Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Servicios REST, aplicación web y Android para la Gestión de Parkings con soporte para pagos PayPal y navegación con Google Maps

Autor: José María Galindo Rodríguez
Tutor: María Teresa Ariza Gómez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2024

Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Servicios REST, aplicación web y Android para la Gestión de Parkings con soporte para pagos PayPal y navegación con Google Maps

Autor:
José María Galindo Rodríguez

Tutor:
María Teresa Ariza Gómez
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo de Fin de Grado: Servicios REST, aplicación web y Android para la Gestión de Parkings con soporte para pagos PayPal y navegación con Google Maps

Autor: José María Galindo Rodríguez

Tutor: Maria Teresa Ariza Gómez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia

A mis profesores

Agradecimientos

En primer lugar, agradecer a mi tutora Maria Teresa Ariza Gómez, por su ayuda, consejos, tiempo y conocimientos que me han ayudado a crecer tanto personal como académicamente. Además, agradecer la oportunidad de haber realizado este trabajo bajo su tutela.

En segundo lugar, agradezco a mis padres, mi hermana y toda mi familia por confiar siempre en mí y por todo el esfuerzo y sacrificio que han realizado durante esta etapa de mi vida. Sus palabras cada día han sido una fuente de motivación y un apoyo incondicional durante todos estos años.

Mi agradecimiento a mi pareja Censi, por ser el mayor apoyo día a día y a su vez por su infinita paciencia, motivación y ayuda. Su apoyo constante ha sido un faro de esperanza y fortaleza en cada paso de esta etapa.

Por último, a todos mis compañeros con los que he compartido este camino durante estos años y las amistades que han surgido en esta etapa. Además, a todos los profesores por su ayuda y conocimiento brindado durante toda la carrera.

José María Galindo Rodríguez

Sevilla, 2024

Resumen

En el mundo actual, el concepto de ciudad inteligente está ganando más interés y apoyo por parte del ser humano con el objetivo específico de enriquecer la eficiencia y la comodidad relacionadas con nuestras actividades diarias. Este fenómeno está respaldado por una variedad de avances tecnológicos que han redefinido nuestra relación con los elementos del entorno urbano. Toda esta evolución ha sido impulsada por la proliferación del uso de dispositivos móviles y el avance de la conectividad a Internet. Estos desarrollos nos han permitido poder tener la oportunidad de interactuar con los sistemas de parkings de nuestro entorno de otra forma distinta.

En este proyecto se ha desarrollado una aplicación móvil que permite a los usuarios gestionar los vehículos, las reservas en los estacionamientos, visualizar en el mapa la información de los espacios de estacionamiento cercanos usando la API de Google Maps, control de entrada y salida del estacionamiento mediante el escaneo de QR y la posibilidad de realizar pagos de las tarifas a través de la plataforma de pago PayPal usando su API. Por otro lado, se han implementado dos servicios REST, uno para la gestión del centro de gestión de parkings y otro para la gestión del parking. Finalmente, se ha desarrollado una aplicación web para el gestor del parking la cual permite visualizar información para una gestión eficaz.

El objetivo principal del proyecto es proporcionar a los usuarios herramientas fáciles y cómodas de usar para la interacción con los sistemas de estacionamiento. Además, proporcionar herramientas para la gestión del parking y los centros de gestión de parkings. Esto permitirá a los usuarios que la búsqueda de estacionamiento no sea una tarea dificultosa en los núcleos urbanos.

Abstract

In today's world, the concept of a smart city is gaining more interest and support from humans with the specific goal of enhancing efficiency and convenience related to our daily activities. This phenomenon is supported by a variety of technological advancements that have redefined our relationship with urban environment elements. This entire evolution has been driven by the proliferation of mobile device usage and advancements in Internet connectivity. These developments have allowed us the opportunity to interact with parking systems in our surroundings in a different way.

In this project, a mobile application has been developed that allows users to manage their vehicles, make parking reservations, view information about nearby parking spaces on a map using the Google Maps API, control entry and exit from the parking area through QR scanning, and make payments for fees via the PayPal payment platform using its API. Additionally, two REST services have been implemented, one for managing the parking management center and another for managing the parking facility. Finally, a web application has been developed for the parking manager, which allows for the visualization of information for effective management.

The main objective of the project is to provide users with easy and convenient tools for interacting with parking systems. Furthermore, it aims to provide tools for managing the parking facility and parking management centers. This will make the search for parking less challenging in urban centers.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

Notación

REST	Transferencia de Estado Representacional
ART-IT	Arquitectura de Referencia para el Transporte Cooperativo e Inteligente
QR	Código de Respuesta Rápida
HTTP	Protocolo de Transferencia Hipertexto
API	Interfaz de Programación de Aplicaciones
IDE	Entorno de Desarrollo Integrado
URL	Localizador Uniforme de Recursos
JPA	API de Persistencia Java
DNI	Documento Nacional de Identidad
CRUD	Crear, Leer, Actualizar y Borrar
SQL	Lenguaje de Consulta Estructurado
BBDD	Base de Datos
POM	Modelo de Objetos de Proyecto
IP	Protocolo de Internet
JSON	Notación de Objetos JavaScript
NIF	Número de Identificación Fiscal
HTML	Lenguaje de Marcas de Hipertexto

1 INTRODUCCIÓN

La ciencia de hoy es la tecnología de mañana.

– Edward Teller –

1.1 Motivación

Este proyecto surge ante la necesidad de transformar la experiencia de estacionamiento en entornos urbanos mediante la utilización de las nuevas tecnologías y convertirlo en un entorno más eficiente. Se implementa un sistema para que una tarea cotidiana como puede ser encontrar estacionamiento en grandes núcleos urbanos se haga más llevadera y no sea un desafío para los ciudadanos. Dicho sistema ayudará a realizar una búsqueda eficiente de espacios de estacionamiento, cobro de tarifas y gestionar los distintos estacionamientos, con tecnologías que se detallarán más adelante.

Este proyecto no solo proporciona la ocasión especial de investigar, aprender y aplicar tecnologías avanzadas, como el desarrollo de servicios REST y la integración de un sistema para ciudades inteligentes, sino que también desempeña un papel esencial en mi desarrollo personal. Al participar en este proyecto a nivel académico, tengo la oportunidad de adquirir y perfeccionar mis habilidades en áreas fundamentales como el estudio de cómo funciona una ciudad inteligente, el diseño de aplicaciones móviles y la gestión de los espacios de estacionamiento, lo cual contribuirá significativamente a mi crecimiento y formación en estos campos clave.

El objetivo principal del proyecto es proporcionar a los usuarios herramientas fáciles y cómodas de usar para la interacción con los sistemas de estacionamiento. Esto incluye reserva de estacionamiento, visualización de espacios de estacionamiento cercanos, control de entrada y salida del parking y la posibilidad de realizar pagos a través de plataformas de pago móvil. Además, proporcionar herramientas para la gestión del parking y los centros de gestión de parkings.

Todo ello se realizará utilizando la tecnología que se describirá en los siguientes capítulos de este proyecto con el fin de mejorar la experiencia de estacionamiento del usuario e incrementar el uso de los recursos disponibles en estas áreas. Una solución beneficiosa para el usuario final y el medio ambiente. Este proyecto es una oportunidad para combinar nuevas tecnologías con un enfoque en la sostenibilidad y así dar ejemplo para futuras investigaciones y desarrollo sobre gestión inteligente de aparcamientos y ciudades inteligentes.

1.2 Objetivos

En esta sección, se muestran los objetivos de este proyecto:

- Implementar un servicio REST que permita simular el centro de gestión de parkings en el que se obtiene información de los diversos parkings del territorio donde se encuentre e información sobre los usuarios y los vehículos que hacen uso del sistema.
- Implementar un servicio REST que permita monitorizar el estado en tiempo real del parking y comunicarse con el centro de gestión de parkings para el intercambio de información.
- Desarrollar una aplicación Android intuitiva y centralizada con el fin de soportar las actividades diarias de los usuarios que hacen uso de la misma en su interacción con los parkings y el centro de gestión de parkings. Además, integrar los servicios de Google Maps y PayPal en la aplicación.

- Desarrollar una aplicación web para que el usuario encargado de la gestión del parking pueda obtener y modificar datos a tiempo real para permitir su gestión.

1.3 Descripción de la solución

En este apartado se presentará la descripción de la solución del sistema. Para la solución de este proyecto se ha tomado como referencia el paquete de servicios que fue desarrollado para el Departamento de Transporte de Estados Unidos, este presenta una Arquitectura de Referencia para el Transporte Cooperativo e Inteligente (ARC-IT) [1] que ayuda a la planificación e integración de sistemas inteligentes de transporte. En concreto, se ha considerado únicamente el paquete de gestión de parkings. Este paquete, establece una relación entre un centro de gestión de parkings que contiene información de los vehículos y los usuarios que hacen uso del sistema además de con los demás parkings del sistema. Por otro lado, también contempla la habitual relación entre los usuarios y el parking del cuál hacen uso en un determinado momento. A continuación, se detallarán las funcionalidades y arquitectura del sistema para poder alcanzar los objetivos mencionados anteriormente en esta memoria.

1.3.1 Funcionalidades del sistema

Las funcionalidades principales de este proyecto son las siguientes:

- Interfaz de usuario sencilla para facilitar su uso.
- Autenticación de los usuarios que hacen uso de la aplicación móvil.
- Permitir a los usuarios gestionar sus vehículos y las reservas realizadas en los parkings.
- Consultar la ubicación de los parkings de la red y los datos de los mismos mediante Google Maps.
- Permitir a los usuarios la entrada y salida del parking mediante el escaneo de QR con la aplicación Android.
- Pago directo de tarifas desde la aplicación Android mediante la plataforma de pago PayPal.
- Facilitar la comunicación y el intercambio de datos entre el centro de gestión de parkings y los parkings.
- Permitir al operario del parking conocer datos del mismo a tiempo real y poder modificarlos para su gestión eficaz.
- Dar de alta a un nuevo parking en el centro de gestión.

1.3.2 Arquitectura del sistema

La arquitectura del sistema es la mostrada en la ilustración 1-1. Se presenta un esquema que resume la estructura general del proyecto, se pueden distinguir cuatro elementos principales:

- **Aplicación móvil**: este componente se corresponde con la aplicación Android desarrollada para el proyecto. La aplicación es emulada en el entorno de Android Studio. Este entorno, nos permite simular un terminal móvil con las mismas funcionalidades que presenta un terminal físico. La aplicación usa la API de Google Maps para mostrar los parkings y su información en el mapa y PayPal para el pago de tarifas desde el dispositivo móvil.
- **API REST del centro de gestión de parkings**: este componente se encarga de gestionar las solicitudes y las respuestas relacionadas con el centro de gestión de parkings y se puede considerar uno de los pilares principales de la arquitectura. Se encarga de recopilar datos de todos los parkings que tenga asociados y los persiste en su base de datos vinculada. Por otro lado, gestiona los datos de todos los usuarios y vehículos que hacen uso del sistema.

- **API REST de gestión de parking**: este componente se encarga de gestionar las solicitudes y las respuestas relacionadas con la gestión del parking y se considera otro punto principal de la arquitectura. Este componente simula lo que conocemos como parking y existe uno por parking gestionado. Se encarga de recopilar todos los datos necesarios para su gestión y persistirlos en su base de datos asociada.
- **Aplicación web gestor de parking**: este componente se corresponde con una aplicación web que tiene una interfaz de usuario sencilla para permitir la gestión del parking. Interacciona con el servicio de gestión de parking mencionado en el punto anterior. Se encarga de obtener y modificar los datos correspondientes al parking que gestiona. El gestor del parking se corresponde con el operario que administra el uso del mismo.

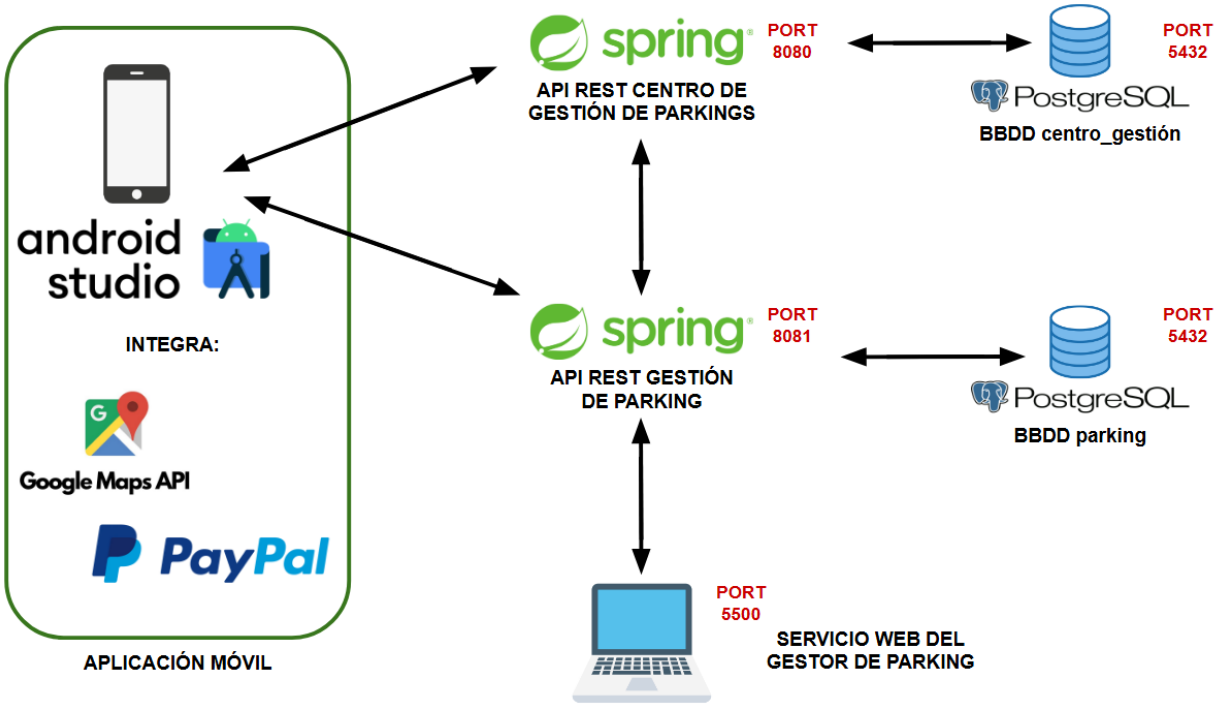

Ilustración 1-1. Arquitectura del sistema

Hay que destacar que todo el proyecto se ha realizado en la misma máquina por lo que la dirección IP ha sido la misma para todos los servicios, la 127.0.0.1 (localhost).

1.4 Estructura de la memoria

La estructura de la memoria de este proyecto se presenta de la siguiente forma:

- **Introducción**: en este apartado se explican de forma general los objetivos, el contexto, la razón de llevar a cabo este proyecto y cómo se organiza la aplicación.
- **Recursos utilizados**: se explican todos los recursos que se han utilizado para la creación del proyecto. Presenta una breve descripción tanto de los recursos hardware como de las tecnologías software que han sido utilizadas en el desarrollo de este proyecto.
- **Servicios REST**: se presenta la organización de los dos servicios desarrollados, los cuales se distribuyen de la siguiente forma:
    - **Servicio REST del centro de gestión de parkings**: se detallarán las características

fundamentales que presenta el servicio REST implementado para el centro de gestión de parkings. Se describirá la estructura, la seguridad que implementa y su conexión con la base de datos.

  - **Servicio REST de gestión de parking**: se explicarán los aspectos clave que presenta el servicio REST para la gestión del parking. Al igual que el anterior, se detallará la estructura del servicio y su conexión con la base de datos.

- **Aplicación Android**: se detalla la estructura y el análisis de la aplicación Android. Además, se describen las distintas funcionalidades que ofrece la aplicación de cara al usuario.
- **Aplicación web gestor de parking**: se explica la estructura y las funcionalidades que la aplicación web ofrece al gestor de parking.
- **Conclusión y líneas futuras**: en este capítulo se presenta una conclusión del proyecto y las líneas futuras que son ideas que podrían mejorarlo de cara al futuro y las próximas versiones que puedan surgir del mismo.

2 RECURSOS UTILIZADOS

Las grandes oportunidades nacen de haber sabido aprovechar las pequeñas.

– Bill Gates –

En este capítulo se indican los recursos hardware y software que se han utilizado para la implementación y desarrollo de este proyecto.

2.1 Recursos Hardware

2.1.1 Ordenador portátil

La implementación, desarrollo y pruebas de este proyecto, así como la redacción de este documento, se ha realizado en el ordenador portátil HP Omen 15-ax205ns tal y como se puede observar en la ilustración 2-1.

Ilustración 2-1. Portátil HP Omen 15-ax205ns

Las características de este ordenador son las siguientes [2]:

- Pantalla de 15.6 pulgadas con resolución Full HD (1920 x 1080 píxeles).
- Procesador: Intel Core i7-7700HQ.
- Tarjeta Gráfica: NVIDIA GeForce GTX 1050.
- Memoria RAM: 16,0 GB.
- Sistema Operativo: Windows 10.

2.2 Recursos Software

2.2.1 Android Studio

Para el desarrollo de la aplicación móvil se ha empleado Android Studio que es un entorno de desarrollo integrado (IDE) y la herramienta oficial para crear aplicaciones Android. Este recurso, ayuda a la creación de código, la depuración del mismo y la creación de interfaces de usuario que facilitan la vista de la aplicación hacia al usuario. Integra un emulador de Android el cuál se ha usado para probar la aplicación y realizar distintas pruebas en la misma. [3]

Ilustración 2-2. Android Studio

2.2.2 Ubuntu

El sistema operativo para el despliegue de los servicios REST, alojamiento de la base de datos y desarrollo de la aplicación móvil, ha sido Ubuntu 22.04. Para el alojamiento de este sistema operativo se ha usado la máquina virtual VMware Workstation Pro. [4]

Ilustración 2-3. Ubuntu 22.04

2.2.3 Spring

Spring es un framework para el desarrollo de aplicaciones Java que ofrece una variedad de herramientas y funcionalidades destinadas a la construcción eficiente y escalable de diversos componentes, incluidos servicios RESTful. [5]

Ilustración 2-4. Spring

2.2.4 Postman

Postman es una plataforma que permite a los desarrolladores probar y documentar APIs. Esta plataforma ofrece un entorno en el cual se pueden realizar peticiones HTTP especificando distintos tipos de solicitudes (POST, GET, DELETE…) configurando los parámetros necesarios. Por otro lado, ofrece la posibilidad de documentar y monitorizar el rendimiento de las APIs lo cual facilita la identificación de posibles problemas y mejorar la calidad del servicio de los sistemas. [6]

Ilustración 2-5. Postman

2.2.5 PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto. Se distingue por su amplio conjunto de funciones y su capacidad de adaptarse eficazmente para el almacenamiento y manipulación de datos. Este sistema proporciona un entorno confiable y versátil, cumpliendo con altos estándares de funcionalidad y flexibilidad en la gestión de información. [7]

Ilustración 2-6. PostgreSQL

2.2.6 Visual Studio Code

Visual Studio Code es un editor de código fuente gratuito y desarrollado por Microsoft. Ofrece funcionalidades como depuración de código, control de versiones, autocompletado inteligente de código y extensiones que pueden facilitar la creación de código, así como la ejecución del mismo. Este programa está diseñado para programar en diversos lenguajes de programación. [8]

Ilustración 2-7. Visual Studio Code

2.2.7 Wireshark

Wireshark es un analizador de protocolos gratuito y de código abierto. Se utiliza para realizar análisis de redes de comunicaciones, desarrollo de software y protocolos y auditorías de seguridad. Este software captura todo tipo de paquetes que atraviesan una conexión y nos permite filtrar las capturas por tipo de protocolo. Es compatible con los principales sistemas operativos como Windows, IOS, Ubuntu, etc. [9]

Ilustración 2-8. Wireshark

2.2.8 Emulador de Google Pixel 4 API 33

Android Studio proporciona una variedad de emuladores, y para este proyecto se ha seleccionado el emulador Google Pixel 4 API 33. Este emulador imita las características tanto de hardware como de software del dispositivo físico Android, facilitando así la prueba y depuración de las aplicaciones desarrolladas en el entorno de Android Studio de manera sencilla y comprensible.

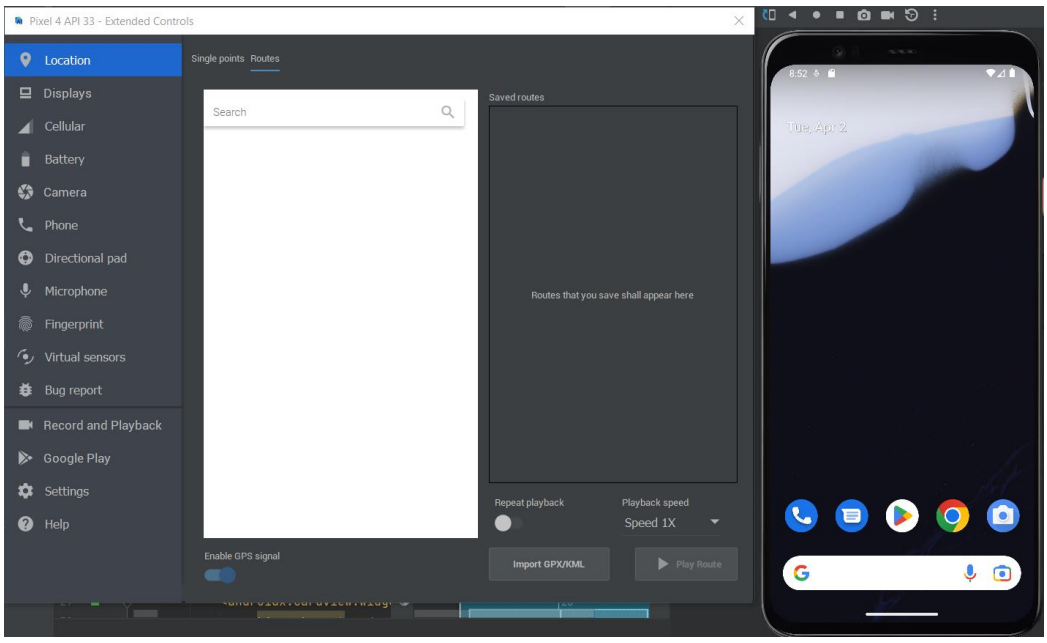

Ilustración 2-9. Emulador de Google Pixel 4 API 33

2.2.9 API Google Maps

La API de Google Maps ofrece la posibilidad de personalizar mapas con imágenes y contenido propio para su visualización en páginas web y dispositivos móviles. Esta API incluye cuatro tipos de mapas básicos (ruta, satélite, híbrido y terreno) que pueden ser modificados con capas y diseños, controles y eventos, así como diversos servicios y bibliotecas. Para ello se necesita una cuenta de desarrollador de Google y una clave que proporciona la plataforma para hacer uso de esta API. [10]

Ilustración 2-10. API Google Maps

2.2.10 API PayPal

La API de PayPal es una interfaz de programación de aplicaciones (API) proporcionada por PayPal que permite a los desarrolladores integrar funciones y servicios de PayPal en sus aplicaciones y sitios web. Con esta API, los desarrolladores pueden facilitar pagos en línea en su plataforma, administrar transacciones, acceder a información de cuentas, crear y enviar facturas, administrar listados y más. Esto brinda a los usuarios finales la capacidad de realizar pagos seguros y sin problemas sin abandonar el entorno familiar de la aplicación o el sitio web que utilizan. [11]

Ilustración 2-11. API PayPal

2.2.11 Swagger OpenAPI

La especificación OpenAPI, también conocida como Swagger, es un conjunto de reglas, especificaciones y herramientas de software de código abierto que nos ayudan a documentar servicios web RESTful. Esta herramienta permite documentar y definir claramente la estructura de las API REST que desees, esto conlleva una comprensión más fácil para los desarrolladores. Esta especificación posee una interfaz web que nos facilita la visualización de la estructura de la API REST y realizar las pruebas oportunas. [12]

Ilustración 2-12. Swagger OpenAPI

2.2.12 Spring Security

Spring Security es una biblioteca de seguridad diseñada para aplicaciones Java. Ofrece una amplia gama de herramientas que permiten autenticar y autorizar usuarios, así como proteger recursos y administrar sesiones de manera flexible. Gracias a sus características avanzadas, como la autenticación de usuarios, el control de acceso y la protección contra amenazas de seguridad. Spring Security ayuda a los desarrolladores a reforzar la seguridad de sus aplicaciones de forma efectiva. [13]

Ilustración 2-13. Spring Security

3 SERVICIOS REST

De vez en cuando, una nueva tecnología, un antiguo problema
y una gran idea se convierten en una innovación.

– Dean Kamen –

3.1 Servicio REST del centro de gestión de parkings

Esta sección es uno de los pilares fundamentales de este proyecto. Se explorarán minuciosamente las características fundamentales del servicio creado para la administración del centro de gestión de parkings. En este punto, se ofrecerá un análisis detallado tanto de la estructura completa de este servicio como de su interacción con la base de datos subyacente. Además, se abordará la documentación de los endpoints de este servicio mediante la herramienta Swagger. Este segmento proporcionará una descripción detallada sobre cómo opera el servicio REST del centro de gestión de parkings, destacando su relevancia dentro del panorama general de la aplicación al permitir la supervisión y control de este.

3.1.1 Servidor REST del centro de gestión de parkings

Este subapartado se centra en explicar con detalle la estructura y configuración del servidor correspondiente con el centro de gestión de parkings. Se incluyen las dependencias en este servicio, la definición de endpoints y las clases de las que está compuesto. Por otro lado, se detallará cómo interaccionan las clases controladoras con la base de datos de este servicio. Para el manejo de los datos del parking se ha empleado el modelo de datos Fiware [14], en concreto, se usará el enfocado a las ciudades inteligentes y los parkings. Este modelo de datos define la estructura y el tipo de los datos en cada contexto, además, proporciona ejemplos.

3.1.1.1 Definición de endpoints

En esta sección, se detallará un aspecto fundamental para el desarrollo del proyecto el cual consiste en detallar y explicar los endpoints de la API REST del centro de gestión de parkings implementada. Los endpoints se corresponden con los puntos de entrada a través de los cuales las aplicaciones se conectan con el servidor. Cada endpoint representa una acción particular que la API puede ejecutar, lo cual nos facilita el intercambio de datos y la comunicación de manera organizada y estructurada.

Los diferentes endpoints del servicio REST correspondiente al centro de gestión de parkings pueden verse en la tabla 3-1 mostrada a continuación:

PATH	Método HTTP	Descripción
/api/login	POST	Permite autenticar a los usuarios en el sistema.

/api/registro	POST	Permite dar de alta un nuevo usuario en el sistema.
/api/parking	GET	Permite obtener una lista con todos los datos de los parkings de la base de datos.
	POST	Permite dar de alta un nuevo parking en el sistema.
/api/vehiculos	GET	Permite obtener una lista con todos los vehículos que tiene asociado un determinado usuario.
	POST	Permite dar de alta en el sistema un vehículo asociado a un usuario.
/api/reserva	GET	Permite obtener una lista con las reservas que tiene un determinado usuario asociadas al vehículo que tiene en propiedad.
	POST	Permite crear una nueva reserva en la base de datos.
	DELETE	Permite eliminar una reserva de la base de datos.
/api/reserva/{id_parking}	GET	Permite obtener una lista con las reservas que están asociadas a un determinado parking.
/api/reserva/confirmacion/{reserva_id}	PATCH	Permite a un parking confirmar o cancelar la confirmación de una reserva.

