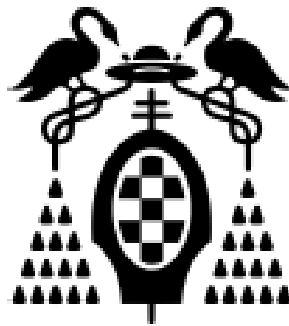


Computación Ubicua

*Sistema de Reserva de Parkings Inteligente
(UBIPARK)*

Grado en Ingeniería Informática
Universidad de Alcalá



Adrián Ajo Matarredona
David Fernández Sanz
Javier José Guzmán Rubio
Carlos Javier Prado Vázquez

Junio 2023-2024

Índice

Índice	2
1. Introducción	5
2. Análisis del problema	5
3. Objetivos y alcance del proyecto	8
4. Ideas descartadas	8
5. Desarrollo	12
5.1. Tecnología a utilizar	12
5.2. Arquitectura del proyecto	13
5.2.1. Capa de percepción.....	13
5.2.2. Capa de transporte	35
5.2.3. Capa de procesado.....	35
5.2.4. Capa de aplicación.....	42
5.3. Plan de desarrollo	54
6. Conclusiones.....	55
7. Bibliografía.....	56
Anexo I – Manual de instalación	56
Anexo II – Manual de Usuario de la aplicación Web	60
Anexo III – Simulación de datos	66
Anexo V – Hojas de características de los componentes	68

Índice de Figuras

- Figura 1. Sensor Temperatura, 9*
Figura 10. Sensor HC-SR04, 14
Figura 11. Semáforo LED, 14
Figura 12. Pantalla LCD, 15
Figura 13. Keypad, 16
Figura 14. Lector RFID, 16
Figura 15. Protoboard, 17
Figura 16. Logo Arduino, 18
Figura 17. ESP-32, 18
Figura 18. Programa de Arduino del primer ESP-32, 26
Figura 19. Programa de Arduino del segundo ESP-32, 30
Figura 2. Sensor de infrarrojos, 9
Figura 20. Programa de Arduino del tercer ESP-32, 33
Figura 21. Topic Entrada, 35
Figura 22. Topic Salida, 35
Figura 23. Topic estado de la plaza, 36
Figura 24. Topic Pago, 36
Figura 25. Conexión a la Base de Datos, 37
Figura 26. Procesar nuevas reservas. Véase
Figura 27. Flujo Broker MQTT, 38
Figura 28. Estructura carpetas Tomcat, 38
Figura 29. Esquema E/R de nuestra BBDD, 40
Figura 3. Sensor de presión, 10
Figura 30. Carpetas front, 42
Figura 31. Carpetas back, 43
Figura 32. Carpetas Java, 43
Figura 33. Pool de conexiones, 43
Figura 34. Ejemplo prepared statement, 44
Figura 35. Carpetas database, 44
Figura 36. Carpetas logic, 44
Figura 37. Función de logic, 45
Figura 38. Project Initializer, 46
Figura 39. Configuración Logs, 46
Figura 4. Sensor de presencia, 10
Figura 40. Esquema de Home, 48
Figura 41. Esquema de Login, 49
Figura 42. Esquema de Register, 49
Figura 43. Esquema de Register Admin, 50
Figura 44. Esquema de Reservar una plaza, 50
Figura 45. Esquema de las reservas del usuario, 51
Figura 46. Esquema de Información, 51
Figura 47. Esquema de información generada, 52
Figura 48. Esquema de estadísticas de Ubipark, 52
Figura 49. Mosquitto config, 56
Figura 5. Placa Arduino, 11
Figura 50. Servicio Mosquitto, 56
Figura 51. CMD, 57
Figura 52. Dirección de los archivos de Mosquitto, 57
Figura 53. Pruebas con Mosquitto, 58
Figura 54. Home.html, 58
Figura 55. Registro exitoso, 59
Figura 56. Login, 59
Figura 57. Failed login, 60
Figura 58. Account information, 60
Figura 59. Confirm, 60
Figura 6. Raspberry Pi, 11
Figura 60. Eliminar Reserva, 60
Figura 61. Reservar plaza, 61
Figura 62. Admin Login, 61
Figura 63. Edit Users, 62
Figura 64. Generar Usuario, 62
Figura 65. Random Data, 63
Figura 66. Generar email, 63
Figura 67. CódigoServlet generarUsuario, 64
Figura 7. Logo MariaDB, 12
Figura 8. Logo de Android, 12
Figura 9. Capas, 13

Resumen

Nuestro proyecto integra diversas tecnologías de hardware y software para crear un sistema inteligente de gestión y control de parkings y de estacionamiento. Utilizando placas de desarrollo como el ESP32 y sensores como el HC-SR04 y el lector RFID MFRC522, hemos diseñado un sistema que monitorea la ocupación de los espacios de estacionamiento. El sistema detecta si un espacio está reservado y si un vehículo está presente, utilizando un sensor ultrasónico para medir la distancia y un lector RFID para validar tarjetas. A través del LCD y del keypad se gestionan las entradas y salidas de vehículos al parking, así como el pago mediante el sensor RC522. Toda la información captada por los sensores se comunica mediante el protocolo MQTT con una página web y se almacena en una base de datos Azure SQL, permitiendo la monitorización y gestión en tiempo real. Los usuarios pueden ver el estado actual del estacionamiento y realizar reservas a través de la página web, que está conectada al broker MQTT. Todos los datos son accesibles y seguros gracias a la integración con Azure SQL. Este proyecto muestra cómo el Internet de las cosas (IoT) puede resolver problemas cotidianos como encontrar un espacio de estacionamiento disponible y gestionar la entrada y salida de un parking. La implementación y configuración del sistema han seguido el plan establecido, demostrando la capacidad de manejar componentes de hardware y servicios de nube.

Palabras clave: Estacionamiento, Arduino, Servidor, Sensores, Arquitectura 4 capas.

1. Introducción

Ubipark es una empresa dedicada al control y gestión de parkings, ofreciendo soluciones innovadoras para facilitar el proceso de aparcamiento. Este proyecto implementa un sistema de gestión de parkings utilizando tres microcontroladores ESP32, diferentes sensores como ultrasonido, un lector de tarjetas RFID, un LCD, un keypad y un semáforo de 3 leds (verde, rojo y amarillo), el proyecto también incluye una página web interactiva, todo interconectado a través del protocolo MQTT.

Cuando un cliente llega al parking, interactúa con un parquímetro que le presenta un menú con opciones para entrar o salir del parking. Si el cliente selecciona la opción de entrada (pulsando 1), deberá introducir la matrícula de su vehículo. Esta información se envía a la página web mediante MQTT, iniciando el conteo del tiempo para calcular el importe a cobrar al abandonar el parking.

Cada plaza del estacionamiento tiene un sensor ultrasónico que detecta la presencia de un vehículo. Basado en esta información, un semáforo en la plaza cambia de color en función de si está ocupado o libre. Los clientes también pueden reservar una plaza de garaje en el horario que deseen registrándose en la página web de Ubipark. Una vez completada la reserva, el semáforo de la plaza elegida se pondrá amarillo durante el período de tiempo seleccionado.

Cuando el cliente desea abandonar el parking, debe volver al parquímetro de Ubipark y seleccionar la opción de salida (pulsando 2). Luego, introduce nuevamente su matrícula para recibir el importe a pagar, que se debe abonar mediante tarjeta de crédito usando el lector RC-522 del parquímetro.

Este sistema integrado mejora significativamente la eficiencia y comodidad en la gestión de parkings, utilizando tecnología avanzada para ofrecer un servicio automatizado y fácil de usar.

2. Análisis del problema

Contexto y Necesidad

En el mundo actual, la gestión eficiente de los espacios de estacionamiento es crucial tanto para los usuarios como para los propietarios de los parkings. Con el aumento de la urbanización y el número de vehículos, encontrar un lugar para estacionar puede ser una tarea desafiante y a menudo estresante. Un sistema avanzado de gestión de parkings que controle las entradas y salidas, y que proporcione información en tiempo real sobre la disponibilidad de plazas, puede solucionar muchos de estos problemas.

