLLADO

Para dar con la solución tecnológica innovadora e intuitiva necesaria, primero se han tenido que analizar los requisitos del sistema. Teniéndolos en cuenta, se han propuesto diferentes esquemas de desarrollo que han derivado en la implementación real de la app. A continuación, se explican estas distintas fases de forma técnica y detallada.

5.1 ANÁLISIS DEL SISTEMA

El análisis del sistema tiene como eje central la mejora de la experiencia del usuario. Para ello se han estudiado los **requisitos funcionales** que son las capacidades mínimas que debe de cumplir el sistema para dar una solución eficiente y de calidad. A partir de estos requisitos mínimos funcionales, se puede añadir cualquier otro tipo de mejoras.

Pero no solo es importante analizar qué se va a hacer el sistema, sino que también es importante estudiar el cómo debe hacerlo. Es aquí donde entran en juego los **requisitos no funcionales**. Ambos tipos de requisitos dan una base sólida y especifican correctamente el sistema a desarrollar.

Por último, cabe resaltar los **supuestos** mínimos indispensables para que se pueda dar el correcto funcionamiento de la aplicación y revisar las **restricciones** existentes externas al sistema.

5.1.1 REQUISITOS FUNCIONALES

Los requisitos funcionales de la aplicación son los que a continuación se explican:

- **Visualización del mapa y de la ubicación del conductor en el mismo.** Es primordial que la aplicación tenga una interfaz de usuario intuitiva en la que el usuario pueda localizarse fácilmente e interactuar con su entorno. Este primer requisito funcional supone la base del resto de requisitos funcionales.

- **Acceso a la información en tiempo real de la ubicación y disponibilidad de los parkings.** El cliente debe de poder visualizar la ubicación de los parkings próximos y debe de ser capaz de consultar su disponibilidad en tiempo real siempre que lo necesite. Se han integrado datos de distintas fuentes para garantizar la veracidad de los mismos.
- **Uso de datos colaborativos entre usuarios para gestionar los aparcamientos.** Los usuarios pueden notificar instantáneamente al resto de usuarios las acciones de aparcar y de abandonar un parking. Esto no solo ayuda a que la información de la ocupación de los aparcamientos esté actualizada en tiempo real, sino que, además, facilita la escalabilidad de la aplicación al no depender de sensores o cámaras de videovigilancia de alto coste.
- **Procesamiento automático de imágenes de video para gestionar los aparcamientos.** De manera opcional para cada aparcamiento, el sistema debe de ser capaz de proporcionar una infraestructura de uso de cámaras de videovigilancia para la detección automática de huecos libres. Para ello, se han empleado métodos y algoritmos de visión artificial [4] que definen la disponibilidad de los estacionamientos en tiempo real.
- **Integración de la información proporcionada por sensores físicos.** Los sensores están instalados, de forma opcional, en las plazas de parking de manera individual. A pesar de que su instalación es opcional para cada parking, el sistema debe de ser capaz de integrarlos correctamente y proporcionar la correcta cobertura necesaria. Además, estos son más fiables que los procesos de visión artificial y que el uso de datos colaborativos entre usuarios (ya que los algoritmos de visión artificial tienen mayor probabilidad de fallar y los usuarios pueden olvidarse de señalar las acciones realizadas).
- **Ofrecer una experiencia e interfaz de usuario fácil e intuitiva.** El diseño de la aplicación debe ayudar a que el cliente sea capaz de utilizarlo de forma fácil e intuitiva. El objetivo es que cualquier conductor, sin necesidad de tener grandes conocimientos técnicos, pueda utilizar la aplicación sin dificultad, incluso bajo situaciones de estrés.

5.1.2 REQUISITOS NO FUNCIONALES

A su vez, los requisitos no funcionales derivados del sistema son los siguientes:

- **Rendimiento.** El sistema no requiere de un uso excesivo y constante de recursos, pero es cierto que la carga del mapa y la implementación en tiempo real de los métodos http se deben de realizar con cierta agilidad y baja latencia. De esta manera, los datos de la disponibilidad de los aparcamientos son lo más fiables posibles en tiempo real. Para ello, el sistema debe de contar con los correctos recursos hardware y de rendimiento.
- **Seguridad.** En los procesos de autenticación, es vital que la información referente al usuario se guarde de forma segura e inexpugnable. Para ello, se ha utilizado encriptación usando tokens JWT.
- **Disponibilidad y fiabilidad.** Las distintas partes del sistema, tanto el backend como el frontend, deben de ser accesibles de forma continua. Al crear un software que trabaja con datos en tiempo real, la existencia de fallos en la disponibilidad supone que los datos mostrados no son fiables. Es por ello por lo que se ha debido de implementar una arquitectura tolerante a fallos.
- **Compatibilidad multiplataforma.** La solución desarrollada está pensada para que sea compatible con los dos sistemas operativos móviles más populares; Android e iOS. Por este motivo, el desarrollo de la app se ha realizado utilizando el framework de desarrollo Flutter [1].

5.1.3 SUPUESTOS Y RESTRICCIONES

Para el correcto funcionamiento de la aplicación y para que se pueda dar solución al problema propuesto se dan por hecho los siguientes supuestos:

- El conductor tiene un dispositivo móvil con conexión a Internet.
- El dispositivo móvil del usuario dispone de geolocalizador GPS y este acepta los permisos de uso por parte de la aplicación.
- La infraestructura del servidor (backend) se encuentra desplegada de forma continua.

- Las cámaras y los sensores que están desplegados en algunos parkings [13] se encuentran funcionando correctamente.

Algunas restricciones que hay que tener en cuenta a la hora de evaluar posibles problemas en el futuro son:

- Que las cámaras utilizadas para hacer el recuento automático de huecos disponibles son de baja calidad o no cuentan con opciones de visión nocturna. Esto hace que los algoritmos de visión artificial no puedan determinar de forma fiable la disponibilidad de los parkings.
- El servicio depende de la correcta conexión entre el servidor y el dispositivo final del usuario. La operatividad del sistema puede verse comprometida por una conexión débil o por la falta de cobertura, especialmente, en ciertos aparcamientos subterráneos.

5.2 DISEÑO DEL SISTEMA

Antes de comenzar con la larga tarea del desarrollo software, es altamente conveniente realizar una serie de diseños del sistema, para luego más adelante, agilizar el proceso de desarrollo.

5.2.1 ARQUITECTURA GENERAL DEL SISTEMA

El sistema está compuesto por diferentes bloques que se comunican entre sí siguiendo distintos protocolos.

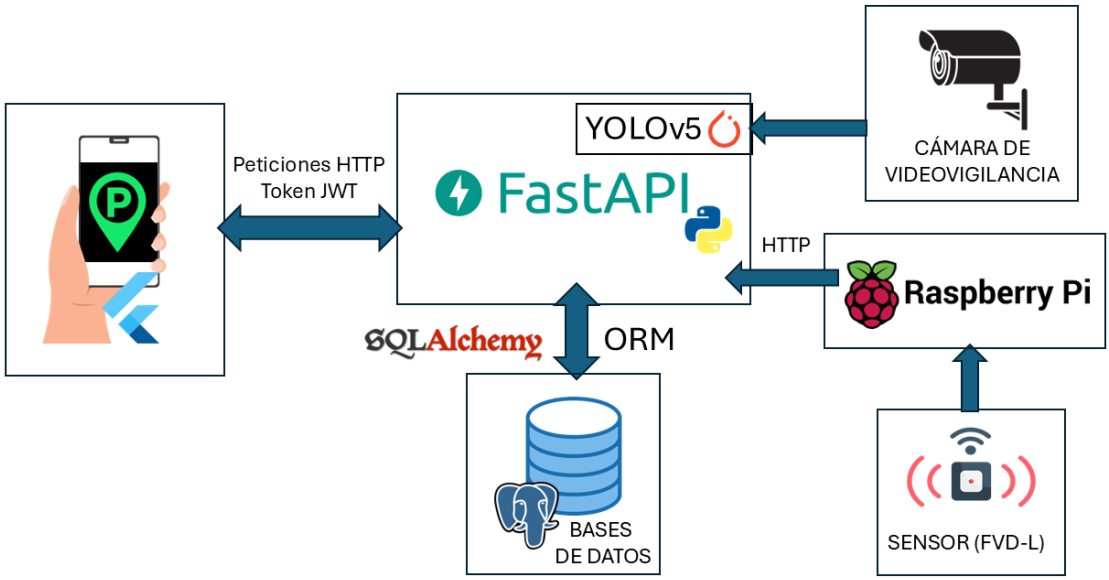

Figura 1: Arquitectura general del sistema

5.2.1.1 App de Flutter (frontend)

El primer bloque analizado es la propia app desarrollada en Flutter (frontend). Su función principal es la de ofrecer una interfaz de usuario intuitiva para que el usuario pueda consultar la información de los parkings, su historial, cambiar los datos de forma colaborativa y ver su ubicación en tiempo real. Este bloque interactúa con el backend desarrollado en FastAPI a través de peticiones HTTP y también gestiona sesiones autenticadas con JWT. El backend devuelve sus respuestas en formato JSON.

5.2.1.2 FastAPI (backend en Python)

Este bloque es el encargado de llevar a cabo toda la lógica del sistema y sirve como nexo de unión entre el frontend (app), la base de datos y las tecnologías de los sensores y las imágenes procesadas por YOLOv5 y capturadas por cámaras de videovigilancia.

Sus módulos clave son los siguientes (todos ellos se encuentran adjuntados en el Anexo II al final de este documento):

- Auth.py: realiza la gestión de usuarios en su proceso de autenticación (JWT) y de incio de sesión.
- Parking_actions.py: lleva a cabo la gestión de los datos colaborativos entre usuarios. Se encarga de actualizar el backend cuando el conductor señaliza un aparcamiento o un abandono de parking.
- Profile_user.py: este módulo gestiona la información personal del perfil del usuario. Es llamado para mostrarse en la parte del menú de usuario en el frontend.
- Sensor_routes.py: este archivo fija los métodos que actualizan la disponibilidad de los parkings en función de los datos procesado del sensor por el dispositivo Raspberry Pi [13].

Se dice que este bloque es el punto de unión entre el frontend y la información del backend porque es consultado directamente por la aplicación y abarca todos los archivos necesarios para procesar las imágenes, integrar la información de los sensores y guardar la información persistente de forma segura y eficiente (con un ORM).

5.2.1.3 Base de datos PostgreSQL

Este módulo tiene la función de almacenar toda la información persistente. Esa información abarca las credenciales de autenticación del usuario, toda la información de los parkings y las tecnologías (sensores y cámaras) asociadas a cada parking. Además, en este bloque se incluyen las herramientas de SQLAlchemy para ORM y PgAdmin como herramienta auxiliar para gestionar la información.

Este módulo se comunica con el backend FastAPI mediante el maepo de objetos-relacionales y haciendo uso de sesiones (SessionLocal) que encapsulan conexiones SQL.

5.2.1.4 Sensores físicos (gpiozero)

Esta parte del sistema es la más fiable detectando la disponibilidad de los parkings. Usa scripts en Python con la herramienta gpiozero para simular la implementación real de dispositivos Raspberry Pi y, mediante peticiones POST y pasando por el servidor FastAPI, se actualiza la información de los parkings almacenada en PostgreSQL. Es decir, se

comunica directamente con el backend y se actualiza el estado del sensor para recalcular el parámetro de las plazas libres mostrado al usuario.