Tabla 3–1. Endpoints del servicio de centro de gestión de parkings

A continuación, se explicará cómo se utiliza Swagger para documentar el centro de gestión de parkings. Swagger es una herramienta que automáticamente genera documentación interactiva para las API REST. La documentación detallada proporciona información clara sobre los puntos finales de la API, los parámetros que aceptan y los formatos de respuesta esperados. La integración de Swagger facilita la comprensión y el uso de la API por parte de los desarrolladores, mejora la comunicación y la colaboración en el equipo de desarrollo. También ofrece una manera fácil de probar los puntos finales de la API directamente desde la interfaz de usuario de Swagger, lo que hace que el proceso de desarrollo y depuración sea más sencillo. La siguiente ilustración 3-1 nos muestra la vista desde la interfaz web de Swagger que nos permite visualizar los endpoints de un modo más gráfico.

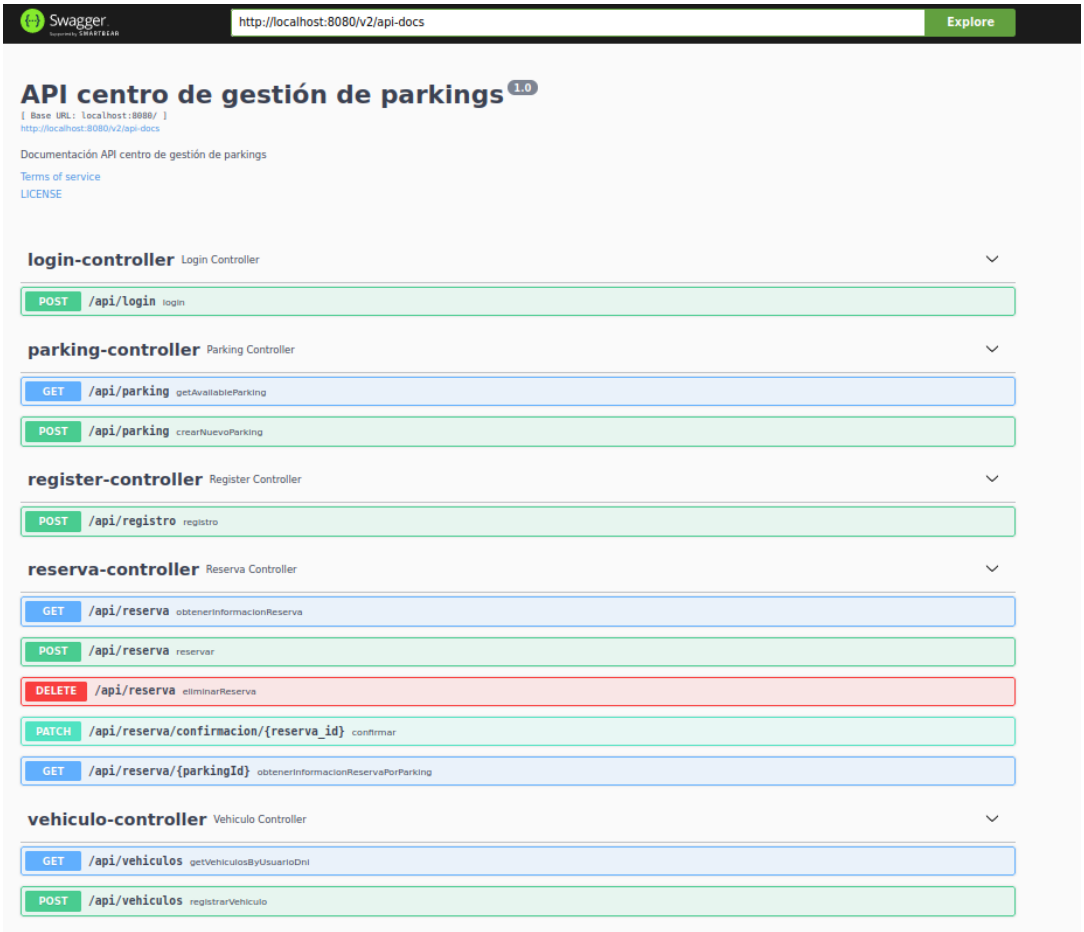

Ilustración 3-1. Endpoints en Swagger del centro de gestión de parkings

Algunos ejemplos de peticiones al servidor del centro de gestión de parkings son los siguientes:

- **Petición GET a http://localhost:8080/api/parking**: esta petición se realiza para obtener los datos de los parkings del centro de gestión. La respuesta de esta petición sigue el formato del modelo de datos Fiware [14] para los parkings, se muestra en la siguiente ilustración 3-2:

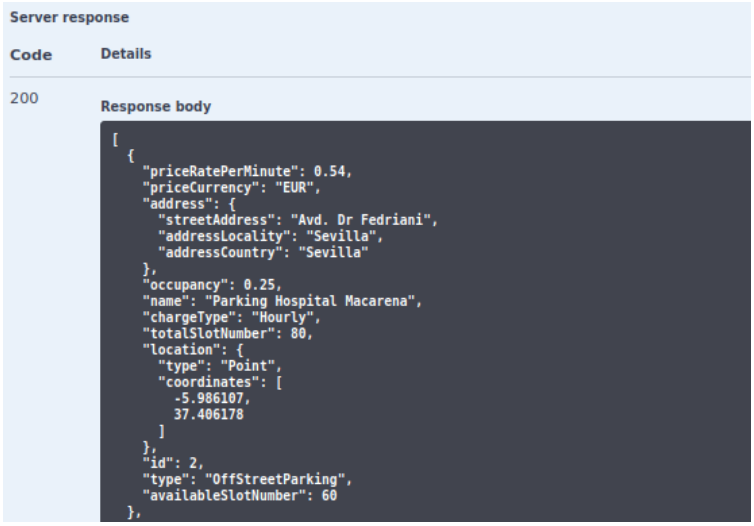

Ilustración 3-2. Respuesta petición GET a /api/parking

- **Petición POST a http://localhost:8080/api/vehiculos**: esta petición se realiza para dar de alta un nuevo vehículo en el centro de gestión. El cuerpo de la solicitud contiene, dni, matrícula, marca y modelo y número de plazas. En la ilustración 3-3 se puede ver el cuerpo de esta petición:

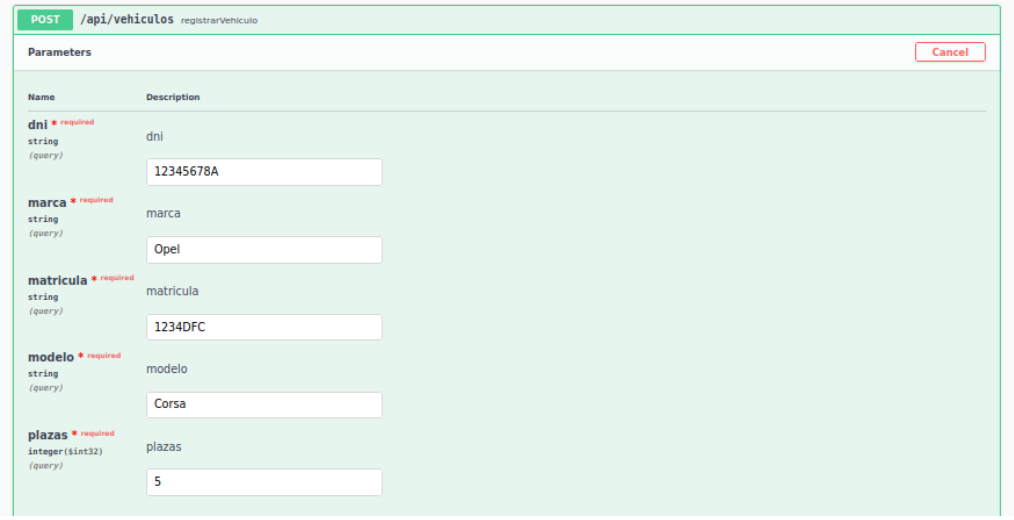

Ilustración 3-3. Petición POST a /api/vehículos

La respuesta obtenida la podemos ver en la ilustración 3-4. Obtenemos una respuesta exitosa con el código 200 OK en caso de que el registro se realice correctamente:

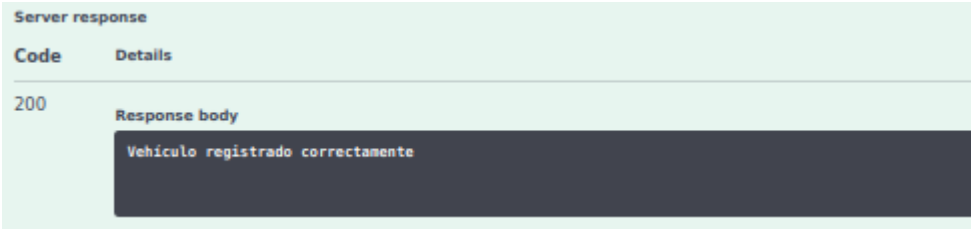

Ilustración 3-4. Respuesta exitosa del servidor a la petición POST

- **Petición PATCH a http://localhost:8080/api/reserva/confirmacion/{reserva_id}**: esta petición se realiza para modificar el estado de una reserva. En el cuerpo de esta petición se envía el estado de la reserva en el que confirmada puede ser true o false, ilustración 3-5:

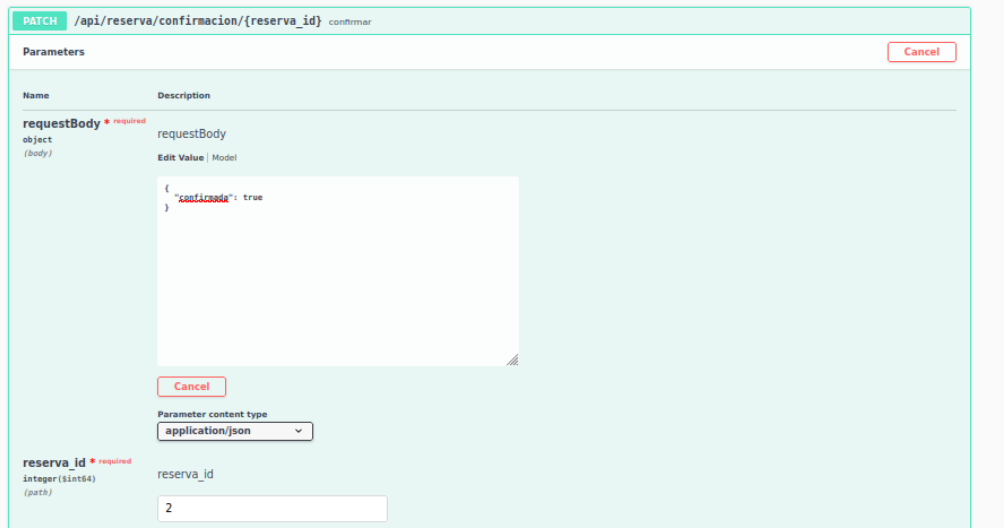

Ilustración 3-5. Petición PATCH a /api/reserva/confirmacion/{reserva_id}

La respuesta obtenida la podemos ver en la ilustración 3-6 para la confirmación. Obtenemos una respuesta exitosa con el código 200 OK en caso de que la modificación del estado se realice correctamente:

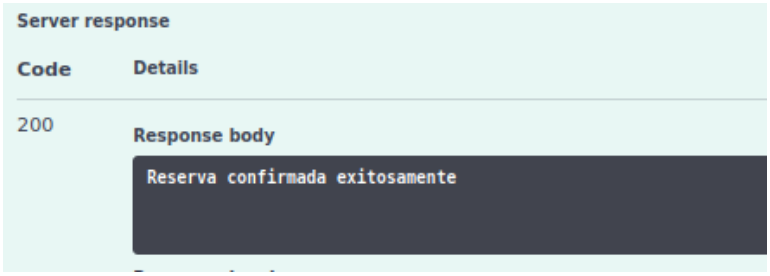

Ilustración 3-6. Respuesta exitosa del servidor a la petición PATCH para confirmar

La respuesta obtenida para cancelar la confirmación de la reserva la podemos ver en la ilustración 3-7:

Ilustración 3-7. Respuesta exitosa del servidor a la petición PATCH para cancelar la confirmación

Si queremos modificar el estado de una reserva con un identificador que no existe, el servidor nos devuelve una respuesta 404 NOT FOUND.

- **Petición DELETE a** http://localhost:8080/api/reserva: esta petición se realiza para eliminar una reserva. En el cuerpo de esta petición se envía el identificador de la reserva, ilustración 3-8:

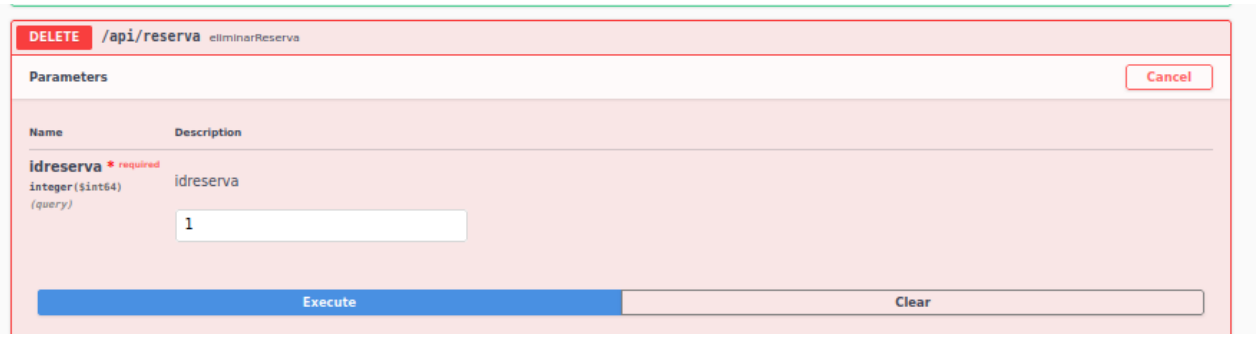

Ilustración 3-8. Petición DELETE a /api/reserva

La respuesta obtenida la podemos ver en la ilustración 3-9. Obtenemos una respuesta exitosa con el código 200 OK en caso de que la eliminación se realice correctamente:

Ilustración 3-9. Respuesta exitosa del servidor a la petición DELETE

Si queremos borrar una reserva con un identificador que no existe, el servidor nos devuelve una respuesta 404 NOT FOUND.

3.1.1.2 Clases

En este apartado se detallarán las clases que se utilizan en el servidor del centro de gestión de parkings. Están organizadas en directorios en función de su uso, se pueden distinguir por controladores, entidades, servicios y repositorios. En la ilustración 3-10 se puede observar la distribución de las mismas:

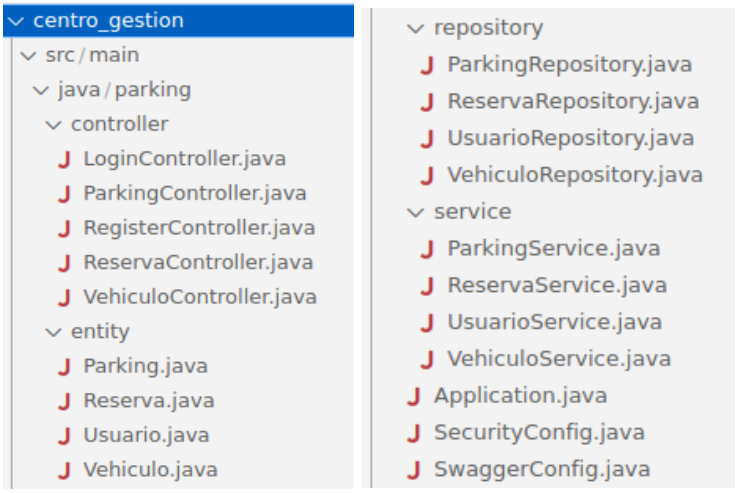

Ilustración 3-10. Clases y estructura del servidor del centro de gestión de parkings

A continuación, se van a detallar las clases principales de este servidor:

- **Application.java** (Ilustración 3-11): se corresponde con la clase principal que contine el método main.

```java
package parking;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableScheduling
public class Application {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource:Application.class, args);
    }

    //Para hacer peticiones a los parkings
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Ilustración 3-11. Application.java del servidor del centro de gestión de parkings

- **SwaggerConfig.java** (Ilustración 3-12): esta clase configura la documentación de la API utilizando Swagger. Define un bean que especifica la ubicación de los controladores de la API, el tipo de documentación a generar y la información descriptiva de la API.

```java
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket apiDocket() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage(basePackage:"parking.controller"))
                .paths(PathSelectors.any())
                .build()
                .apiInfo(getApiInfo())
                ;
    }

    private ApiInfo getApiInfo() {
        return new ApiInfo(
                title:"API centro de gestión de parkings",
                description:"Documentación API centro de gestión de parkings",
                version:"1.0",
                termsOfServiceUrl:"http://terms",
                new Contact(name:"", url:"", email:""),
                license:"LICENSE",
                licenseUrl:"LICENSE URL",
                Collections.emptyList()
                );
    }
}
```

Ilustración 3-12. SwaggerConfig.java del servidor del centro de gestión de parkings

- **SecurityConfig.java** (Ilustración 3-13): esta clase se encarga de configurar la seguridad de la aplicación utilizando Spring Security. Se define un bean llamado passwordEncoder() que utiliza el algoritmo de encriptación BCrypt para encriptar las contraseñas. Esta encriptación garantiza un mayor nivel de seguridad al almacenar las contraseñas en la base de datos, lo que ayuda a proteger la información confidencial de los usuarios. Se describirá más en profundidad en un apartado posterior.

```java
@Configuration
@EnableWebSecurity
@ComponentScan(basePackages = {"parking", "parking.controller"})
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    // Configuración de Spring Security
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/**").permitAll()
            .and()
            .csrf().disable(); // Deshabilitar CSRF para simplificar la configuración
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Ilustración 3-13. SecurityConfig.java del servidor del centro de gestión de parkings

- **Vehiculo.java** (Ilustración 3-14): esta clase representa la entidad que describe un vehículo que hace uso del centro de gestión de parkings. Contiene las etiquetas de JPA necesarias para su persistencia en la base de datos y contiene atributos como la matrícula del vehículo, la marca, el modelo, el número de plazas y una referencia al usuario asociado. Además, proporciona constructores para crear instancias de la clase con los datos necesarios, así como métodos de acceso para manipular y acceder a estos datos. Esta estructura facilita la interacción con los datos de cada vehículo y su relación con otros elementos del sistema.

```java
package parking.entity;

import javax.persistence.*;
import java.util.*;

@Entity
@Table(name = "vehiculo")
public class Vehiculo {


    @Id
    private String matricula;

    private String marca;
    private String modelo;
    private Integer plazas;

    @ManyToOne
    @JoinColumn(name = "usuarios_dni", nullable = false)
    private Usuario usuario;

    // Constructores

    public Vehiculo() {
    }

    public Vehiculo(String matricula, String marca, String modelo, Integer plazas, Usuario usuario) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.plazas = plazas;
        this.usuario = usuario;
    }

    // Getters y Setters

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public Integer getPlazas() {
        return plazas;
    }

    public void setPlazas(Integer plazas) {
        this.plazas = plazas;
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }
}
```

Ilustración 3-14. Vehiculo.java del servidor del centro de gestión de parkings

- **Usuario.java**: representa una entidad que almacena información sobre los usuarios registrados en el sistema del centro de gestión de parkings. Contiene atributos como el DNI del usuario, su nombre, apellidos, contraseña y dirección de correo electrónico. Además de los constructores requeridos para crear instancias de la clase con los datos proporcionados, la clase ofrece métodos de acceso para modificar y acceder a esta información. Esta estructura permite una gestión eficiente de los usuarios

del sistema, facilitando su registro y autenticación en el sistema de parkings. Esta clase está configurada con las etiquetas JPA necesarias para su persistencia en la base de datos. La estructura de esta clase es similar a la de la ilustración mostrada en el punto anterior, por lo que no se mostrará.

- **Reserva.java**: esta clase representa una reserva de estacionamiento realizada por un usuario para un vehículo en un parking específico dentro del centro de gestión de parkings. Contiene atributos como el identificador de la reserva, el vehículo asociado, el parking reservado, la fecha de inicio y fin de la reserva, así como el estado de confirmación y el precio de la reserva. Los constructores permiten crear instancias de la clase con los datos proporcionados, y los métodos de acceso facilitan la modificación y el acceso a esta información. Al igual que el anterior, esta estructura es similar a la mostrada en Vehículo.java y por lo tanto no se mostrará.

- **Parking.java**: se corresponde con un lugar de estacionamiento dentro del centro de gestión de parkings. Contiene atributos que describen las características del parking, como su tipo, nombre, categoría, tipo de carga, moneda de precio, tarifa por minuto, ubicación geográfica, capacidad de estacionamiento, ocupación, dimensiones máximas permitidas y tipo de estacionamiento. Estos atributos se han definido usando el modelo de datos Fiware [14] para los parkings. Al igual que los anteriores, esta estructura es similar a la mostrada en Vehículo.java y por lo tanto no se mostrará.

- **VehiculoRepository** (Ilustración 3-15): esta clase proporciona métodos para interactuar con la tabla de vehículos en la base de datos dentro del contexto del sistema del centro de gestión de parkings. Esta interfaz extiende JpaRepository de Spring Data JPA, lo que le otorga métodos predeterminados para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Define el siguiente método:
  - findByUsuarioDni (String dni): Permite buscar registros en la tabla de vehículos por el DNI del usuario al que están asociados, devolviendo una lista de vehículos correspondientes al usuario con el DNI especificado.

```java
package parking.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import parking.entity.Vehiculo;
import java.util.List;

public interface VehiculoRepository extends JpaRepository<Vehiculo, String> {

    List<Vehiculo> findByUsuarioDni(String dni);

}
```

Ilustración 3-15. VehiculoRepository.java del servidor del centro de gestión de parkings

- **UsuarioRepository.java**: esta clase es muy similar a la anterior, en este caso, representa un repositorio que interactúa con la tabla de usuarios en el contexto del centro de gestión de parkings. Esta clase, está diseñada para gestionar entidades de tipo Usuario, utilizando su DNI como clave primaria.

- **ReservaRepository** (Ilustración 3-16): define métodos para interactuar con la tabla de reservas en la base de datos en el contexto del sistema de gestión de parkings. Además de los métodos heredados, esta interfaz define tres métodos adicionales:
  - findByVehiculoMatricula (String matricula): permite buscar reservas basadas en la matrícula del vehículo.

- findByParkingId (Long parkingId): permite buscar reservas basadas en el ID del parking.
- findByVehiculoAndParkingAndFechaInicio (Vehiculo vehiculo, Parking parking, LocalDateTime fecha_inicio): permite buscar reservas basadas en el vehículo, el parking y la fecha de inicio.

```java
package parking.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import parking.entity.Parking;
import parking.entity.Reserva;
import parking.entity.Vehiculo;
import java.time.LocalDateTime;
import java.util.*;

public interface ReservaRepository extends JpaRepository<Reserva, Long> {
    List<Reserva> findByVehiculo_Matricula(String matricula);
    List<Reserva> findByParking_Id(Long parkingId);
    List<Reserva> findByVehiculoAndParkingAndFechaInicio(Vehiculo vehiculo, Parking parking, LocalDateTime fecha_inicio);
}
```

Ilustración 3-16. ReservaRepository.java del servidor del centro de gestión de parkings

- **ParkingRepository.java**: esta clase es muy similar a la anterior, en este caso, representa un repositorio que interactúa con la tabla de parkings dentro del contexto del sistema del centro de gestión de parkings. Define el siguiente método:
  - findByLatitudeAndLongitude(Float latitude, Float longitude): permite buscar un registro en la tabla de parkings basado en la latitud y longitud especificadas.

- **VehiculoService.java** (Ilustración 3-17): esta clase representa un servicio que facilita la interacción con la base de datos para gestionar los vehículos que hacen uso del centro de gestión de parkings. A través de métodos como getVehiculosByUsuarioDni, existeVehiculo y registrarVehiculo, proporciona funcionalidades para recuperar información detallada sobre los vehículos, verificar su existencia y registrar nuevos vehículos. Además, implementa la lógica necesaria para garantizar la integridad de los datos y la coherencia del sistema.

```java
@Service
public class VehiculoService {

    private final VehiculoRepository vehiculoRepository;

    @Autowired
    public VehiculoService(VehiculoRepository vehiculoRepository) {
        this.vehiculoRepository = vehiculoRepository;
    }

    public List<Vehiculo> getVehiculosByUsuarioDni(String dni) {
        return vehiculoRepository.findByUsuarioDni(dni);
    }

    public boolean existeVehiculo(String matricula, String dni) {
        Vehiculo vehiculo = vehiculoRepository.findById(matricula).orElse(null);
        return vehiculo != null && vehiculo.getUsuario().getDni().equals(dni);
    }

    public void registrarVehiculo(Vehiculo vehiculo) {
        vehiculoRepository.save(vehiculo);
    }

    public Vehiculo getByMatricula(String matricula) {
        return vehiculoRepository.findById(matricula).orElse(null);
    }


}
```

Ilustración 3-17. VehiculoService.java del servidor del centro de gestión de parkings

- **UsuarioService.java** (Ilustración 3-18): esta clase se corresponde con un servicio que facilita la

interacción con la base de datos para gestionar los usuarios que hacen uso del centro de gestión de parkings. A través de métodos como getUsuarioByDni, existeUsuario y getLoginByDni, proporciona funcionalidades para recuperar información detallada sobre los usuarios, verificar su existencia, así como para gestionar el inicio de sesión y el registro de nuevos usuarios en el sistema.

```java
@Service
public class UsuarioService {

    private final UsuarioRepository usuarioRepository;

    @Autowired
    public UsuarioService(UsuarioRepository usuarioRepository) {
        this.usuarioRepository = usuarioRepository;
    }

    public Usuario getUsuarioByDni(String dni) {
        Optional<Usuario> optionalUsuario = usuarioRepository.findById(dni);
        return optionalUsuario.orElse(null);
    }

    public boolean existeUsuario(String dni) {
        return usuarioRepository.existsById(dni);
    }


    public Usuario getLoginByDni(String dni) {
        return usuarioRepository.findById(dni).orElse(null);
    }

    public void saveLogin(Usuario usuario) {
        usuarioRepository.save(usuario);
    }
}
```

Ilustración 3-18. UsuarioService.java del servidor del centro de gestión de parkings

- **ReservaService.java** (Ilustración 3-19): esta clase se corresponde con un servicio que facilita la interacción con la base de datos para gestionar las reservas en el centro de gestión de parkings. A través de métodos como registrarReserva, obtenerReservaPorMatricula, obtenerReservaPorParkingId, obtenerReservaPorId, existeReserva y eliminarReserva, proporciona funcionalidades para registrar nuevas reservas, recuperar reservas existentes por matrícula, ID de parking o ID de reserva, verificar la existencia de una reserva para un vehículo en un determinado parking y horario, así como eliminar reservas existentes.

```java
@Service
public class ReservaService {

    private final ReservaRepository reservaRepository;

    @Autowired
    public ReservaService(ReservaRepository reservaRepository) {
        this.reservaRepository = reservaRepository;
    }

    @Transactional
    public void registrarReserva(Reserva reserva) {
        reservaRepository.save(reserva);
    }

    public List<Reserva> obtenerReservaPorMatricula(String matricula) {
        return reservaRepository.findByVehiculo_Matricula(matricula);
    }

    public List<Reserva> obtenerReservaPorParkingId(Long parkingId) {
        return reservaRepository.findByParking_Id(parkingId);
    }

    public Reserva obtenerReservaPorId(Long id) {
        return reservaRepository.findById(id).orElse(null);
    }

    public boolean existeReserva(Vehiculo vehiculo, Parking parking, LocalDateTime fecha_inicio) {
        // Buscar reservas para el vehículo, en el mismo parking y con la misma fecha de inicio
        List<Reserva> reservas = reservaRepository.findByVehiculoAndParkingAndFechaInicio(vehiculo, parking, fecha_inicio);
        return !reservas.isEmpty();
    }

    @Transactional
    public void eliminarReserva(Reserva reserva) {
        reservaRepository.delete(reserva);
    }
}
```

Ilustración 3-19. ReservaService.java del servidor del centro de gestión de parkings