Objetivos del Sistema

El sistema propuesto tiene varios objetivos clave:

- **Control de Entradas y Salidas:** Monitorizar y registrar las entradas y salidas de vehículos para llevar un control preciso de la ocupación del parking en tiempo real.
- **Indicadores Visuales de Disponibilidad:** Proveer a los usuarios información visual inmediata sobre la disponibilidad de plazas, reduciendo así el tiempo y la frustración del usuario con la búsqueda de espacios libres.
- **Reservas Online y Remotas:** Permitir a los usuarios reservar plazas de estacionamiento de forma online evitando la necesidad de verificar la disponibilidad de manera presencial.
- **Estadísticas para Propietarios:** Generar informes detallados y estadísticas útiles para los propietarios del parking, como las horas de mayor flujo de coches y la recaudación diaria, ayudando a optimizar la gestión y rentabilidad del espacio.

Análisis de mercado

Descripción del Producto

El sistema de control y gestión de parkings incluye sensores de ultrasonidos para detectar la ocupación de plazas, indicadores visuales de disponibilidad, una plataforma de reservas online y un módulo de análisis de datos. Este sistema tiene como objetivo mejorar la experiencia de los usuarios y optimizar la gestión de los parkings.

El principal mercado objetivo para este sistema se puede dividir en las siguientes categorías:

1. **Parkings Públicos Urbanos:** Localizados en áreas metropolitanas con alta densidad de tráfico y escasez de espacios de estacionamiento.
2. **Centros Comerciales:** Grandes superficies que necesitan gestionar un alto volumen de vehículos de forma eficiente.

Análisis de la Competencia

El mercado de sistemas de gestión de parkings es competitivo y está en crecimiento. A continuación, se describen algunos competidores y sus características:

1. **ParkMe:** Ofrece una aplicación móvil para encontrar y reservar plazas de estacionamiento, con un fuerte enfoque en datos de disponibilidad en tiempo real.
2. **SpotHero:** Se especializa en la reserva de estacionamiento a través de una plataforma en línea, con una amplia red de parkings afiliados.
3. **ParkMobile:** Proporciona soluciones de pago y reserva de estacionamiento mediante una app, integrada con servicios municipales en varias ciudades.

Análisis DAFO

Fortalezas	Debilidades
<ul style="list-style-type: none">● Proporcionamos un sistema que combina detección, visualización, reserva y análisis en una sola solución.● Mejora la experiencia del usuario reduciendo el tiempo de búsqueda de una plaza y pudiendo reservar de forma online.● Utilización de tecnologías accesibles y económicas haciendo pudiendo hacer una instalación económica.	<ul style="list-style-type: none">● Requiere mantenimiento y actualización constante para asegurar su correcto funcionamiento.
Oportunidades	Amenazas
<ul style="list-style-type: none">● Somos un equipo joven con ilusión y ganas de llevar un proyecto como este.● Existe una creciente demanda de soluciones ubicuas en el sector● Integración de nuestro sistema de gestión de parkings en proyectos de ciudades inteligentes.	<ul style="list-style-type: none">● La presencia de múltiples competidores con soluciones similares puede dificultar la diferenciación.● Carecemos de experiencia previa lo que puede generar dudas a la hora de contratar nuestro servicio

Conclusión

El sistema de control y gestión de parkings tiene un gran potencial de mercado debido a la creciente demanda de soluciones de estacionamiento eficientes y la tendencia hacia la digitalización de servicios urbanos, así como de las ciudades inteligentes.

3. Objetivos y alcance del proyecto

Objetivo General: Desarrollar e implementar un sistema de control y gestión de parkings que optimice la utilización de plazas de estacionamiento, mejore la experiencia del usuario y proporcione herramientas de análisis y gestión para los propietarios.

Objetivos Específicos:

1. **Monitorización en Tiempo Real:** Implementar sensores de ultrasonidos para detectar la ocupación de plazas de estacionamiento en tiempo real.
2. **Indicadores Visuales de Disponibilidad:** Instalar luces LED que indiquen visualmente la disponibilidad de las plazas (verde para libre, rojo para ocupado y amarillo para reservas activas).
3. **Plataforma de Reservas Online:** Desarrollar una plataforma web y una aplicación móvil que permitan a los usuarios reservar plazas de estacionamiento de forma remota.
4. **Generación de Estadísticas y Análisis:** Crear herramientas de análisis de datos que proporcionen a los propietarios informes sobre la ocupación, los patrones de uso y la recaudación diaria.

Usuarios Finales:

- Conductores que buscan una experiencia de estacionamiento menos frustrante y más eficiente sin tener que perder el tiempo buscando un aparcamiento cuando llegas a tu destino.

Propietarios y Administradores de Parkings:

- Administradores de parkings públicos y privados que buscan optimizar la gestión de sus espacios, además de obtener información en tiempo real del parking.
- Propietarios de parkings urbanos, centros comerciales o zonas con un tráfico muy concurrido que requieran de una optimización del aparcamiento.

4. Ideas descartadas

En esta sección enumeramos las ideas descartadas, así como los motivos de que no hayan sido implementadas.

- **Sensor de temperatura:** la idea era utilizar el sensor de temperatura para saber cuando una plaza está ocupada.

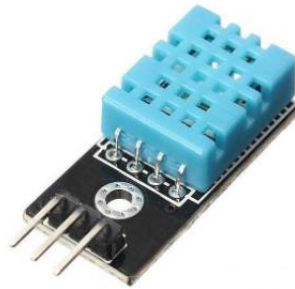


Figura 1. Sensor Temperatura

La idea fue descartada ya que podría no ser muy efectiva.

- **Sensor de infrarrojos:** la idea era detectar la presencia de un vehículo en la plaza mediante este sensor.

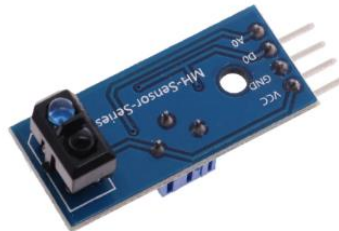


Figura 2. Sensor de infrarrojos

Se descartó al ser poco efectivo para lo que requiere nuestro proyecto.

- **Sensor de presión:** se planteó la idea de utilizar este sensor para saber si la plaza estaba ocupada.



Figura 3. Sensor de presión

Se descartó la idea al no ser muy certero.

- **Sensor de presencia:** Se utilizará colocándolo en cada uno de las plazas para detectar si la presencia del vehículo en la plaza.



Figura 4. Sensor de presencia

Se descartó ya que el uso de un sensor de ultrasonidos es más eficiente y más manejable.

- **Placa Arduino:** no se ha utilizado una placa Arduino. Si escogemos una placa estándar Arduino (por ejemplo, Arduino Zero), veremos que no dispone de módulo wi (la mayoría) o bluetooth. Además de que requieren el uso de un escudo ethernet adicional. En cambio, una ESP32, dispone de un módulo Wi. Arduino Zero utiliza un chip ATSAMD21G18 con una frecuencia de reloj de 48 MHz. En comparación, el Esp32 funciona con un chip de microprocesador Tensilica Xtensa LX6 con una frecuencia de reloj de entre 160 y 240 MHz. Finalmente, otra diferencia radica en el precio.



Figura 5. Placa Arduino

- **Raspberry Pi:** se descartó la idea, dado que, para el correcto funcionamiento de contenidos web amplios y dinámicos, el rendimiento normal de una R-Pi es insuficiente. Ya que este pequeño ordenador es apto, más bien, como entorno de prueba local para páginas web, aunque las páginas estáticas sencillas que no tienen muchas visitas también pueden alojarse en una R-Pi. Otro factor es el económico, dado que no hace falta el gasto en una R-Pi, sabiendo que podemos hacer el uso de nuestros propios equipos portátiles. Aunque cabe la idea de poner en marcha en futuras implementaciones con el uso de estos miniordenadores.