5.2.1.5 Imágenes de cámaras de videovigilancia (detección con YOLOv5)

Este subsistema analiza, en tiempo real, las imágenes de las cámaras instaladas en los garajes y detecta los vehículos mediante la herramienta YOLOv5 de visión artificial entrenada previamente con millones de imágenes. Esto ocurre dentro del servidor FastAPI. Y las comunicaciones son directas con el archivo main.py del bloque FastAPI (dentro del bucle_car_loop).

5.2.2 DIAGRAMA DE CASOS DE USO

Un diagrama de casos de uso es un esquema de tipo UML (Unified Modeling Language). En él se representan los actores del sistema software a desarrollar. Sirve para representar las distintas funcionalidades de estos actores, (principalmente usuarios y administradores software) y como estas se relacionan entre sí.

A continuación, se representa gráficamente el diagrama de casos de uso del programa software desarrollado:

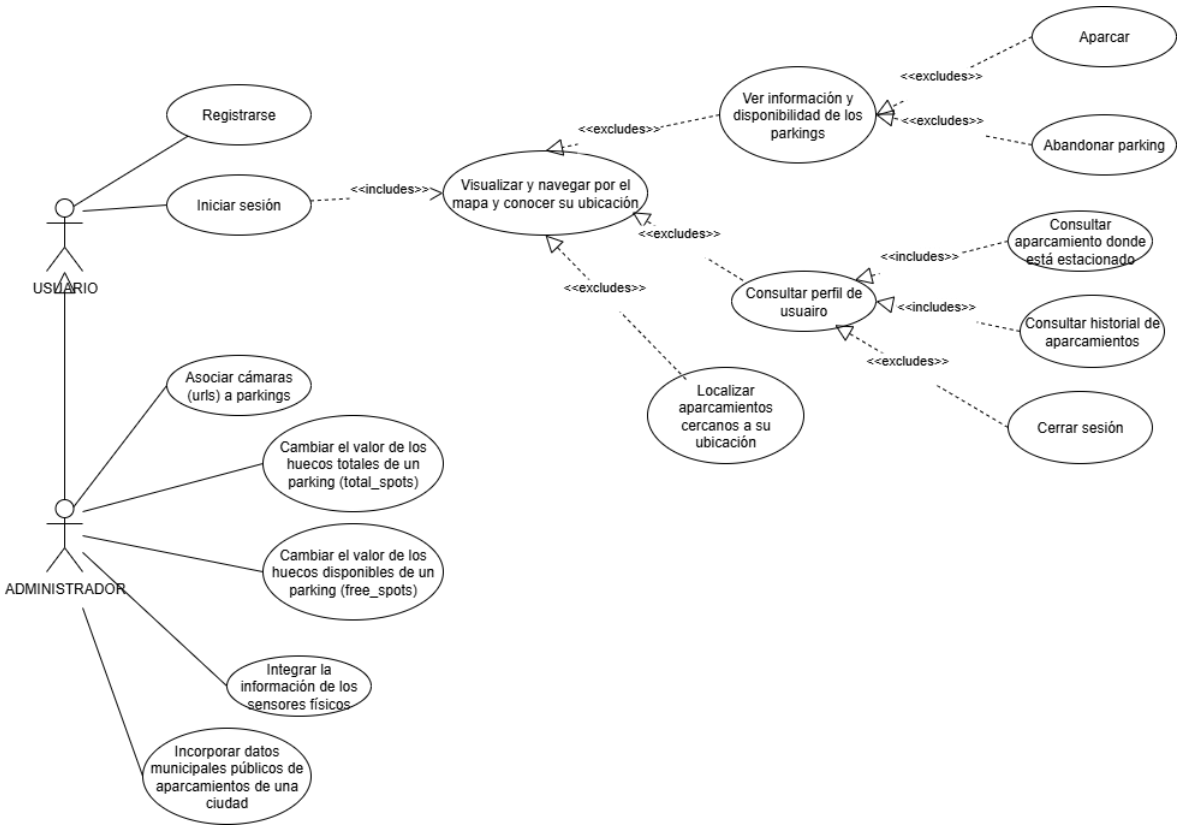

Figura 2: Diagrama de casos de uso

El usuario, tras registrarse por primera vez e iniciar sesión, será redirigido a la ventana del mapa, el cual visualizará y podrá navegar por él. Tras aceptar los permisos de acceso a la ubicación correspondientes, podrá ver su ubicación en el mapa. Visualizando el mapa verá también todos los marcadores de los parkings de la ciudad de Madrid. Pinchando en cada parking, podrá acceder a la información y disponibilidad del mismo. Si no ha aparcado en ningún estacionamiento con anterioridad, podrá aparcar. Sin embargo, si el usuario ya se encuentra aparcado en cualquier otro parking, podrá marcar la opción de abandonar parking. A parte de esto, el usuario podrá acceder a su perfil y consultar su historial de aparcamiento. En dicho menú lateral, también tendrá la opción de cerrar sesión. Pinchando en los botones que aparecen en el mapa podrá resaltar para localizar más fácilmente, por un periodo limitado de tiempo, los cinco parkings más próximos.

Por otro lado, el administrador puede realizar todas las tareas que puede realizar el usuario con la particularidad, de que este además puede realizar acciones únicas. Dichas acciones

abarcan la recopilación, el procesado y la publicación de endpoints de parking a partir de la información pública (accesible a través de portales web municipales) Además, el administrador podrá añadir las urls de los vídeos de la entrada y salida de los garajes o también de todos los aparcamientos en general. Además, tendrá la posibilidad de cambiar el parámetro de aparcamientos totales dentro del mismo garaje y de los huecos disponibles. Por último, será el responsable de añadir la información recolectada por los sensores físicos instalados en los estacionamientos.

5.2.3 DISEÑO DE BASES DE DATOS

A continuación, se muestran capturas extraídas usando la herramienta auxiliar de PgAdmin, de los distintos parámetros de las tablas de la base de datos.

	id [PK] integer	url character varying	parking_id integer	tipo character varying
1	3	http://170.249.152.2:8080/cgi-bin/viewer/video.jpg?r=1740941740	13470	plaza
2	5	http://67.61.139.162:8080/jpg/image.jpg?1742026508	39871	plaza

Figura 3: Parámetros de la tabla 'cameras'

	id [PK] integer	user_id integer	parking_id integer	date timestamp without time zone
1	1	1	81696	2025-06-15 19:19:01.305007
2	2	1	24131	2025-06-15 20:40:11.660355
3	3	2	24108	2025-06-15 20:41:08.142293
4	4	2	24108	2025-06-15 20:41:57.592511

Figura 4: Parámetros de la tabla 'parking_records'

Esta tabla es usada para acceder al historial de parkings del usuario.

	id [PK] integer	nombre character varying	tipo character varying	latitud double precision	longitud double precision	total_spots integer	free_spots integer
1	13462	Aparcamiento mixto. Fuencarral	/contenido/entidadesYorganismos/AparcamientosRe...	40.4296270729669	-3.70279096159205	521	259
2	5898885	Aparcamiento mixto. Avenida de Portugal	/contenido/entidadesYorganismos/AparcamientosRe...	40.4112391063366	-3.73808787568741	873	436
3	19060	Aparcamiento para residentes. Aluche	/contenido/entidadesYorganismos/AparcamientosRe...	40.39260709014263	-3.7586798849896357	208	104
4	13475	Aparcamiento mixto. Brasil	/contenido/entidadesYorganismos/AparcamientosRe...	40.4560925110791	-3.69349688854822	628	314
5	183747	Aparcamiento mixto. Casino de la Reina	/contenido/entidadesYorganismos/AparcamientosRe...	40.40709468568204	-3.7038875277131464	584	292
6	183748	Aparcamiento mixto. Daoiz y Velarde	/contenido/entidadesYorganismos/AparcamientosRe...	40.40255344121901	-3.67763144165245	488	244
7	13464	Aparcamiento mixto. Felipe II	/contenido/entidadesYorganismos/AparcamientosRe...	40.4242207643363	-3.67342185936768	1790	895
8	36270	Aparcamiento mixto. Hernani	/contenido/entidadesYorganismos/AparcamientosRe...	40.44817621258857	-3.6976113125092684	495	247

Figura 5: Parámetros de la tabla 'parkings'

Sin duda esta es la tabla más importante del sistema.

	id [PK] integer	parking_id integer	spot_number integer	status boolean	last_updated timestamp without time zone
1	2	81696	6	false	2025-06-23 21:16:15.989524
2	1	81696	5	true	2025-06-23 21:16:54.075957
3	3	13567	4	false	2025-06-23 21:46:25.187466

Figura 6: Parámetros de la tabla 'sensors'

Esta tabla registra los sensores

	id [PK] integer	username character varying	hashed_password character varying	current_parking_id integer	is_admin boolean
1	4	lola	$2b$12$ermaGevk.KzxvsFHz23Wd.QKts8Hd0AE908jrN3UOeTcNxR8rKYLC	[null]	false
2	5	mama	$2b$12$A.HgbPWr1KVUfsaaiUSVAu6B/a396i.3ja5iJVMX16xfQdwPERiq6	[null]	false
3	11	Pepe	$2b$12$nNtaQGhTgYz..gfgT4dEH.ic3sq.8vuSc3zOjrFkWuZQTj253oLvK	[null]	false
4	1	jesus	$2b$12$fXTaMxsKvCzh8rMtLP.yIuoGcdVetjT7Fn7SR.fObgFS7Ql5ucUw2	[null]	false
5	9	suso	$2b$12$Bfsp3b1BkqCZFl4FEyuwhe6oJs0GpIehfrnwWzgcR4Vi1uy8g73lO	[null]	false
6	3	javi	$2b$12$x/e/x6ZECyafzYPWGjfF6.vKgPSalviniGerTQZnH6DrcE1yAaH7y	[null]	false
7	6	papa	$2b$12$vb5qb0Jw0lHcKzp.VflNeOnGXirK3qLv/5SCHrqCcLNPFNj0cYaHO	13462	false
8	7	Marco	$2b$12$k8mtuxJTPIsj9K3g6xNajurkZCSxAd49FeGs2hU79e5RKyMGf66...	[null]	false
9	8	admin	$2b$12$5ng3qPzqUmaNuHk6JJ4VEODxs0Vt.AfpUzUdv64kpw8S8O/B9w...	[null]	true

Esta tabla es la que se utiliza en el proceso de autenticación.

5.3 IMPLEMENTACIÓN DEL SISTEMA

5.3.1 ACCESO Y TRATAMIENTO DE LAS APIS PÚBLICAS

Este proyecto nace para dar solución a la difícil búsqueda de aparcamiento en la ciudad de Madrid, no obstante, este proyecto tiene la ambición de escalar a otras muchas ciudades masificadas del mundo. Toda la información necesaria acerca de los parkings de Madrid se

ha extraído directamente de la página web del Ayuntamiento de Madrid [12] Concretamente, se obtuvieron los archivos CSV de los parkings municipales, los parkings PAR y los parkings para residentes. No obstante, dicha información pública ha tenido que ser procesada ya que, en una primera instancia, no contaba con la estructura deseada. Entre las columnas que si han sido de interés se encuentran la clave de identificación del parking PK, su NOMBRE, las coordenadas; LATITUD y LONGITUD y una DESCRIPCION. De esta última columna se han extraído casi la totalidad de los huecos disponibles de los parkings. No obstante, para su extracción, ha sido necesario procesar la información de esta columna, ya que esta no tenía una estructura uniforme. Para esta labor se ha utilizado la función extraer_total_spots del archivo de Python cargar_parkings.py, archivo se adjunta en el anexo II:

Tras procesar todos los datos y quedarnos con la información de interés del endpoint de parking, estos datos fueron incorporados en la base de datos "parkings" de PostgreSQL. Desde este momento, el frontend ya ha podido consultar esta tabla de datos y actualizar su información en tiempo real.