- **ParkingService.java** (Ilustración 3-20): esta clase se corresponde con un servicio que facilita la interacción con la base de datos para gestionar los parkings en el centro de gestión de parkings. A través de métodos como existeParkingConLatitudYLongitud, getAllParking, getParkingById, registrarParking y saveAll, proporciona funcionalidades para verificar la existencia de un parking con una determinada latitud y longitud, recuperar todos los parkings, obtener un parking por su ID, registrar nuevos parkings y guardar todos los parkings actualizados en la base de datos.

```java
@Service
public class ParkingService {

    private final ParkingRepository parkingRepository;

    @Autowired
    public ParkingService(ParkingRepository parkingRepository) {
        this.parkingRepository = parkingRepository;
    }

    public boolean existeParkingConLatitudYLongitud(Float latitude, Float longitude) {
        return parkingRepository.findByLatitudeAndLongitude(latitude, longitude) != null;
    }

    public List<Parking> getAllParking() {
        return parkingRepository.findAll();
    }

    public Parking getParkingById(Long id) {
        return parkingRepository.findById(id).orElse(null);
    }

    public void registrarParking(Parking parking) {

        // Guardar el nuevo parking en la base de datos
        parkingRepository.save(parking);
    }
```

```java
public void saveAll(List<Parking> newParkingList) {
    // Obtener la lista de parkings existentes
    List<Parking> existingParkingList = parkingRepository.findAll();

    for (Parking newParking : newParkingList) {
        // Buscar el parking correspondiente en la lista existente
        Optional<Parking> existingParkingOptional = existingParkingList.stream()
                .filter(p -> p.getId() == newParking.getId())
                .findFirst();

        existingParkingOptional.ifPresent(existingParking -> {
            // Actualizar los parámetros deseados
            existingParking.setName(newParking.getName());
            existingParking.setChargeType(newParking.getChargeType());
            existingParking.setPriceRatePerMinute(newParking.getPriceRatePerMinute());
            existingParking.setTotalSlotNumber(newParking.getTotalSlotNumber());
            existingParking.setAvailableSlotNumber(newParking.getAvailableSlotNumber());
            existingParking.setOccupiedSlotNumber(newParking.getOccupiedSlotNumber());
            existingParking.setOccupancy(newParking.getOccupancy());

            // Guardar el parking actualizado en la base de datos
            parkingRepository.save(existingParking);
        });
    }
}
```

Ilustración 3-20. ReservaService.java del servidor del centro de gestión de parkings

- **VehiculoController.java** (Ilustración 3-21): esta clase corresponde al controlador encargado de gestionar las solicitudes relacionadas con la información de los vehículos. Contiene dos métodos principales:
  - El método getVehiculosByUsuarioDni() permite obtener una lista de vehículos asociados a un usuario específico mediante su número de identificación (DNI) a través de una solicitud GET.
  - Por otro lado, el método registrarVehiculo() permite registrar un nuevo vehículo en la base de datos a través de una solicitud POST en el endpoint "/api/vehiculos". Este método recibe parámetros como la matrícula, marca, modelo, número de plazas y DNI del propietario del vehículo. Antes de realizar el registro, verifica si el vehículo ya existe en la base de datos. Si el vehículo no existe, obtiene el usuario asociado al DNI proporcionado, crea un nuevo objeto de tipo Vehiculo y lo registra en la base de datos utilizando el servicio VehiculoService. Finalmente, devuelve un mensaje indicando el resultado de la operación.

```java
@RestController
@RequestMapping("/api/vehiculos")
public class VehiculoController {

    private final VehiculoService vehiculoService;
    private final UsuarioService usuarioService;

    @Autowired
    public VehiculoController(VehiculoService vehiculoService, UsuarioService usuarioService) {
        this.vehiculoService = vehiculoService;
        this.usuarioService = usuarioService;
    }

    @GetMapping()
    public List<Vehiculo> getVehiculosByUsuarioDni(@RequestParam String dni) {
        return vehiculoService.getVehiculosByUsuarioDni(dni);
    }

    @PostMapping()
    public ResponseEntity<String> registrarVehiculo(
        @RequestParam String matricula,
        @RequestParam String marca,
        @RequestParam String modelo,
        @RequestParam Integer plazas,
        @RequestParam String dni) {

    // Verificar si el vehículo ya existe en la base de datos
    if (vehiculoService.existeVehiculo(matricula, dni)) {
        return ResponseEntity.badRequest().body("El vehículo ya existe");
    }

    // Obtener el usuario asociado al DNI
    Usuario usuario = usuarioService.getUsuarioByDni(dni);

    // Crear un nuevo vehículo
    Vehiculo nuevoVehiculo = new Vehiculo(matricula, marca, modelo, plazas, usuario);
    // Insertar el vehículo en la base de datos
    vehiculoService.registrarVehiculo(nuevoVehiculo);

    System.out.println("Vehículo registrado correctamente con matricula: " + matricula);
    return ResponseEntity.ok("Vehículo registrado correctamente");
    }
}
```

Ilustración 3-21. VehiculoController.java del servidor del centro de gestión de parkings

- **LoginController.java** (Ilustración 3-22): esta clase gestiona las solicitudes de inicio de sesión mediante una solicitud POST. Su método login() autentica usuarios comparando el DNI y la contraseña proporcionados con los datos almacenados en la base de datos. Si la autenticación es exitosa, devuelve un mensaje de "Login exitoso" con un estado HTTP OK (200), de lo contrario, devuelve un mensaje de "Usuario o contraseña incorrectos" con un estado HTTP UNAUTHORIZED (401). Podemos destacar también el uso de passwordEncoder para la comprobación de la contraseña encriptada usando la clase SpringSecurity.java, esto se detallará en la sección 3.1.3 de esta memoria.

```java
@RestController
@RequestMapping("/api/login")
@CrossOrigin(origins = "*")
public class LoginController {

    private final UsuarioRepository usuarioRepository;
    private final PasswordEncoder passwordEncoder;

    @Autowired
    public LoginController(UsuarioRepository usuarioRepository, PasswordEncoder passwordEncoder) {
        this.usuarioRepository = usuarioRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @PostMapping()
    public ResponseEntity<String> login(@RequestParam(name="dni") String dni,
                                        @RequestParam(name="contrasena") String contrasena) {
        Optional<Usuario> optionalLogin = usuarioRepository.findById(dni);
        if (optionalLogin.isPresent()) {
            Usuario usuario = optionalLogin.get();
            if (passwordEncoder.matches(contrasena, usuario.getContrasena())) {
                return new ResponseEntity<>("Login exitoso", HttpStatus.OK);
            }
        }
        return new ResponseEntity<>("Usuario o contraseña incorrectos", HttpStatus.UNAUTHORIZED);
    }
}
```

Ilustración 3-22. LoginController.java del servidor del centro de gestión de parkings

- **RegisterController.java** (Ilustración 3-23): gestiona las solicitudes de registro de usuarios. Su método registro() verifica si el usuario ya existe en la base de datos a través del DNI mediante una solicitud POST. Si el usuario no existe, encripta la contraseña proporcionada y guarda la información del usuario en la base de datos. Devuelve un mensaje indicando el éxito del registro si se completa correctamente. Si el DNI ya está registrado, devuelve un mensaje de error indicando que el DNI ya existe. Por otro lado, se usa passwordEncoder definido en SpringSecurity.java para la encriptación de la contraseña.

```java
@RequestMapping("/api/registro")
@CrossOrigin(origins = "*")
public class RegisterController {

    private final UsuarioRepository usuarioRepository;
    private final PasswordEncoder passwordEncoder;

    @Autowired
    public RegisterController(UsuarioRepository usuarioRepository, PasswordEncoder passwordEncoder) {
        this.usuarioRepository = usuarioRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @PostMapping()
    public ResponseEntity<String> registro(
            @RequestParam(name = "dni") String dni,
            @RequestParam(name = "nombre") String nombre,
            @RequestParam(name = "apellidos") String apellidos,
            @RequestParam(name = "contrasena") String contrasena,
            @RequestParam(name = "email") String email) {

        // Verificar si el usuario ya existe en la base de datos
        if (usuarioRepository.existsById(dni)) {
            return ResponseEntity.badRequest().body("El DNI del usuario ya existe");
        }

        // Encriptar la contraseña
        String contrasenaEncriptada = passwordEncoder.encode(contrasena);

        // Insertar el usuario en la base de datos
        Usuario nuevoUsuario = new Usuario();
        nuevoUsuario.setDni(dni);
        nuevoUsuario.setNombre(nombre);
        nuevoUsuario.setApellidos(apellidos);
        nuevoUsuario.setContrasena(contrasenaEncriptada);
        nuevoUsuario.setEmail(email);

        usuarioRepository.save(nuevoUsuario);

        System.out.println("Usuario registrado correctamente con dni: " + dni);
        return ResponseEntity.ok("Usuario registrado correctamente");
    }
}
```

Ilustración 3-23. RegisterController.java del servidor del centro de gestión de parkings

- **ReservaController.java**: esta clase se corresponde con el controlador que gestiona las reservas en el centro de gestión de parkings. Contiene implementados los métodos que se detallan a continuación:
    - reservar (Ilustración 3-24): permite crear una nueva reserva de un vehículo en un parking específico, verificando previamente la existencia del vehículo y si existe una reserva para el mismo. Se implementa como restricción que las reservas son de días completos. Por otro lado, no se permite la reserva de un vehículo para la misma fecha en un parking.

```java
@PostMapping()
public ResponseEntity<String> reservar(
        @RequestParam String vehiculo_matricula,
        @RequestParam Long parking_id,
        @RequestParam String fecha_inicio,
        @RequestParam String fecha_fin,
        @RequestParam(defaultValue = "false") Boolean confirmada) {

    // Buscar el vehículo por matrícula
    Vehiculo vehiculo = vehiculoService.getByMatricula(vehiculo_matricula);

    // Verificar si el vehículo existe
    if (vehiculo == null) {
        return ResponseEntity.badRequest().body("El vehículo con matrícula " + vehiculo_matricula + " no está dado de alta en el sistema.");
    }

    // Buscar el parking por su id
    Parking parking = parkingService.getParkingById(parking_id);

    // Convertir las cadenas de fecha a objetos LocalDateTime
    LocalDateTime fechaInicio = LocalDateTime.parse(fecha_inicio + "T00:00:00");
    LocalDateTime fechaFin = LocalDateTime.parse(fecha_fin + "T23:59:59");

    // Verificar si ya existe una reserva para este vehículo y fecha de inicio en el mismo parking
    if (reservaService.existeReserva(vehiculo, parking, fechaInicio)) {
        return ResponseEntity.badRequest().body("Ya existe una reserva para esta fecha y este vehículo en el mismo parking.");
    }

    //Calcular el precio de la reserva
    double precioTotal = calcularPrecioReserva(parking_id, fecha_inicio, fecha_fin);

    // Crear una nueva reserva
    Reserva nuevaReserva = new Reserva(vehiculo, parking, fechaInicio, fechaFin, confirmada, precioTotal);

    // Registrar la reserva en el servicio
    reservaService.registrarReserva(nuevaReserva);

    System.out.println("Se ha creado una reserva para la matricula: " + vehiculo_matricula + " en el parking con id: " + parking_id);
    return ResponseEntity.ok("Reserva creada exitosamente");
}
```

Ilustración 3-24. Método reservar de ReservaController.java

- obtenerInformacionReservaPorParking (Ilustración 3-25): se obtiene la información de las reservas de un parking.

```java
@GetMapping("/{parkingId}")
public ResponseEntity<List<Map<String, Object>>> obtenerInformacionReservaPorParking(@PathVariable Long parkingId) {
    List<Reserva> reservas = reservaService.obtenerReservaPorParkingId(parkingId);

    // Obtener la fecha actual sin tener en cuenta la hora
    LocalDate fechaActual = LocalDate.now();

    if (!reservas.isEmpty()) {
        List<Map<String, Object>> responseList = new ArrayList<>();

        for (Reserva reserva : reservas) {
            // Obtener la fecha de fin de la reserva sin tener en cuenta la hora
            LocalDate fechaFinReserva = reserva.getFechaFin().toLocalDate();

            // Comparar solo el día, el mes y el año de la fecha de fin de la reserva con la fecha actual
            if (!fechaFinReserva.isBefore(fechaActual)) {
                Map<String, Object> response = new HashMap<>();
                response.put("reserva_id", reserva.getReservaId());
                response.put("matricula", reserva.getVehiculo().getMatricula());
                response.put("fecha_inicio", reserva.getFechaInicio());
                response.put("fecha_fin", reserva.getFechaFin());
                response.put("confirmada", reserva.getConfirmada());
                response.put("precio_reserva", reserva.getPrecio_reserva());

                responseList.add(response);
            }
        }

        return ResponseEntity.ok(responseList);
    } else {
        // No se encontró reserva para el parking dado
        return ResponseEntity.notFound().build();
    }
}
```

Ilustración 3-25. Método obtenerInformacionReservaPorParking de ReservaController.java

- obtenerInformacionReserva: este método es similar al anterior, en este caso, se obtiene la información de las reservas de un vehículo.

- confirmar (Ilustración 3-26): permite confirmar o cancelar la confirmación de una reserva existente.

```
@PatchMapping("/confirmacion/{reserva_id}")
public ResponseEntity<String> confirmar(
        @PathVariable Long reserva_id,
        @RequestBody Map<String, Boolean> requestBody) {

    // Obtener el valor de confirmada del cuerpo de la solicitud
    Boolean confirmada = requestBody.get("confirmada");

    // Buscar la reserva por su ID
    Reserva reserva = reservaService.obtenerReservaPorId(reserva_id);

    // Verificar si se encontró la reserva
    if (reserva != null) {
        if (confirmada == true){
            // Establecer el valor de confirmada true
            reserva.setConfirmada(confirmada);

            // Guardar la reserva actualizada en la base de datos
            reservaService.registrarReserva(reserva);

            System.out.println("Se ha confirmado la reserva con id: " + reserva_id);
            return ResponseEntity.ok("Reserva confirmada exitosamente");
        }else{
            // Establecer el valor de confirmada false
            reserva.setConfirmada(confirmada);

            // Guardar la reserva actualizada en la base de datos
            reservaService.registrarReserva(reserva);

            System.out.println("Se ha cancelado la confirmación de la reserva con id: " + reserva_id);
            return ResponseEntity.ok("Confirmación de reserva cancelada exitosamente");

        }
    } else {
        // No se encontró la reserva con el ID dado
        return ResponseEntity.notFound().build();
    }
}
```

Ilustración 3-26. Método confirmar de ReservaController.java

- eliminarReserva (Ilustración 3-27): permite eliminar una reserva existente.

```
@DeleteMapping()
public ResponseEntity<String> eliminarReserva(@RequestParam("idreserva") Long reserva_id) {
    // Buscar la reserva por su ID
    Reserva reserva = reservaService.obtenerReservaPorId(reserva_id);

    // Verificar si se encontró la reserva
    if (reserva != null) {
        // Eliminar la reserva
        reservaService.eliminarReserva(reserva);

        System.out.println("Se ha eliminado la reserva con id: " + reserva_id);
        return ResponseEntity.ok("Reserva eliminada exitosamente");
    } else {
        // No se encontró la reserva con el ID dado
        return ResponseEntity.notFound().build();
    }
}
```

Ilustración 3-27. Método eliminarReserva de ReservaController.java

- **ParkingController.java**: esta clase gestiona las solicitudes de los parkings. Estos datos son manejados siguiendo el modelo de datos Fiware para los parkings. Presenta los siguientes métodos:

  - getAvailableParking: obtiene una lista con los datos de los parkings que tiene asociados el centro de gestión. Estos datos siguen la estructura y el formato definidos en el modelo de datos Fiware [14] para los parkings.

- updateParkingData: este método está marcado con @Scheduled para ejecutarse cada 2 minutos. Su función se corresponde con recopilar la información de los parkings para actualizarla en el centro de gestión de parkings mediante una petición a cada uno de ellos.
- crearNuevoParking: este método permite dar de alta un nuevo parking en el centro de gestión verificando que no exista otro en la ubicación que se ha indicado.

3.1.1.3 Configuración del servidor

Para acceder y utilizar la aplicación, se requiere establecer la dirección y el puerto en los cuales el servidor estará activo. En este proyecto, el servidor se ha configurado para escuchar en la dirección IP y el puerto específico:

- Dirección IP del servidor: localhost (127.0.0.1).
- Puerto del servidor: 8080 (indicado en el archivo application.properties).

Con esta configuración, la aplicación se vuelve accesible a través de la dirección http://localhost:8080.

3.1.1.4 Dependencias

Las dependencias en el servicio son elementos de vital importancia que permiten que el proyecto funcione correctamente. Estas dependencias han sido incluidas en el archivo POM (Project Object Model) del proyecto, las principales son las siguientes:

- **PostgreSQL**: se ha incluido esta dependencia para facilitar la integración con la base de datos PostgreSQL, permitiendo así el acceso y la manipulación de datos de manera eficiente.
- **Spring Data JPA**: esta herramienta simplifica el manejo de datos en aplicaciones Java utilizando el framework Spring. Proporciona una capa de abstracción sobre la base de datos, lo que facilita la escritura de consultas y la gestión de la persistencia de los objetos. Su objetivo es simplificar al desarrollador la persistencia de datos entre los distintos repositorios de información.
- **Swagger**: se ha integrado Swagger para generar documentación interactiva de las API REST del proyecto. Esta herramienta simplifica la creación y el mantenimiento de la documentación, lo que mejora la comprensión y el uso de las APIs por parte de los desarrolladores y usuarios.
- **Spring Security**: se ha incluido la dependencia de Spring Security para mejorar la seguridad de la aplicación, en concreto, enfocada a las contraseñas que se usan en la misma.

A continuación, se muestra un fragmento del archivo POM en el que se pueden ver reflejadas estas dependencias:

```
<!-- PostgreSQL -->
 <dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
 </dependency>
<!-- Spring Data JPA -->
 <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
 </dependency>
<!-- Swagger -->
 <dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
```

```
  <version>3.0.0</version>
 </dependency>
<!—Spring Security -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
 </dependency>
```

3.1.2 Base de datos del centro de gestión de parkings

En este apartado se detallará la implementación de la base de datos del centro de gestión de parkings. Esta base de datos juega un papel fundamental en la conservación de la información del sistema como pueden ser los datos de los usuarios, los vehículos y los distintos parkings. Mediante un marco robusto, se consigue retener y gestionar información pertinente, lo que posibilita una gestión y análisis eficaz por parte del centro de gestión.

La siguiente ilustración 3-28 muestra el diagrama entidad relación de la base de datos del centro de gestión de parkings:

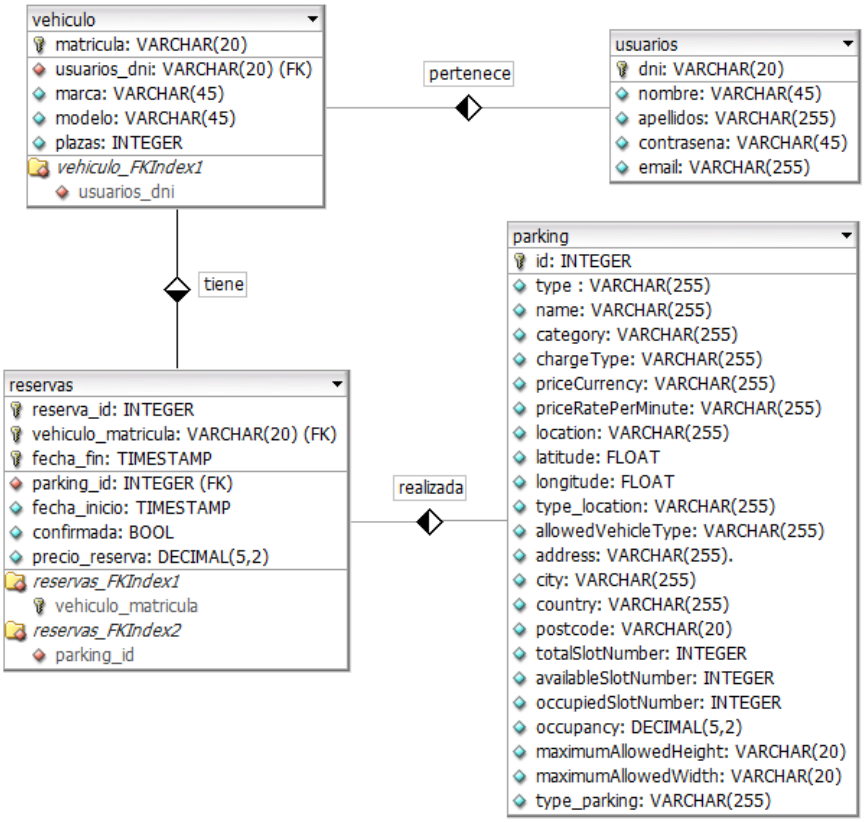

Ilustración 3-28. Diagrama entidad relación BBDD centro de gestión de parkings

3.1.2.1 Tabla usuarios

La tabla “usuarios”, se utiliza para almacenar información sobre los usuarios dados de alta en el sistema del centro de gestión. Cada fila en la tabla representa los datos de un usuario y contiene detalles como el DNI, el nombre y los apellidos, la contraseña asociada a su cuenta y el email. El propósito de esta tabla es mantener un registro de los usuarios que hacen uso del centro de gestión. El código SQL es el mostrado a continuación:

```
DROP TABLE IF EXISTS usuarios;
CREATE TABLE usuarios (
  dni VARCHAR(20) NOT NULL,
  nombre VARCHAR(45) NULL,
  apellidos VARCHAR(255) NULL,
  contrasena VARCHAR(255) NULL,
  email VARCHAR(255) NULL,
  PRIMARY KEY(dni)
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO usuarios (dni, nombre, apellidos, contrasena, email)
VALUES ('12345678A', 'Pepito', 'Apellido1 Apellido2', 'micontrasena123', 'pepito@example.com');
```

3.1.2.2 Tabla parking

La tabla "parking", se utiliza para almacenar información detallada sobre cada parking asociado a un centro de gestión. Cada fila en la tabla contiene detalles como el ID del estacionamiento, el tipo de estacionamiento, el nombre, la categoría, el tipo de tarifa de cobro, la moneda de la tarifa, el precio por minuto, la ubicación geográfica (latitud y longitud), el tipo de ubicación, el tipo de vehículo permitido, la dirección (incluyendo ciudad, país y código postal), el número total de plazas de estacionamiento, el número de plazas disponibles, el número de plazas ocupadas, la tasa de ocupación actual, la altura máxima permitida, el ancho máximo permitido y el tipo de estacionamiento. El propósito de esta tabla es mantener un registro detallado de los datos de cada estacionamiento asociado a un centro de gestión. Esto permite una gestión eficaz y proporciona información útil. El código SQL es el siguiente:

```
DROP TABLE IF EXISTS parking;
CREATE TABLE parking (
  id INTEGER NOT NULL,
  type VARCHAR(255) NULL,
  name VARCHAR(255) NULL,
  category VARCHAR(255) NULL,
  chargeType VARCHAR(255) NULL,
  priceCurrency VARCHAR(255) NULL,
  priceRatePerMinute DECIMAL(5, 2) NULL,
  latitude FLOAT NULL,
  longitude FLOAT NULL,
  type_location VARCHAR(255) NULL,
  allowedVehicleType VARCHAR(255) NULL,
  address VARCHAR(255) NULL,
  city VARCHAR(255) NULL,
  country VARCHAR(255) NULL,
  postcode VARCHAR(20) NULL,
  totalSlotNumber INTEGER NULL,
  availableSlotNumber INTEGER NULL,
  occupiedSlotNumber INTEGER NULL,
  occupancy DECIMAL(5, 2) DEFAULT NULL,
  maximumAllowedHeight VARCHAR(20) NULL,
  maximumAllowedWidth VARCHAR(20) NULL,
  type_parking VARCHAR(255) NULL,
  PRIMARY KEY(id)
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO info_parking (id, type, name, category, chargeType, priceCurrency,
```

```
priceRatePerMinute, latitude, longitude, type_location, allowedVehicleType, address, city, country,
postcode, totalSlotNumber, availableSlotNumber, occupiedSlotNumber, occupancy,
maximumAllowedHeight, maximumAllowedWidth, type_parking)
VALUES (3, 'Subterraneo', 'Parking Nervion', 'Public', 'Hourly', 'EUR', 0.14, 37.385345, -5.971967,
'Point', 'Car', 'Avd. Luis Montoto', 'Sevilla', 'Sevilla', '41009', 80, 0, 80, 1.00, '2 meters', '2.5 meters',
'OffStreetParking');
```

3.1.2.3 Tabla vehiculo

La tabla "vehiculo", se emplea para almacenar información sobre los vehículos registrados en el sistema del centro de gestión. Cada fila en la tabla representa los datos de un vehículo detallando la matrícula del vehículo, el DNI del usuario propietario, la marca y el modelo del vehículo, así como el número de plazas disponibles. El propósito de esta tabla es mantener un registro de los vehículos asociados a los usuarios que utilizan el centro de gestión.

Además, la tabla incluye una clave externa que referencia la columna "usuarios_dni", la cual está vinculada a la clave primaria "dni" en la tabla "usuarios" explicada anteriormente. Esta relación establece una restricción referencial entre ambas tablas, asegurando que cada entrada en la tabla "vehiculo" esté asociada a un usuario existente en la tabla "usuarios". Esta restricción se configura con la opción "ON DELETE CASCADE", lo que significa que, si se elimina un usuario de la tabla "usuarios", también se eliminarán automáticamente todos los registros asociados en la tabla "vehiculo" para ese usuario. El código SQL es el mostrado a continuación:

```
DROP TABLE IF EXISTS vehiculo;
CREATE TABLE vehiculo (
  matricula VARCHAR(20) NOT NULL,
  usuarios_dni VARCHAR(20) NOT NULL,
  marca VARCHAR(45) NULL,
  modelo VARCHAR(45) NULL,
  plazas INTEGER NULL,
  PRIMARY KEY(matricula),
  FOREIGN KEY(usuarios_dni)
    REFERENCES usuarios(dni)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO vehiculo (matricula, usuarios_dni, marca, modelo, plazas)
VALUES ('1234ABC', '12345678A', 'Toyota', 'Yaris', 5);
```

3.1.2.4 Tabla reservas

La tabla "reservas", se utiliza para almacenar información sobre las reservas realizadas en el sistema del centro de gestión. Cada fila en la tabla representa una reserva y contiene detalles como el ID de la reserva, la matrícula del vehículo asociado, el ID del parking donde se realiza la reserva, las fechas de inicio y fin de la reserva, el estado de confirmación y el precio de la reserva.

Esta tabla, incluye una clave externa que referencia la columna "vehiculo_matricula", la cual está vinculada a la clave primaria "matricula" en la tabla "vehiculo" explicada anteriormente. Esta relación establece una restricción referencial entre ambas tablas, asegurando que cada entrada en la tabla "reservas" esté asociada a un vehículo existente en la tabla "vehiculo". Por otro lado, incluye una clave externa que referencia la columna "parking_id", la cual está vinculada a la clave primaria "id" en la tabla "parking" también, explicada anteriormente. El código SQL es el mostrado a continuación:

```
DROP TABLE IF EXISTS reservas;
```

```
CREATE TABLE reservas (
   reserva_id SERIAL,
   vehiculo_matricula VARCHAR(20) NOT NULL,
   parking_id INTEGER NOT NULL,
   fecha_inicio TIMESTAMP NULL,
   fecha_fin TIMESTAMP NULL,
   confirmada BOOL NULL,
   precio_reserva DECIMAL(5, 2) NULL,
   PRIMARY KEY(reserva_id, vehiculo_matricula, fecha_inicio),
   FOREIGN KEY (vehiculo_matricula)
      REFERENCES vehiculo(matricula)
         ON DELETE CASCADE
         ON UPDATE CASCADE,
   FOREIGN KEY (parking_id)
      REFERENCES parking(id)
         ON DELETE CASCADE
         ON UPDATE CASCADE
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO reservas (vehiculo_matricula, parking_id, fecha_inicio, fecha_fin, confirmada,
precio_reserva)
VALUES ('1234ABC', 3, '2024-04-17 10:00:00', '2024-04-17 22:00:00', true, 10.50);
```

3.1.3 Spring Security

La seguridad de los datos y los recursos es esencial en el diseño de cualquier aplicación. Este apartado se centra en la implementación de la seguridad utilizando Spring Security en el servidor REST del centro de gestión de parkings. Se detallará el proceso utilizado para garantizar que las contraseñas de los usuarios estén almacenadas de manera segura en la base de datos mediante la encriptación. A través de esta medida de seguridad, se fortalece la integridad de la información y se promueve la confianza de los usuarios en la aplicación.