Figura 6. Raspberry Pi

- **MariaDB:** Al principio planteamos construir la base de datos con MariaDB, pero finalmente hemos optado por utilizar Azure SQL.



Figura 7. Logo MariaDB

- **Aplicación Android:** está idea ha sido descartada ya que, para el tipo de negocio desarrollado, no tendría mucho éxito una aplicación y es más accesible y eficiente para el cliente la página web.



Figura 8. Logo de Android

5. Desarrollo

En este apartado se aportará todo el estudio realizado para abordar el desarrollo del proyecto. El contenido se divide en la siguientes subsecciones:

5.1. Tecnología a utilizar

Durante este apartado abordaremos el cómo se construirá nuestro proyecto, cómo será el diseño de su funcionamiento, que tipo de arquitectura emplearemos.

5.2. Arquitectura del proyecto

Antes de comenzar a desarrollar qué tecnologías van a hacer funcionar nuestro proyecto y cómo será este funcionamiento, debemos elegir una arquitectura que se adapte correctamente a este. Una arquitectura se define como “estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos”. Por la forma de nuestro proyecto hemos considerado que la mejor arquitectura que se adapta es una arquitectura en 4 capas.

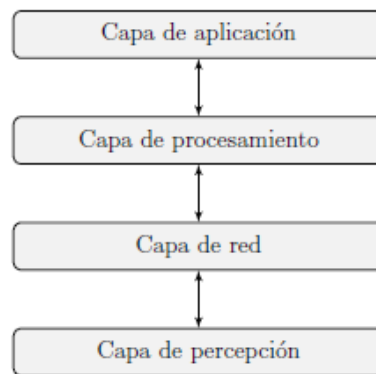


Figura 9. Capas

5.2.1. Capa de percepción

Es en esta capa es donde se encuentran todos los sensores, actuadores, y controladores que nos permitirán comunicarnos con el entorno. En nuestro caso la capa de percepción se compone de diferentes sensores como son ultrasonidos, LCD, keypad, lector de tarjetas RFID y semáforo de leds; que recogen información de los diferentes vehículos que entran al parking y del estado de ocupación de las plazas del mismo. Los sensores de ultrasonido detectarán si la plaza está ocupado o no, el semáforo de leds mostrará rojo en caso de que sea una plaza ocupada, verde si está libre y amarilla si es una plaza que ha sido reservada, el LCD y el keypad permiten al cliente interactuar con el parquímetro, donde deberán introducir la matrícula para entrar y salir al parking y se les mostrará el importe en función del tiempo que hayan permanecido en el parking, por último el lector de tarjetas RFID permite a los clientes efectuar el pago.

Sensores

Analizar los sensores que componen la capa de percepción y qué es lo que aportan al sistema.

Para detectar si una plaza está ocupada, hemos hecho uso del sensor de ultrasonidos HC-SR04. El principal motivo, es que por un precio reducido podemos resolver el problema de detectar si hay un vehículo en la plaza.



Figura 10. Sensor HC-SR04

Tenemos los pines de corriente (VCC y GND), y para llevar a cabo su funcionalidad tenemos los de Trig y Echo, por donde se envía el pulso, y por donde se recibe. Dependiendo del tiempo que tarda en volver la señal podremos calcular la distancia.

Para indicar el estado de ocupación de la plaza al usuario, hemos usado el semáforo de leds AZDelivery.



Figura 11. Semáforo LED

Cuenta con un pin GND, y otros tres que permiten seleccionar los colores a encender. Los elegimos ya que vienen con resistencias integradas y con un mismo sensor podemos indicar fácilmente al usuario el estado de la plaza.

- Ocupado
- Libre
- Reservado

Para la interacción del parquímetro con el usuario se ha hecho uso de un LCD.

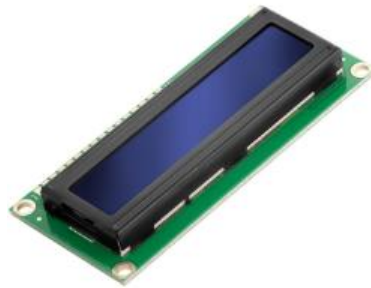


Figura 12. Pantalla LCD

El LCD cuenta con:

- **Pines de corriente (VCC y GND):** Estos pines proporcionan la alimentación eléctrica necesaria para el funcionamiento del LCD. VCC se conecta a la fuente de alimentación, mientras que el GND se conecta a tierra.
- **Pines de control (RS, RW y E):**
 - RS (Register Select): Este pin indica al LCD si los datos enviados se refieren a un comando (como limpiar la pantalla o configurar el cursor) o a caracteres a mostrar en la pantalla.
 - RW (Read/Write): Este pin determina si se está leyendo desde el LCD o escribiendo en él. En la mayoría de los casos, se configura en modo de escritura (RW conectado a tierra) para enviar datos al LCD.
 - E (Enable): Este pin habilita la lectura o escritura de datos. Un pulso de alto nivel en este pin le indica al LCD que los datos en el bus de datos son válidos y deben ser procesados.

Para que el usuario pueda introducir la matrícula en la entrada y la salida al parking, se ha utilizado un keypad.



Figura 13. Keypad

- **Pines de filas y columnas:**
 - Las filas (R1-R4) se conectan a pines de entrada/salida del microcontrolador y se configuran como salidas.
 - Las columnas (C1-C4) se conectan a pines de entrada/salida del microcontrolador y se configuran como entradas con resistencias de pull-up.

Cuando se presiona una tecla en el teclado, se cierra el circuito entre una fila y una columna específicas. El microcontrolador puede detectar qué tecla se ha presionado escaneando secuencialmente cada fila y comprobando el estado de las columnas. Con esta información, se puede determinar qué tecla se ha presionado y tomar la acción correspondiente en el programa.

Para que el usuario pueda efectuar el pago una vez vaya a abandonar el parking, se ha hecho uso de un lector de tarjetas RFID (Radio-Frequency Identification) RC-522.



Figura 14. Lector RFID

El lector RC-522 consta de varios pines que se utilizan para la conexión y comunicación con el microcontrolador:

- **Pines de corriente (VCC y GND):** Estos pines proporcionan la alimentación eléctrica necesaria para el funcionamiento del lector. VCC se conecta a la fuente de alimentación de 3.3V o 5V, mientras que GND se conecta a tierra.
- **Pines de datos (MISO, MOSI, SCK y SS):**
 - MISO (Master In Slave Out): Este pin se utiliza para la transmisión de datos desde el lector al microcontrolador.
 - MOSI (Master Out Slave In): Este pin se utiliza para la transmisión de datos desde el microcontrolador al lector.
 - SCK (Serial Clock): Este pin se utiliza para sincronizar la comunicación entre el lector y el microcontrolador.
 - SS (Slave Select): Este pin se utiliza para seleccionar el lector dentro de un bus SPI múltiple.
- **Pines de control (RST y IRQ):**
 - RST (Reset): Este pin se utiliza para restablecer el lector a su estado predeterminado.
 - IRQ (Interrupt Request): Este pin puede utilizarse para generar interrupciones en el microcontrolador cuando se produce una acción específica en el lector, como la detección de una tarjeta.

Todos estos componentes electrónicos mencionados deben ir conectados a una protoboard para trabajar de forma cómoda.

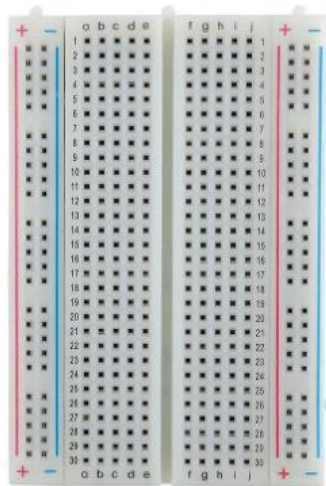


Figura 15. Protoboard

Para desarrollar el software que controla todos estos componentes mencionados, hemos hecho uso del IDE de Arduino.