5.3.2 MÉTODOS HTTP

Se han usado dependencias de seguridad que verifican el usuario actual que puede usar las rutas protegidas. Además, el administrador tiene acceso a ciertos métodos exclusivos para gestionar las tecnologías inteligentes y cambiar datos relativos a los parkings.

A continuación, se muestran los métodos HTTP implementados. Se han obtenidos capturas directamente desde la ruta /docs.

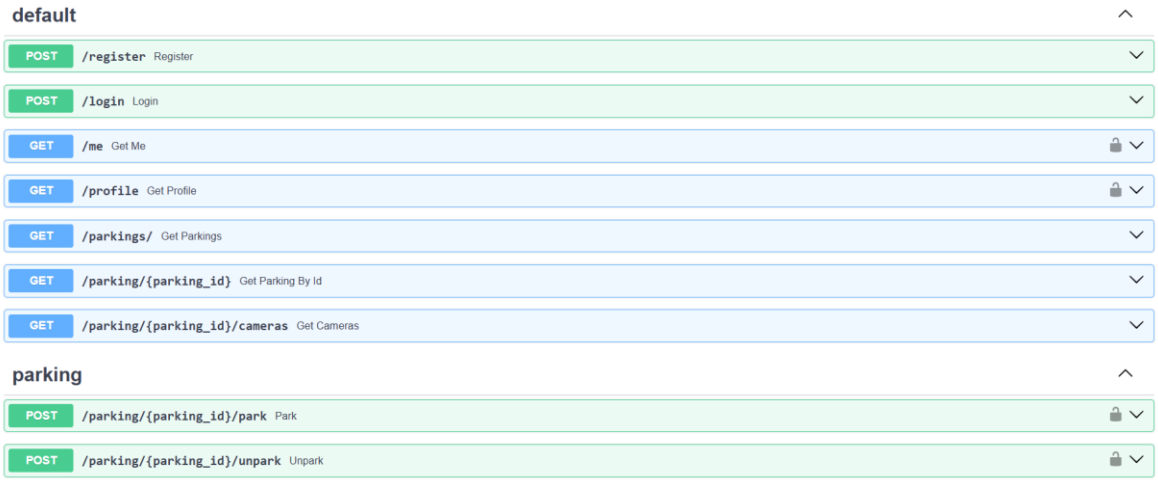

Figura 7: Instantánea de los métodos HTTP implementados I

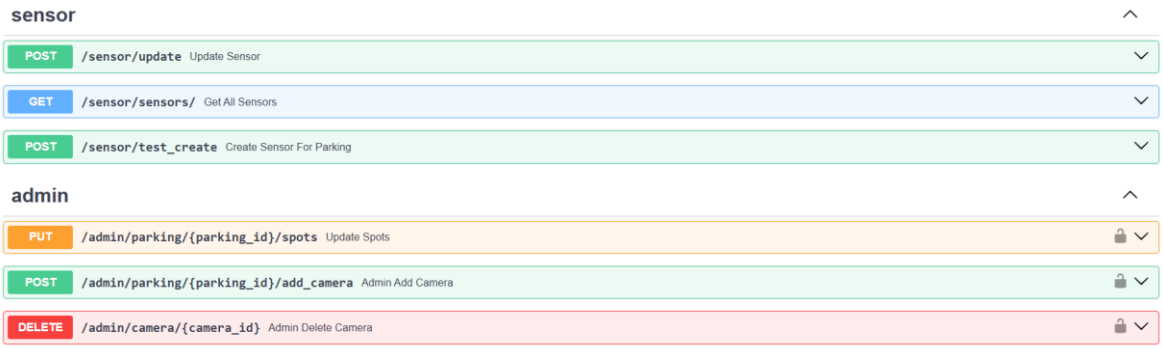

Figura 8: Instantánea de los métodos HTTP implementados II

5.3.3 AUTENTICACIÓN Y SEGURIDAD

Esta se gestiona a través de un sistema robusto que sigue los estándares modernos. Se implementa el protocolo OAuth2 con flujo de tipo "passwordbearer" y utilizando JWT como principal mecanismo de autenticación.

El endpoint /login valida las credenciales introducidas por el usuario; su nombre de perfil y su contraseña hasheada (con bcrypt). Además, el token JWT es generado y se usa el plugin flutter_secure_storage para almacenarlo. Todas las peticiones posteriores que requieren autenticación deben incluir el encabezado Authorization: Bearer.

Por último, cabe destacar que se establecieron distintos niveles de acceso. Algunas rutas están protegidas para que seas accesibles por administradores (únicamente). Así, se garantiza toda la seguridad en la app con su arquitectura de cliente-servidor.

5.3.4 SIMULACIÓN Y SOPORTE DE SENSORES

Cuando se extraen datos de los sensores, res elementos principales entran en juego:

- Sensor físico FVD-L
- Microservicio en Raspberry Pi
- Backend FastAPI

Hasta la fecha no se ha implementado ningún dispositivo RaspBerry Pi real. Lo que se ha hecho ha sido simularlo con un script de Python que usa (gpiozero).

Existen muchos tipos de sensores como los sensores magnéticos de presión por plaza, los sensores ultrasónicos/lidar o los sistemas conectados por LoRa, WiFi o Bluetooth.

Ya que este proyecto busca la fiabilidad, escalabilidad y precisión la recomendación es que se siga un estándar de sensores de tipo FVD-L. Se trata de un sensor magnetorresistivo pasivo que altera el campo magnético natural de la Tierra en presencia de grandes objetos metálicos (como un automóvil).

Este es un tipo de sensor que requiere menos componentes electrónicos y es muy escalable y fácilmente integrable con el backend del proyecto con el uso de GPIO (un microservicio que notifica a FastAPI).

El objetivo es detectar en tiempo real si un sensor FVD-L está activado por la presencia de un vehículo y notificar dicha información al backend FastAPI para actualizar el estado de las plazas del parking.

Capítulo 6. ANÁLISIS DE RESULTADOS

Tras exponer detalladamente el estudio inicial del sistema, el diseño y la forma en la que se ha implementado para la consecución de este proyecto, a continuación, se destaca, de forma crítica, los resultados más relevantes del trabajo.

Se ha publicado un vídeo en el que se muestran los principales casos de uso de la aplicación. Para acceder a dicho video haz clic aquí.

6.1 COMPARACIÓN DE LOS REQUISITOS FUNCIONALES CON LOS RESULTADOS REALES

A continuación, se inserta una tabla que analiza el cumplimiento satisfactorio de los requisitos funcionales mediante la relación de estos con instantáneas de la app funcional:

6.1.1 VISUALIZACIÓN DEL MAPA Y DE LA UBICACIÓN DEL CONDUCTOR EN EL MISMO

A través del uso e implementación de ciertas librerías y otros elementos, entre los que destacan el mapa de flutter (flutter_map) y del uso de la librería open-source de OpenStreetMap, se ha podido construir satisfactoriamente el mapa de la ciudad. Además, con la dependencia de flutter de geolocator se ha podido representar la ubicación exacta del usuario con un marcador. El resultado es el siguiente:

Figura 9: ParkiNeo: Permisos de ubicación

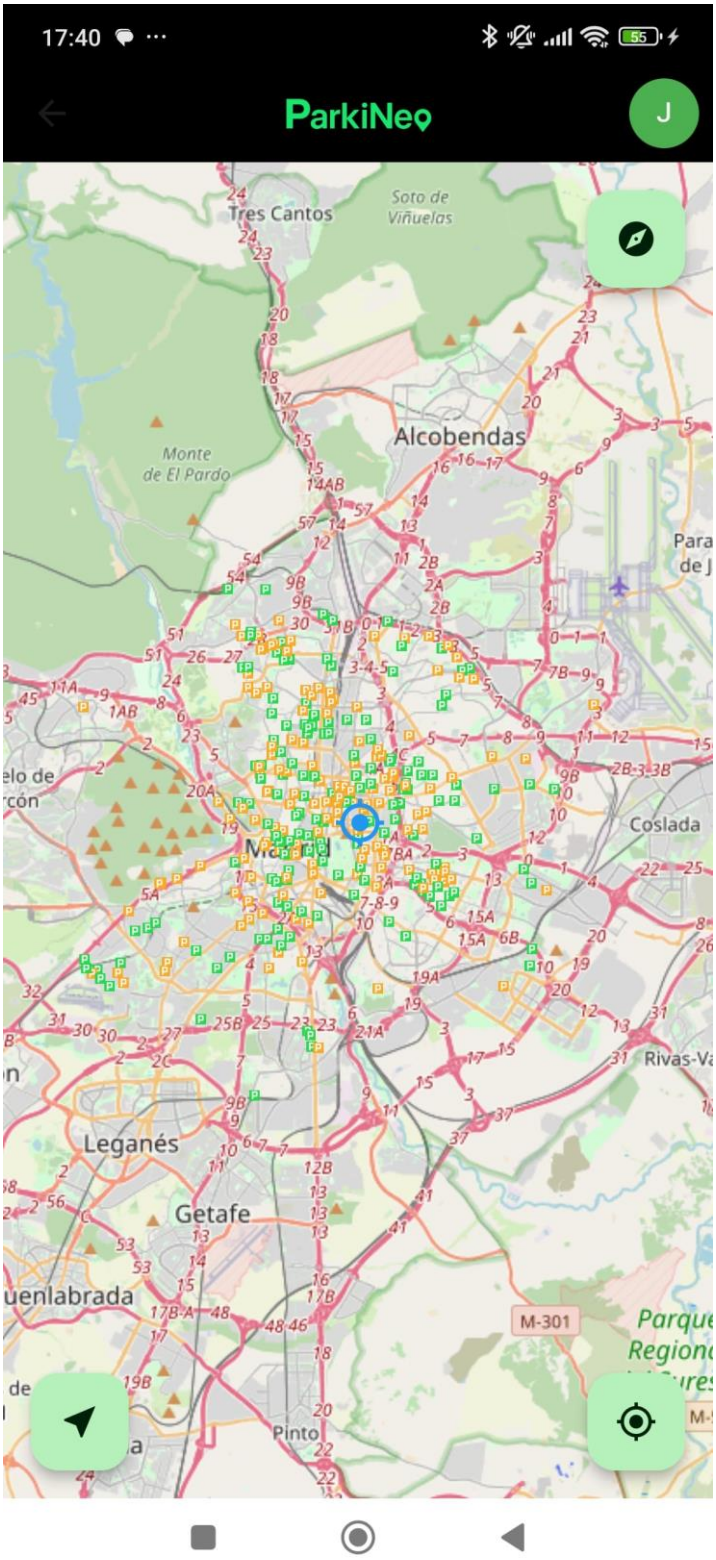

Figura 10: ParkiNeo: Mapa y ubicación

6.1.2 ACCESO A LA INFORMACIÓN EN TIEMPO REAL DE LA UBICACIÓN Y DISPONIBILIDAD DE LOS PARKINGS

Los usuarios además, visualizan todos los marcadores de los parkings y, al pinchar en uno, ven una tarjeta de parking con la información de la disponibilidad y la opción de aparcar/abandonar parking. Si pulsan dicho botón, se actualiza automáticamente el