La dependencia añadida está definida en el apartado 3.1.1.4 de esta memoria, para ello se crea la clase SecurityConfig que se va a explicar a continuación:

- Se configura un bean llamado PasswordEncoder que utiliza un algoritmo para cifrar contraseñas de manera segura.
- En la configuración de Spring Security el filtro permitirá el acceso a todas las URL sin necesidad de autenticación ya que en la aplicación móvil se requerirá previamente para acceder a todas las funcionalidades.

```
@Configuration
@EnableWebSecurity
@ComponentScan(basePackages = {"parking", "parking.controller"})
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   // Configuración de Spring Security
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests()
         .antMatchers("/**").permitAll() // Permitir acceso sin autenticación a todas las URL
         .and()
         .csrf().disable(); // Deshabilitar CSRF para simplificar la configuración
   }
```

```
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

3.2 Servicio REST de gestión de parking

Esta sección al igual que la anterior, constituye otro de los elementos fundamentales de este proyecto. Se examinarán detalladamente las características esenciales del servicio desarrollado para la gestión de un parking concreto. Aquí se presentará un análisis exhaustivo tanto de la estructura integral de este servicio como de su interacción con la base de datos subyacente. Asimismo, se abordará la documentación de los endpoints de este servicio a través de la herramienta Swagger. Este apartado proporcionará una visión detallada de cómo funciona el servicio REST de gestión de parking, resaltando su importancia en el contexto general de la aplicación al permitir la supervisión y el control efectivo del mismo.

3.2.1 Servidor REST de gestión de parking

Este subapartado se centra en explicar con detalle la estructura y configuración del servidor correspondiente con un parking concreto. Se incluyen las dependencias en este servicio, la definición de endpoints y las clases de las que está compuesto. Al igual que el apartado anterior, para el manejo de los datos del parking se ha empleado el modelo de datos Fiware [14], en concreto, se usará el enfocado a las ciudades inteligentes y los parkings.

3.2.1.1 Definición de endpoints

En esta sección, se detallará un aspecto fundamental para el desarrollo del proyecto el cual consiste en detallar y explicar los endpoints de la API REST de gestión de parking implementada. Los endpoints se corresponden con los puntos de entrada a través de los cuales las aplicaciones se conectan con el servidor. Cada endpoint representa una acción particular que la API puede ejecutar, lo cual nos facilita el intercambio de datos y la comunicación de manera organizada y estructurada.

Los diferentes endpoints del servicio REST de la gestión de un determinado parking pueden verse en la tabla 3-2 mostrada a continuación:

PATH	Método HTTP	Descripción
/api/parking	GET	Permite obtener una lista con los datos del parking almacenados en la base de datos.
	POST	Permite modificar los datos del parking en la base de datos.
/api/parking/entrada	POST	Permite registrar una nueva entrada de un vehículo al parking.
/api/parking/salida	POST	Permite registrar una salida de un vehículo al parking.

/api/parking/pago	POST	Permite realizar el pago a los clientes tras indicar la salida del parking.

Tabla 3–2. Endpoints del servicio de gestión de parking

A continuación, se explicará cómo se utiliza Swagger para documentar el servicio de gestión de parking. La siguiente ilustración 3-29 nos muestra la vista desde la interfaz web de Swagger que nos permite visualizar los endpoints de un modo más gráfico.

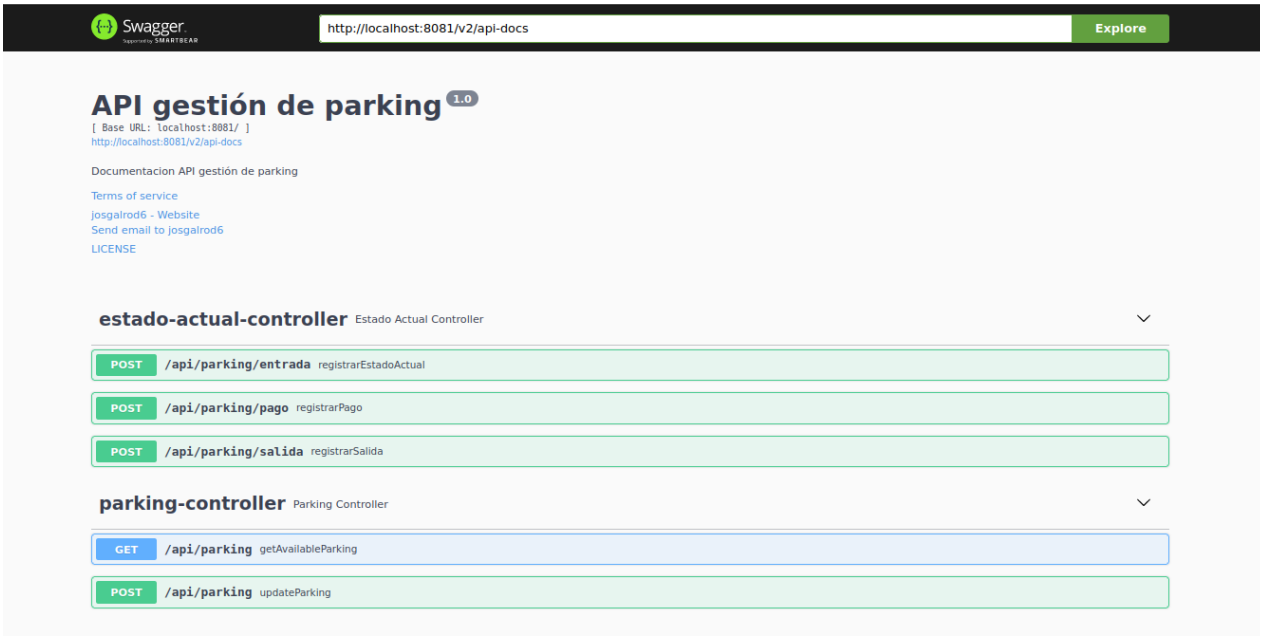

Ilustración 3-29. Endpoints en Swagger de gestión de parking

Algunos ejemplos de peticiones al servidor de la gestión de parking son los siguientes:

- **Petición GET a http://localhost:8081/api/parking**: esta petición se realiza para conocer los datos del parking a tiempo real y devuelve algunos datos como plazas ocupadas, precio por minuto, total de plazas, etc. En la ilustración 3-30 se puede ver un ejemplo de esta petición:

Ilustración 3-30. Respuesta petición GET a /api/parking

- **Petición POST a http://localhost:8081/api/parking/entrada**: esta petición ocurre cuando un vehículo quiere entrar en el parking. En el cuerpo de esta petición se envía la matrícula del vehículo que realiza la entrada al parking, ilustración 3-31:

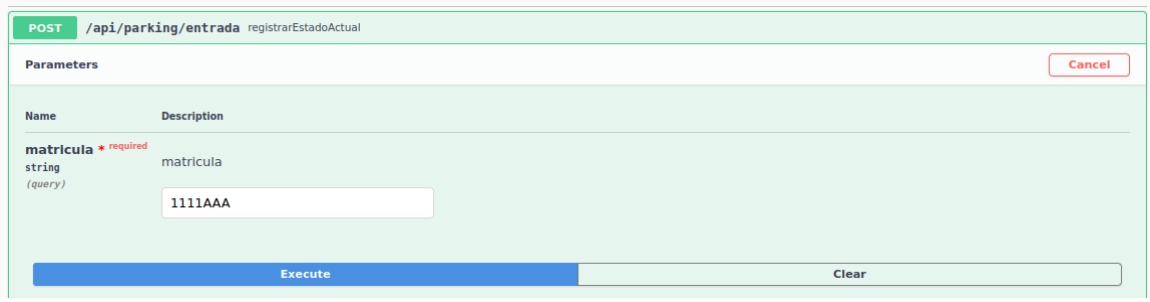

Ilustración 3-31. Petición POST a /api/parking/entrada

La respuesta obtenida la podemos ver en la ilustración 3-32. Obtenemos una respuesta exitosa con el código 200 OK en caso de que la entrada se realice correctamente:

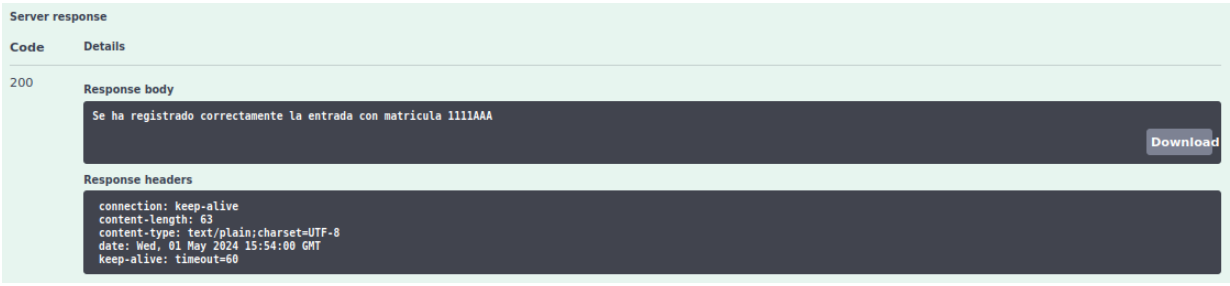

Ilustración 3-32. Respuesta exitosa del servidor a la petición POST

Caso contrario sería obtener una respuesta no exitosa a la entrada del vehículo al parking. En este caso el servidor devuelve 400 BAD REQUEST tal y como podemos observar en la siguiente ilustración 3-33:

Ilustración 3-33. Respuesta no exitosa del servidor a la petición POST

3.2.1.2 Clases

En este apartado se detallarán las clases que se utilizan en el servidor de gestión de parking. Están organizados en directorios en función de su uso, se pueden distinguir por controladores, entidades, servicios y repositorios. En la ilustración 3-34 se puede ver cómo están distribuidas las mismas:

Ilustración 3-34. Clases y estructura del servidor de gestión de parking

A continuación, se van a detallar las clases principales de este servidor:

- **Application.java** (Ilustración 3-35): esta clase es la clase principal que contine el método main.

```java
package parking1;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource:Application.class, args);
    }
}
```

Ilustración 3-35. Application.java del servidor de gestión de parking

- **SwaggerConfig.java** (Ilustración 3-36): esta clase configura la documentación de la API utilizando Swagger. Define un bean que especifica la ubicación de los controladores de la API, el tipo de documentación a generar y la información descriptiva de la API.

```java
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket apiDocket() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage(basePackage:"parking1.controller"))
                .paths(PathSelectors.any())
                .build()
                .apiInfo(getApiInfo())
                ;
    }

    private ApiInfo getApiInfo() {
        return new ApiInfo(
                title:"API gestión de parking",
                description:"Documentacion API gestión de parking",
                version:"1.0",
                termsOfServiceUrl:"",
                new Contact(name:"", url:"", email:""),
                license:"LICENSE",
                licenseUrl:"LICENSE URL",
                Collections.emptyList()
                );
    }
}
```

Ilustración 3-36. SwaggerConfig.java del servidor de gestión de parking

- **EstadoActual.java** (Ilustración 3-37): esta clase se corresponde con una entidad y representa el estado actual de un vehículo estacionado en un sistema de gestión de parking. Contiene atributos que describen la matrícula del vehículo, la fecha y hora de entrada y salida, el tiempo total de estacionamiento, el precio total a pagar, así como el estado del pago. Está anotada con las etiquetas de JPA necesarias para su persistencia en la base de datos, y proporciona constructores y métodos de acceso para manipular los datos asociados con cada instancia de la clase.

```java
package parking1.entity;

import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "estado_actual")
public class EstadoActual {

    @Id
    private String matricula;
    @Column(name = "fechayhora_entrada")
    private LocalDateTime fechaHoraEntrada;
    @Column(name = "fechayhora_salida")
    private LocalDateTime fechaHoraSalida;
    @Column(name = "total_minutos")
    private double totalMinutos;
    private Float precio_total;
    private Boolean pagada;
    private String id_pago;

    // Constructores

    public EstadoActual() {
    }

    public EstadoActual(String matricula, LocalDateTime fechaHoraEntrada, LocalDateTime fechaHoraSalida,
    double totalMinutos, Float precio_total, Boolean pagada, String id_pago) {
        this.matricula = matricula;
        this.fechaHoraEntrada = fechaHoraEntrada;
        this.fechaHoraSalida = fechaHoraSalida;
        this.totalMinutos = totalMinutos;
        this.precio_total = precio_total;
        this.pagada = pagada;
        this.id_pago = id_pago;
    }

    // Getters y Setters

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public LocalDateTime getFechaHoraEntrada() {
        return fechaHoraEntrada;
    }

    public void setFechaHoraEntrada(LocalDateTime fechaHoraEntrada) {
        this.fechaHoraEntrada = fechaHoraEntrada;
    }

    public LocalDateTime getFechaHoraSalida() {
        return fechaHoraSalida;
    }

    public void setFechaHoraSalida(LocalDateTime fechaHoraSalida) {
        this.fechaHoraSalida = fechaHoraSalida;
    }

    public double getTotalMinutos() {
        return totalMinutos;
    }

    public void setTotalMinutos(double totalMinutos) {
        this.totalMinutos = totalMinutos;
    }

    public void setPrecio_total(Float precio_total) {
        this.precio_total = precio_total;
    }
```

Ilustración 3-37. EstadoActual.java del servidor de gestión de parking

- **Parking.java**: esta clase se corresponde con una entidad y representa el parking que es controlado. Sus atributos incluyen detalles como el tipo de estacionamiento, nombre, categoría, tipo de cargo, moneda de precio, tarifa por minuto, coordenadas de ubicación, tipo de ubicación, tipos de vehículos permitidos, dirección, ciudad, país, código postal, número total de espacios, número de espacios

disponibles, número de espacios ocupados, ocupación, altura máxima permitida, ancho máximo permitido y tipo de estacionamiento. La estructura de esta clase es similar a la de la ilustración mostrada en el punto anterior, por lo que no se mostrará.

- **EstadoActualRepository.java** (Ilustración 3-38): esta clase define métodos para interactuar con la tabla correspondiente en la base de datos en el contexto del sistema de gestión de parking. Extiende la interfaz JpaRepository de Spring Data JPA, lo que le proporciona métodos predeterminados para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Además de los métodos heredados, como guardar, eliminar y buscar por ID, esta interfaz define un método adicional:
  - findByMatricula(String matricula): Permite buscar un registro en la tabla de estado actual por su matrícula, devolviendo el estado actual correspondiente a esa matrícula.

```
package parking1.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import parking1.entity.EstadoActual;

public interface EstadoActualRepository extends JpaRepository<EstadoActual, String> {
    EstadoActual findByMatricula(String matricula);

}
```

Ilustración 3-38. EstadoActualRepository.java del servidor de gestión de parking

- **ParkingRepository.java**: esta clase es muy similar a la anterior, en este caso, representa un repositorio que interactúa con la tabla correspondiente en la base de datos dentro del contexto del sistema de gestión de parking.

- **EstadoActualService.java** (Ilustración 3-39): esta clase se corresponde con un servicio que facilita la interacción con la base de datos para gestionar el estado actual de los vehículos estacionados. A través de métodos como getEstadoActualByMatricula, registrarEstadoActual y actualizarPago, proporciona funcionalidades para recuperar información detallada sobre los vehículos, registrar su estado actual y procesar pagos. Además, implementa lógica para ajustar la disponibilidad de espacios de estacionamiento y calcular tarifas en función del tiempo de estacionamiento y las tarifas definidas en la configuración. La siguiente ilustración muestra las funciones que contiene esta clase.

```java
public EstadoActual getEstadoActualByMatricula(String matricula) {
    return estadoActualRepository.findByMatricula(matricula);
}

public void registrarEstadoActual(EstadoActual estadoActual) {
    // Obtener el estacionamiento por su ID
    Parking parking = parkingRepository.findById(ID_PARKING).orElse(null);

    if (parking != null) {
        // Incrementar occupiedSlotNumber y decrementar availableSlotNumber
        int occupiedSlots = parking.getOccupiedSlotNumber() + 1;
        int availableSlots = parking.getAvailableSlotNumber() - 1;
        int totalSlots = parking.getTotalSlotNumber();
        parking.setOccupiedSlotNumber(occupiedSlots);
        parking.setAvailableSlotNumber(availableSlots);

        float occupancy;
        if (totalSlots > 0) {
            occupancy = ((float) occupiedSlots / totalSlots);
        } else {
            occupancy = 0.0f;
        }
        parking.setOccupancy(occupancy);
        // Guardar los cambios en el estacionamiento
        parkingRepository.save(parking);

        // Guardar el estado actual en la tabla estado_actual
        estadoActualRepository.save(estadoActual);
    }
}

public void actualizarEstadoActual(EstadoActual estadoActual) {
    // Obtener el estacionamiento por su ID
    Parking parking = parkingRepository.findById(ID_PARKING).orElse(null);

    if (parking != null) {
        // Guardar el estado actual en la tabla estado_actual
        estadoActualRepository.save(estadoActual);
    }
}
```

```java
public void actualizarPago(EstadoActual estadoActual) {
    // Obtener el estacionamiento por su ID
    Parking parking = parkingRepository.findById(ID_PARKING).orElse(null);

    if (parking != null) {

        // Incrementar occupiedSlotNumber y decrementar availableSlotNumber
        int occupiedSlots = parking.getOccupiedSlotNumber() - 1;
        int availableSlots = parking.getAvailableSlotNumber() + 1;
        int totalSlots = parking.getTotalSlotNumber();
        parking.setOccupiedSlotNumber(occupiedSlots);
        parking.setAvailableSlotNumber(availableSlots);

        float occupancy;
        if (totalSlots > 0) {
            //occupancy = ((float) occupiedSlots / totalSlots) * 100;
            occupancy = ((float) occupiedSlots / totalSlots);
        } else {
            occupancy = 0.0f;
        }
        parking.setOccupancy(occupancy);
        // Guardar los cambios en el estacionamiento
        parkingRepository.save(parking);

        // Guardar el pago
        estadoActualRepository.save(estadoActual);
    }
}

public float calcularPrecio(float minutosEstacionado) {
    // Obtener el precio del parking
    Parking parking = parkingRepository.findById(ID_PARKING).orElse(null);
    Float precioParking = parking.getPriceRatePerMinute();

    Float total = minutosEstacionado * precioParking;
    // Calcular el precio total en base a los minutos estacionados y el precio del parking
    return total;
}
```

Ilustración 3-39. EstadoActualService.java del servidor de gestión de parking

- **ParkingService.java** (Ilustración 3-40): esta clase se corresponde con un servicio que facilita operaciones como información detallada sobre el parking y actualizar dinámicamente la información del mismo. Destaca el método updateParking, que permite no solo actualizar los detalles del estacionamiento, como el número de espacios disponibles y la tarifa por minuto, sino también calcular automáticamente la ocupación y los espacios disponibles.

```java
@Service
public class ParkingService {

    private final ParkingRepository parkingRepository;

    @Autowired
    public ParkingService(ParkingRepository parkingRepository) {
        this.parkingRepository = parkingRepository;
    }

    public List<Parking> getAllParking() {
        return parkingRepository.findAll();
    }

    public Parking getParkingById(Long id) {
        return parkingRepository.findById(id).orElse(null);
    }

    public void updateParking(Parking parking, int totalSlotNumber, int occupiedSlotNumber, float priceRatePerMinute, String chargeType) {
        // Actualizar los campos del parking
        parking.setTotalSlotNumber(totalSlotNumber);
        parking.setOccupiedSlotNumber(occupiedSlotNumber);
        parking.setPriceRatePerMinute(priceRatePerMinute);
        parking.setChargeType(chargeType);

        // Calcular el valor de ocupación
        //float occupancy = (float) occupiedSlotNumber / totalSlotNumber * 100;
        float occupancy = ((float) occupiedSlotNumber / totalSlotNumber);
        parking.setOccupancy(occupancy);

        // Calcular el número de espacios disponibles
        int availableSlotNumber = totalSlotNumber - occupiedSlotNumber;
        parking.setAvailableSlotNumber(availableSlotNumber);

        // Guardar los cambios en la base de datos
        parkingRepository.save(parking);
    }
}
```

Ilustración 3-40. ParkingService.java del servidor de gestión de parking

- **EstadoActualController.java**: esta clase se corresponde con el controlador y gestiona las solicitudes relacionadas con el estado actual de los vehículos en el sistema de gestión de parking. Este controlador proporciona endpoints para registrar la entrada y salida de vehículos, así como para procesar pagos. Se presentan tres métodos entre los que se distinguen:
    - registrarEstadoActual (Ilustración 3-41): se verifica si el vehículo ya está en el estacionamiento y si hay espacios disponibles antes de registrar su entrada. Se comprueba antes de entrar, que el vehículo con esa matrícula no esté en el interior.

```
@PostMapping("/entrada")
    public ResponseEntity<String> registrarEstadoActual(@RequestParam String matricula) {
        // Verificar si ya existe un estado actual para la matrícula
        EstadoActual estadoActualExistente = estadoActualService.getEstadoActualByMatricula(matricula);

        if (estadoActualExistente != null && estadoActualExistente.getFechaHoraSalida() == null && estadoActualExistente.getPagada() != true) {
            System.out.println("El vehículo con matrícula " + matricula + " ya está en el parking.");
            return ResponseEntity.badRequest().body("El vehículo con matrícula " + matricula + " ya está en el parking.");
        }

        // Verificar si hay espacios disponibles en el parking
        Parking parking = parkingService.getParkingById(ID_PARKING);
        if (parking.getAvailableSlotNumber() <= 0) {
            // No hay espacios disponibles en el parking
            System.out.println("El parking está lleno. No se puede permitir la entrada del vehículo con matrícula " + matricula);
            return ResponseEntity.badRequest().body("El parking está lleno. No se puede permitir la entrada del vehículo con matrícula " + matricula);
        }

        // Obtener la fecha y hora actual
        LocalDateTime fechaHoraEntrada = LocalDateTime.now();

        // Crear un nuevo estado actual
        EstadoActual nuevoEstadoActual = new EstadoActual();
        nuevoEstadoActual.setMatricula(matricula);
        nuevoEstadoActual.setFechaHoraEntrada(fechaHoraEntrada);
        nuevoEstadoActual.setPagada(false);

        // Insertar el estado actual en la base de datos
        estadoActualService.registrarEstadoActual(nuevoEstadoActual);

        System.out.println("Ha entrado un vehículo con matrícula: " + matricula);
        return ResponseEntity.ok("Se ha registrado correctamente la entrada con matricula " + matricula);
    }
```

Ilustración 3-41. Método registrarEstadoActual de EstadoActualController.java servidor de gestión de parking

- registrarSalida (Ilustración 3-42): se verifica si el vehículo está en el parking antes de registrar su salida y calcular el precio total del estacionamiento.

```
    @PostMapping("/salida")
    public ResponseEntity<String> registrarSalida(@RequestParam String matricula) {
        // Verificar si el vehículo está registrado en el estado actual
        EstadoActual estadoActual = estadoActualService.getEstadoActualByMatricula(matricula);

        if (estadoActual == null) {
            System.out.println("El vehículo con matrícula " + matricula + " no está en el parking.");
            return ResponseEntity.badRequest().body("El vehículo con matrícula " + matricula + " no está en el parking.");
        }

        // Verificar si ya se ha registrado una salida para este vehículo y el pago es true
        if (estadoActual.getFechaHoraSalida() != null && estadoActual.getPagada() == true) {
            System.out.println("La salida del vehículo con matrícula " + matricula + " ya ha sido registrada.");
            return ResponseEntity.badRequest().body("La salida del vehículo con matrícula " + matricula + " ya ha sido registrada.");
        }

        // Obtener la fecha y hora actual
        LocalDateTime fechaHoraSalida = LocalDateTime.now();

        // Calcular los minutos totales
        float minutosDiferencia = ChronoUnit.MINUTES.between(estadoActual.getFechaHoraEntrada(), fechaHoraSalida);

        // Calcular el precio total
        float precioTotal = estadoActualService.calcularPrecio(minutosDiferencia);

        // Actualizar el estado actual con la fecha y hora de salida, minutos totales y precio
        estadoActual.setFechaHoraSalida(fechaHoraSalida);
        estadoActual.setTotalMinutos(minutosDiferencia);
        estadoActual.setPrecio_total(precioTotal);
        estadoActual.setPagada(pagada:false);

        // Actualizar el estado actual en la base de datos
        estadoActualService.actualizarEstadoActual(estadoActual);

        System.out.println("Ha salido un vehículo con matrícula: " + matricula);
        return ResponseEntity.ok("Salida registrada correctamente con matrícula: " + matricula + " y precio: " + precioTotal);
    }
```

Ilustración 3-42. Método registrarSalida de EstadoActualController.java servidor de gestión de parking

- registrarPago (Ilustración 3-43): se registra el pago de un vehículo y se actualiza su estado en la base de datos.

```java
@PostMapping("/pago")
public ResponseEntity<String> registrarPago(@RequestParam String matricula, @RequestParam String id_pago) {
    // Verificar si ya existe un estado actual para la matrícula
    EstadoActual estadoActual = estadoActualService.getEstadoActualByMatricula(matricula);

    // Verificar si el estado actual es nulo, lo que significa que la matrícula no está registrada en el parking
    if (estadoActual == null) {
        System.out.println("No se encontró ningún vehículo con la matrícula " + matricula + " en el parking.");
        return ResponseEntity.badRequest().body("No se encontró ningún vehículo con la matrícula " + matricula + " en el parking.");
    }

    // Verificar si ya se ha realizado el pago
    if (estadoActual.getPagada() == true && estadoActual.getId_pago() != null) {
        System.out.println("El pago del vehículo con matrícula " + matricula + " ya ha sido realizado.");
        return ResponseEntity.badRequest().body("El pago del vehículo con matrícula " + matricula + " ya ha sido realizado.");
    }

    // Actualizar datos de pagado e id del pago
    estadoActual.setPagada(pagada:true);
    estadoActual.setId_pago(id_pago);

    // Insertar el estado actual en la base de datos
    estadoActualService.actualizarPago(estadoActual);


    System.out.println("Pago registrado correctamente de matrícula: " + matricula);
    return ResponseEntity.ok("Pago registrado correctamente de matricula: " + matricula);
}
```

Ilustración 3-43. Método registrarPago de EstadoActualController.java servidor de gestión de parking

- **ParkingController.java** (Ilustración 3-44): esta clase se corresponde con el controlador que se encarga de gestionar las solicitudes relacionadas con la información del parking. Contiene el método getInfoParking que devuelve detalles del parking como tipo de cargo, moneda de precio, tarifa por minuto, cantidad total de espacios, espacios disponibles, ocupados y su nivel de ocupación.

  Además, este controlador permite la actualización de la información de un estacionamiento específico mediante una solicitud POST en el mismo endpoint. El método updateParking recibe parámetros como el número total de espacios, espacios ocupados, tarifa por minuto y tipo de cargo, y actualiza los datos correspondientes en la base de datos a través del servicio ParkingService.

  Estos datos que se usan en esta clase siguen la estructura y el formato definidos en el modelo de datos Fiware [14] para los parkings.