Figura 16. Logo Arduino

Utilizamos este pues tenemos las ventajas de que nos permite trabajar con nuestro ESP-32 como si de una placa de Arduino se tratara, lo que además implica que estaremos programando con los lenguajes C/C++, lo que es muy interesante, pues son lenguajes de alto nivel que ya manejamos y cuentan con diversas librerías ya listas para manejar los puertos y otros periféricos de manera sencilla.

ESP-32

El núcleo del proyecto es el ESP-32 que es el hardware donde irán conectados todos los sensores y actuadores.

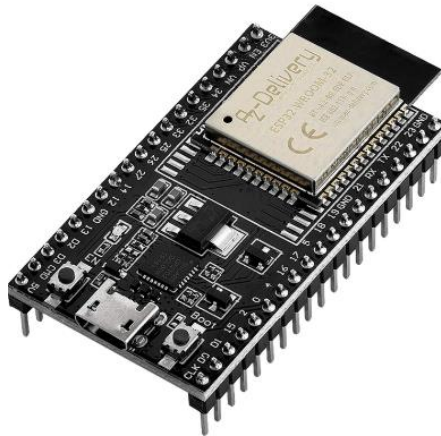


Figura 17. ESP-32

La hemos elegido por diversos motivos:

- Relación calidad-precio, pues nos ofrece montones de cosas por un precio razonable (12 euros) para un proyecto de estas características.
- Compatibilidad con el IDE de Arduino, lo que implica poder programar en C.
- Módulo Wi-Fi integrado, nos ayudará a poder enviar a nuestro servidor todos los datos que recojamos.
- Cuenta con mas de 30 pines entre analogicos y digitales (nuestro proyecto hara uso de
- ambos), para poder conectar todo aquello que necesitemos.
- Podemos alimentarlo mediante un puerto USB o aplicando tensión a sus pines, lo que es algo sencillo y además no consume mucho.

En concreto se han hecho uso de tres placas de desarrollo ESP-32.

El primer ESP-32 es el encargado de gestionar el LCD y el keypad. Para ello hemos desarrollado el siguiente código en arduino.

```
#include <Keypad.h>
#include <LiquidCrystal.h>
#include <PubSubClient.h>
#include <WiFi.h>
#include <ArduinoJson.h> // Librería para manejar JSON

const byte ROW_NUM = 4; // four rows
const byte COLUMN_NUM = 4; // four columns
const char *ssid = "UBICUA";
const char *wifi_password = "cjp2001";
const char *mqtt_broker = "192.168.19.117"; // broker address// define topic
const char *mqtt_username = ""; // username for authentication
const char *mqtt_password = ""; // password for authentication
const int mqtt_port = 1883;
const char *Parking_maquina_entrada = "Ubipark/Parking1/Maquina/Entrada";
const char *Parking_maquina_salida = "Ubipark/Parking1/Maquina/Salida";
const char *Parking_maquina_respuesta = "Ubipark/Parking1/Maquina/Respuesta";
const char *Parking_maquina_tarjeta = "Ubipark/Parking1/Maquina/Tarjeta";

WiFiClient espClient;
PubSubClient client(espClient);

// Configuración de los pines de la LCD a los pines del ESP32
LiquidCrystal lcd(22, 23, 5, 18, 19, 21);
```

```

char keys[ROW_NUM][COLUMN_NUM] = {
  { '1', '2', '3', 'A' },
  { '4', '5', '6', 'B' },
  { '7', '8', '9', 'C' },
  { '*', '0', '#', 'D' }
};

byte pin_rows[ROW_NUM] = { 16, 32, 14, 27 }; // Pines de las filas conectados al ESP32
byte pin_column[COLUMN_NUM] = { 26, 17, 13, 15 }; // Pines de las columnas conectados al ESP32

Keypad keypad = Keypad(makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM);

String matricula = ""; // Para almacenar la matrícula
int tiempoEnParking = 0; // Para almacenar el tiempo en el parking
bool esperandoRespuesta = false;
bool esperandoTarjeta = false;
unsigned long tiempoSinRespuesta = 0;
const char* id_parking = "1";

void setup() {
  Serial.begin(9600);
  lcd.begin(16, 2); // Inicializa la pantalla LCD con 16 columnas y 2 filas
  lcd.clear(); // Limpia cualquier cosa que pueda estar en la pantalla

  WiFi.begin(ssid, wifi_password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
  }
  Serial.println("Connected to the WiFi network");

  // Conectando al broker MQTT
  client.setServer(mqtt_broker, mqtt_port);
  client.setCallback(callback);
  while (!client.connected()) {
    String client_id = "Arduino";
    client_id += String(WiFi.macAddress());
    Serial.printf("The client %s connects to the public mqtt broker\n", client_id.c_str());
    if (client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {
      Serial.println("Public emqx mqtt broker connected");
      client.subscribe(Parking_maquina_respuesta);
      client.subscribe(Parking_maquina_tarjeta);
    } else {

```

```

        Serial.print("failed with state ");
        Serial.print(client.state());
        delay(2000);
    }
}

void loop() {
    client.loop();
    if (!esperandoRespuesta) {
        displayMenu(); // Muestra el menú
    }

    char key = keypad.getKey(); // Lee la tecla presionada

    if (key && !esperandoRespuesta) { // Si se presionó una tecla y no se está esperando una respuesta
        Serial.println(key); // Imprime la tecla en el monitor serial
        lcd.clear(); // Limpia la pantalla LCD

        if (key == '1') {
            handleEntradaParking(); // Maneja la opción de Entrada Parking
        } else if (key == '2') {
            handleSalidaParking(); // Maneja la opción de Salida Parking
        } else {
            lcd.setCursor(0, 0); // Establece el cursor en la columna 0, fila 0
            lcd.print("Opcion invalida");
            delay(2000); // Espera 2 segundos para mostrar el mensaje
        }
    }
}

void displayMenu() {
    lcd.setCursor(0, 0); // Establece el cursor en la columna 0, fila 0
    lcd.print("1: Entrada");
    lcd.setCursor(0, 1); // Establece el cursor en la columna 0, fila 1
    lcd.print("2: Salida");
}

void handleEntradaParking() {
    lcd.clear(); // Limpia la pantalla LCD
    lcd.setCursor(0, 0); // Establece el cursor en la columna 0, fila 0
    lcd.print("Introduce mat:");
    lcd.setCursor(0, 1); // Establece el cursor en la columna 0, fila 1
    matricula = ""; // Limpia la variable de matrícula

    while (true) {
        char key = keypad.getKey(); // Lee la tecla presionada
        if (key) { // Si se presionó una tecla
            Serial.println(key); // Imprime la tecla en el monitor serial
            if (key == '#') { // Si se presionó la tecla de confirmación (por ejemplo '#')

```

```

        if (isMatriculaValida(matricula)) {
            break; // Sale del bucle
        } else {
            lcd.clear();
            lcd.setCursor(0, 0);
            lcd.print("Matricula invalida");
            delay(2000);
            lcd.clear();
            lcd.setCursor(0, 0);
            lcd.print("Introduce mat:");
            lcd.setCursor(0, 1);
            lcd.print(matricula);
        }
    } else if (key == '*') { // Si se presionó la tecla de borrar (por ejemplo '*')
        if (matricula.length() > 0) {
            matricula.remove(matricula.length() - 1); // Borra el último carácter
            lcd.clear();
            lcd.setCursor(0, 0);
            lcd.print("Introduce mat:");
            lcd.setCursor(0, 1);
            lcd.print(matricula);
        }
    } else if (key == 'D') { // Si se presionó la tecla para regresar al menú
        lcd.clear();
        displayMenu();
    }

    return; // Sal del bucle y retorna al menú
} else {
    matricula += key; // Añade la tecla presionada a la matrícula
    lcd.setCursor(0, 1); // Establece el cursor en la columna 0, fila 1
    lcd.print(matricula); // Muestra la matrícula en la LCD
}
}
}

// Aquí puedes agregar el código que maneja la matrícula ingresada
Serial.print("Matricula introducida: ");
Serial.println(matricula);
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Matricula guardada");
StaticJsonDocument<200> doc;
doc["id_parking"] = id_parking;
doc["matricula"] = matricula;

char jsonBuffer[256];
serializeJson(doc, jsonBuffer);

// Publicar el mensaje JSON
client.publish(Parking_maquina_entrada, jsonBuffer);
delay(2000);