backend y se muestra al resto de usuarios la nueva disponibilidad

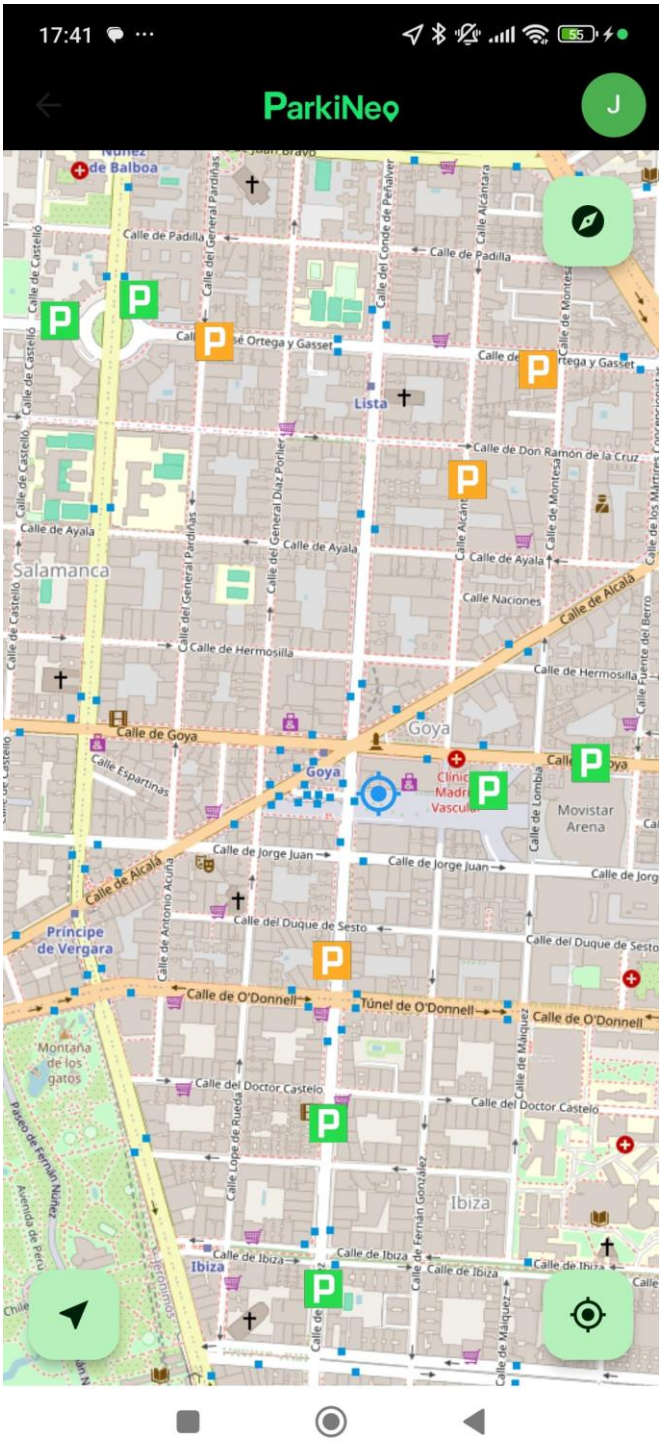

Figura 11: ParkiNeo: Mapa con marcadores de los parkings

Figura 12: ParkiNeo: Tarjeta de aparcamiento

6.1.3 USO DE DATOS COLABORATIVOS ENTRE USUARIOS PARA GESTIONAR LOS APARCAMIENTOS

Los usuarios al señalizar un aparcamiento ven las distintas opciones dónde cambia el color del botón de aparcamiento/abandonar parking. Cabe recordar que cuando un usuario ya se encuentra aparcado en un parking no puede aparcar en ninguno otro:

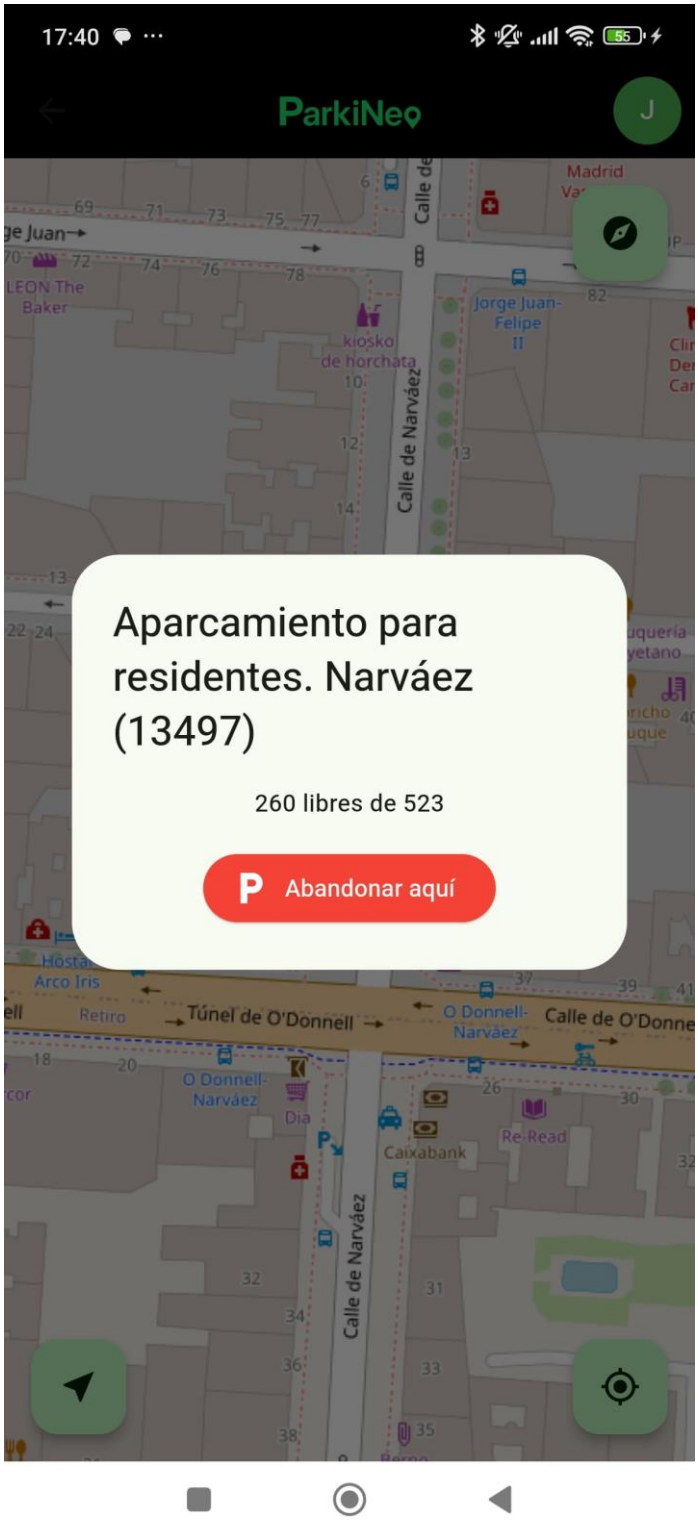

Figura 13: Tarjeta de aparcamiento de usuario aparcado

Figura 14: Tarjeta de parking de un usuario aparcado en otro parking

6.1.4 PROCESAMIENTO AUTOMÁTICO DE IMÁGENES

Hay dos dinámicas a seguir con la incorporación de las cámaras:

- Cámaras de tipo "plaza": Añadir cámaras de todos los huecos. Los huecos totales serán la suma de todos los coches detectados en todas las cámaras. (Metodología similar a la de los sensores).
- Cámaras de tipo "entrada/salida": Añadir las cámaras de videovigilancia a la entrada y a la salida del garaje de los aparcamientos. De esta manera, el número total de plazas ocupadas, (en vez de ser la suma total de los coches detectados), será la suma y la resta de los coches que entran y salen del garaje. Para este segundo método, al detectar un coche en el frame se estudia si en el frame anterior también se había detectado dicho coche o no. De esta forma se evita contabilizar en más de una vez el mismo automóvil.

A continuación, se muestra una captura de imagen de una asociación satisfactoria de una cámara de tipo plaza a un parking:

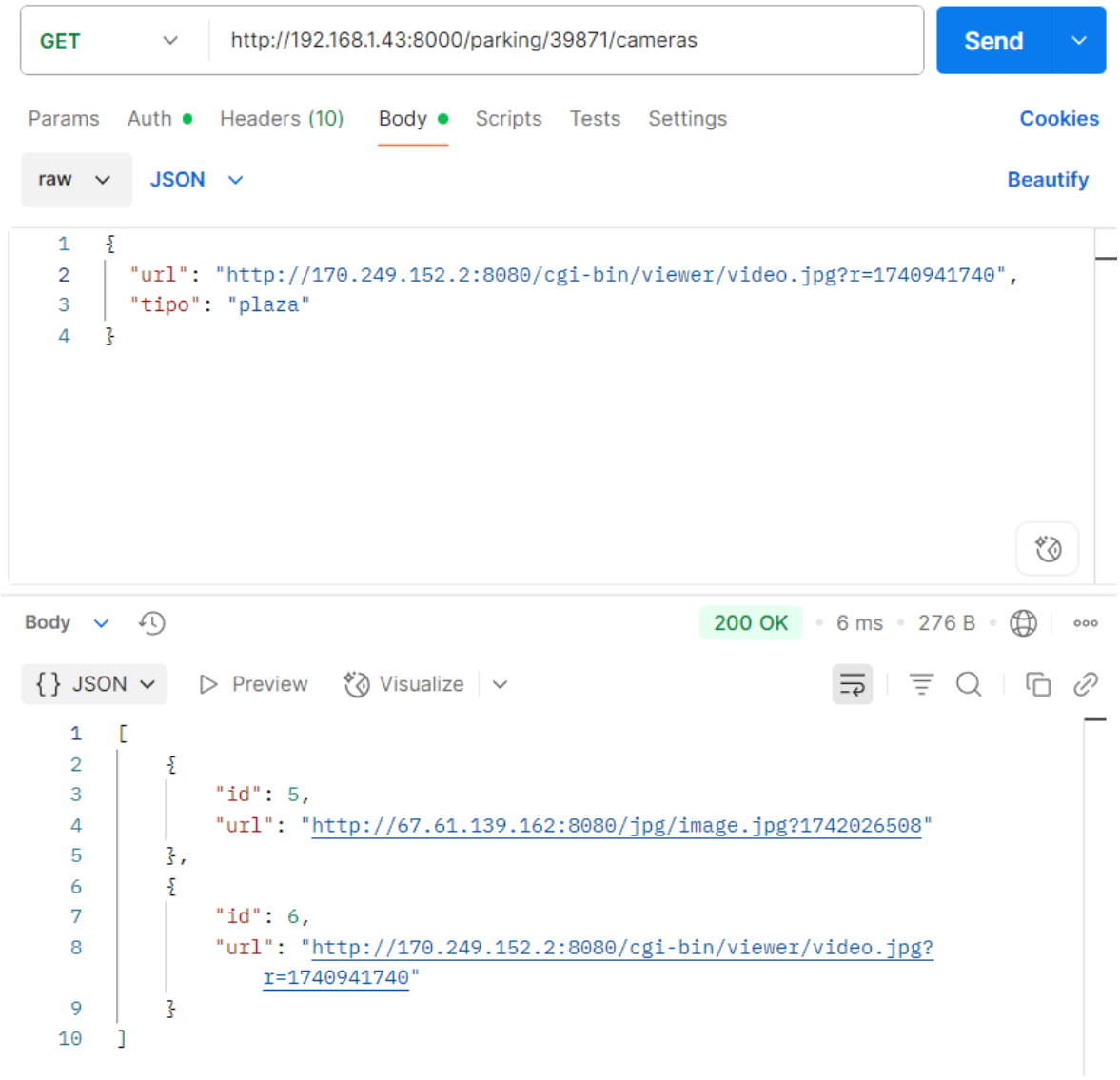

Figura 15: Asociación de cámara tipo 'plaza' a parking usando Postman

6.1.5 INTEGRACIÓN DE LA INFORMACIÓN DE LOS SENSORES FISICOS

Para simular correctamente la instalación y uso de un sensor, se utilizan, principalmente dos métodos HTTP con la herramienta de Postman.