```java
@RestController
@CrossOrigin(origins = "*")
@RequestMapping("/api/parking")
public class ParkingController {

    private final ParkingService parkingService;

    @Autowired
    public ParkingController(ParkingService parkingService) {
        this.parkingService = parkingService;
    }

    @GetMapping()
    public ResponseEntity<List<Object>> getInfoParking() {
        List<Parking> availableParking = parkingService.getAllParking();

        List<Object> formattedResponse = availableParking.stream()
                .map(parking -> formatParkingInfo(parking))
                .collect(Collectors.toList());

        return ResponseEntity.ok(formattedResponse);
    }

    private Object formatParkingInfo(Parking parking) {
        Map<String, Object> parkingInfo = new HashMap<>();

        parkingInfo.put("id", parking.getId());
        parkingInfo.put("name", parking.getName());
        parkingInfo.put("chargeType", parking.getChargeType());
        parkingInfo.put("priceCurrency", parking.getPricecurrency());
        parkingInfo.put("priceRatePerMinute", parking.getPriceRatePerMinute());
        parkingInfo.put("totalSlotNumber", parking.getTotalSlotNumber());
        parkingInfo.put("availableSlotNumber", parking.getAvailableSlotNumber());
        parkingInfo.put("occupiedSlotNumber", parking.getOccupiedSlotNumber());
        parkingInfo.put("occupancy", (parking.getOccupancy()));

        return parkingInfo;
    }

    @PostMapping()
    public ResponseEntity<String> updateParking(
        @RequestParam int totalSlotNumber,
        @RequestParam int occupiedSlotNumber,
        @RequestParam float priceRatePerMinute,
        @RequestParam String chargeType) {

        Parking parking = parkingService.getParkingById(1L); // El id para este parking es el 1

        if (parking == null) {
            return ResponseEntity.badRequest().body("El parking no se encontró en la base de datos.");
        }

        // Llamar al método del servicio para actualizar el parking
        parkingService.updateParking(parking, totalSlotNumber, occupiedSlotNumber, priceRatePerMinute, chargeType);
        System.out.println("Se han actualizado correctamente los datos del parking");
        return ResponseEntity.ok("El parking ha sido actualizado correctamente.");
    }

}
```

Ilustración 3-44. ParkingController.java del servidor de gestión de parking

3.2.1.3 Configuración del servidor

Para acceder y utilizar la aplicación, se requiere establecer la dirección y el puerto en los cuales el servidor estará activo. En este proyecto, el servidor se ha configurado para escuchar en la dirección IP y el puerto específico:

- Dirección IP del servidor: localhost (127.0.0.1).
- Puerto del servidor: 8081 (indicado en el archivo application.properties).

Con esta configuración, la aplicación se vuelve accesible a través de la dirección http://localhost:8081.

3.2.1.4 Dependencias

Las dependencias incluidas en este servicio son las mismas que las explicadas en el servicio anterior, excepto la de Spring Security ya que este servicio no requiere de autenticación. Estas dependencias han sido incluidas en el archivo POM (Project Object Model) del proyecto, las principales son las siguientes:

- **PostgreSQL**: se ha incluido esta dependencia para facilitar la integración con la base de datos PostgreSQL, permitiendo así el acceso y la manipulación de datos de manera eficiente.
- **Spring Data JPA**: esta herramienta simplifica el manejo de datos en aplicaciones Java utilizando el framework Spring. Proporciona una capa de abstracción sobre la base de datos, lo que facilita la escritura de consultas y la gestión de la persistencia de los objetos. Su objetivo es simplificar al

desarrollador la persistencia de datos entre los distintos repositorios de información.

- **Swagger**: se ha integrado Swagger para generar documentación interactiva de las API REST del proyecto. Esta herramienta simplifica la creación y el mantenimiento de la documentación, lo que mejora la comprensión y el uso de las API por parte de los desarrolladores y usuarios.

A continuación, se muestra un fragmento del archivo POM en el que se pueden ver reflejadas estas dependencias:

```
<!-- PostgreSQL -->
 <dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
 </dependency>
<!-- Spring Data JPA -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
 </dependency>
<!-- Swagger -->
 <dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>3.0.0</version>
 </dependency>
```

3.2.2 Base de datos de gestión de parking

En este apartado se detallará la implementación de la base de datos de un parking. Esta base de datos juega un papel fundamental en la conservación de la información del sistema como pueden ser los datos de los vehículos en su interior y los datos del mismo. Mediante un marco robusto, se consigue retener y gestionar información pertinente, lo que posibilita una gestión y análisis eficaz en cada parking.

3.2.2.1 Tabla estado_actual

La tabla “estado_actual”, se utiliza para almacenar información sobre el estado actual del parking relacionado con los vehículos que contiene en el interior. Cada fila en la tabla representa el estado actual de un vehículo dentro del parking y contiene detalles como la matrícula del vehículo, la fecha y hora de entrada al estacionamiento, la fecha y hora de salida (si ya ha salido), la duración total de la estancia en minutos, el precio total a pagar, si la tarifa ha sido pagada o no, y el ID del pago asociado (si corresponde). El propósito de esta tabla es mantener un registro de los vehículos que están actualmente estacionados, así como de su tiempo de estancia y el estado de pago de la tarifa. El código SQL es el siguiente:

```
DROP TABLE IF EXISTS estado_actual;
CREATE TABLE estado_actual (
  matricula VARCHAR(20) NOT NULL,
  fechayhora_entrada TIMESTAMP NOT NULL,
  fechayhora_salida TIMESTAMP DEFAULT NULL,
  total_minutos DECIMAL(5, 2) DEFAULT NULL,
  precio_total DECIMAL(5, 2) DEFAULT NULL,
  pagada BOOL NULL,
  id_pago VARCHAR(30) NULL,
  PRIMARY KEY(matricula, fechayhora_entrada)
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO estado_actual (matricula, fechayhora_entrada, fechayhora_salida, total_minutos,
precio_total, pagada, id_pago)
VALUES ('1234ABC', '2024-04-17 10:36:28.807865', '2024-04-17 10:42:44.162696', 6.00, 0.24,
TRUE, '9DB69837NJ7525606');
```

3.2.2.2 Tabla info_parking

La tabla "info_parking", se utiliza para almacenar información detallada sobre el parking. Cada fila en la tabla contiene detalles como el ID del estacionamiento (proporcionado por el centro de gestión), el tipo de estacionamiento, el nombre, la categoría, el tipo de tarifa de cobro, la moneda de la tarifa, el precio por minuto, la ubicación geográfica (latitud y longitud), el tipo de ubicación, el tipo de vehículo permitido, la dirección (incluyendo ciudad, país y código postal), el número total de plazas de estacionamiento, el número de plazas disponibles, el número de plazas ocupadas, la tasa de ocupación actual, la altura máxima permitida, el ancho máximo permitido y el tipo de estacionamiento. El propósito de esta tabla es mantener un registro detallado de los datos del estacionamiento, incluyendo su ubicación, capacidad, tarifas y otros detalles relevantes. Esto permite una gestión eficaz y proporciona información útil. El código SQL es el siguiente:

```
DROP TABLE IF EXISTS info_parking;
CREATE TABLE info_parking (
  id INTEGER NOT NULL,
  type VARCHAR(255) NULL,
  name VARCHAR(255) NULL,
  category VARCHAR(255) NULL,
  chargeType VARCHAR(255) NULL,
  priceCurrency VARCHAR(255) NULL,
  priceRatePerMinute DECIMAL(5, 2) NULL,
  latitude FLOAT NULL,
  longitude FLOAT NULL,
  type_location VARCHAR(255) NULL,
  allowedVehicleType VARCHAR(255) NULL,
  address VARCHAR(255) NULL,
  city VARCHAR(255) NULL,
  country VARCHAR(255) NULL,
  postcode VARCHAR(20) NULL,
  totalSlotNumber INTEGER NULL,
  availableSlotNumber INTEGER NULL,
  occupiedSlotNumber INTEGER NULL,
  occupancy DECIMAL(5, 2) DEFAULT NULL,
  maximumAllowedHeight VARCHAR(20) NULL,
  maximumAllowedWidth VARCHAR(20) NULL,
  type_parking VARCHAR(255) NULL,
  PRIMARY KEY(id)
);
```

Un ejemplo de inserción en esta tabla puede ser el siguiente:

```
INSERT INTO info_parking (id, type, name, category, chargeType, priceCurrency,
priceRatePerMinute, latitude, longitude, type_location, allowedVehicleType, address, city, country,
postcode, totalSlotNumber, availableSlotNumber, occupiedSlotNumber, occupancy,
maximumAllowedHeight, maximumAllowedWidth, type_parking)
VALUES (1, 'Subterraneo', 'Parking Plaza del Duque', 'Public', 'Hourly', 'EUR', 0.04,
37.39381408691406, -5.996476173400879, 'Point', 'Car', 'Plaza del Duque', 'Sevilla', 'Sevilla', '41003',
100, 1, 99, 0.99, '2 meters', '2.5 meters', 'OffStreetParking');
```

4 APLICACIÓN ANDROID: ESTACIONA2

La paciencia es una virtud de la programación.
Es necesario tomarse tiempo para comprender
el problema antes de intentar resolverlo.

– E.W. Dijkstra –

En este capítulo, se describe la funcionalidad de la aplicación Android con detalle. La aplicación tiene como nombre "Estaciona2". Los siguientes apartados de este capítulo detallan la estructura y funcionalidad de la aplicación y la vista para el usuario.

En la siguiente ilustración 4-1, se muestran los casos de uso de la aplicación Android para el usuario:

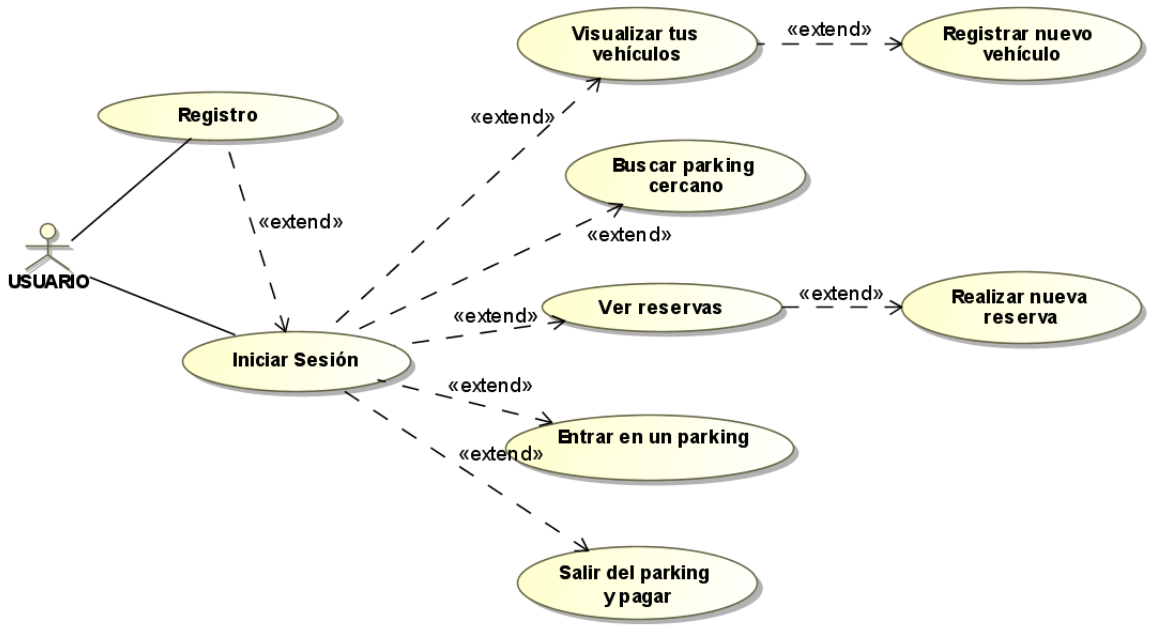

Ilustración 4-1. Casos de uso de la aplicación Android

4.1 Estructura y funcionalidad

En este apartado, se explicarán con detalle las distintas clases de la aplicación, así como la funcionalidad de la misma. Además, se detallará la comunicación entre la aplicación Android, los servidores y la base de datos. Al ser clases muy extensas algunas de ellas, se va a detallar de cada clase sus métodos y utilidad de forma clara.

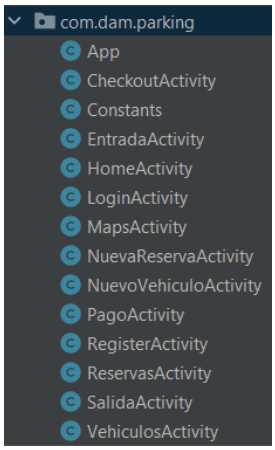

Ilustración 4-2. Clases de la aplicación Android

Las clases que se van a explicar son las mostradas en la ilustración 4-2:

- **App**

  La clase App es una extensión de la clase Application en Android que actúa como un contenedor de datos globales para la aplicación. Representa los siguientes atributos:

  - **dni**: DNI del usuario.
  - **matriculaseleccionada**: Matrícula seleccionada por el usuario.
  - **precio**: Precio asociado.
  - **matriculas**: Lista de matrículas.
  - **nombreparkings**: Lista de nombres de parkings.
  - **idreservas**: Lista de IDs de reservas.
  - **nombreIdParkingMap**: Mapa que relaciona nombres de parkings con sus identificadores correspondientes.

  La clase App proporciona métodos getter y setter para acceder y modificar estos atributos. Esta clase es fundamental para compartir datos en toda la aplicación ya que garantiza que sus datos permanezcan disponibles durante todo el ciclo de vida de la aplicación.

- **Constants**

  La clase Constants es una clase que contiene constantes utilizadas en la aplicación, como la dirección IP y puerto del servidor. Se define una constante llamada API_URL que almacena la dirección IP y puerto del servidor. Esto nos facilita su mantenimiento y reutilización en toda la aplicación ya que en el caso de que cambie el valor de esta constante, solo es necesario actualizar el valor de la constante en esta clase, en lugar de buscar y modificar manualmente todas las referencias en el código.

- **HomeActivity**

  La clase HomeActivity es una actividad de Android que se emplea para abrir las actividades desde el menú principal según el botón que sea pulsado por el usuario.

- **LoginActivity**

  La clase LoginActivity es una actividad de Android que maneja el proceso de inicio de sesión en la aplicación. A continuación, se detallan los métodos y sus funciones en la clase:

  - **onCreate(Bundle savedInstanceState)**: este método se invoca cuando se crea la actividad. Se inicializan y configuran los componentes de la interfaz de usuario que se corresponden con los campos de texto y los botones de inicio de sesión y registro. También se establecen los listeners para los eventos de click en los botones.
  - **LoginTask**: es una clase interna que extiende AsyncTask y se utiliza para realizar la autenticación del usuario en el servidor.
  - **doInBackground(String... params)**: se ejecuta en segundo plano y realiza una solicitud HTTP POST al servidor con las credenciales que introduce el usuario. Devuelve un resultado que indica si el inicio de sesión es correcto o no.
  - **onPostExecute(Integer resultado)**: se invoca posterior a finalización de la ejecución en segundo plano. Actualiza la interfaz de usuario en función del resultado del inicio de sesión. Si el inicio de sesión fue correcto, muestra un mensaje de "Login correcto" y redirige al usuario a la actividad HomeActivity. Si las credenciales son incorrectas, muestra un mensaje de "Usuario o contraseña incorrectos".
  - **getPostDataString(Map<String, String> params)**: este método se utiliza para convertir los parámetros de la solicitud en una cadena de consulta que se envía al servidor.

- **RegisterActivity**

  La clase RegisterActivity es una actividad de Android que se utiliza para el alta de nuevos usuarios en el sistema. Los métodos y sus funciones en esta clase son los mismos que los explicados en la clase anterior, LoginActivity, y por lo tanto no se entrará en detalle de nuevo. Se realiza una solicitud POST al servidor y la respuesta contiene el resultado del registro del nuevo usuario en el sistema. Muestra un mensaje de Toast con la respuesta del servidor.

  Por otro lado, se comprueba que los campos del formulario sean correctos y que las contraseñas introducidas coincidan.

La siguiente ilustración 4-3 muestra el diagrama de secuencia entre la aplicación Android, el servidor del centro de gestión de parkings y su base de datos para el inicio de sesión y el registro en la aplicación:

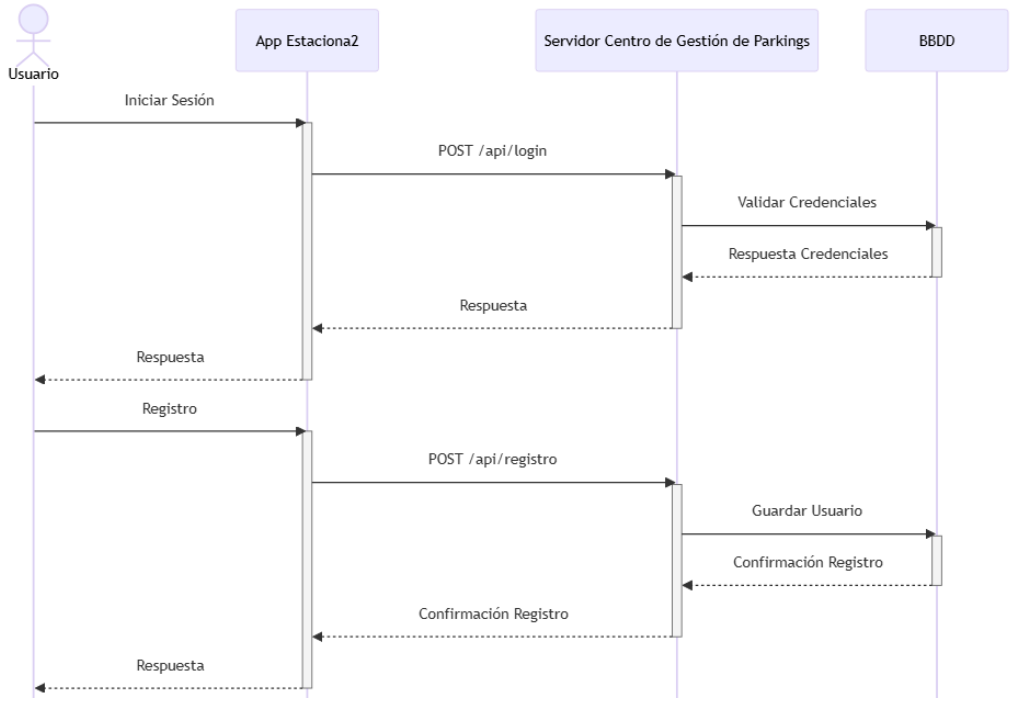

Ilustración 4-3. Diagrama de secuencia aplicación Android para inicio de sesión y registro

- **VehiculosActivity**

  La clase VehiculosActivity es una actividad de Android que se utiliza para mostrar la lista de vehículos asociados a un usuario. A continuación, se detallan los métodos y sus funciones en la clase:

  - **onCreate(Bundle savedInstanceState)**: se invoca cuando se crea la actividad y se configura la lista de vehículos y los botones para agregar vehículos y regresar al menú. Por otro lado, también obtiene el DNI del usuario actualmente autenticado y llama a la tarea GetVehiculosTask para obtener los vehículos asociados a ese usuario.
  - **GetVehiculosTask**: se emplea para realizar una solicitud GET al servidor y obtener la lista de vehículos asociados al usuario. Se ejecuta en segundo plano y devuelve una cadena JSON que consiste en la información de los vehículos.
  - **mostrarListaVehiculos(String json)**: este método recibe una cadena JSON que contiene la lista de vehículos del usuario. Recorre esta cadena y muestra la lista de vehículos en la interfaz de usuario mediante un adaptador de lista.

- **NuevoVehiculoActivity**

  La clase NuevoVehiculoActivity es una actividad de Android que se utiliza para permitir al usuario registrar un nuevo vehículo. Esta clase presenta los mismos métodos y funciones de RegisterActivity. Se realiza una solicitud POST al servidor y la respuesta contiene el resultado del registro del nuevo usuario en el sistema. Además, se muestra un mensaje de Toast con la respuesta del servidor.

- **MapsActivity**

  La clase MapsActivity es una actividad de Android que muestra un mapa utilizando la API de Google Maps y coloca marcadores en ubicaciones de estacionamiento basadas en los datos obtenidos del servidor del centro de gestión de parkings. A continuación, se detallan los métodos y sus funciones:

- **onCreate(Bundle savedInstanceState)**: se invoca este método cuando se crea la actividad. Aquí se inicializan los componentes de la interfaz de usuario.
- **onMapReady(GoogleMap googleMap)**: se configura el mapa y se agrega un listener para los clicks en los marcadores.
- **GetParkingData**: se utiliza para realizar una solicitud GET al servidor para obtener datos de los parkings. Se ejecuta en segundo plano y devuelve una cadena de texto que contiene la respuesta del servidor.
- **showParkingLocations(String json)**: muestra los marcadores de ubicaciones de estacionamiento en el mapa utilizando los datos obtenidos del servidor.
- **getMarkerColor(double occupancy)**: determina el color del marcador según el nivel de ocupación del estacionamiento.
- **ParkingInfoDialog(String title)**: muestra un cuadro de diálogo con información detallada sobre un estacionamiento cuando se hace click en su marcador en el mapa.

La siguiente ilustración 4-4 muestra el diagrama de secuencia entre la aplicación Android, el servidor del centro de gestión de parkings y su base de datos para obtener la lista de los parkings que se muestran en el mapa y para consultar o dar de alta nuevos vehículos en el sistema:

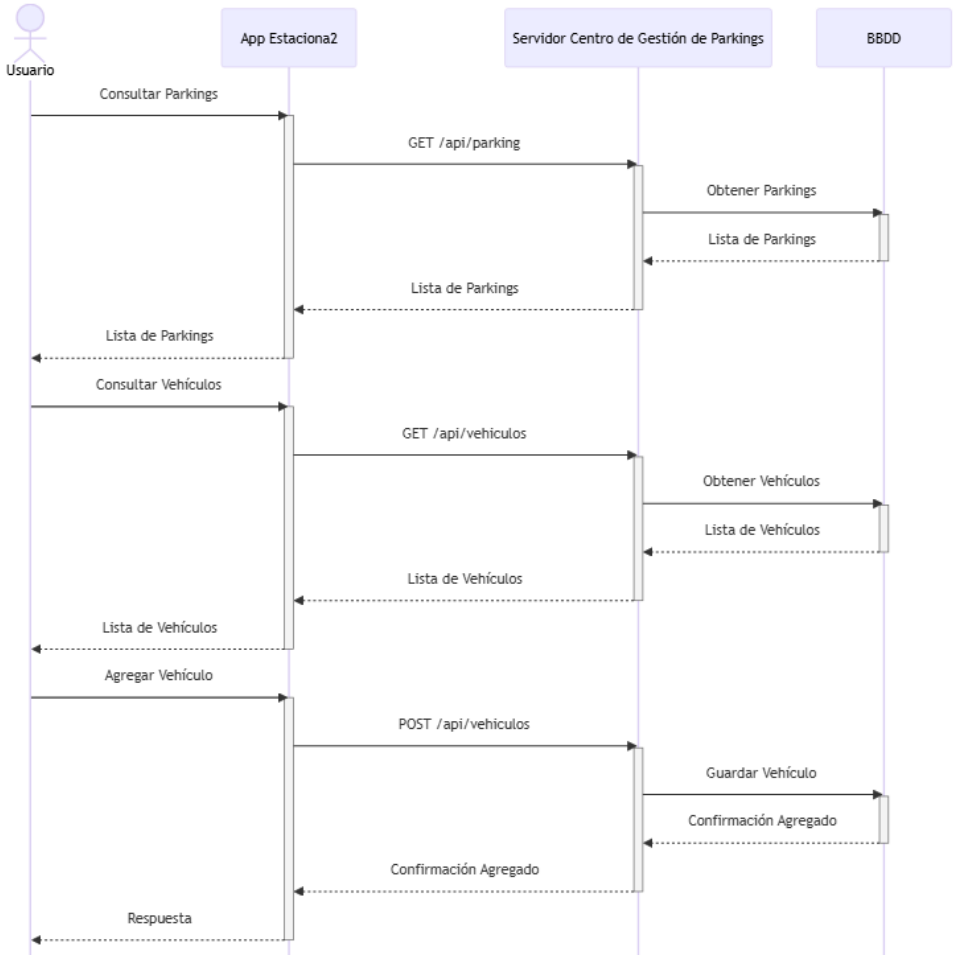

Ilustración 4-4. Diagrama de secuencia aplicación Android para consultar parkings y consultar y dar de alta nuevos vehículos

- **ReservasActivity**

  La clase ReservasActivity es una actividad de Android que muestra las reservas de estacionamiento asociadas a un vehículo seleccionado por el usuario. A continuación, se detallan los métodos y sus funciones:

  - **onCreate(Bundle savedInstanceState)**: se invoca cuando se crea la actividad y se inicializan los componentes de la intefaz de usuario.
  - **GetVehiculosTask**: se utiliza para realizar la solicitud GET al servidor para obtener la lista de vehículos asociados al usuario. Se ejecuta en segundo plano y devuelve una cadena de texto que contiene la lista de vehículos en formato JSON.
  - **mostrarListaMatriculas(String json)**: este método muestra la lista de matrículas obtenida del servidor en el spinner de matrículas.
  - **GetReservasTask**: se utiliza para realizar la solicitud GET al servidor para obtener la lista de reservas asociadas a una matrícula seleccionada. Se ejecuta en segundo plano y devuelve una cadena de texto que contiene la lista de reservas en formato JSON.
  - **mostrarListaReservas(String json)**: muestra la lista de reservas obtenida del servidor en la lista de reservas de la interfaz de usuario. También configura el listener para el botón de cancelar reserva en cada elemento de la lista.
  - **mostrarConfirmacionCancelacion(String idReserva)**: muestra un diálogo de confirmación para que el usuario confirme la cancelación de una reserva.
  - **CancelarReservaTask**: se utiliza para realizar la solicitud DELETE al servidor para cancelar una reserva. Se ejecuta en segundo plano y devuelve un booleano que indica si la cancelación fue exitosa o no.
  - **GetParkingsTask**: se utiliza para realizar la solicitud GET al servidor para obtener los nombres e IDs de parkings.

- **NuevaReservaActivity**

  La clase NuevaReservaActivity es una actividad de Android que permite al usuario realizar una nueva reserva de estacionamiento. Contiene el método onCreate que tiene la misma funcionalidad que todos los explicados anteriormente. A continuación, se detallan los demás métodos y sus funciones en la clase:

  - **RealizarReservaTask**: se utiliza para realizar la solicitud POST al servidor para realizar una nueva reserva de estacionamiento. Se ejecuta en segundo plano y devuelve una cadena de texto que contiene la respuesta del servidor.
  - **mostrarDatePicker(final EditText editTextFecha)**: muestra un diálogo de selección de fecha para el campo de fecha correspondiente. Cuando se selecciona una fecha, se muestra en el EditText correspondiente.
  - **GetParkingsTask**: se utiliza para realizar la solicitud GET al servidor para obtener la lista de nombres e IDs de parkings disponibles.
  - **obtenerIdParkingSeleccionado()**: obtiene el ID del parking seleccionado en el spinner de parkings.
  - **onPostExecute(String responseBody)**: se invoca después de que se complete la tarea de realizar la reserva. Muestra un mensaje con la respuesta del servidor en un Toast.

La siguiente ilustración 4-5 muestra el diagrama de secuencia entre la aplicación Android, el servidor del centro de gestión de parkings y su base de datos para obtener la lista de reservas, crear o cancelar una reserva de estacionamiento:

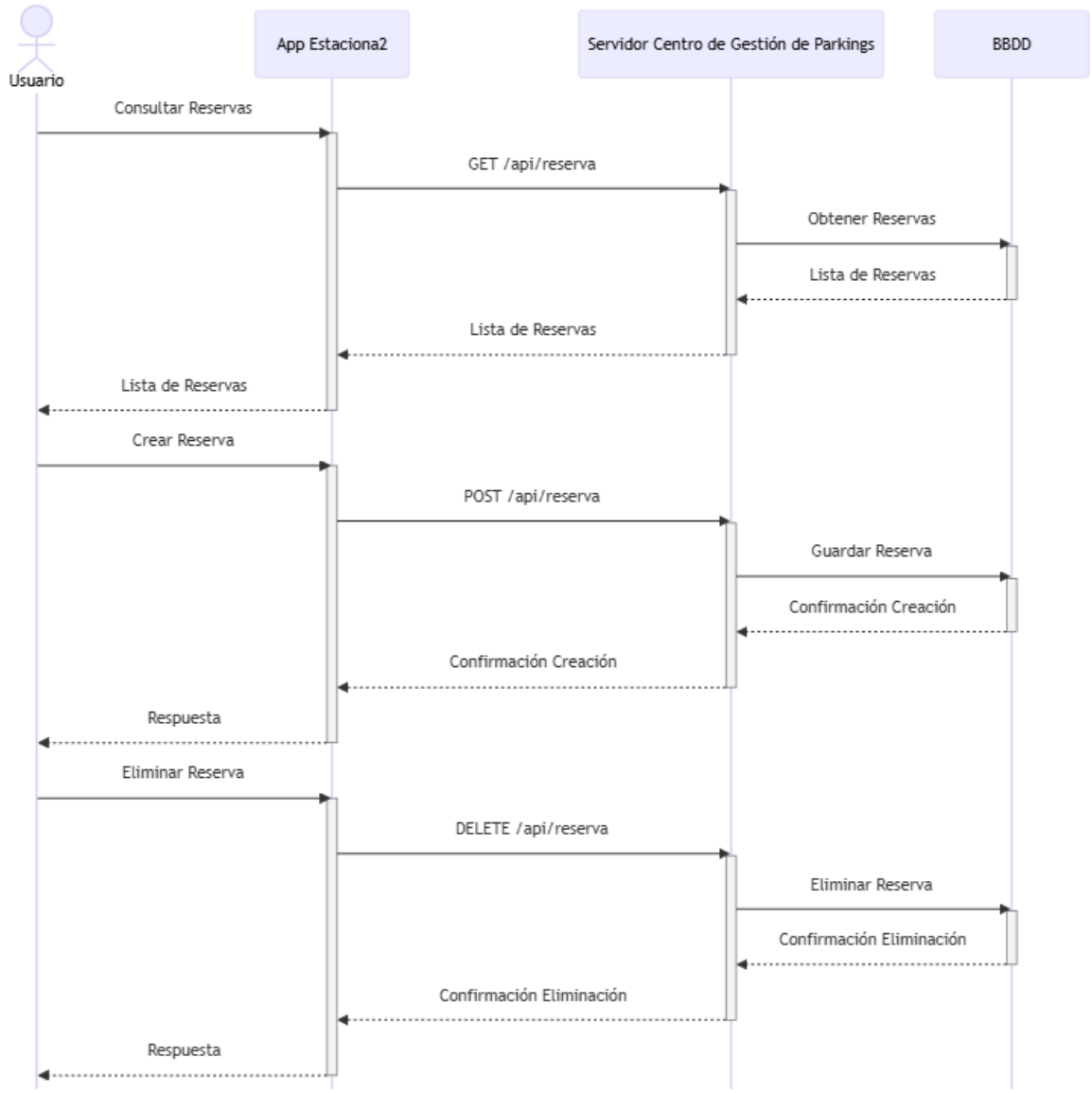

Ilustración 4-5. Diagrama de secuencia aplicación Android para consultar, crear o eliminar reservas

- **EntradaActivity**

  La clase EntradaActivity es una actividad de Android que permite escanear códigos QR para confirmar la entrada de un vehículo en un estacionamiento. A continuación, se detallan los métodos y sus funciones en la clase:

  - **onCreate(Bundle savedInstanceState)**: este método se invoca cuando se crea la actividad. Aquí se inicializan los componentes de la interfaz de usuario y se configuran los listeners para el escáner QR y el spinner de matrículas. También se inicia una tarea asíncrona para obtener las matrículas asociadas al DNI del usuario.
  - **onResume()**: se invoca este método cuando la actividad vuelve a estar en primer plano. En este caso, se reanuda el escáner QR.
  - **onPause()**: se invoca cuando la actividad pasa a segundo plano. En este caso, se pausa el escáner QR para liberar recursos de la aplicación.
  - **GetVehiculosTask**: se utiliza para realizar una solicitud GET al servidor para obtener las

matrículas asociadas al DNI del usuario. Se ejecuta en segundo plano y devuelve una cadena de texto que contiene la respuesta del servidor.

- **mostrarListaMatriculas(String json)**: muestra las matrículas obtenidas del servidor en un spinner.
- **realizarSolicitudPOST(String apiUrl, String matricula)**: realiza una solicitud POST al servidor para confirmar la entrada del vehículo en el estacionamiento. Se ejecuta cuando se escanea un código QR y el usuario confirma la entrada.

- **SalidaActivity**

  La clase SalidaActivity es una actividad de Android que permite escanear códigos QR para confirmar la salida de un vehículo en un estacionamiento. Los métodos y sus funciones en la clase son los mismos que los explicados en la clase anterior, EntradaActivity, y por lo tanto no se explicarán de nuevo. La respuesta del servidor ante la solicitud POST desde esta clase contiene los datos del precio de la estancia que serán usados en las actividades explicadas posteriormente.

- **PagoActivity**

  La clase PagoActivity es una actividad de Android que se encarga de gestionar el proceso de pago utilizando la API de PayPal. Contiene el método onCreate que tiene la misma funcionalidad que los explicados anteriormente. A continuación, se detallan los demás métodos y sus funciones en la clase:

  - **createOrder()**: se utiliza para crear una orden de pago en PayPal. Se realiza una solicitud POST a la API de PayPal para crear la orden de pago con el precio obtenido. Una vez que se crea la orden, se obtiene el enlace de aprobación y se abre en una pestaña del navegador en la aplicación utilizando Custom Tabs.
  - **onSuccess(int statusCode, Header[] headers, String response)**: este método se invoca cuando la solicitud para crear una orden de pago es exitosa. Aquí se obtiene el enlace de aprobación de la respuesta JSON y se abre en una pestaña del navegador en la misma aplicación utilizando Custom Tabs.

  Esta actividad utiliza la API de PayPal para procesar los pagos tal y como se ha mencionado anteriormente. Además, al abrir el enlace a la plataforma PayPal en una pestaña del navegador sobre la aplicación (Custom Tabs), proporciona una experiencia de usuario fluida sin necesidad de salir de la aplicación y abrir el navegador por separado. Es importante destacar que esta actividad es posterior a la de SalidaActivity en el flujo de la aplicación.

- **CheckoutActivity**

  La clase CheckoutActivity es una actividad de Android que gestiona el proceso de verificación después de que se haya realizado un pago exitoso con PayPal. Al igual que los anteriores, contiene el método onCreate. Presenta los siguientes métodos adicionales:

  - **captureOrder(String orderID)**: se utiliza para capturar el pago en PayPal una vez que se ha completado la orden. Se realiza una solicitud POST a la API de PayPal para capturar el pago con el ID de la compra realizada. Después de capturar el pago con éxito, se realiza una solicitud POST adicional para registrar el pago en el servidor del parking.
  - **realizarPagoPOST(String matricula, String orderID)**: inicia una tarea asíncrona para realizar una solicitud POST al servidor del estacionamiento para registrar el pago. Contiene como cuerpo de la solicitud la matrícula del vehículo y el ID del pago de PayPal.

  Esta actividad es parte del flujo de la aplicación después de que se haya completado el proceso de pago con PayPal. Permite capturar el pago y registrar la transacción en el servidor del parking.

La siguiente ilustración 4-6 muestra el diagrama de secuencia entre la aplicación Android, el servidor de gestión de parking y su base de datos para registrar una entrada o salida del parking y realizar el pago de la tarifa:

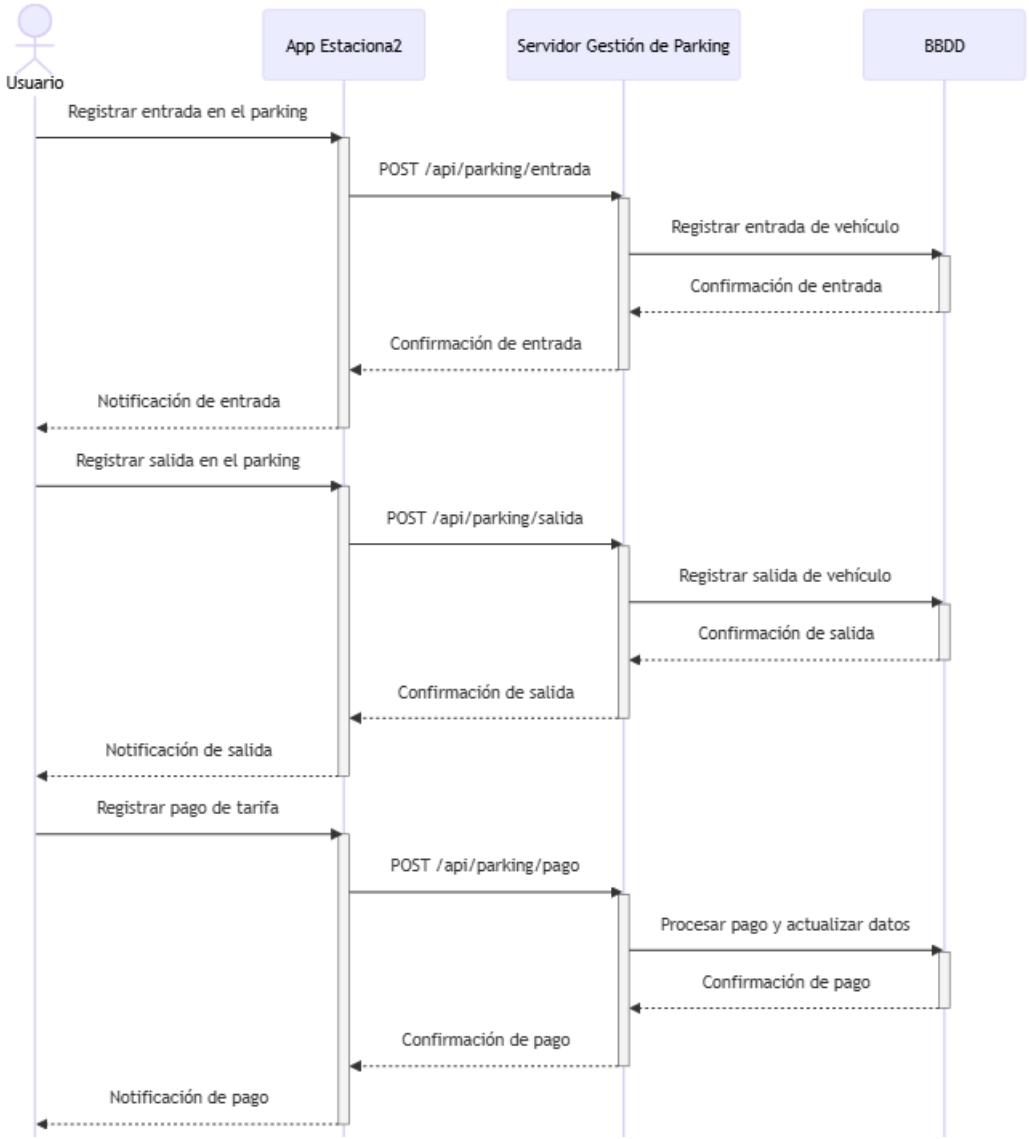

Ilustración 4-6. Diagrama de secuencia aplicación Android para registrar una entrada o salida y realizar el pago de la tarifa

4.2 Interfaz de usuario de la aplicación Android

En este apartado, se presentará cómo están distribuidos los elementos y qué funciones tienen, además de especificar las diferentes formas de interactuar con cada componente. La interfaz de usuario la podemos identificar como el punto de unión entre la funcionalidad que presenta una aplicación con la experiencia para el usuario que hace uso de esta.

Cada detalle del diseño, desde la organización de los elementos hasta la selección de colores y fuentes, se ha examinado minuciosamente para garantizar que los usuarios puedan interactuar de manera sencilla y eficaz con el centro de gestión de parkings o con un parking en concreto.

A continuación, se describen y muestran cada una de las pantallas que contiene la aplicación Android.

4.2.1 Pantalla de inicio de sesión

En la ilustración 4-7, se puede observar la pantalla de inicio de sesión de la aplicación:

Ilustración 4-7. Pantalla de inicio de sesión

Esta pantalla se muestra cuando se abre la aplicación. En la parte superior de la misma, se presenta el logo de la aplicación junto con el nombre de la misma y debajo de este, un título que se denomina "Inicio de sesión". A continuación, se presenta un formulario con los campos de identificación (EditText) que se corresponde con el DNI o NIF del usuario y la contraseña que tiene asociada a su cuenta en la aplicación. Estos campos del formulario son obligatorios de rellenar por lo que si se deja alguno incompleto aparece una alerta en la aplicación que indica al usuario que debe rellenarlos. Por otro lado, en el caso de introducir un DNI o contraseña erróneos, aparecerá una alerta indicando que esos datos son erróneos.

En el caso de que el usuario no esté registrado, en la parte inferior de la vista de esta pantalla se muestra un título denominado "Registro Nuevo Usuario" que va a permitir dar de alta en el sistema al dar click sobre él. Cuando un usuario inicia sesión con su DNI, será guardado en la aplicación para que cualquier gestión dentro de la misma, el usuario no tenga que volver a introducirlo.

4.2.2 Pantalla de registro

La pantalla de registro en el sistema se muestra en la ilustración 4-8:

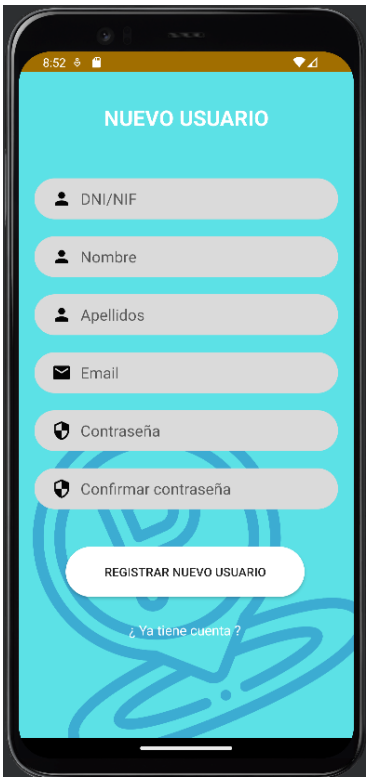

Ilustración 4-8. Pantalla de registro

En el registro de un nuevo usuario en la aplicación, la pantalla muestra un formulario con los campos (EditText): DNI/NIF, Nombre, Apellidos, Email, Contraseña y un campo de repetir contraseña. Estos campos del formulario son obligatorios de rellenar por lo que si se deja alguno incompleto salta una alerta en la aplicación que indica al usuario que debe rellenarlos. Por otro lado, los campos de contraseña y repetir contraseña deben coincidir ya que en caso de que no coincidan se muestra una alerta. Se contempla el caso de que un usuario ya esté dado de alta en el sistema y quiera volver a darse de alta, en este caso, saltará una alerta que indica que ese usuario ya está registrado en el sistema y por tanto no puede volver a hacerlo.

Finalmente, en la parte inferior de la misma se muestra un botón para el registro y un título denominado "¿Ya tiene cuenta?" que permite volver a la pantalla de inicio de sesión.

4.2.3 Pantalla principal

La pantalla principal de la aplicación se muestra una vez el usuario inicia sesión correctamente. Esta pantalla principal se corresponde con la ilustración 4-9, está compuesta por distintos botones que se corresponden con CardView de Android que a su vez contienen una imagen (ImageView) y un texto (TextView). Las funcionalidades de los botones son las siguientes:

- Ir a la pantalla de listar tus vehículos y registrar uno nuevo.
- Ir a la pantalla para buscar un parking cercano.
- Ir a la pantalla para listar tus reservas o realizar una.
- Ir a la pantalla para registrar la entrada en un parking.
- Ir a la pantalla para salir de un parking y realizar el pago.
- Volver a la pantalla de inicio de sesión.

Ilustración 4-9. Pantalla principal

4.2.4 Pantalla de listar tus vehículos

En la ilustración 4-10, se muestra la pantalla en la que se listan los vehículos que tiene asociados el DNI con el que se ha iniciado sesión en la aplicación:

Ilustración 4-10. Pantalla lista de vehículos

Esta pantalla presenta en la parte superior un título con el texto “Tus vehículos” y a continuación una lista con los vehículos (ListView) que tiene el usuario dados de alta en el sistema. En la parte inferior de la pantalla se

muestran dos botones, el primero de ellos es para volver a la pantalla principal de la aplicación y el segundo de ellos para dar de alta un nuevo vehículo en el sistema.

4.2.4.1 Pantalla de registro de nuevo vehículo

Esta pantalla se corresponde con el registro de un nuevo vehículo en el sistema y es consecuencia de haber pulsado el botón de agregar un nuevo vehículo explicado en el apartado anterior. En la ilustración 4-11 se muestra la vista:

Ilustración 4-11. Pantalla registro de nuevo vehículo

En el registro de un nuevo vehículo en el sistema, podemos observar en la parte superior un título con el texto de "Nuevo vehículo" y a continuación un logo de un vehículo. En la parte central de la pantalla se presenta un formulario con los campos (EditText): Matrícula, Marca, Modelo y Número de plazas. Estos campos del formulario son obligatorios de rellenar por lo que si se deja alguno incompleto salta una alerta en la aplicación que indica al usuario que debe rellenarlos. Finalmente, en la parte inferior de la pantalla aparece un botón para realizar el registro del nuevo vehículo y un texto "Volver a mis vehículos" que permite volver a la pantalla anterior.

Cuando un usuario da de alta un nuevo vehículo en el sistema, la aplicación ya tiene guardado previamente el DNI con el que se ha iniciado sesión, esto conlleva que el usuario no tenga que volver a introducirlo tal y como se ha explicado en apartados anteriores. Se contempla el caso de que un usuario introduzca una matrícula que ya tiene dada de alta en el sistema que, en este caso, la aplicación muestra un mensaje de alerta de que esa matricula ya existe en el sistema.

4.2.5 Pantalla de ubicación de parkings

En la ilustración 4-12, se muestra la pantalla en la que se muestra la ubicación de los parkings en el mapa:

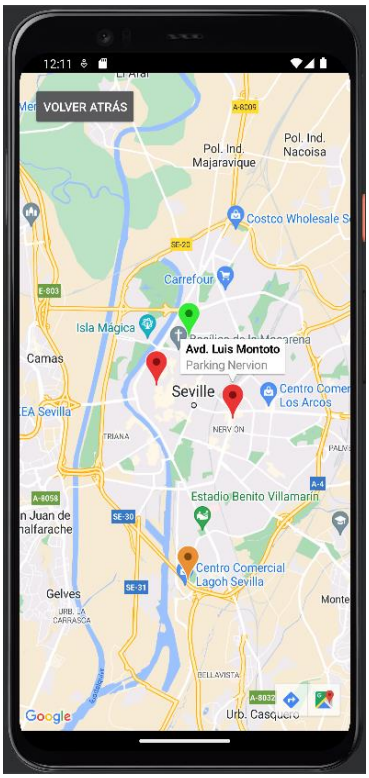

Ilustración 4-12. Pantalla ubicación de parkings

En esta pantalla se puede observar un botón en la parte superior que nos permite volver a la pantalla principal de la aplicación. En el mapa mostrado aparecen las ubicaciones de los parkings y un color asociado que se corresponde con la ocupación de este, los colores son: verde si la ocupación es menor al 75%, naranja si está entre un 75% y un 90% y rojo si es mayor al 90%. En la esquina inferior derecha podemos solicitar que nos indique el camino para llegar al parking marcado en esta pantalla.

Por otro lado, cuando queremos saber más información sobre un parking en concreto hay que dar click encima del mismo y nos muestra los datos: dirección, plazas libres, precio por minuto y la ocupación del mismo. Para cerrar el mensaje debemos pulsar en “OK”. Esto se puede ver en la siguiente ilustración 4-13:

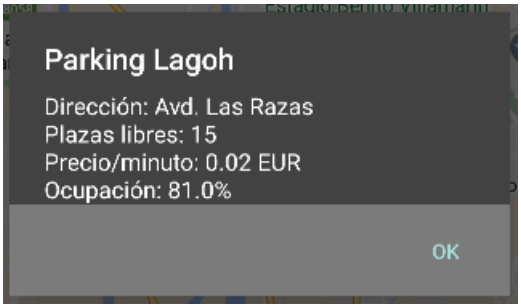

Ilustración 4-13. Alerta con detalle del parking

4.2.6 Pantalla de lista de reservas

La pantalla de reservas presenta la vista mostrada en la ilustración 4-14:

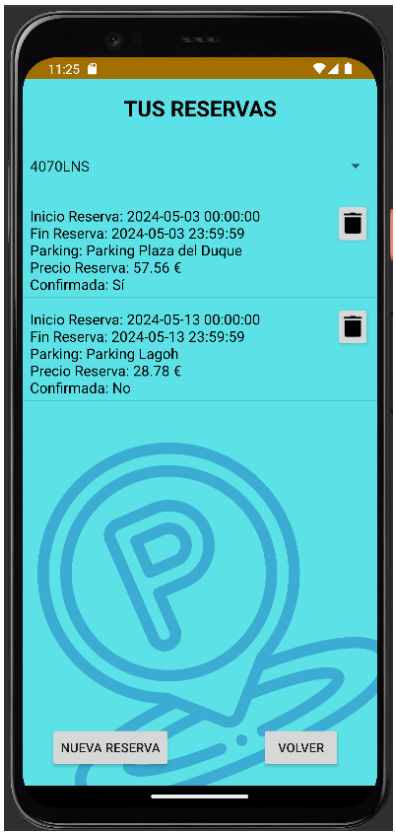

Ilustración 4-14. Pantalla lista de reservas