```

```

    lcd.clear();
}

bool isMatriculaValida(String matricula) {
    if (matricula.length() != 7) {
        return false; // La matrícula debe tener 7 caracteres
    }
    for (int i = 0; i < 4; i++) {
        if (!isDigit(matricula[i])) {
            return false; // Los primeros 4 caracteres deben ser dígitos
        }
    }
    for (int i = 4; i < 7; i++) {
        if (!isAlpha(matricula[i])) {
            return false; // Los últimos 3 caracteres deben ser letras
        }
    }
    return true; // La matrícula es válida
}

void handleSalidaParking() {
    lcd.clear(); // Limpia la pantalla LCD
    lcd.setCursor(0, 0); // Establece el cursor en la columna 0, fila 0
    lcd.print("Introduce mat:");
    lcd.setCursor(0, 1); // Establece el cursor en la columna 0, fila 1

```

matricula = ""; // Limpia la variable de matrícula

```

while (true) {
    char key = keypad.getKey(); // Lee la tecla presionada
    if (key) { // Si se presionó una tecla
        Serial.println(key); // Imprime la tecla en el monitor serial
        if (key == '#') { // Si se presionó la tecla de confirmación (por ejemplo '#')
            if (isMatriculaValida(matricula)) {
                StaticJsonDocument<200> doc;
                doc["id_parking"] = id_parking;
                doc["matricula"] = matricula;

                char jsonBuffer[256];
                serializeJson(doc, jsonBuffer);

                // Publicar el mensaje JSON
                client.publish(Parking_maquina_salida, jsonBuffer);
                esperandoRespuesta = true; // Esperar la respuesta del servidor
                lcd.clear();
                lcd.setCursor(0, 0);
                lcd.print("Comprobando");
                lcd.setCursor(0, 1);
                lcd.print("Matricula");
                break; // Sale del bucle
            } else {

```

```

        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Matricula invalida");
        delay(2000);
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Introduce mat:");
        lcd.setCursor(0, 1);
        lcd.print(matricula);
    }
} else if (key == '*') { // Si se presionó la tecla de borrar (por ejemplo '*')
    if (matricula.length() > 0) {
        matricula.remove(matricula.length() - 1); // Borra el último carácter
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Introduce mat:");
        lcd.setCursor(0, 1);
        lcd.print(matricula);
    }
} else if (key == 'D') { // Si se presionó la tecla para regresar al menú
    lcd.clear();
    displayMenu();
    return; // Sal del bucle y retorna al menú
} else {
    matricula += key; // Añade la tecla presionada a la matrícula
    lcd.setCursor(0, 1); // Establece el cursor en la columna 0, fila 1
    lcd.print(matricula); // Muestra la matrícula en la LCD
}
}
}

void callback(char* topic, byte* payload, unsigned int length) {
    payload[length] = '\0'; // Asegura que el payload es una cadena de caracteres válida
    String message = String((char*)payload);

    Serial.print("Mensaje recibido [");
    Serial.print(topic);
    Serial.print("]: ");
    Serial.println(message);

    if (String(topic) == Parking_maquina_respuesta) {
        if (message == "0") {
            lcd.clear();
            lcd.setCursor(0, 0);
            lcd.print("Matricula invalida");
            delay(2000);
            lcd.clear();
            esperandoRespuesta = false; // Permite al usuario intentar de nuevo
        } else {

```



```

    tiempoEnParking = message.toInt();
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Tiempo: ");
    lcd.print(tiempoEnParking);
    lcd.print(" secs");

    int importe = calcularImporte(tiempoEnParking);

    // Crear documento JSON para la publicación del importe
    StaticJsonDocument<200> doc;
    doc["matricula"] = matricula;
    doc["importe"] = importe;
    char jsonBuffer[256];
    serializeJson(doc, jsonBuffer);

    // Publicar el mensaje JSON con el importe y la matrícula
    client.publish(Parking_maquina_tarjeta, jsonBuffer);

    lcd.setCursor(0, 1);
    lcd.print("Importe: $");
    lcd.print(importe);
    delay(5000);
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Esperando Pago");
    esperandoTarjeta = true;
    tiempoSinRespuesta = millis(); // Inicia el contador de tiempo
}
} else if (String(topic) == Parking_maquina_tarjeta) {
    delay(11000);
    if (message == "1") { // Corrige el operador de comparación
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Pagado");
        lcd.setCursor(0, 1);
        lcd.print("Buen viaje");
        esperandoTarjeta = false;
        delay(5000); // Espera para que el usuario vea el mensaje "Pagado"
        lcd.clear();
        esperandoRespuesta = false;
    } else {
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Error en el pago");
        delay(2000);
        esperandoRespuesta = false;
        esperandoTarjeta = false;
    }
}
}

```

```

}

int calcularImporte(int tiempo) {
    int tarifaPorSegundo = 1; // Puedes ajustar la tarifa por segundo según tus necesidades
    return tiempo * tarifaPorSegundo;
}

```

Figura 18. Programa de Arduino del primer ESP-32

En primer lugar, se realiza una configuración inicial, donde se definen las bibliotecas y se inicializan las variables. Se establecen los parámetros de conexión WIFI y MQTT, así como los pines del keypad y del LCD.

En Setup(), se inicia la comunicación serial, se configura el LCD, se conecta al WIFI y se establece la conexión MQTT. Además, se hacen las suscripciones a los topics para recibir mensajes del servidor.

En Loop(), se encuentra el programa principal, en cada iteración se comprueba si hay algún mensaje recibido por MQTT. Si no está esperando una respuesta, se muestra el menú en el LCD y se espera a que el usuario presione una tecla en el keypad. En función de la tecla pulsada se ejecuta la acción correspondiente.

Funciones de entrada y salida. Estas funciones se encargan de manejar la entrada de la matrícula del vehículo y su posterior envío al MQTT. También se manejan respuestas del servidor, como la validación de la matrícula y la cantidad a pagar.

Callback(), esta función es llamada cuando se recibe un mensaje por MQTT. Se procesan los mensajes recibidos y se ejecutan las acciones correspondientes, como la validación de la matrícula y la cantidad a pagar.

La función del cálculo del importe se encarga de calcular el importe a pagar en función del tiempo que el vehículo ha estado estacionado en el parking. Se utiliza una tarifa fija por segundo que puede ser modificada.

El segundo ESP-32 es el encargado de manejar las plazas del parking y de mostrar al cliente mediante el semáforo de leds el estado de la plaza (libre, ocupada o reservada).

El código es el siguiente:

```

#include <PubSubClient.h>
#include <WiFi.h>
#include <ArduinoJson.h> // Librería para manejar JSON

const char *ssid = "UBICUA";
const char *wifi_password = "cjp2001";

const char *mqtt_broker = "192.168.19.117"; // broker address// define topic
const char *mqtt_username = ""; // username for authentication
const char *mqtt_password = ""; // password for authentication
const char *P1_topic_estado = "Ubipark/Parking1/P1/Estado"; // Tópico para publicar el estado
const char *P1_topic_reserva = "Ubipark/Parking1/P1/Reserva"; // Tópico para recibir reservas

WiFiClient espClient;
PubSubClient client(espClient);

const int mqtt_port = 1883;
const int P1_ROJO = 19;
const int P1_AMARILLO = 22;
const int P1_VERDE = 21;

// Definimos los pines del ESP32 a los que conectaremos el Trig y Echo del HC-SR04
#define P1_TRIG_PIN 5
#define P1_ECHO_PIN 18

bool P1_reserved = false; // Variable para almacenar si hay una reserva
const char* id_parking = "1";
const char* id_plaza = "1";

void setup() {
    // Iniciamos la comunicación serie a 115200 baudios
    Serial.begin(9600);

    // Configuramos los pines de los LEDs como salida
    pinMode(P1_ROJO, OUTPUT);
    pinMode(P1_AMARILLO, OUTPUT);
    pinMode(P1_VERDE, OUTPUT);

    // Configuramos los pines del Trig y Echo
    pinMode(P1_TRIG_PIN, OUTPUT);
    pinMode(P1_ECHO_PIN, INPUT);
    WiFi.begin(ssid, wifi_password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.println("Connecting to WiFi..");
    }
    Serial.println("Connected to the WiFi network");
    //connecting to a mqtt broker
    client.setServer(mqtt_broker, mqtt_port);
    client.setCallback(callback);
    while (!client.connected()) {
        String client_id = "Arduino";
        client_id += String(WiFi.macAddress());
        Serial.printf("The client %s connects to the public mqtt broker\n", client_id.c_str());
        if (client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {
            Serial.println("Public emqx mqtt broker connected");
            client.subscribe(P1_topic_reserva);
        } else {
            Serial.print("failed with state ");