Con /sensor/test_create, (en parámetros), se introduce el parking_id asociado y el número de spot_number, y ya se crea y devuelve un id. El supuesto sensor acaba de ser creado/registrado.

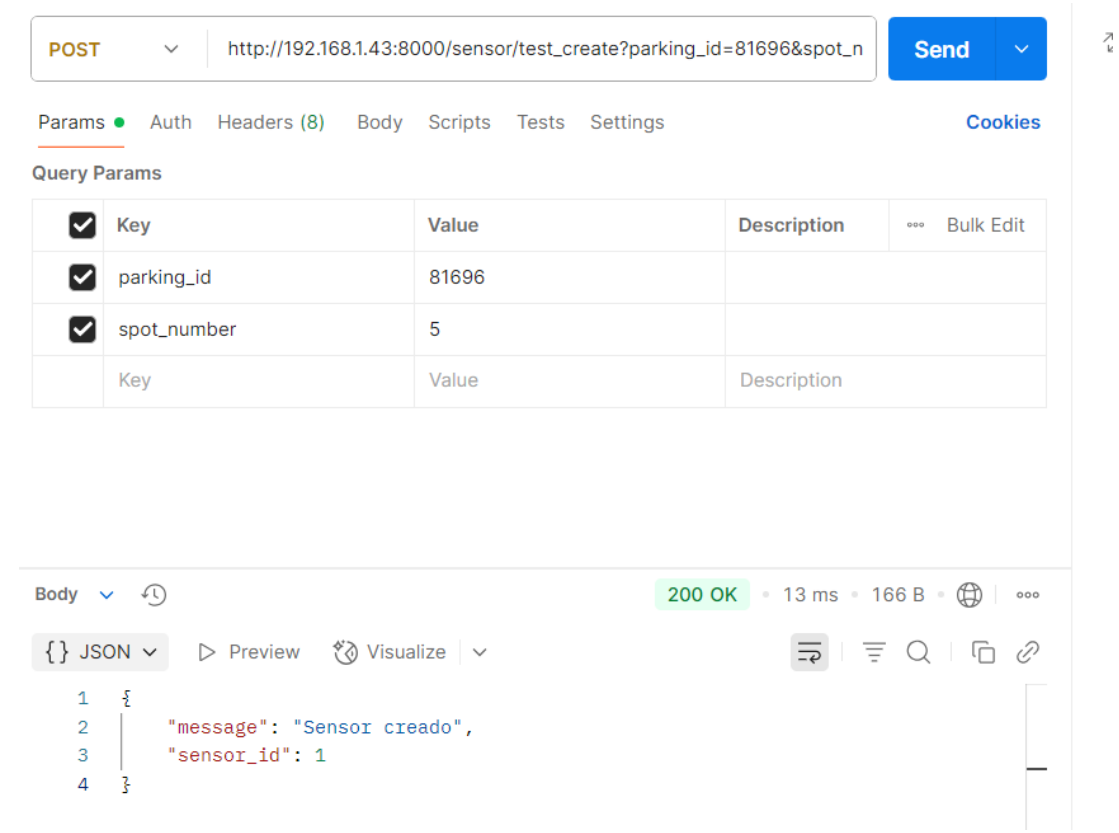

Figura 16: Registro de un sensor usando Postman

El siguiente paso es simular que el sensor detecta un coche:

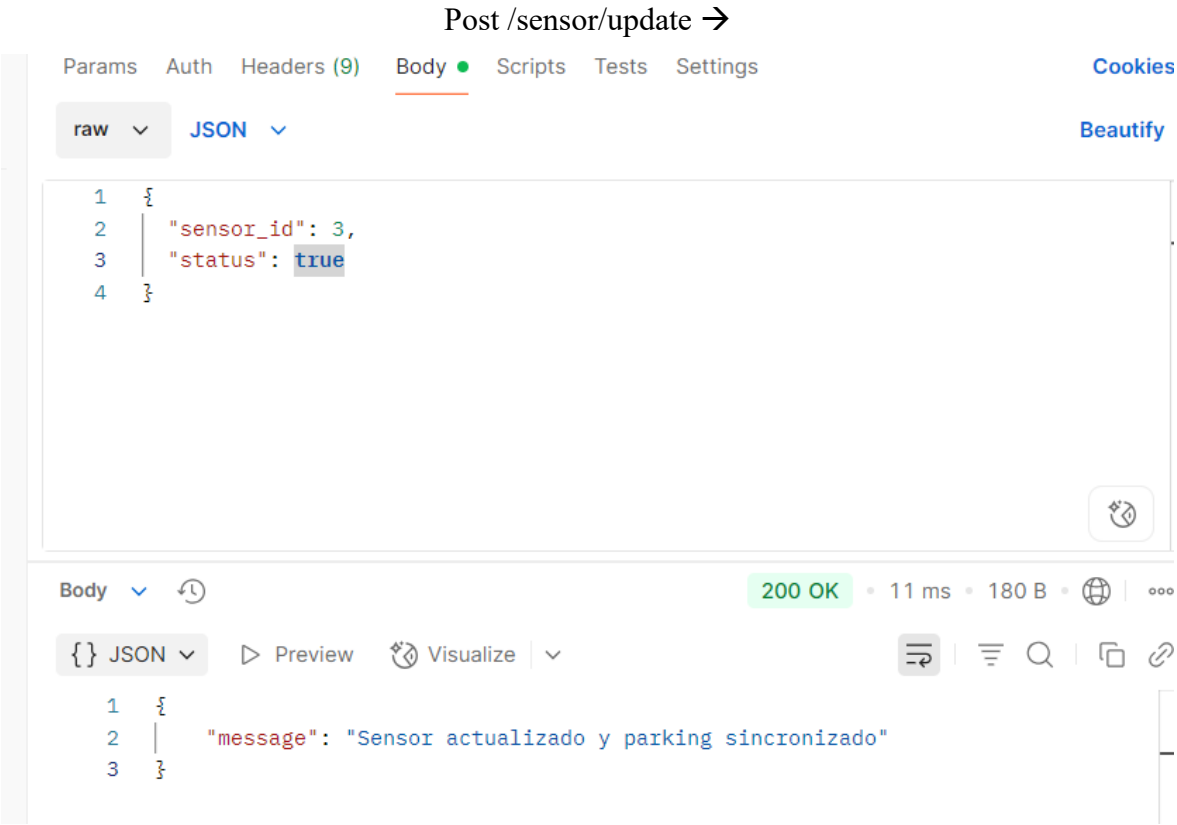

Figura 17: Simulación de detección de vehículo por sensor usando Postman

La petición POST funciona correctamente.

6.2 USER INTERFACE & USER EXPERIENCE (UI & UX)

Tanto la interfaz de usuario (UI por sus siglas en inglés) como la experiencia de usuario (UX como es conocido por sus siglas en inglés) son elementos centrales de cualquier desarrollo software y que mejoran la interacción del usuario con el mismo. En este proyecto se ha tratado de cuidar ambos aspectos.

6.2.1 USER INTERFACE (UI)

La interfaz de usuario (UI) abarca todos los elementos visuales y gráficos de la aplicación software que hacen más atractiva la estética y el uso de la aplicación. La interfaz de usuario

se encuentra en cada uno de los elementos usados, desde la apariencia de los más sencillos botones hasta el logo de la aplicación.

El logotipo de esta app es el siguiente:

Figura 18: Logotipo de ParkiNeo

Figura 19: Logotipo nombre de ParkiNeo

El nombre de la aplicación es ParkiNeo. Este nombre está compuesto por el anglicismo parking y por el nombre del protagonista de la saga de ficción de Matrix; Neo. Saga que ha inspirado a muchas personas de todo el mundo y a muchos ingenieros telemáticos.

Además, los colores corporativos de la aplicación son los mismos que los de esta serie de ficción:

- Verde menta brillante: Refleja notas de verde-azulado. Evoca aspectos como frescura, dinamismo y un cierto grado de compromiso con la sostenibilidad y el medioambiente.
- Negro: Crea un contraste que resalta las sensaciones evocadas por el color verde. Además, aporta elegancia.

6.2.2 MEJORAS DE LA UX (USER EXPERIENCE)

A su vez, la experiencia de usuario abarca aspectos como la facilidad de navegación y la rapidez de respuesta.

A pesar de que en un primer lugar no se especificó como un requisito funcional, se ha tratado de mejorar la UX para que la app sea lo más funcional posible. Se ha querido que el usuario pueda realizar sus funciones (aquellas especificadas en el diagrama de casos de uso anterior), en el menor número de acciones posible.

Para ello, se ha añadido un menú de usuario que incrementa, considerablemente, la calidad de la experiencia del usuario. Dicho menú consta de las siguientes partes: un componente visual (ListTile) que informa al usuario dónde ha aparcado su vehículo, un historial que muestra los parkings de los últimos cinco estacionamientos y un botón que al ser presionado cierra la sesión del usuario. Cabe destacar que, si el usuario está aparcado en algún estacionamiento, el usuario podrá pinchar el componente visual de tipo ListTile y, al realizar esta acción, se abrirá la tarjeta informativa de dicho parking. Esto ahorra mucho tiempo al conductor que desea señalizar un abandono de su parking y, o bien no se acuerda de dónde

estacionó o bien no encuentra dicho parking en el mapa. Esto supone una mejora destacable de la experiencia del usuario de ParkiNeo.

Figura 20: Instantánea del perfil del usuario

Otra mejora reseñable de la UX de la app es el fácil acceso a la información de los parkings pinchando en los marcadores de estos. A parte de estar situados en su localización exacta dentro del mapa, estos escalan su tamaño automáticamente en función del zoom del usuario mediante esta función (llegando incluso a desaparecer si el usuario decide disminuir mucho el nivel de zoom del mapa):

```
  double getMarkerScale() {
    if (_currentZoom < 6) return 0;
    if (_currentZoom < 12) return 0.1;
    if (_currentZoom < 15) return 0.2;
    return 0.4;
}
```

Otras mejoras de la experiencia de uso abarcan el acceso y la actualización automática de la ubicación del usuario, la persistencia de la sesión con el uso de tokens JWT y la retroalimentación inmediata ante errores (mostrados en forma de barra inferior emergente).