Esta pantalla, contiene en la parte superior un desplegable (Spinner) para seleccionar la matrícula asociada al DNI con el que se ha iniciado sesión de la que se quiere obtener la lista de reservas que tiene asociadas. Si esa matrícula tiene reservas en el sistema se muestran ordenadas de fecha más actual a más futura, en el caso de que no existan reservas se muestra un mensaje "No hay reservas para esta matrícula".

Cada elemento de la lista se compone de lo siguiente: fecha y hora del inicio de la reserva, fecha y hora del fin de la reserva, nombre del parking en la que se ha realizado, precio de la reserva y si está confirmada o no. Para crear la lista se usa el elemento ListView, en la que cada componente de la misma tiene un TextView con los datos correspondientes y un botón (Button). El botón de cada elemento de la lista contiene el icono de la papelera, cuando el usuario pulsa este botón se muestra un mensaje para confirmar la eliminación de la misma. En la parte inferior de la pantalla se muestran dos botones: botón volver a la pantalla principal de la aplicación y botón para crear una nueva reserva que se detalla en el punto siguiente.

4.2.6.1 Pantalla de realizar una nueva reserva

Esta pantalla se corresponde con el registro de una nueva reserva en el sistema y es consecuencia de haber pulsado el botón de nueva reserva indicado en el apartado anterior. En la ilustración 4-15 se muestra la vista de esta pantalla:

Ilustración 4-15. Pantalla nueva reserva

En el registro de una nueva reserva en el sistema, podemos observar en la parte superior un título con el texto de "Nueva reserva" y a continuación un logo de un vehículo. En la parte central de la pantalla se presenta un formulario con los campos: matrícula (Spinner con las matrículas asociadas al DNI del usuario), nombre del parking (Spinner con los parkings), fecha de inicio y fecha de fin de la reserva. Estos campos del formulario son obligatorios de rellenar por lo que si se deja alguno incompleto salta una alerta en la aplicación que indica al usuario que debe rellenarlos. Finalmente, en la parte inferior de la pantalla aparece un botón para realizar la reserva y un texto "Volver a mis reservas" que permite volver a la pantalla anterior.

Los campos matrícula y nombre del parking son campos desplegables en los que el usuario puede elegir entre las matrículas que están asociadas a su DNI en el sistema y entre los parkings que hay existentes. Por otro lado, la fecha de inicio no puede ser inferior a la actual y además la fecha de fin de la reserva debe de ser igual o posterior a la de inicio, en el caso de que esto no se cumpla, la aplicación muestra una alerta al usuario.

4.2.7 Pantalla de entrada al parking

La pantalla de entrada al parking se compone de un escáner de QR y un campo desplegable (Spinner con las matrículas asociadas al DNI del usuario) con un mensaje que le indica al usuario que debe seleccionar la matrícula con la que va a entrar al parking antes de escanear el QR. Una vez seleccionada y escaneado el QR, se muestra un mensaje de confirmación de la entrada en la que el usuario puede cancelar o aceptar. Finalmente, en la parte inferior de la pantalla se muestra un botón para volver a la pantalla principal de la aplicación.

La ilustración 4-16 muestra la pantalla de entrada al parking y el mensaje de confirmación:

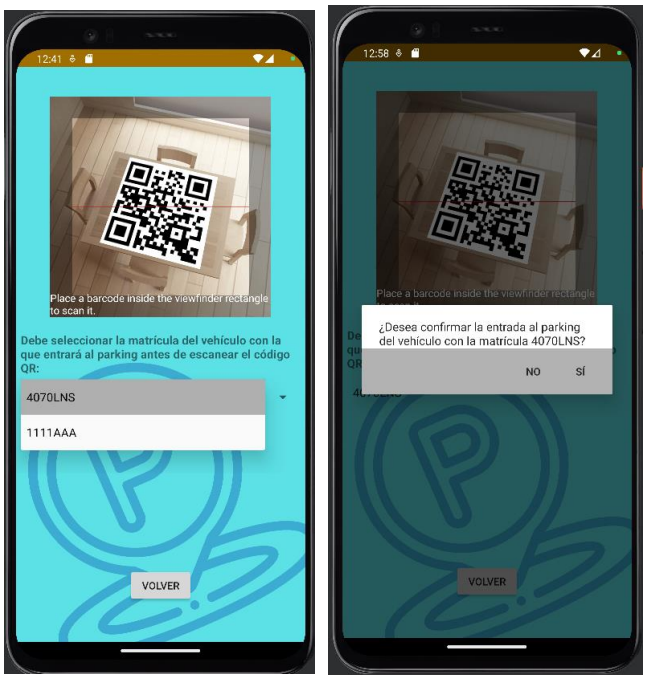

Ilustración 4-16. Pantalla entrada al parking

4.2.8 Pantalla de salida del parking

La pantalla de salida del parking es similar a la de entrada al parking explicada detalladamente en el apartado anterior por lo que no volveré a explicarla con detalle. Esta pantalla incluye un botón con el texto “Pagar” que se activa cuando se realiza una salida correctamente. La vista es la mostrada en la ilustración 4-17:

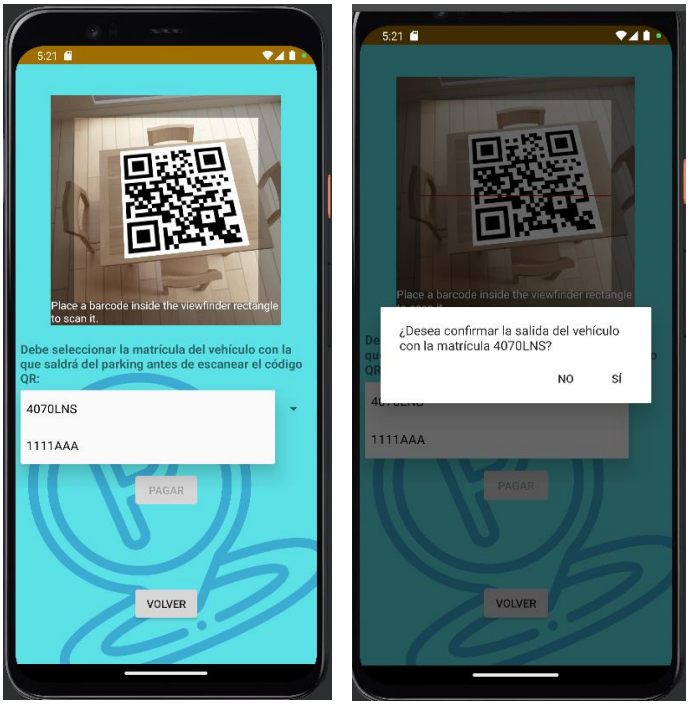

Ilustración 4-17. Pantalla salida del parking

4.2.8.1 Pantalla de pagos PayPal

Esta pantalla se corresponde con la pantalla de pagos de PayPal y es consecuencia de haber pulsado el botón de pago indicado en el apartado anterior. En primer lugar, aparece una pantalla en la que te indica el precio de la estancia en el parking y un botón con el texto de "Pagar con PayPal" que te redirecciona a la página de pagos de PayPal. Posteriormente, en la página de pagos indicas tu cuenta de PayPal y realizas el pago pulsando en el botón correspondiente, esta pantalla (Custom Tabs), se corresponde con el navegador directamente integrado en la aplicación sin necesidad de ir a la aplicación del navegador. Finalmente, nos redirige a la vista de la confirmación del pago en la que aparece el importe a pagar, el identificador de la compra y un botón con el texto de "Confirmar Pago". En la ilustración 4-18 se muestra la secuencia que sigue esta pantalla:

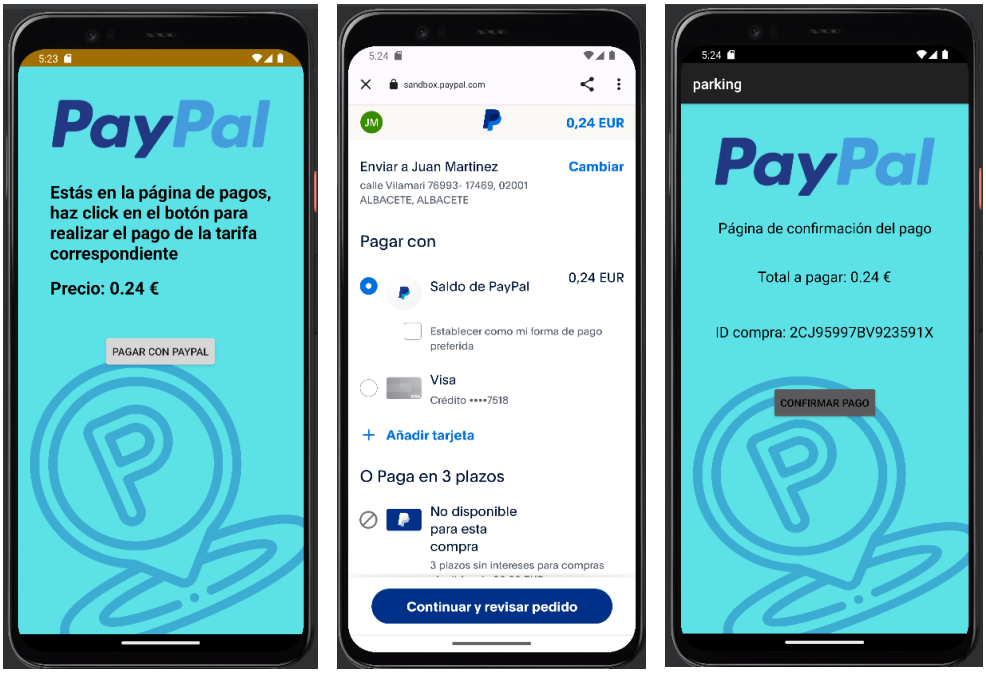

Ilustración 4-18. Pantalla de pago PayPal

4.2.9 Salir de la aplicación

Para salir de la aplicación, se debe pulsar el botón "Cerrar Sesión" mostrado en la pantalla principal de la aplicación tal y como se muestra en la ilustración 4-9.

5 APLICACIÓN WEB GESTOR DE PARKING

La computadora fue diseñada para resolver problemas que antes no existían.

– Bill Gates –

En este capítulo, se describe la funcionalidad de la aplicación web para el gestor de parking con detalle. El gestor del parking se corresponde con el operario que administra el uso del mismo. Este operario realiza las funciones como el control de acceso, monitoreo de plazas y gestión de reservas en el estacionamiento.

Los siguientes apartados de este capítulo detallan la estructura y funcionalidad de la aplicación y la vista para el usuario que se corresponde con el gestor del parking. En la siguiente ilustración 5-1, se muestran los casos de uso de la aplicación web para el usuario:

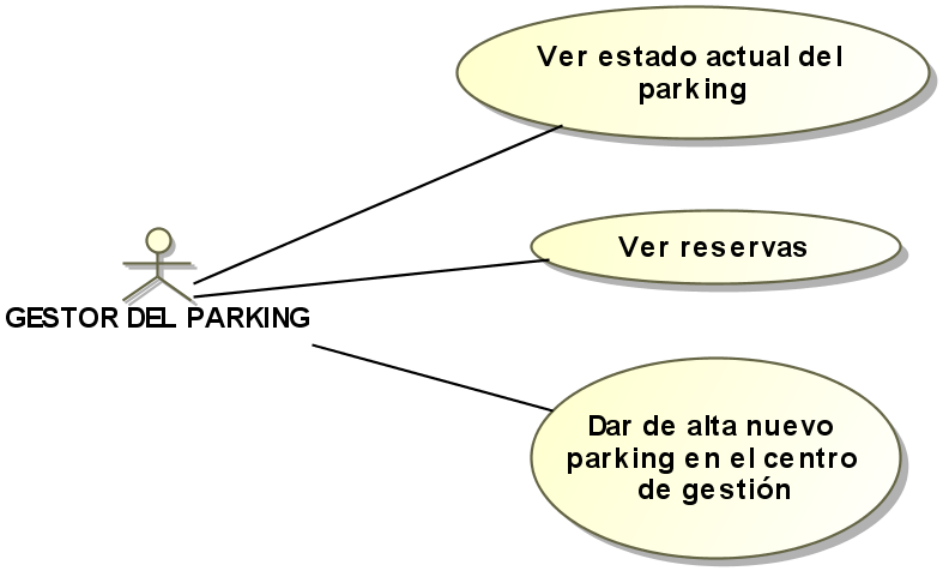

Ilustración 5-1. Casos de uso de la aplicación web

5.1 Estructura y funcionalidad

En este apartado, se explicarán con detalle la estructura y la funcionalidad de la aplicación web. Además, se detallará la comunicación entre la aplicación web, los servidores y la base de datos. La estructura de esta aplicación web se muestra en la ilustración 5-2:

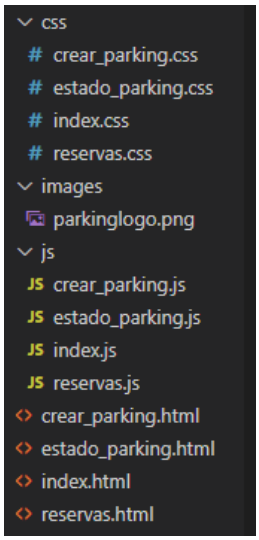

Ilustración 5-2. Estructura de la aplicación web

Como podemos observar en la ilustración anterior, cada archivo HTML se corresponde con una funcionalidad distinta de la aplicación web y los archivos dependientes de estos están separados por hojas de estilos, scripts e imágenes. Las funcionalidades van a ser descritas con más detalle a continuación:

- **index.html**: se corresponde con el archivo principal HTML que se carga automáticamente cuando el usuario inicia la aplicación. En función de la actividad que desee realizar el usuario, en la aplicación se redireccionará a reservas.html, crear_parking.html o estado_parking.html. El script "index.js" es el encargado de realizar esta función de redirección.

- **reservas.html**: este archivo se corresponde con el apartado de la gestión de las reservas que han realizado los usuarios en un parking concreto. Contiene un script llamado "reservas.js" que se encarga de realizar solicitudes al servidor para obtener los datos de las reservas. Además, se pueden gestionar las confirmaciones o cancelaciones de las reservas por parte del operador del parking.

  Este script utiliza la función fetch() que se encarga de realizar una solicitud GET al servidor y obtener los datos correspondientes a las reservas en formato JSON. Una vez obtenidas las reservas, las inserta dinámicamente en la tabla HTML ordenadas de fecha más reciente a fecha más posterior. Por otro lado, se definen dos funciones adicionales llamadas confirmarReserva() y noConfirmarReserva() que al ser pulsados los botones asociados a las mismas, se realizan solicitudes POST al servidor y confirma o cancela la confirmación de la reserva. Finalmente, estas funciones actualizan la interfaz de usuario y recargan la página después de completar las operaciones.

- **crear_parking.html**: este archivo corresponde al apartado de creación de un nuevo parking en el sistema. Presenta un formulario que permite ingresar los detalles del nuevo parking. Además, se incluye un script llamado "crear_parking.js" que se encarga de validar el formulario antes de enviarlo y realizar la solicitud POST al servidor para crear el nuevo parking.

  El formulario contiene campos para ingresar información relevante, como tipo, nombre, categoría, tarifa por minuto, ubicación geográfica, cantidad de plazas disponibles, entre otros. También se incluye un botón "Crear Parking" que activa la función registrarNuevoParking() al ser pulsado.

  El script "crear_parking.js" contiene las siguientes dos funciones:

  - **validarFormulario()**: esta función valida que todos los campos del formulario estén

completos antes de enviar la solicitud al servidor. Si algún campo está vacío, se muestra una alerta indicando al usuario que complete todos los campos necesarios.

- **registrarNuevoParking()**: esta función se encarga de enviar una solicitud POST al servidor con los datos del nuevo parking ingresados en el formulario. Antes de enviar la solicitud, se valida el formulario utilizando la función validarFormulario(). Si el formulario está validado correctamente, se crea un objeto con los valores de los campos del formulario y se realiza la solicitud POST al servidor. Una vez completada la operación, se muestra una alerta indicando al usuario que el nuevo parking ha sido creado exitosamente o se maneja el error en caso de que ocurra algún problema durante la creación del parking.

La siguiente ilustración 5-3 muestra el diagrama de secuencia entre la aplicación web para el gestor del parking, el servidor del centro de gestión de parkings y su base de datos para dar de alta un nuevo parking, consultar las reservas realizadas en el parking y confirmar o cancelar la confirmación de reservas:

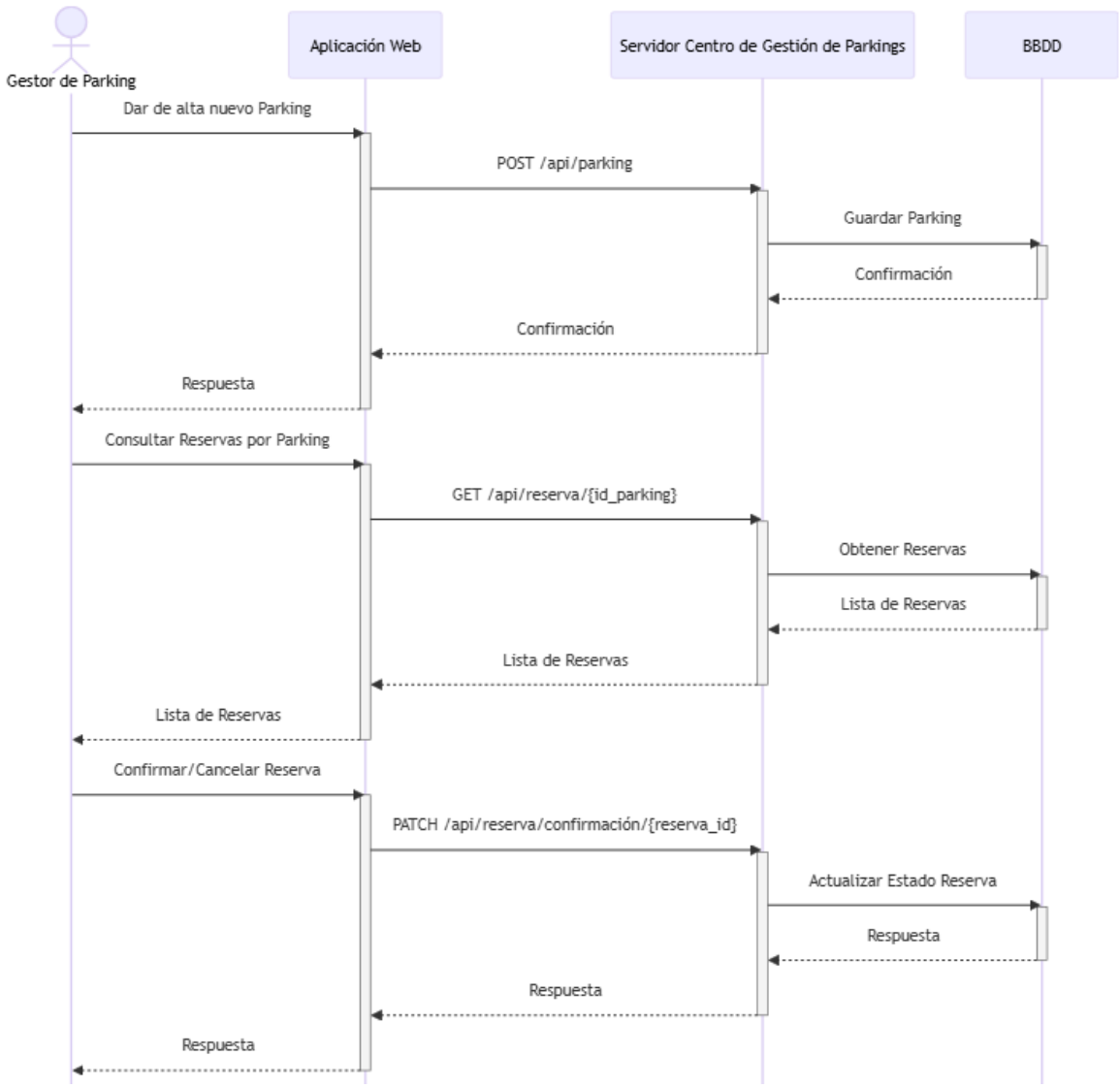

Ilustración 5-3. Diagrama de secuencia comunicación aplicación web, servidor centro de gestión de parkings y base de datos para dar de alta un nuevo parking, consultar y confirmar o cancelar reservas

- **estado_parking.html**: este archivo representa el apartado que muestra y permite modificar el estado

actual de un parking en el sistema. Contiene un formulario con campos para visualizar y actualizar información como el nombre del parking, el número total de plazas, el número de plazas ocupadas, la tarifa por minuto y el tipo de carga de la tarifa. Además, incluye un botón "Guardar Cambios" que activa la función para enviar los cambios al servidor.

El formulario tiene el campo del nombre del parking deshabilitado, solo para verlo y no se permite su modificación en esta interfaz.

El script "estado_parking.js" realiza una solicitud GET al servidor de la cuál se obtiene el estado actual del parking y así mostrar los datos en el formulario. Además, agrega un controlador para permitir el envío de una solicitud POST al servidor con los datos actualizados del parking.

El script contiene lo siguiente:

- Obtener el estado del parking: utiliza la función fetch() para realizar una solicitud GET al servidor y obtener los datos del parking. Luego, actualiza los campos del formulario con la información obtenida del servidor.
- Enviar los cambios al servidor: agrega un controlador de eventos al formulario para interceptar el envío. Luego, se recopilan los datos del formulario y se envían al servidor utilizando una solicitud POST. Si la solicitud se completa con éxito, se muestra una alerta indicando que los cambios han sido guardados correctamente. En caso de error, se muestra una alerta indicando que ha ocurrido un error al guardar los cambios.

La siguiente ilustración 5-4 muestra el diagrama de secuencia entre la aplicación web para el gestor del parking, el servidor de gestión de parking y su base de datos para mostrar o modificar los datos del parking que administra:

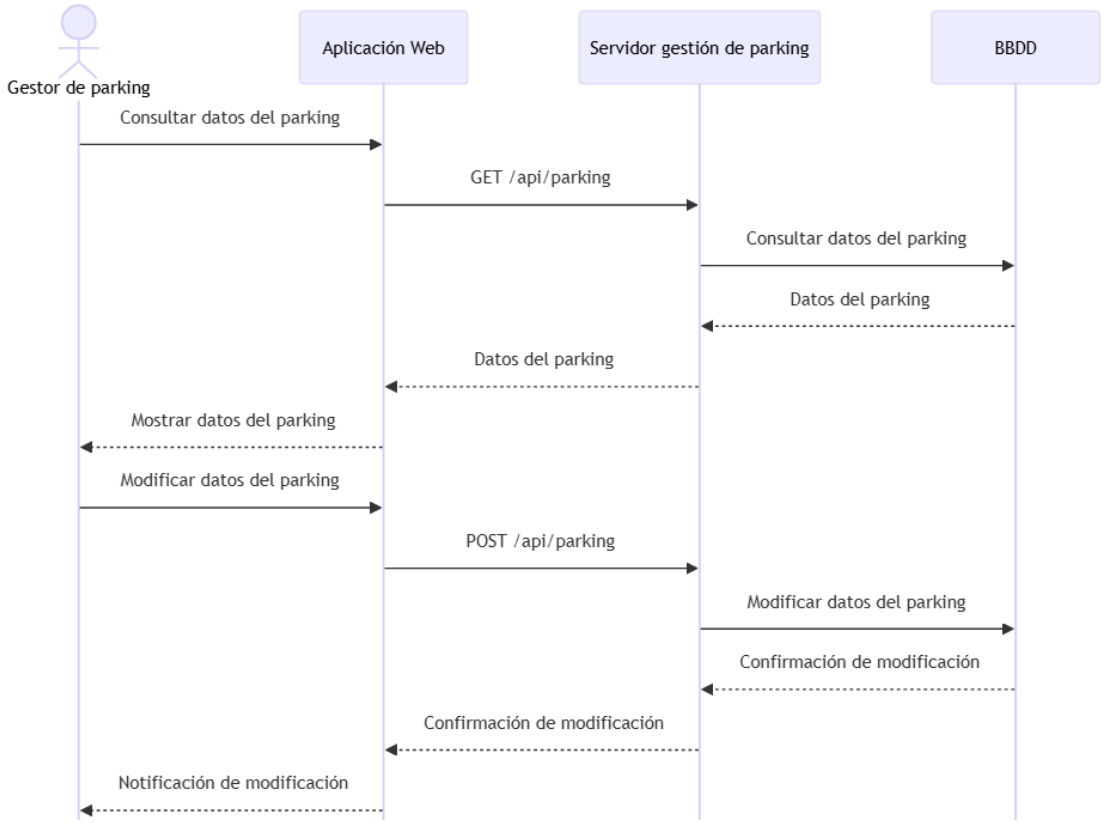

Ilustración 5-4. Diagrama de secuencia comunicación aplicación web, servidor gestión de parking y base de datos para mostrar o modificar los datos del parking

5.2 Interfaz de usuario

En este apartado, se presentará cómo están distribuidos los elementos y qué funciones tiene la aplicación web, además de especificar las diferentes formas de interactuar con cada componente. La interfaz de usuario la podemos identificar como el punto de unión entre la funcionalidad que presenta una aplicación con la experiencia para el usuario que hace uso de esta.

Cada detalle del diseño, desde la organización de los elementos hasta la selección de colores y fuentes, se ha examinado para garantizar que los usuarios puedan interactuar de manera sencilla y eficaz dentro de la aplicación.

A continuación, se describen y muestran cada una de las pantallas que contiene la aplicación web de gestión de un parking.

5.2.1 Pantalla principal

La pantalla principal de la aplicación web se muestra cada vez que el usuario abre la aplicación, esta pantalla principal se corresponde con la ilustración 5-5, está compuesta por 3 botones con las siguientes funcionalidades:

- Ir a la pantalla para ver el estado actual del parking.
- Ir a la pantalla para ver las reservas solicitadas.
- Ir a la pantalla para dar de alta un nuevo parking en el centro de gestión.

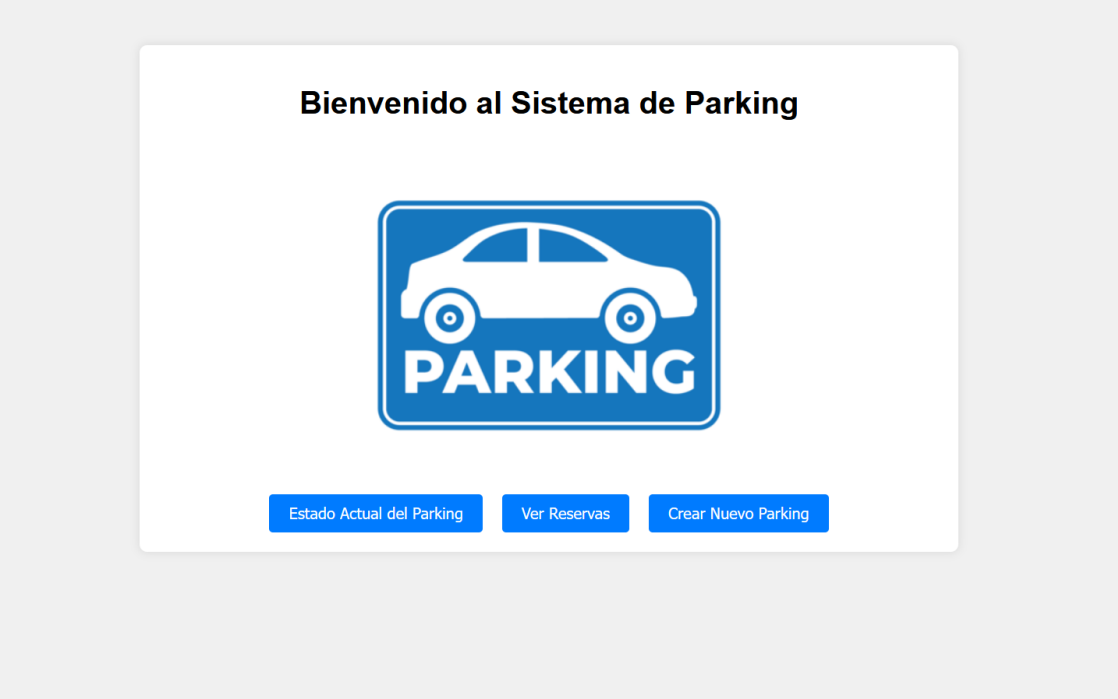

Ilustración 5-5. Pantalla principal aplicación web

5.2.2 Pantalla estado actual del parking

Esta pantalla se corresponde con la visualización de datos que permiten ver el estado actual del parking tal y como podemos ver en la ilustración 5-6. Estos datos se podrán visualizar o modificar en función de la

necesidad del mismo. Esta pantalla se compone de varios campos que muestran los siguientes datos:

- Nombre del parking, este campo no se puede modificar.
- Número total de plazas de las que dispone el parking, este campo se puede modificar según la necesidad.
- Número de plazas ocupadas actuales, este campo se puede modificar.
- Precio por minuto, este campo también es modificable.
- Tipo de carga de la tarifa, este campo también se puede modificar. Este campo indica si la tarifa es fija o puede cambiar según la hora.

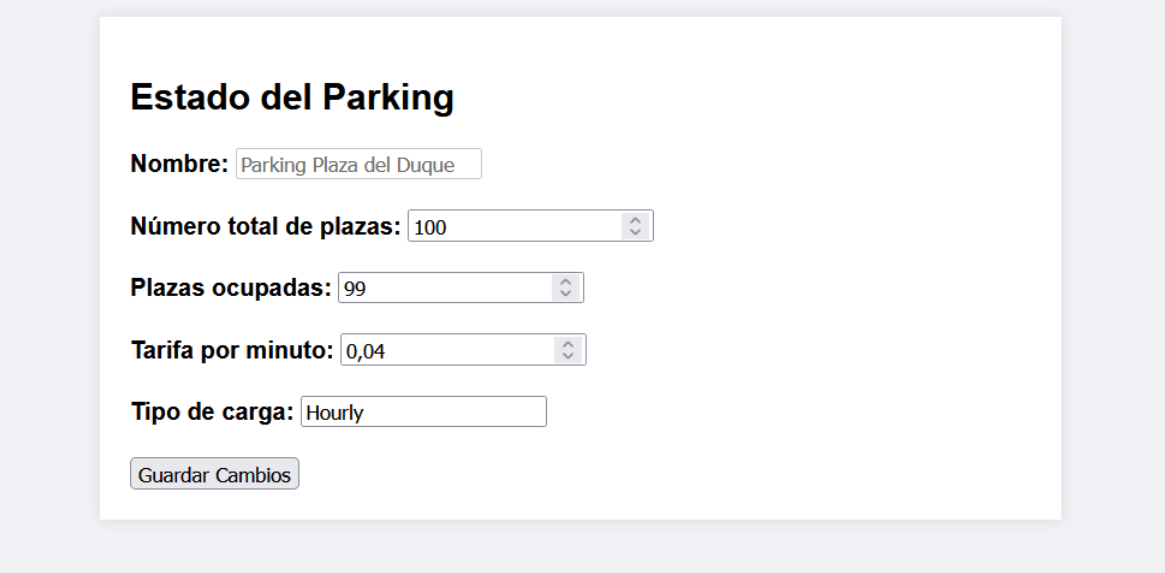

Ilustración 5-6. Pantalla estado actual del parking

La pantalla presenta un botón con el texto “Guardar Cambios”. Al ser pulsado, nos muestra un mensaje que indica si se han realizado los cambios correctamente o no. Y como consecuencia de este acto los datos nuevos son actualizados y mostrados, ilustración 5-7.

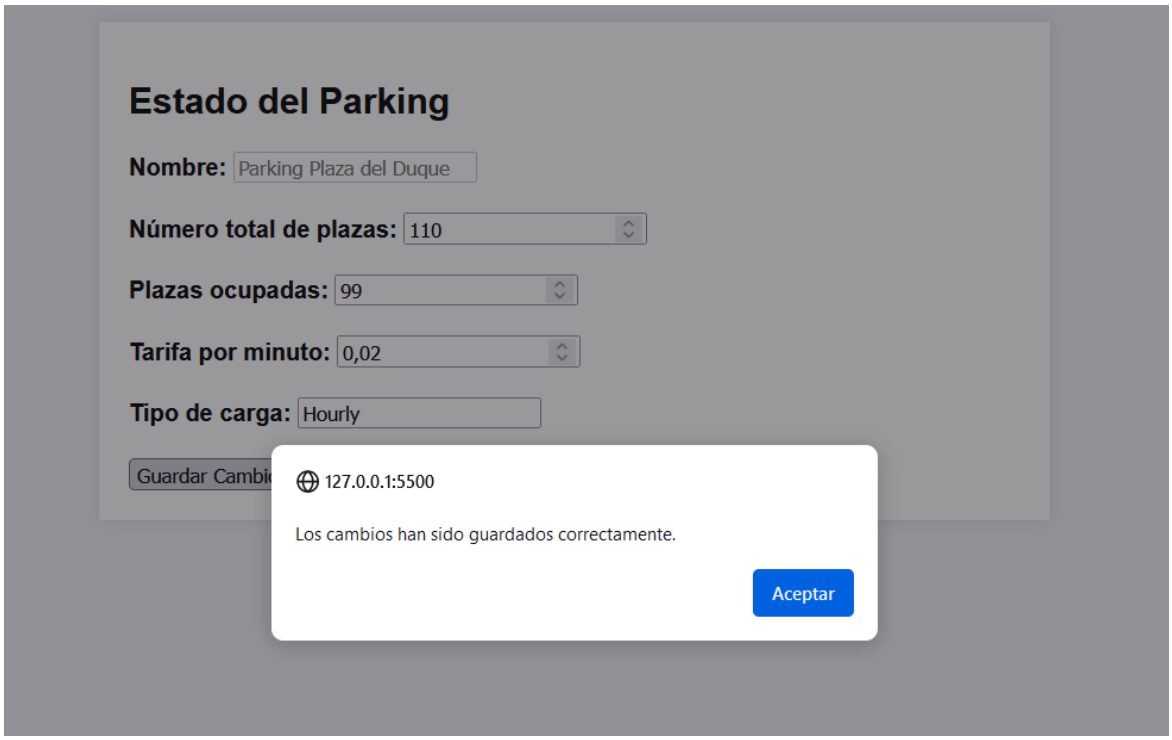

Ilustración 5-7. Alerta cambios estado actual del parking

5.2.3 Pantalla para ver reservas solicitadas

La pantalla para ver las reservas solicitadas por los clientes tiene el aspecto mostrado en la ilustración 5-8. Esta pantalla se compone de una tabla con los campos: identificador, matrícula asociada, fecha y hora de inicio y fin, el estado (confirmada o no confirmada) y los botones para confirmar o no confirmar las reservas. Estos botones son mostrados siguiendo la siguiente lógica:

- En el caso de que la reserva tenga el estado "No Confirmada" aparece un botón con el color verde con el texto "Confirmar" que llama a una función para confirmarla.
- En el caso contrario, si la reserva tiene el estado "Confirmada" aparece un botón de color rojo con el texto "No Confirmar" que, al ser pulsado, llama a la función respectiva para cambiar la reserva al estado no confirmada.

Cuando cualquiera de los botones es pulsado, se muestra una alerta de si se ha confirmado o no correctamente.

Ilustración 5-8. Pantalla reservas solicitadas

5.2.4 Pantalla para crear nuevo parking

Esta pantalla se corresponde con un formulario para poder dar de alta en el centro de gestión un nuevo parking tal y como podemos ver en la ilustración 5-9. El formulario se compone de un campo de datos compuesto por: tipo de parking, nombre, categoría, tipo de carga de la tarifa, moneda, precio por minuto, latitud y longitud, tipo de vehículo permitido, dirección del parking, ciudad, país, código postal, total de plazas de las que dispone, altura y anchura máxima y el tipo de parking que es un desplegable con opciones a elegir.

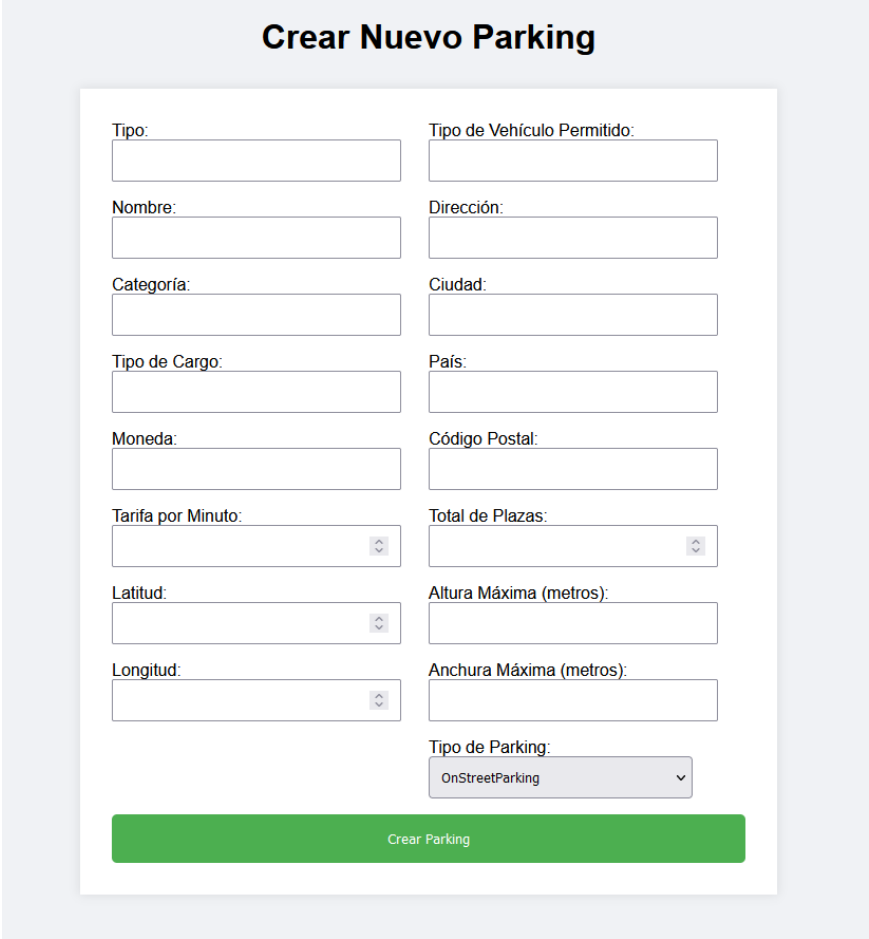

Ilustración 5-9. Pantalla dar de alta nuevo parking

Los campos del formulario son obligatorios de rellenar por lo que si se deja alguno incompleto salta una alerta en la aplicación que indica al usuario que debe rellenarlos. Finalmente, en la parte inferior existe un botón verde con el texto de “Crear Parking”, cuando es pulsado se muestra una alerta en la aplicación que indica si se ha dado de alta correctamente o no en el centro de gestión. La siguiente ilustración 5-10 muestra la alerta de que el parking se ha creado correctamente y el identificador que tiene asociado:

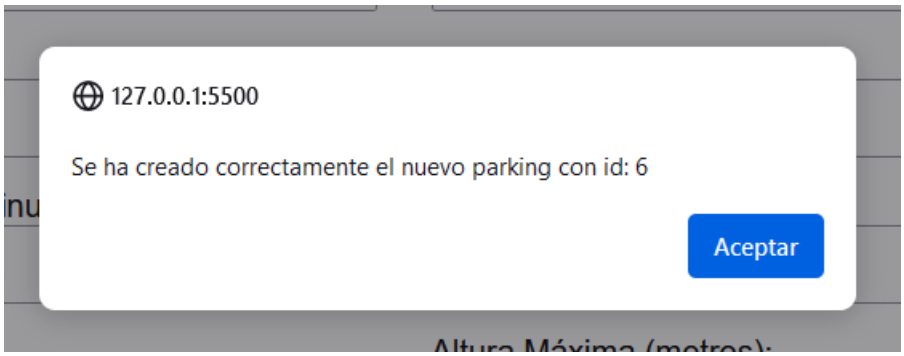

Ilustración 5-10. Alerta al dar de alta nuevo parking correctamente

En cambio, si hay un error al crearlo, nos muestra la alerta mostrada en la ilustración 5-11 que nos indica que ya existe un parking en la ubicación que se intenta dar de alta:

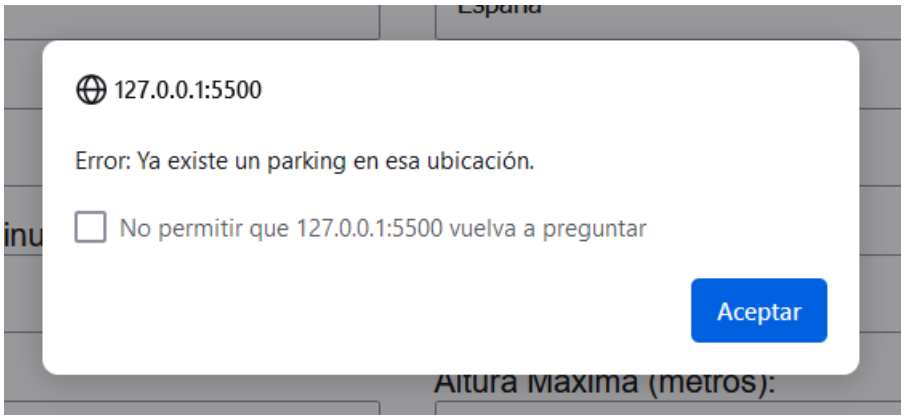

Ilustración 5-11. Alerta de error al dar de alta nuevo parking

6 CONCLUSIÓN Y LÍNEAS FUTURAS

Como conclusión de esta memoria, resulta fundamental resaltar los hitos y progresos alcanzados por este proyecto en el desarrollo de una aplicación para Android, una aplicación web para el gestor de parking y la implementación de servicios REST, para el centro de gestión de parkings y para la gestión de parking. Esta sección de conclusiones recapitula las conclusiones técnicas y traza las posibles direcciones futuras que podrían elevar este proyecto a niveles superiores de innovación y rendimiento.

6.1 Conclusiones técnicas

Para este proyecto se ha creado una aplicación Android con el nombre "Estaciona2" que permite a los usuarios interaccionar con el centro de gestión de parkings para dar de alta nuevos vehículos, realizar reservas y ver información relevante de los parkings de la red a través de la API de Google Maps. Por otro lado, permite la entrada y la salida del parking mediante el escaneo de QR, así como el pago de la tarifa correspondiente a la estancia en el mismo mediante la API de PayPal. Además, se han implementado dos servicios REST, uno de ellos correspondiente con el centro de gestión de parkings y el otro con la gestión de parking. Finalmente, para el gestor de parking se ha creado una aplicación web la cual permite la gestión del mismo.

La integración de estos sistemas permite al usuario tener una mejor experiencia para interactuar con los estacionamientos. La integración con los sistemas de pago y los mapas permiten mejorar esta experiencia y hacerla más cómoda.

Finalmente, este proyecto ha significado un relevante aprendizaje en el desarrollo de servidores REST, bases de datos y aplicaciones web y móviles Android. He podido aplicar los conocimientos teóricos y prácticos adquiridos en esta etapa educativa y además adquirir nuevos conocimientos en los sistemas de estacionamiento aplicados a ciudades inteligentes.

6.2 Conclusiones personales

Este proyecto define una etapa en mi recorrido académico que muestra la dedicación y constancia durante este tiempo para llegar a conseguir nuevos logros. Puedo concluir que lo propuesto al inicio de este proyecto se ha cumplido con éxito y puede verse reflejado en el mismo.

Por otro lado, me ha ayudado a mejorar y reforzar mis habilidades en el ámbito tecnológico. Además, supone una satisfacción personal la evolución que he tenido durante todo este periodo y el logro de haber superado los nuevos desafíos que ha supuesto. Este proyecto me ha permitido conocer nuevas tecnologías y reforzar mis conocimientos en otras en las que ya había tenido oportunidad de conocer previamente. A pesar de mi falta de experiencia en este tipo de proyectos, mis conocimientos adquiridos durante mi etapa académica en la Ingeniería y la ayuda de mi tutora han supuesto una ayuda fundamental para alcanzar los objetivos propuestos.

Este proyecto ha mejorado mis habilidades para elaborar esta memoria técnica que abarca la estructura fundamental y análisis de las aplicaciones y servicios implementados. Además, ha enriquecido mi capacidad para tener una importante planificación y organización de todos los pasos del proyecto de forma ordenada.

Me ha permitido conocer nuevos ámbitos tecnológicos aplicables a la mejora de la vida del ser humano. Además, he podido mejorar la confianza para afrontar nuevos proyectos similares a este y continuar en un futuro teniendo nuevas experiencias.

6.3 Líneas futuras

En este apartado se definirán las posibles direcciones en las que este proyecto podría evolucionar y tener más funcionalidades. Se definirán nuevos aspectos y nuevas tecnologías en distintas áreas en las que el proyecto puede potenciar y enriquecer su alcance y funcionalidad.

6.3.1 Diseño

En el apartado del diseño y sus posibles mejoras podemos continuar con las siguientes:

- Creación e implementación de una barra lateral de navegación para que la experiencia del usuario sea más cómoda y gratificante. Por otro lado, aprovechar para mostrar como pantalla principal el mapa con los estacionamientos y usar la barra lateral para acceder a las otras funcionalidades.
- Redefinir nuevas formas de figuras y alertas más atractivas visualmente para el usuario.

6.3.2 Funcionalidades

Las funcionalidades futuras que puede implementar son las siguientes:

- Añadir notificaciones mediante email y en la misma aplicación móvil para las reservas que realice el usuario. Esto permite que el usuario tenga presente en todo momento el estado de sus reservas mediante alertas.
- Permitir el pago de tarifas mediante otras plataformas de pago como por ejemplo Bizum, transferencia bancaria, etc.
- Añadir al servidor REST del centro de gestión de parkings una función para poder calcular las horas de máxima afluencia en los parkings y la ocupación promedio.
- Añadir al servidor REST de gestión del parking la generación de informes para permitir al gestor del parking acceder a información detallada por periodos. Por otro lado, añadir un sistema de sugerencias o incidencias que puedan tener los usuarios que hacen uso del mismo.
- Expandir el sistema para los estacionamientos en las vías urbanas y usar sensores para permitir conocer el estado de los mismos.
- Adaptar la aplicación para el sistema operativo iOS.
- Integrar nuevas conexiones con otros sistemas como el centro de gestión de tráfico definido por la Arquitectura de Referencia para el Transporte Cooperativo e Inteligente (ARC-IT) [1].

6.3.3 Seguridad

En el aspecto de la seguridad, se podrían mejorar los siguientes puntos:

- Implementación de requerimiento de autenticación usando SpringSecurity para acceso a otras rutas que no sean /api/login o /api/registro.
- Implementar el protocolo seguro HTTPS en las consultas al servidor. Esto implica obtener un certificado SSL/TLS y configurar el servidor y la aplicación para soportar esta funcionalidad.
- Implementar mecanismos para limitar los intentos de inicio de sesión fallidos, como el bloqueo de cuentas temporalmente después de varios intentos fallidos.

6.3.4 Despliegue e infraestructura

En el despliegue y la infraestructura podemos concluir con los siguientes aspectos que se podrían implementar:

- Desplegar los servidores en plataformas como Azure o Google Cloud Platform. Estas plataformas ofrecen servidores virtuales y alojamiento para bases de datos relacionales.
- Desplegar la aplicación móvil en la tienda de aplicaciones de Android, Play Store. Esto permitirá a cualquier usuario realizar su descarga desde la misma.

- Desplegar la aplicación web en plataformas como Netlify o Heroku que nos ofrecen herramientas sencillas para el despliegue de aplicaciones web.
- Implementar herramientas para el monitoreo de logs de las aplicaciones y para obtener información sobre el uso de las mismas y las incidencias que pueda tener.

ANEXO A: INSTALACIÓN Y DESPLIEGUE DE LOS SERVICIOS REST IMPLEMENTADOS

El código de los servicios REST implementados y las aplicaciones Android y web está disponible en el siguiente repositorio de GitHub:

https://github.com/jmgalin/TFG_Parking_ETSI

Desde este repositorio, descargue el archivo "serv_REST_TFG.zip" correspondiente a los servicios REST del proyecto y descomprímalo en la ubicación deseada del sistema.

1. Instalación de los servidores REST

Antes de instalar los servidores REST, verifique que tenga Maven instalado en su sistema. Puede obtenerlo ejecutando el siguiente comando de descarga.