```

```

        Serial.print(client.state());
        delay(2000);
    }
}

void callback(char *topic, byte *payload, unsigned int length) {
    String reserve;
    Serial.print("Message arrived in topic: ");
    Serial.println(topic);
    Serial.print("Message:");

    // Imprime el payload recibido
    for (int i = 0; i < length; i++) {
        Serial.print((char)payload[i]);
    }
    Serial.println();
    Serial.println("-----");

    // Convierte el payload en una cadena
    for (int j = 0; j < length; j++) {
        reserve += (char) payload[j];
    }
    int reserva = reserve.toInt();

    // Compara el t3pico recibido
    if (String(topic) == "Ubipark/Parking1/P1/Reserva") {
        Serial.println(reserva);
        // Compara el valor de reserva
        if(reserva == 1){
            P1_reserved = true;
            return;
        }
        else if(reserva == 0){
            P1_reserved = false;
            return;
        }
    }
}
}

```

```

long CalcularDistancia(int SensorEcho, int SensorTriger) {
    // Generamos un pulso de 10 microsegundos en el pin Trig
    digitalWrite(SensorTriger, LOW);
    delayMicroseconds(2);
    digitalWrite(SensorTriger, HIGH);
    delayMicroseconds(10);
    digitalWrite(SensorTriger, LOW);

    // Medimos el tiempo que tarda en llegar el eco
    long duration = pulseIn(SensorEcho, HIGH);

    // Convertimos la duración en distancia (en centímetros)
    long distance = duration * 0.034 / 2;

    return distance;
}

void loop() {
    client.loop();

    // Crear un documento JSON
    StaticJsonDocument<200> doc;
    char jsonBuffer[256];

    if (P1_reserved) {
        if (CalcularDistancia(P1_ECHO_PIN, P1_TRIG_PIN) < 10) {
            digitalWrite(P1_ROJO, HIGH);
            digitalWrite(P1_VERDE, LOW);
            digitalWrite(P1_AMARILLO, HIGH);

            // Preparar el mensaje JSON
            doc["id_parking"] = id_parking;
            doc["id_plaza"] = id_plaza;
            doc["estado"] = 1;

            serializeJson(doc, jsonBuffer);
            client.publish(P1_topic_estado, jsonBuffer);
        } else {
            digitalWrite(P1_ROJO, LOW);
            digitalWrite(P1_VERDE, LOW);
        }
    }
}

```

```

        digitalWrite(P1_AMARILLO, HIGH);

        // Preparar el mensaje JSON
        doc["id_parking"] = id_parking;
        doc["id_plaza"] = id_plaza;
        doc["estado"] = 0;

        serializeJson(doc, jsonBuffer);
        client.publish(P1_topic_estado, jsonBuffer);
    }
} else {
    if (CalcularDistancia(P1_ECHO_PIN, P1_TRIG_PIN) < 10) {
        digitalWrite(P1_ROJO, HIGH);
        digitalWrite(P1_VERDE, LOW);
        digitalWrite(P1_AMARILLO, LOW);

        // Preparar el mensaje JSON
        doc["id_parking"] = id_parking;
        doc["id_plaza"] = id_plaza;
        doc["estado"] = 1;

        serializeJson(doc, jsonBuffer);
        client.publish(P1_topic_estado, jsonBuffer);
    } else {
        digitalWrite(P1_ROJO, LOW);
        digitalWrite(P1_VERDE, HIGH);
        digitalWrite(P1_AMARILLO, LOW);

        // Preparar el mensaje JSON
        doc["id_parking"] = id_parking;
        doc["id_plaza"] = id_plaza;
        doc["estado"] = 0;

        serializeJson(doc, jsonBuffer);
        client.publish(P1_topic_estado, jsonBuffer);
    }
}

// Esperamos cinco segundos antes de la siguiente medición
delay(10000);
}

```

Figura 19. Programa de Arduino del segundo ESP-32

Al igual que con el anterior ESP-32 se realiza una configuración inicial donde de nuevo se definen las bibliotecas y se inicializan las variables. Se establecen los parámetros de conexión WiFi y MQTT, así como los pines del sensor de ultrasonido

HC-SR04 y los pines de los leds que indicarán el estado de la plaza de estacionamiento.

Setup(), en esta función se inicia la comunicación serial, se configuran los pines de los leds y del sensor de ultrasonido, se conecta al WiFi y se establece la conexión con el servidor MQTT. Además, se suscribe al topic de MQTT donde se recibirán las reservas de la plaza de estacionamiento.

Callback(), Esta función se llama cuando se recibe un mensaje del servidor MQTT. Aquí se procesan los mensajes recibidos y se actualiza la variable P1_reserved que indica si la plaza de estacionamiento está reservada o no.

Función del cálculo de distancia, esta función utiliza el sensor de ultrasonido para medir la distancia entre el sensor y el objeto más cercano, que en este caso sería el vehículo estacionado en la plaza. La distancia se calcula en centímetros y se devuelve como un valor entero.

Loop(), es el bucle principal del programa, en cada iteración se mide la distancia mediante el sensor de ultrasonido. En función de la distancia y del estado de la reserva de la plaza de estacionamiento, se encienden o apagan los leds para indicar el estado de la plaza, El estado de la plaza se publica mediante MQTT.

El último ESP-32 es el utilizado por el lector RFID-RC522 para realizar el pago.

El código es el siguiente:

```
#include <SPI.h>
#include <MFRC522.h>
#include <PubSubClient.h>
#include <WiFi.h>
#include <ArduinoJson.h> // Librería para manejar JSON

const char *ssid = "UBICUA";
const char *wifi_password = "cjp2001";
const char *mqtt_broker = "192.168.19.117"; // broker address
const char *mqtt_username = ""; // username for authentication
const char *mqtt_password = ""; // password for authentication
const char *Parking_maquina_tarjeta = "Ubipark/Parking1/Maquina/Tarjeta";
const char *Parking_maquina_pago = "Ubipark/Parking1/Maquina/Pago";
#define SS_PIN 5 // SDA pin connected to GPIO 5
#define RST_PIN 22 // RST pin connected to GPIO 22
WiFiClient espClient;
PubSubClient client(espClient);

const int mqtt_port = 1883;
MFRC522 mfrc522(SS_PIN, RST_PIN); // Create MFRC522 instance
int importe = 0; // Variable to store the received amount
String matricula = ""; // Variable to store the received license plate
bool esperar_tarjeta = false; // Flag to indicate whether to wait for card

void setup() {
  Serial.begin(9600); // Initialize serial communications with the PC
```

```

SPI.begin();           // Initialize SPI bus
mfrc522.PCD_Init();    // Initialize MFRC522
Serial.println("Place your card near the reader...");
Serial.println();
WiFi.begin(ssid, wifi_password);
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
}
Serial.println("Connected to the WiFi network");
// Connecting to the MQTT broker
client.setServer(mqtt_broker, mqtt_port);
client.setCallback(callback);
while (!client.connected()) {
    String client_id = "Arduino";
    client_id += String(WiFi.macAddress());
    Serial.printf("The client %s connects to the public MQTT broker\n", client_id.c_str());
    if (client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {
        Serial.println("Public EMQX MQTT broker connected");
        client.subscribe(Parking_maquina_tarjeta);
    } else {
        Serial.print("Failed with state ");
        Serial.print(client.state());
        delay(2000);
    }
}

void callback(char* topic, byte* payload, unsigned int length) {
    if (strcmp(topic, Parking_maquina_tarjeta) == 0) {
        payload[length] = '\0'; // Asegura que el payload es una cadena de caracteres válida
        String message = String((char*)payload);

        // Deserializar el mensaje JSON
        StaticJsonDocument<200> doc;
        DeserializationError error = deserializeJson(doc, message);
        if (error) {
            Serial.print("deserializeJson() failed: ");
            Serial.println(error.c_str());
            return;
        }

        // Obtener los valores del JSON
        importe = doc["importe"];
        matricula = doc["matricula"].as<String>();
        esperar_tarjeta = true;

        Serial.print("Importe recibido: ");
        Serial.println(importe);
        Serial.print("Matricula: ");
        Serial.println(matricula);
    }
}