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

Comentar las conclusiones del proyecto, destacando lo que se ha hecho, dejando claros qué objetivos se han cubierto y cuáles son las aportaciones hechas.

7.1 CONCLUSIONES

Tras diseñar, desarrollar y analizar los resultados de la solución tecnológica propuesta se han extraído las siguientes conclusiones.

- Se ha dado una solución tecnológica moderna e innovadora que implementa las tecnológicas más modernas en cuanto a visión artificial. Dichos modelos permiten detectar con alto grado de fiabilidad de la presencia de vehículos.
- Se ha conseguido que la información referente a la disponibilidad de los parkings esté actualizada a tiempo real. Cualquier usuario verá en todo momento la información de los parkings actualizada.
- Se ha conseguido realizar una solución intuitiva y fácil de usar para todos los públicos, especialmente los más mayores. Para ello, se ha cuidado la interfaz de usuario y la experiencia de usuario poniendo especial atención en ambas.
- Se han cumplido el diagrama de casos de uso inicial. El usuario otorga una experiencia completa al usuario final.
- Se ha demostrado que el sistema es escalable y que podría ser implementado en otras ciudades del mundo. Esto es debido a su diseño modular y a cómo se ha diseñado la arquitectura general del sistema.
- Se ha conseguido dar una solución tecnológica moderna, robusta y eficiente.

7.2 TRABAJOS FUTUROS

- La incorporación y asociación de sensores y cámaras se realiza con herramientas auxiliares de gestión de APIs como es Postman. Quizás en un futuro debería añadirse

una ventana de administrador en la parte del frontend para realizar los cambios al backend desde ahí.

- En este proyecto, el concepto de sensor de presencia ha sido simulado (gpiozero). En futuros trabajos podrían adquirirse dichos sensores y probarlos en un entorno real controlado.
- Actualmente solo se pueden usar sensores magnetorresistivos FVD – L. En el futuro podría hacerse que se usasen otros tipos.
- En el proceso de autenticación podría darse la posibilidad de que el usuario realice un cambio de contraseña por si ha este se le ha olvidado la que tenía. Esto mejoraría la experiencia de usuario.
- En la función de datos colaborativos entre usuarios, añadir mayor grado de fiabilidad al asegurarse que el usuario que dice "aparcar aquí", relamente se encuentra físicamente en un radio cercano a dicho parking.
- Para las primeras pruebas del desarrollo del proyecto, se ha hecho uso de un aparcamiento como zona objeto de pruebas. Para la implementación real del proyecto será necesario estudiar los recursos ya existentes, como las cámaras de videovigilancia, y analizar si es posible el uso de estas para minimizar costes. Ello implica estudiar la gestión de permisos y ver si resulta más rentable su uso que la instalación de nuevos dispositivos de captura de imágenes.

Capítulo 8. REFERENCIAS

[1] Flutter, "Flutter – Build apps for any screen," Google, [Online]. Available: https://flutter.dev.

[2] OpenStreetMap contributors, "OpenStreetMap," [Online]. Available: https://www.openstreetmap.org.

[3] S. Ramírez, "FastAPI," [Online]. Available: https://fastapi.tiangolo.com.

[4] Ultralytics, "YOLOv5 by Ultralytics," GitHub, [Online]. Available: https://github.com/ultralytics/yolov5.

[5] OpenCV.org, "Open Source Computer Vision Library," [Online]. Available: https://opencv.org.

[6] PostgreSQL Global Development Group, "PostgreSQL," [Online]. Available: https://www.postgresql.org.

[7] Microsoft, "Visual Studio Code," [Online]. Available: https://code.visualstudio.com.

[8] Parclick, Parkopedia, ElParking: Servicios de reserva de aparcamiento. [Online]. Available: https://www.parclick.com, https://www.parkopedia.es, https://elparking.com.

[9] SFpark & SMOU, "Smart parking initiatives in San Francisco and Barcelona." [Online]. Available: https://sfpark.org, https://smou.cat.

[10] NYC Open Data, "Mobility data and parking APIs," [Online]. Available: https://opendata.cityofnewyork.us.

[11] K. Beck et al., *Manifesto for Agile Software Development*. [Online]. Available: https://agilemanifesto.org

[12] Ayuntamiento de Madrid, "Datos abiertos: Aparcamientos," Portal de datos abiertos. [Online]. Available: https://datos.madrid.es

[13] Raspberry Pi Foundation, "Raspberry Pi 4 Model B," [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

Los Objetivos de Desarrollo Sostenible (ODS) fueron definidos por primera vez en 2015 y forman parte de la Agenda 2030 impulsada por la ONU con el fin de promulgar un desarrollo intergeneracional sostenible. El presente proyecto se alinea con la ONU cumpliendo los siguientes ODS:

- ODS 11: Ciudades y comunidades sostenibles. Al reducir el tiempo de espera de búsqueda de aparcamiento se consigue que la ciudad sea un espacio más accesible y sostenible (al reducir también el gasto de combustible). Además, se reduce el estrés psicológico de conductores y transeúntes mejorando así sus respectivas calidades de vida.
- ODS 13: Acción por el clima. La implementación real de este proyecto consigue reducir la huella de carbono producida por el ser humano. Al hacer más eficientes los desplazamientos en busca de aparcamiento, se consigue disminuir considerablemente el consumo de combustible y los gases de efecto invernadero emitidos a la atmósfera.
- ODS 9: Industria, innovación e infraestructura. Este ODS busca, entre otros aspectos, fomentar la innovación tecnológica a la hora de dar solución a las demandas de los ciudadanos. Este proyecto satisface este objetivo en la medida que promueve la innovación tecnológica en el ámbito del transporte urbano, mediante el desarrollo de una app sencilla, intuitiva y accesible para todos los ciudadanos.

ANEXO II

Aquí se adjuntan los principales códigos, tanto del frontend (.dart) como del backend (.py) usados en el proyecto.

El archivo pubscpec.yaml de Flutter define todas las dependencias que se incluyen en el proyecto además de otros recursos visuales y metadatos.

A continuación, se presenta dicho archivo de este proyecto:

```
name: map
description: "A new Flutter project."
publish_to: 'none' # Remove this line if you wish to publish to pub.dev

version: 1.0.0+1

environment:
  sdk: ^3.5.4

dependencies:
  flutter:
    sdk: flutter

  flutter_map: ^6.1.0
  latlong2: ^0.9.0
  geolocator: ^10.1.1 #10.1.1
  http: ^1.3.0
  flutter_secure_storage: ^8.0.0
  provider: ^6.1.5
  #url_launcher: ^6.1.5

  cupertino_icons: ^1.0.8

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^4.0.0
  flutter_launcher_icons: ^0.14.3

flutter:

  uses-material-design: true

  assets:
```

```
    - assets/parking.png
    - assets/parking_verde.png
    - assets/parking_amarillo.png
    - assets/parking_rojo.png
    - assets/F_icon.png
    - assets/letrasrect.png

flutter_launcher_icons:
  android: true
  ios: true
  image_path: "./assets/F_icon.png"
```

```
#cargar_parkings.py
import pandas as pd
import re
from sqlalchemy.orm import Session
from database import SessionLocal
from model_db import ParkingDB

#archivos oficiales del ayuntamiento
archivos_csv = [
    "municipales.csv",
    "disuasorios.csv",
    "PAR.csv"
]

#columnas de interés
columnas_relevantes = ["PK", "NOMBRE", "LATITUD", "LONGITUD", "TIPO",
"DESCRIPCION"]

#función refinada para extraer total_spots desde el campo de la descripción
def extraer_total_spots(descripcion: str) -> int:
    if not isinstance(descripcion, str):
        return 10  #si no es posible de extraer devolvera el valor por defecto 10

    descripcion = descripcion.lower()
    numeros = []

    #a continuación, se analizan distintos escenarios de cómo este dato puede ser
dado:
    #"número de plazas: 224"
    match_numero = re.search(r'n[uú]mero de plazas[:\s]+(\d+)', descripcion)
    if match_numero:
        numeros.append(int(match_numero.group(1)))

    #"plazas: 123"
    match_total = re.search(r'plazas:\s*(\d+)', descripcion)
    if match_total:
        numeros.append(int(match_total.group(1)))

    #"131 públicas y 408 para residentes". En este caso se suman
```

```
    matches = re.findall(
        r'(\d+)\s*(?:p[úu]blicas|para residentes|rotacionales|residentes|para
autom[óo]viles)',
        descripcion
    )
    numeros += [int(n) for n in matches]

    #"para automóviles: 344"
    matches_colon = re.findall(r'para autom[óo]viles:\s*(\d+)', descripcion)
    numeros += [int(n) for n in matches_colon]

    numeros_unicos = list(set(numeros))

    if numeros_unicos:
        return sum(numeros_unicos)

    return 10  #si no se puede extraer nada

#procesa e importa datos a postgre
def importar():
    registros = []

    for archivo in archivos_csv:
        try:
            df = pd.read_csv(archivo, delimiter=";", encoding="latin1",
usecols=columnas_relevantes)

        except Exception as e:
            print(f"Error leyendo {archivo}: {e}")
            continue

        df["NOMBRE"] = df["NOMBRE"].apply(lambda x:
x.encode("latin1").decode("utf-8") if isinstance(x, str) else x) #esto es muy
util para el tema tildes

        df = df.dropna(subset=["LATITUD", "LONGITUD"])
        df["total_spots"] = df["DESCRIPCION"].apply(extraer_total_spots)

        df_unificado = df.rename(columns={
            "PK": "id",
            "NOMBRE": "nombre",
            "TIPO": "tipo",
            "LATITUD": "latitud",
            "LONGITUD": "longitud"
        })[["id", "nombre", "tipo", "latitud", "longitud", "total_spots"]]

        df_unificado["free_spots"] = df_unificado["total_spots"] // 2 #por
defecto pone el campo free_spots a la mitad del total_spots
        registros.append(df_unificado)

    df_final = pd.concat(registros, ignore_index=True)

    db: Session = SessionLocal()
```

```
    for _, row in df_final.iterrows():
        try:
            lat = float(str(row["latitud"]).replace("--", "-"))
            lon = float(str(row["longitud"]).replace("--", "-"))

            parking = ParkingDB(
                id=int(row["id"]),
                nombre=str(row["nombre"])[:255],
                tipo=str(row.get("tipo", ""))[:50],
                latitud=lat,
                longitud=lon,
                total_spots=int(row["total_spots"]),
                free_spots=int(row["free_spots"])
            )
            db.merge(parking)
            db.commit()
        except Exception as e:
            print(f"Error insertando parking {row['id']}: {e}")
            db.rollback()
    db.close()
    print("Todos los parkings han sido importados.")

if __name__ == "__main__":
    importar()


#parkin_actions
#parking_actions.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from datetime import datetime

from auth import get_db, oauth2_scheme, SECRET_KEY, ALGORITHM
from jose import jwt

from model_user import UserDB
from model_db import ParkingDB
from model_profile import ParkingRecordDB

router = APIRouter(prefix="/parking", tags=["parking"])

def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db)
) -> UserDB:
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        if not username:
            raise HTTPException(status_code=401, detail="Token inválido")
    except Exception:
        raise HTTPException(status_code=401, detail="Token inválido")
    user = db.query(UserDB).filter_by(username=username).first()
```

```
    if not user:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
    return user

@router.post("/{parking_id}/park")
def park(
    parking_id: int,
    db: Session = Depends(get_db),
    user: UserDB = Depends(get_current_user)
):
    parking = db.query(ParkingDB).get(parking_id)
    if not parking:
        raise HTTPException(status_code=404, detail="Parking no encontrado")
    if parking.free_spots < 1:
        raise HTTPException(status_code=400, detail="No hay plazas libres")
    if user.current_parking_id is not None:
        raise HTTPException(status_code=400, detail="Ya estás aparcado en otro
parking")

    #actualizar plazas y estado del usuario
    parking.free_spots -= 1
    user.current_parking_id = parking.id

    #registrar histórico
    record = ParkingRecordDB(
        user_id=user.id,
        parking_id=parking.id,
        date=datetime.utcnow()
    )
    db.add(record)
    db.commit()

    return {
        "message": "Aparcado correctamente",
        "free_spots": parking.free_spots
    }

@router.post("/{parking_id}/unpark")
def unpark(
    parking_id: int,
    db: Session = Depends(get_db),
    user: UserDB = Depends(get_current_user)
):
    if user.current_parking_id != parking_id:
        raise HTTPException(status_code=400, detail="No estás aparcado en ese
parking")

    parking = db.query(ParkingDB).get(parking_id)
    if not parking:
        raise HTTPException(status_code=404, detail="Parking no encontrado")

    #restaurar plazas, sin exceder el total
    parking.free_spots = min(parking.free_spots + 1, parking.total_spots)
```

```
    user.current_parking_id = None
    db.commit()

    return {
        "message": "Has abandonado el parking",
        "free_spots": parking.free_spots
    }

#auth.py
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from database import SessionLocal
from model_user import UserDB
from pydantic import BaseModel
from passlib.context import CryptContext
from jose import JWTError, jwt
from datetime import datetime, timedelta
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

#configuración
SECRET_KEY = "mi_clave_super_secreta"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60

router = APIRouter()
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

#db session dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

#pydantic models
class UserCreate(BaseModel):
    username: str
    password: str

class UserOut(BaseModel):
    id: int
    username: str

    class Config:
        orm_mode = True

class Token(BaseModel):
    access_token: str
    token_type: str

#funciones auxiliares
```

```
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def hash_password(password):
    return pwd_context.hash(password)

def create_access_token(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def get_user(db, username: str):
    return db.query(UserDB).filter(UserDB.username == username).first()

def authenticate_user(db, username: str, password: str):
    user = get_user(db, username)
    if not user or not verify_password(password, user.hashed_password):
        return None
    return user

#rutas
@router.post("/register", response_model=UserOut)
def register(user: UserCreate, db: Session = Depends(get_db)):
    if get_user(db, user.username):
        raise HTTPException(status_code=400, detail="Usuario ya existe")
    hashed_pw = hash_password(user.password)
    db_user = UserDB(username=user.username, hashed_password=hashed_pw)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user

@router.post("/login", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session =
Depends(get_db)):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Credenciales inválidas")
    access_token = create_access_token(data={"sub": user.username})
    return {"access_token": access_token, "token_type": "bearer"}

@router.get("/me", response_model=UserOut)
def get_me(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        if username is None:
            raise HTTPException(status_code=401, detail="Token inválido")
    except JWTError:
        raise HTTPException(status_code=401, detail="Token inválido")

    user = get_user(db, username)
```

```
    if user is None:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
    return user

#car.py
import sys
print(sys.executable)

import torch
import numpy as np
import cv2
np.bool = bool  #parche para versiones antiguas de yolo

#url del stream de la cámara
STREAM_URL = "http://170.249.152.2:8080/cgi-bin/viewer/video.jpg?r=1740941740"

#cargar modelo yolo preentrenado
yolo = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)

def detect_cars_from_stream():
    """
    captura una imagen del stream y detecta si hay coches, devolviendo:
    - un booleano indicando si hay al menos un coche.
    - un entero indicando el número de coches detectados.
    """
    cap = cv2.VideoCapture(STREAM_URL)
    if not cap.isOpened():
        print("error: no se pudo abrir el stream.")
        return False, 0

    ret, frame = cap.read()
    cap.release()

    if not ret:
        print("error: no se pudo capturar la imagen.")
        return False, 0

    #realizar la detección con yolo
    results = yolo(frame)
    detections = results.pandas().xyxy[0]

    #filtrar detecciones de coches
    car_detections = detections[detections['name'] == 'car']
    num_cars = len(car_detections)

    return num_cars > 0, num_cars  #devuelve (bool, int)

if __name__ == "__main__":
    car_found, num_cars = detect_cars_from_stream()
    print(f"¿hay un coche en el stream?: {car_found} | cantidad detectada:
{num_cars}")

#database.py
```