```
sudo apt install maven
```

Podemos comprobar que se ha instalado correctamente con el siguiente comando:

```
maven -version
```

Una vez instalado Maven, abra un terminal y navegue hasta el directorio donde descomprimió el proyecto utilizando el comando:

```
cd ruta/directorio
```

2. Inicio de las bases de datos

En el directorio del proyecto se encuentra un script que permite inicializar las bases de datos requeridas para el funcionamiento del proyecto. El script "bbddsetup.sh" crea las bases de datos correspondientes y a cada una de ellas les inserta las tablas y datos de ejemplo en ellas. Previamente a la ejecución del mismo debemos de darle los permisos adecuados de ejecución con el siguiente comando:

```
chmod +x bbddsetup.sh
```

Para la ejecución del mismo una vez concedidos los permisos hay que ejecutar el siguiente comando:

```
./bbddsetup.sh
```

3. Despliegue de los servidores REST

Una vez iniciadas las bases de datos, podemos desplegar los servidores. Para ello hay que hacerlo desde el directorio del proyecto y cada uno en un terminal distinto. En primer lugar, debemos de desplegar el servidor asociado a un parking por lo que tenemos que ubicarnos en su directorio correspondiente:

```
cd parking1
```

Una vez en esta ubicación podemos iniciar Spring con el siguiente comando:

```
mvn spring-boot:run
```

En segundo lugar, en otro terminal debemos de desplegar el servidor asociado al centro de gestión de parkings por lo que tenemos que ubicarnos en su directorio correspondiente:

```
cd centro_gestion
```

Una vez en esta ubicación podemos iniciar Spring con el siguiente comando:

```
mvn spring-boot:run
```

ANEXO B: CONFIGURACIÓN DE LA API DE PAYPAL Y LA API DE GOOGLE MAPS

En diversos apartados anteriores de esta memoria se ha detallado cómo se usa la API de PayPal para el pago de tarifas a través del dispositivo móvil y el uso de la API de Google Maps para la visualización de información de los distintos parkings de la red. A continuación, se va a detallar la configuración de ambas APIs.

1. Configuración de la API de PayPal

En este apartado, se explicará la configuración [15] que se debe realizar en la API de PayPal para su uso en las aplicaciones. En primer lugar, se debe crear una cuenta personal en la plataforma PayPal Developer para poder gestionar tus aplicaciones. Posteriormente, se debe crear un usuario ficticio para poder probar la aplicación. Para ello, en el apartado Sandbox Accounts se debe pulsar sobre el botón "Create account" y crear una cuenta de tipo Business, ilustración B-1.

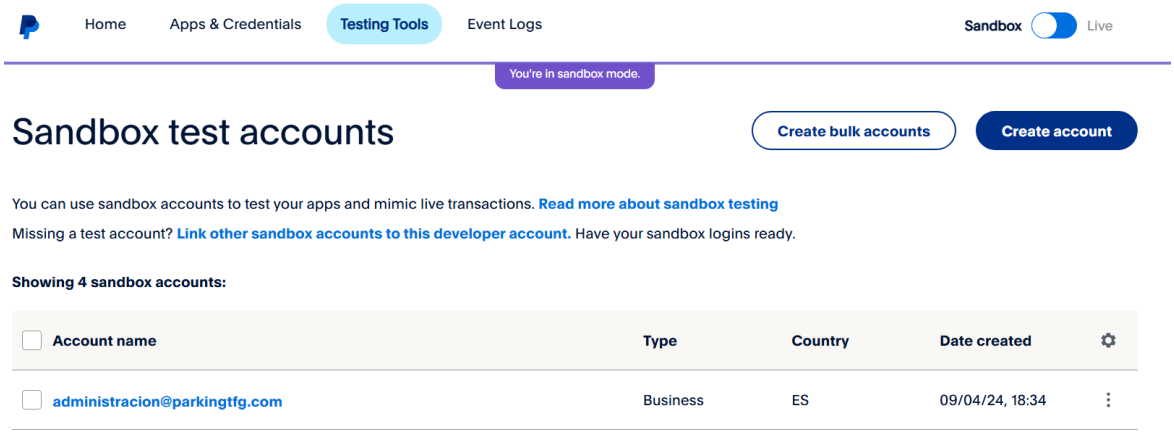

Ilustración B-1. Configurar usuario en PayPal Developer

Además, se puede crear una cuenta de tipo Personal para hacer uso de la aplicación, en esta cuenta se puede indicar un saldo ficticio para en futuras ocasiones realizar pagos en la aplicación y probar su uso.

Posteriormente, en el apartado Apps & Credentials se debe pulsar sobre el botón "Create App" para crear un nuevo proyecto rellenando los campos correspondientes y asociando esta aplicación con la cuenta creada anteriormente. Una vez configurada la aplicación, se generan unas credenciales de cliente que contienen un identificador de cliente y una clave secreta que se deben guardar para su posterior uso, ilustración B-2.

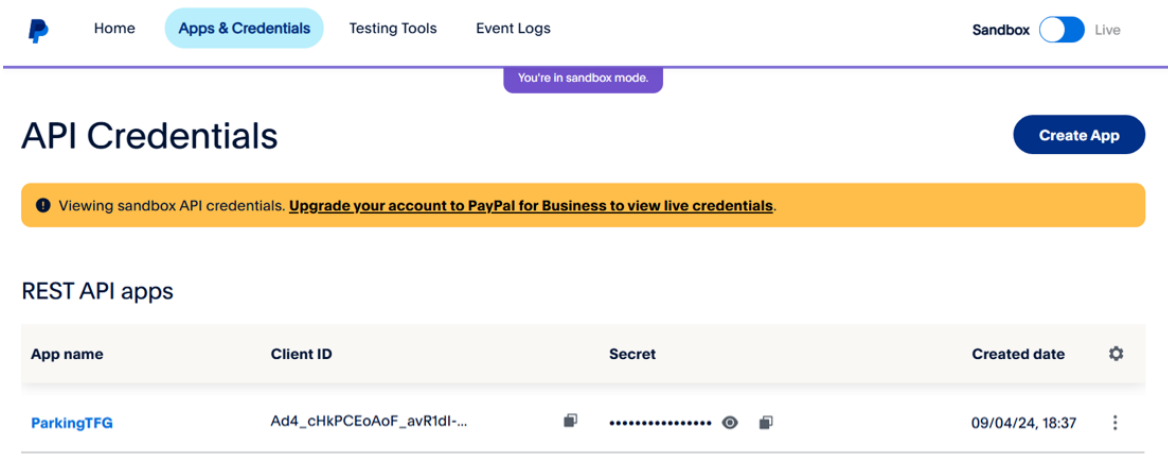

Ilustración B-2. Configurar aplicación en PayPal Developer

Una vez realizados los pasos detallados anteriormente, para que podamos hacer uso de la API en nuestro proyecto, debemos de obtener el token de acceso. Para ello, se debe realizar una solicitud POST a la URL https://api-m.sandbox.paypal.com/v1/oauth2/token con las credenciales de cliente generadas anteriormente con el formado "CLIENT_ID:CLIENT_SECRET. Se debe establecer el encabezado Content-Type a application/x-www-form-urlencoded y finalmente, definir el cuerpo de la solicitud para indicar que se está solicitando un token de acceso basado en las credenciales del cliente (grant_type=client_credentials).

Una vez obtenido el token de acceso se puede usar en la aplicación para realizar futuras solicitudes a la API de PayPal, permitiendo así a la aplicación realizar operaciones como pagos, reembolsos y consultas de transacciones de manera segura y autorizada.

2. Configuración de la API de Google Maps

En este apartado, se detallará la configuración [16] que se debe realizar para usar esta API de Google Maps. En primer lugar, se debe crear un proyecto en Android Studio que use como lenguaje Java. Se debe seleccionar una versión superior a la mínima requerida por el SDK de Maps para Android, en este caso, superior a la versión 18.0.x. Actualmente se corresponde con la API nivel 19 (Android 4.4). Una vez creado el proyecto, se debe tener presente el archivo AndroidManifest.xml ya que es el archivo de configuración de nuestra aplicación móvil.

El paso siguiente consiste en crear una cuenta en la plataforma Google Cloud y crear un proyecto. El uso de esta plataforma es gratuito siempre y cuando no se exceda un límite. Como paso posterior, debemos habilitar en la plataforma la API de Google Maps para Android y debemos tener un resultado como el mostrado en la ilustración B-3:

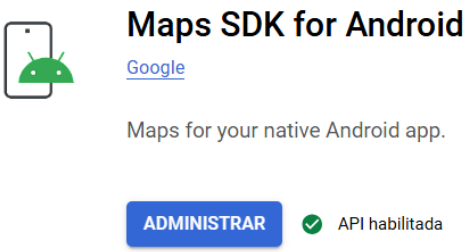

Ilustración B-3. Habilitar API Google Maps para Android

Una vez habilitada la API, debemos dirigirnos a la pestaña de “Credenciales” y pulsar sobre el botón de “Crear credenciales” para crear una clave única que nos permitirá hacer uso de esta API. Finalmente, una vez creada tenemos que guardar la clave para su posterior uso. La vista de este paso se muestra en la siguiente ilustración B-4:

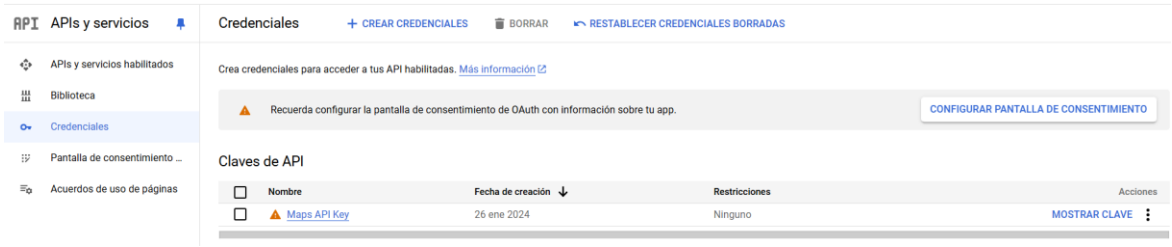

Ilustración B-4. Creación de clave para API Google Maps

Como paso siguiente y una vez obtenida la clave, abrir el archivo build.gradle a nivel del proyecto y agregar el siguiente código mostrado en la ilustración B-5.

```
plugins {
    // ...
    id 'com.google.android.libraries.mapsplatform.secrets-gradle-plugin' version '2.0.1' apply false
}

dependencies {
    implementation 'com.google.android.gms:play-services-maps:18.1.0'
    // ...
}
```

Ilustración B-5. Código build.gradle a nivel de proyecto para API Google Maps

A continuación, abrir el archivo build.gradle a nivel del módulo y agregar el código mostrado en la ilustración B-6 dentro del elemento plugins.

```
id 'com.google.android.libraries.mapsplatform.secrets-gradle-plugin'
```

Ilustración B-6. Código build.gradle a nivel del módulo para API Google Maps

Una vez añadidos estos códigos anteriores, se debe guardar el proyecto y sincronizar con Gradle. Finalmente, en el archivo AndroidManifest.xml debemos agregar dentro de application la línea siguiente mostrada en la ilustración B-7 en la que debemos sustituir MAPS_API_KEY por la clave que hemos generado previamente.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="MAPS_API_KEY" />
```

Ilustración B-7. Código AndroidManifest.xml para API Google Maps

Una vez seguidos los pasos descritos anteriormente, ya tenemos configurada la aplicación Android para el uso de la API de Google Maps. Como pasos posteriores, podemos tomar plantillas que nos proporciona Android Studio para crear la vista del mapa y administrarlo.

REFERENCIAS

[1] The National ITS Reference Architecture, "Home" [Online]. Available: https://www.arc-it.net/index.html.

[2] HP Omen, «Especificaciones del producto» [En línea]. Available: https://support.hp.com/co-es/document/c05489251.

[3] Android Studio, «Home | Android Studio» [En línea]. Available: https://developer.android.com/studio?hl=es-419.

[4] Ubuntu, «Home | Ubuntu» [En línea]. Available: https://ubuntu.com/.

[5] Spring, «Spring | Home» [En línea]. Available: https://spring.io/.

[6] Postman, «Postman | Home» [En línea]. Available: https://www.postman.com/.

[7] PostgreSQL, «Home | PostgreSQL» [En línea]. Available: https://www.postgresql.org/.

[8] Visual Studio Code, «Home | VSC» [En línea]. Available: https://code.visualstudio.com/.

[9] Wireshark, «Home | Wireshark» [En línea]. Available: https://www.wireshark.org/.

[10] Google Developers, «Home | Google Maps Platform» [En línea]. Available: https://developers.google.com/maps?hl=es-419.

[11] PayPal Developer, «Home | PayPal Developer» [En línea]. Available: https://developer.paypal.com/home.

[12] Swagger, «Home | Swagger» [En línea]. Available: https://swagger.io/.

[13] Spring Security, «Home | Spring Security» [En línea]. Available: https://spring.io/projects/spring-security.

[14] FIWARE, «Smart Data Models - FIWARE» [En línea]. Available: https://www.fiware.org/smart-data-models/.

[15] PayPal Developer, «Get Started with PayPal REST API» [En línea]. Available: https://developer.paypal.com/api/rest/.

[16] Google Maps Platform, «Guía de inicio rápido del SDK de Maps para Android» [En línea]. Available: https://developers.google.com/maps/documentation/android-sdk/start.