```



```

void loop() {
    client.loop();

    // Only proceed with card reading if waiting for card
    if (esperar_tarjeta) {
        unsigned long start_time = millis();
        while (millis() - start_time < 10000) { // Wait for 10 seconds
            if (mfrc522.PICC_IsNewCardPresent() && mfrc522.PICC_ReadCardSerial()) {
                Serial.print("UID tag: ");
                String content = "";
                for (byte i = 0; i < mfrc522.uid.size; i++) {
                    Serial.print(mfrc522.uid.uidByte[i] < 0x10 ? " 0" : " ");
                    Serial.print(mfrc522.uid.uidByte[i], HEX);
                    content.concat(String(mfrc522.uid.uidByte[i] < 0x10 ? " 0" : " "));
                    content.concat(String(mfrc522.uid.uidByte[i], HEX));
                }
                Serial.println();
                Serial.print("Message: ");
                content.toUpperCase();
                Serial.println(content);

                // Crear documento JSON para la publicación del importe y la matrícula
                StaticJsonDocument<200> doc;
                doc["importe"] = importe;
                doc["matricula"] = matricula;

                char jsonBuffer[256];
                serializeJson(doc, jsonBuffer);

                // Publicar el mensaje JSON
                client.publish(Parking_maquina_pago, jsonBuffer);
                Serial.print("Published amount and license plate: ");
                Serial.println(jsonBuffer);

                // Publish '1' to the tarjeta topic
                client.publish(Parking_maquina_tarjeta, "1");
                Serial.println("Published '1' to the tarjeta topic");

                // Halt PICC
                mfrc522.PICC_HaltA();

                // Reset the flag
                esperar_tarjeta = false;
                break; // Exit the while loop after reading the card
            }
        }
        // If 10 seconds pass without reading a card, reset the flag
        esperar_tarjeta = false;
    }
}

```

Figura 20. Programa de Arduino del tercer ESP-32

Al igual que con los dos ESP-32 anteriores hay una configuración inicial en la que se incluyen las bibliotecas necesarias y se definen las variables y constantes, incluyendo los parámetros de conexión WiFi y MQTT, así como los pines del lector de tarjetas RFID.

Setup(), en esta función se inicializan las comunicaciones serial y SPI, se configura el lector de tarjetas RFID y se establece la conexión WiFi. Luego, se conecta al servidor MQTT y se suscribe al topic donde se recibirán las solicitudes de pago.

Callback(), Esta función se llama cuando se recibe un mensaje del servidor MQTT. Aquí se procesa el mensaje recibido, que contiene el importe a pagar y la matrícula del vehículo. Además, se activa esperar_tarjeta para indicar que se espera que se presente una tarjeta RFID.

Loop(), el bucle principal del programa, en cada iteración se comprueba si se está esperando que se pase una tarjeta RFID. Si es así, se esperan 10 segundos para que se pase una tarjeta. Si se lee una tarjeta se lee su UID, se publica el importe a pagar y la matrícula del vehículo por MQTT y restablece esperar_tarjeta. Si no se presenta ninguna tarjeta en el tiempo establecido, se restablece la variable esperar_tarjeta sin realizar ninguna acción.

5.2.2. Capa de transporte

Explicar el protocolo utilizado en esta capa y por qué se ha elegido.

5.2.3. Capa de procesado

En este apartado se analizarán los elementos que componen la capa de procesado, indicando primero el objetivo de esta capa y después descomponiendo el resto en subapartados que traten sobre cada uno de los elementos de la capa. En este caso los elementos en los que se descompondrá serían el bróker, apache Tomcat y la base de datos.

Broker

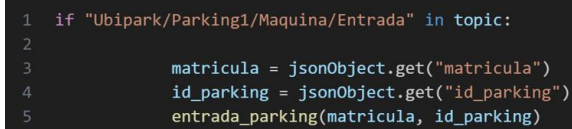
Para el intercambio de mensajes por el protocolo MQTT hemos utilizado un script en python, para ejecutarlo se necesitan varias bibliotecas. A continuación detallaremos los pasos para su instalación y configurar el entorno:

Primero deberemos instalar el lenguaje Python, para ello vamos a la web oficial [Download Python | Python.org](https://www.python.org/) y seguimos los pasos en el instalador. Una vez instalado será necesario instalar varias librerías externas para que nuestro proyecto funcione, la primera de ellas es Paho MQTT, esta biblioteca la utilizamos para interactuar con el Broker MQTT, para instalarla ejecutamos el siguiente comando: “*pip install paho-mqtt*”, la segunda librería es PyMySQL utilizada para interactuar con nuestra base de datos MySQL, la instalaremos con el siguiente comando: “*pip install pymysql*”. Otras dependencias estándar como json, threading y datetime son bibliotecas estándar de Python y no necesitan instalación adicional.

El script se organiza en varias funciones y métodos para manejar diferentes operaciones como conexión a la base de datos, inserciones, actualizaciones y manejo de mensajes MQTT. Aquí se explica la estructura y las etiquetas utilizadas.

Las etiquetas MQTT se usan para identificar y procesar diferentes tipos de mensajes que llegan al broker MQTT. Estas etiquetas incluyen:

Entrada al Parking:

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is in Python and shows an MQTT topic filter. It starts with a line number '1' followed by 'if "Ubipark/Parking1/Maquina/Entrada" in topic:'. Line '2' is empty. Line '3' has 'matricula = jsonObject.get("matricula")'. Line '4' has 'id_parking = jsonObject.get("id_parking")'. Line '5' has 'entrada_parking(matricula, id_parking)'.

```
1 if "Ubipark/Parking1/Maquina/Entrada" in topic:
2
3     matricula = jsonObject.get("matricula")
4     id_parking = jsonObject.get("id_parking")
5     entrada_parking(matricula, id_parking)
```

Figura 21. Topic Entrada

Utilizamos la etiqueta “Ubipark/Parking1/Maquina/Entrada” para registrar las entradas de coches al parking, registramos cual es la matrícula del coche y en qué parking ha entrado.