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

#url de la base de datos
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"  #reemplazar por postgresql u
otro en producción

#motor de la base de datos
engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}  #solo
para sqlite
)

#sesiones de base de datos
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

#modelo base
Base = declarative_base()

#main.py
from fastapi import FastAPI, HTTPException, BackgroundTasks, Depends
import cv2
import torch
import asyncio
import pandas as pd
from fastapi.middleware.cors import CORSMiddleware
from typing import List
from parkings import Parking, ParkingOut
from sqlalchemy.orm import Session
from database import SessionLocal
from model_db import ParkingDB, CameraDB
from auth import router as auth_router
from pydantic import BaseModel
from profile_user import router_profile
from parking_actions import router as parking_router
from sensor_routes import router as sensor_router
from admin_routes import router as admin_router

class CameraIn(BaseModel):
    url: str
    tipo: str = "plaza"

app = FastAPI(debug=True)

@app.on_event("startup")
async def startup_event():
    print("startup_event: lanzando detect_car_loop")
    asyncio.create_task(detect_car_loop())

app.include_router(auth_router)
app.include_router(router_profile)
app.include_router(parking_router)
```

```
app.include_router(sensor_router)
app.include_router(admin_router)

#carga el modelo yolo al iniciar la api
yolo = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)

#estados para cámaras de entrada/salida
entry_exit_state: dict[int, bool] = {}  #estado anterior (había coche) por cámara
car_counters: dict[int, int] = {}  #contador neto de coches por parking

def process_entry_exit_camera(cam: CameraDB, frame) -> int:
    """
    retorna +1 si un coche acaba de entrar (tipo 'entrada'),
    -1 si acaba de salir (tipo 'salida'), o 0 en otro caso.
    """
    results = yolo(frame)
    detections = results.pandas().xyxy[0]
    has_car = bool((detections['name'] == 'car').any())
    prev = entry_exit_state.get(cam.id, False)
    entry_exit_state[cam.id] = has_car

    if has_car and not prev:
        return 1 if cam.tipo == "entrada" else -1 if cam.tipo == "salida" else 0
    return 0

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

def load_parkings_from_csv(csv_path: str):
    df = pd.read_csv(csv_path)
    parkings = []
    for _, row in df.iterrows():
        parkings.append(Parking(
            id=int(row["id"]),
            nombre=row.get('nombre', ''),
            latitude=float(row["latitud"]),
            longitude=float(row["longitud"]),
            total_spots=10,
            free_spots=5
        ))
    return parkings

async def detect_car_loop():
    while True:
        db: Session = SessionLocal()
        try:
            parkings = db.query(ParkingDB).all()
            for parking in parkings:
                #separamos cámaras por tipo
```

```
                plaza_urls = [cam.url for cam in parking.cameras if cam.tipo ==
"plaza"]
                entry_exit_cams = [cam for cam in parking.cameras if cam.tipo in
("entrada", "salida")]

                #modo plaza: detectamos todos en bloque
                if plaza_urls:
                    total = 0
                    for url in plaza_urls:
                        cap = cv2.VideoCapture(url)
                        if not cap.isOpened(): continue
                        ret, frame = cap.read(); cap.release()
                        if not ret: continue
                        total += len(results :=
yolo(frame).pandas().xyxy[0].query("name=='car'"))
                    parking.free_spots = max(parking.total_spots - total, 0)

                #modo entrada/salida
                if entry_exit_cams:
                    delta = 0
                    for cam in entry_exit_cams:
                        cap = cv2.VideoCapture(cam.url)
                        if not cap.isOpened(): continue
                        ret, frame = cap.read(); cap.release()
                        if not ret: continue
                        delta += process_entry_exit_camera(cam, frame)
                    #inicializamos contador si no existe
                    car_counters.setdefault(parking.id, parking.total_spots -
parking.free_spots)
                    #actualizamos y limitamos entre [0, total_spots]
                    car_counters[parking.id] = min(
                        max(car_counters[parking.id] + delta, 0),
                        parking.total_spots
                    )
                    parking.free_spots = parking.total_spots -
car_counters[parking.id]

            db.commit()
        except Exception as e:
            print("error bucle detección:", e)
        finally:
            db.close()
        await asyncio.sleep(10)

#cargar parkings reales desde csv unificado
parkings_db = load_parkings_from_csv("parkings_unificados.csv")

#configuración cors (para pruebas, permite todas las conexiones)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
```

```
    allow_headers=["*"],
)

#url del stream de la cámara
STREAM_URL = "http://170.249.152.2:8080/cgi-bin/viewer/video.jpg?r=1740941740"

#variable global para almacenar el estado de detección
car_detected_status = {"car_detected": False, "num_cars": 0}

@app.get("/check_car/")
async def check_car():
    """
    endpoint para obtener el estado más reciente de la detección de coches.
    """
    return car_detected_status

@app.get("/parkings/", response_model=List[ParkingOut])
def get_parkings():
    db: Session = SessionLocal()
    try:
        parkings = db.query(ParkingDB).all()
        return [
            ParkingOut(
                id=p.id,
                nombre=p.nombre,
                latitude=p.latitud,
                longitude=p.longitud,
                total_spots=p.total_spots,
                free_spots=p.free_spots
            )
            for p in parkings
        ]
    finally:
        db.close()

@app.get("/parking/{parking_id}", response_model=ParkingOut)
def get_parking_by_id(parking_id: int):
    db: Session = SessionLocal()
    try:
        parking = db.query(ParkingDB).filter(ParkingDB.id == parking_id).first()
        if parking is None:
            raise HTTPException(status_code=404, detail="Parking no encontrado")
        return ParkingOut(
            id=parking.id,
            nombre=parking.nombre,
            latitude=parking.latitud,
            longitude=parking.longitud,
            total_spots=parking.total_spots,
            free_spots=parking.free_spots
        )
    finally:
        db.close()
```

```
@app.get("/parking/{parking_id}/cameras")
def get_cameras(parking_id: int, db: Session = Depends(get_db)):
    parking = db.query(ParkingDB).filter(ParkingDB.id == parking_id).first()
    if not parking:
        raise HTTPException(status_code=404, detail="Parking no encontrado")

    return [{"id": cam.id, "url": cam.url, "tipo": cam.tipo} for cam in
parking.cameras]

@app.delete("/camera/{camera_id}")
def delete_camera(camera_id: int, db: Session = Depends(get_db)):
    camera = db.query(CameraDB).filter(CameraDB.id == camera_id).first()
    if not camera:
        raise HTTPException(status_code=404, detail="Cámara no encontrada")

    db.delete(camera)
    db.commit()
    return {"message": f"Cámara con ID {camera_id} eliminada"}

//splash_page.dart
import 'package:flutter/material.dart';
import 'package:flutter_secure_storage/flutter_secure_storage.dart';
import 'package:http/http.dart' as http;

class SplashPage extends StatefulWidget {
  const SplashPage({super.key});

  @override
  State<SplashPage> createState() => _SplashPageState();
}

class _SplashPageState extends State<SplashPage> {
  final _storage = const FlutterSecureStorage();

  @override
  void initState() {
    super.initState();
    _startSplash();
  }

  Future<void> _startSplash() async {
    //espera 2 segundos como mínimo
    await Future.delayed(const Duration(seconds: 2));
    await checkAuth();
  }

  Future<void> checkAuth() async {
    final token = await _storage.read(key: 'access_token');

    if (!mounted) return;

    if (token != null && token.isNotEmpty) {
      final response = await http.get(
```

```
        Uri.parse('http://192.168.1.43:8000/me'),
        headers: {'Authorization': 'Bearer $token'},
      );

      if (response.statusCode == 200) {
        Navigator.pushReplacementNamed(context, '/home');
      } else {
        await _storage.delete(key: 'access_token');
        Navigator.pushReplacementNamed(context, '/login');
      }
    } else {
      Navigator.pushReplacementNamed(context, '/login');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black, //fondo negro
      body: Center(
        child: Image.asset(
          'assets/F_icon.png',
          width: 100,
          height: 100,
        ),
      ),
    );
  }
}

//login_page.dart
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'package:flutter_secure_storage/flutter_secure_storage.dart';

class LoginPage extends StatefulWidget {
  const LoginPage({Key? key}) : super(key: key);

  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  final _formKey = GlobalKey<FormState>();
  String username = '';
  String password = '';

  Future<void> login() async {
    final url = Uri.parse('http://192.168.1.43:8000/login'); //cambia a tu ip o
dominio real

    final response = await http.post(
```

```
      url,
      headers: {'Content-Type': 'application/x-www-form-urlencoded'},
      body: {'username': username, 'password': password},
    );

    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      final token = data['access_token'];

      //guardamos el token aquí
      final storage = FlutterSecureStorage();
      await storage.write(key: 'access_token', value: token);
      print('token guardado correctamente en almacenamiento seguro.');

      //navega a la pantalla principal de tu app
      Navigator.pushReplacementNamed(context, '/home');
    } else {
      print('error en el login: ${response.body}');
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Usuario o contraseña incorrectos')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      body: Center(
        child: SingleChildScrollView(
          padding: const EdgeInsets.all(24.0),
          child: Card(
            shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(20)),
            elevation: 10,
            color: Colors.white,
            child: Padding(
              padding: const EdgeInsets.all(20.0),
              child: Form(
                key: _formKey,
                child: Column(
                  mainAxisSize: MainAxisSize.min,
                  children: [
                    Image.asset('assets/letrasrect.png', height: 80),
                    const SizedBox(height: 20),
                    const Text(
                      'Iniciar Sesión',
                      style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold,
color: Colors.black),
                    ),
                    const SizedBox(height: 20),
                    TextFormField(
                      decoration: const InputDecoration(
```

```
                        labelText: 'Usuario',
                        border: OutlineInputBorder(),
                      ),
                      validator: (value) => value!.isEmpty ? 'Introduce tu
usuario' : null,
                      onChanged: (value) => username = value,
                    ),
                    const SizedBox(height: 16),
                    TextFormField(
                      obscureText: true,
                      decoration: const InputDecoration(
                        labelText: 'Contraseña',
                        border: OutlineInputBorder(),
                      ),
                      validator: (value) => value!.isEmpty ? 'Introduce tu
contraseña' : null,
                      onChanged: (value) => password = value,
                    ),
                    const SizedBox(height: 24),
                    ElevatedButton(
                      style: ElevatedButton.styleFrom(
                        backgroundColor: Color(0xFF18E76E), //verde corporativo
                        foregroundColor: Colors.black,
                        padding: const EdgeInsets.symmetric(horizontal: 40,
vertical: 16),
                        shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(10)),
                      ),
                      onPressed: () {
                        if (_formKey.currentState!.validate()) {
                          login();
                        }
                      },
                      child: const Text('Entrar'),
                    ),
                    const SizedBox(height: 10),
                    TextButton(
                      onPressed: () {
                        Navigator.pushNamed(context, '/register'); //navega a la
pantalla de registro
                      },
                      child: const Text('¿No tienes cuenta? Regístrate', style:
TextStyle(color: Colors.black)),
                    ),
                  ],
                ),
              ),
            ),
          ),
        ),
      ),
    );
  }
```

```
}

//home_page.dart
import 'dart:convert';
import 'dart:async';

import 'package:flutter/material.dart';
import 'package:flutter_map/flutter_map.dart';
import 'package:latlong2/latlong.dart';
import 'package:http/http.dart' as http;
import 'package:flutter_secure_storage/flutter_secure_storage.dart';

import '../../models/parking.dart';
import 'home_map_view.dart';
import 'parking_markers.dart';
import 'user_drawer.dart';
import '../../services/location_service.dart';
import 'user_location_marker.dart';
import 'parking_card_dialog.dart';
import '../../services/parking_service.dart';

class HomePage extends StatefulWidget {
  const HomePage({super.key, required this.title});
  final String title;

  @override
  State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  final GlobalKey<ScaffoldState> _scaffoldKey = GlobalKey<ScaffoldState>();
  late final MapController _mapController;
  late final LatLng _initialCenter;
  late final double _initialZoom;
  LatLng _currentCenter = LatLng(40.590293067583126, -4.1341927056389975);
  double _currentZoom = 11;

  LatLng? _userLocation;
  Timer? _locationUpdateTimer;
  static const int locationSeconds = 15;

  String? username;
  String? token;
  Map<String, dynamic>? profileData;
  int? _currentParkingId;
  String? _currentParkingName;

  Timer? _parkingUpdateTimer;
  static const int parkingSeconds = 120;
  List<Parking>? _parkings;
  bool _isLoadingParkings = true;

  Timer? _highlightTimer; //este timer es para la animación de parpadeo
```

```
//esto es para el tema de parking cercano. ids de los parkings resaltados
List<int> _highlightedParkingIds = [];

void _highlightNearestParkings({int count = 5}) {
  if (_userLocation == null || _parkings == null) return;
  final dist = Distance();
  final sorted = List<Parking>.from(_parkings!)
    ..sort((a, b) => dist
      .distance(_userLocation!, LatLng(a.latitude, a.longitude))
      .compareTo(
        dist.distance(_userLocation!, LatLng(b.latitude, b.longitude))
      ));
  setState(() {
    _highlightedParkingIds =
      sorted.take(count).map((p) => p.id).toList();
  });
}

void _onHighlightButtonPressed({int count = 5}) {
  //cancela un posible timer anterior
  _highlightTimer?.cancel();

  //calcula y pinta los 5 parkings más cercanos
  _highlightNearestParkings(count: count);

  //tras 15 s, borra los resaltados
  _highlightTimer = Timer(const Duration(seconds: 15), () {
    setState(() => _highlightedParkingIds.clear());
  });
}

@override
void initState() {
  super.initState();
  _mapController = MapController();
  _initialCenter = _currentCenter;
  _initialZoom   = _currentZoom;

  //carga inicial de parkings
  _loadParkings();

  startLocationUpdates();
  loadUserProfile();
}

@override
void dispose() {
  _locationUpdateTimer?.cancel();
  _parkingUpdateTimer?.cancel();
  _highlightTimer?.cancel();
  super.dispose();
}
```

```
void startLocationUpdates() {
  _locationUpdateTimer?.cancel();
  _locationUpdateTimer = Timer.periodic(
    const Duration(seconds: locationSeconds),
    (_) => updateUserLocation(),
  );
}

Future<void> updateUserLocation() async {
  final pos = await LocationService.getCurrentLocation();
  if (pos != null) {
    setState(() => _userLocation = pos);
  }
}

Future<void> loadUserProfile() async {
  const storage = FlutterSecureStorage();
  final savedToken = await storage.read(key: 'access_token');
  if (savedToken == null) return;

  try {
    final data = await fetchUserProfile(savedToken);
    setState(() {
      token = savedToken;
      username = data['username'];
      profileData = data;
      final current = data['current_parking'];
      _currentParkingId =
          current != null ? current['id'] as int : null;
      _currentParkingName =
          current != null ? current['nombre'] as String : null;
    });
  } catch (e) {
    //maneja el error si es necesario
  }
}

Future<Map<String, dynamic>> fetchUserProfile(String token) async {
  final response = await http.get(
    Uri.parse('http://192.168.1.43:8000/profile'),
    headers: {'Authorization': 'Bearer $token'},
  );
  if (response.statusCode == 200) {
    final utf8Body = utf8.decode(response.bodyBytes);
    return jsonDecode(utf8Body) as Map<String, dynamic>;
  } else {
    throw Exception('Error al cargar perfil');
  }
}

double getMarkerScale() {
  if (_currentZoom < 6) return 0;
  if (_currentZoom < 12) return 0.1;
```

```
    if (_currentZoom < 15) return 0.2;
    return 0.4;
  }

  Future<void> _loadParkings() async {
    setState(() => _isLoadingParkings = true);
    try {
      final response = await http.get(
        Uri.parse('http://192.168.1.43:8000/parkings/'),
      );
      if (response.statusCode != 200) throw Exception();
      final utf8Body = utf8.decode(response.bodyBytes);
      final List<dynamic> jsonList = jsonDecode(utf8Body);
      _parkings = jsonList
          .map((j) => Parking.fromJson(j as Map<String, dynamic>))
          .toList();
    } catch (_) {
      _parkings = [];
    } finally {
      setState(() => _isLoadingParkings = false);
    }
  }

  Future<void> _onParkingButtonPressed(int parkingId, bool isCurrentlyParked)
async {
    try {
      if (isCurrentlyParked) {
        //abandonar
        await ParkingService(token!).unpark(parkingId);
        setState(() {
          _currentParkingId   = null;
          _currentParkingName = null;
          //actualiza profileData para que el drawer muestre "ninguno"
          if (profileData != null) {
            profileData!['current_parking'] = null;
          }
        });
      } else {
        //aparcar
        await ParkingService(token!).park(parkingId);
        //busca el nombre del parking en tu lista
        final parking = _parkings!.firstWhere((p) => p.id == parkingId);
        setState(() {
          _currentParkingId   = parkingId;
          _currentParkingName = parking.nombre;
          //actualiza profileData con el nuevo parking
          if (profileData != null) {
            profileData!['current_parking'] = {
              'id': parkingId,
              'nombre': parking.nombre,
            };
          }
        });
```

```
      }
      //recarga todos los parkings para actualizar freeSpots
      await _loadParkings();
    } catch (e) {
      ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text(e.toString())));
    }
  }

//main.dart
import 'package:flutter/material.dart';
import 'pages/register_page.dart';
import 'pages/login_page.dart';
import 'pages/home/home_page.dart'; //tu mapa o pantalla principal
import 'pages/splash_page.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'ParkiNeo',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Color(0xFF18E76E)),
        useMaterial3: true,
      ),
      initialRoute: '/', //aquí iniciaremos
      routes: {
        '/': (context) => const SplashPage(),
        '/login': (context) => const LoginPage(),
        '/register': (context) => const RegisterPage(),
        '/home': (context) => const HomePage(title: 'Mapa Demo'),
      },
    );
  }
}
```