Salida del Parking:

A screenshot of a code editor with a dark background. The code is in Python and shows an MQTT topic filter for the exit. It starts with 'elif "Ubipark/Parking1/Maquina/Salida" in topic:'. Inside the block, it gets the license plate 'matricula = jsonObject.get("matricula")' and checks if there is a reservation 'if(tiene_reserva(matricula)):' with an indented block of actions: 'salida_parking(matricula)', 'salida_reserva(matricula)', 'tiempo = tiempo_parking(matricula)', 'client.publish("Ubipark/Parking1/Maquina/Respuesta", str(tiempo))', 'client.publish("Ubipark/Parking1/P1/Reserva", 0)', and 'mover_datos(matricula)'. An 'else:' block follows with 'salida_parking(matricula)', 'tiempo = tiempo_parking(matricula)', 'client.publish("Ubipark/Parking1/Maquina/Respuesta", str(tiempo))', and 'mover_datos(matricula)'.

```
elif "Ubipark/Parking1/Maquina/Salida" in topic:

    matricula = jsonObject.get("matricula")
    if(tiene_reserva(matricula)):

        salida_parking(matricula)
        salida_reserva(matricula)
        tiempo = tiempo_parking(matricula)
        client.publish("Ubipark/Parking1/Maquina/Respuesta", str(tiempo))
        client.publish("Ubipark/Parking1/P1/Reserva", 0)
        mover_datos(matricula)

    else:
        salida_parking(matricula)
        tiempo = tiempo_parking(matricula)
        client.publish("Ubipark/Parking1/Maquina/Respuesta", str(tiempo))
        mover_datos(matricula)
```

Figura 22. Topic Salida

Para la salida del coche hemos utilizado el topic “Ubipark/Parking1/Maquina/Salida”, cuando un coche sale del parking la máquina nos envía por MQTT la matrícula y en caso de que ese coche tuviera una reserva, gestionaremos la salida del coche, es decir calcularemos cuánto tiempo ha estado, finalizamos la reserva insertando la hora a la que ha salido el coche y enviaremos un 0 concluyendo la reserva de la plaza. En caso

de que no tenga reserva guardamos los datos del coche y el importe y por último enviamos el tiempo a la máquina para que ésta calcule el importe.

Actualizar el estado de las plazas:

```
1 elif "Ubipark/Parking1/P1/Estado" in topic:
2
3     id_plaza = jsonObject.get("id_plaza")
4     id_parking = jsonObject.get("id_parking")
5     estado = jsonObject.get("estado")
6     estado_plaza(id_parking, id_plaza, estado)
```

Figura 23. Topic estado de la plaza

Utilizamos la etiqueta “Ubipark/Parking1/P1/Estado” para obtener si la máquina nos comunica que el estado de una plaza ha cambiado para actualizarlo en la BBDD, si recibimos un 0 en estado es que se libera y 1 si está ocupada.

Pago en Máquina:

```
1 elif "Ubipark/Parking1/Maquina/Pago" in topic:
2
3     cantidad = jsonObject.get("importe")
4     matricula = jsonObject.get("matricula")
5     insertar_importe(matricula, cantidad)
```

Figura 24. Topic Pago

Para recibir los pagos de la máquina hemos utilizado la etiqueta “Ubipark/Parking1/Maquina/Pago” donde nos enviará la matrícula y el importe una vez el usuario pague su estancia en el parking y nosotros lo insertamos en la BBDD.

Conexión a la Base de Datos:

```
1 class ConnectionDDBB:
2     def obtain_connection(self, autocommit=True):
3         # Proporciona tu configuración de conexión aquí
4         connection = pymysql.connect(
5             host='vendifydb.mysql.database.azure.com',
6             user='Usuario',
7             password='IngresaTuContraseña',
8             database='parking',
9             autocommit=autocommit
10        )
11        return connection
```

Figura 25. Conexión a la Base de Datos

Especificamos nuestros parámetros de conexión a la base de datos.

Operaciones que interactúan con la BBDD:

- **entrada_parking(matricula,id_parking):** Registra la entrada de un vehículo.
- **salida_parking(matricula):** Registra la salida de un vehículo.
- **salida_reserva(matricula):** Registra la salida de una reserva.
- **mover_datos(matricula):** Mueve los datos del tiempo de parking al histórico.
- **estado_plaza(id_parking, id_plaza, estado):** Actualiza el estado de una plaza de parking.
- **insertar_importe(matricula, importe):** Inserta el importe del pago en el histórico.
- **tiempo_parking(matricula):** Calcula el tiempo de parking.
- **tiene_reserva(matricula):** Verifica si el vehículo tiene reserva.

Otras funciones:

```
1 def fetch_and_process_inserts():
2     while True:
3         new_inserts = fetch_new_inserts()
4         for insert in new_inserts:
5             user_id = insert[1]
6             parking_id = insert[2]
7             fecha_reserva = insert[3].strftime('%Y-%m-%d')
8             hora_inicio = insert[4].strftime('%H:%M:%S')
9             id_plaza = insert[5]
10
11            data = {
12                "user_id": user_id,
13                "parking_id": parking_id,
14                "fecha_reserva": fecha_reserva,
15                "hora_inicio": hora_inicio,
16                "id_plaza": id_plaza
17            }
18
19            if(parking_id==4):
20                client.publish("Ubipark/Parking1/P1/Reserva", 1)
21            else:
22                print("Se ha recibido una reserva"+json.dumps(data))
```

Figura 26. Procesar nuevas reservas

Función encargada de estar escuchando a la base de datos constantemente y en caso de que se reserve una plaza en el parking actual, le enviamos un 1 a la plaza reservada para que la luz se ponga en naranja.

Flujo del programa:

```
1 client = mqtt.Client()
2 client.on_connect = on_connect
3 client.on_message = on_message
4
5 client.connect(broker_address, broker_port)
6
7 mqtt_thread = threading.Thread(target=client.loop_forever)
8 fetch_thread = threading.Thread(target=fetch_and_process_inserts)
9
10 mqtt_thread.start()
11 fetch_thread.start()
```

Figura 27. Flujo Broker MQTT

Primero creamos el cliente MQTT y nos conectamos al broker, el método `on_message` será el encargado de encaminar los mensajes dependiendo del topic, por otro lado, para que nuestro script no se bloquee, ya que no puede escuchar al MQTT y a la base de datos a la vez, para ello corremos en hilos distintos los procesos.

Apache Tomcat

Para llevar a cabo la instalación de Apache Tomcat, primero de todo, debemos ir a su sitio web oficial <https://tomcat.apache.org/> y en nuestro caso hemos descargado la versión 10.1. Una vez instalado debemos configurar los usuarios con los que accederemos a Tomcat, para ello vamos al archivo 'conf/tomcat-users.xml' y creamos un usuario con las credenciales que queramos, también si quieres cambiar el puerto en el que correrá Tomcat (8080 por defecto) puedes ir al archivo 'conf/server.xml' y cambiarlo. Para iniciar Tomcat podemos ejecutar el archivo 'startup.bat' en el directorio '/bin' de la instalación de Tomcat o si utilizamos el IDE Netbeans lo iniciará automáticamente.

En cuanto a la estructura de carpetas de un proyecto de Tomcat tiene la siguiente forma:

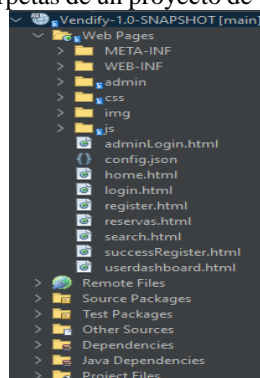


Figura 28. Estructura carpetas Tomcat

- Directorio **META-INF**: contiene metadatos sobre la aplicación web y context.xml
- Directorio **WEB-INF**: contiene archivos de configuración y recursos privados de la aplicación.
- Directorio **admin**: contiene los archivos HTML de la sección de administrador.
- Directorio **css**: contiene los archivos de tipo CSS para dar estilo a nuestra web.
- Directorio **img**: contiene las imágenes que utilizamos en nuestra web.
- Directorio **js**: contiene los scripts de Javascript.
- Directorio **Source Packages**: contiene las clases de Java para el funcionamiento de nuestra aplicación.
- Resto de carpetas: contienen dependencias de java y archivo de configuración de éstas.

Base de datos

Para la base de datos hemos elegido una base de datos MySQL alojada en Azure gracias a la suscripción *Azure for Students* que nos proporciona Universidad de Alcalá. Para crear este recurso iniciamos sesión con nuestra cuenta de la universidad en [Azure](#), vamos a buscar Recurso y escribimos *Servidor flexible de Azure Database for MySQL* y seguimos los pasos para su configuración, una vez terminado podemos ver los parámetros para conectarnos a ella desde MySQL Workbench en la sección *Conectar*.

Para el modelado de tablas hemos utilizado MySQL Workbench y el esquema Entidad/Relación nos ha quedado de la siguiente manera:

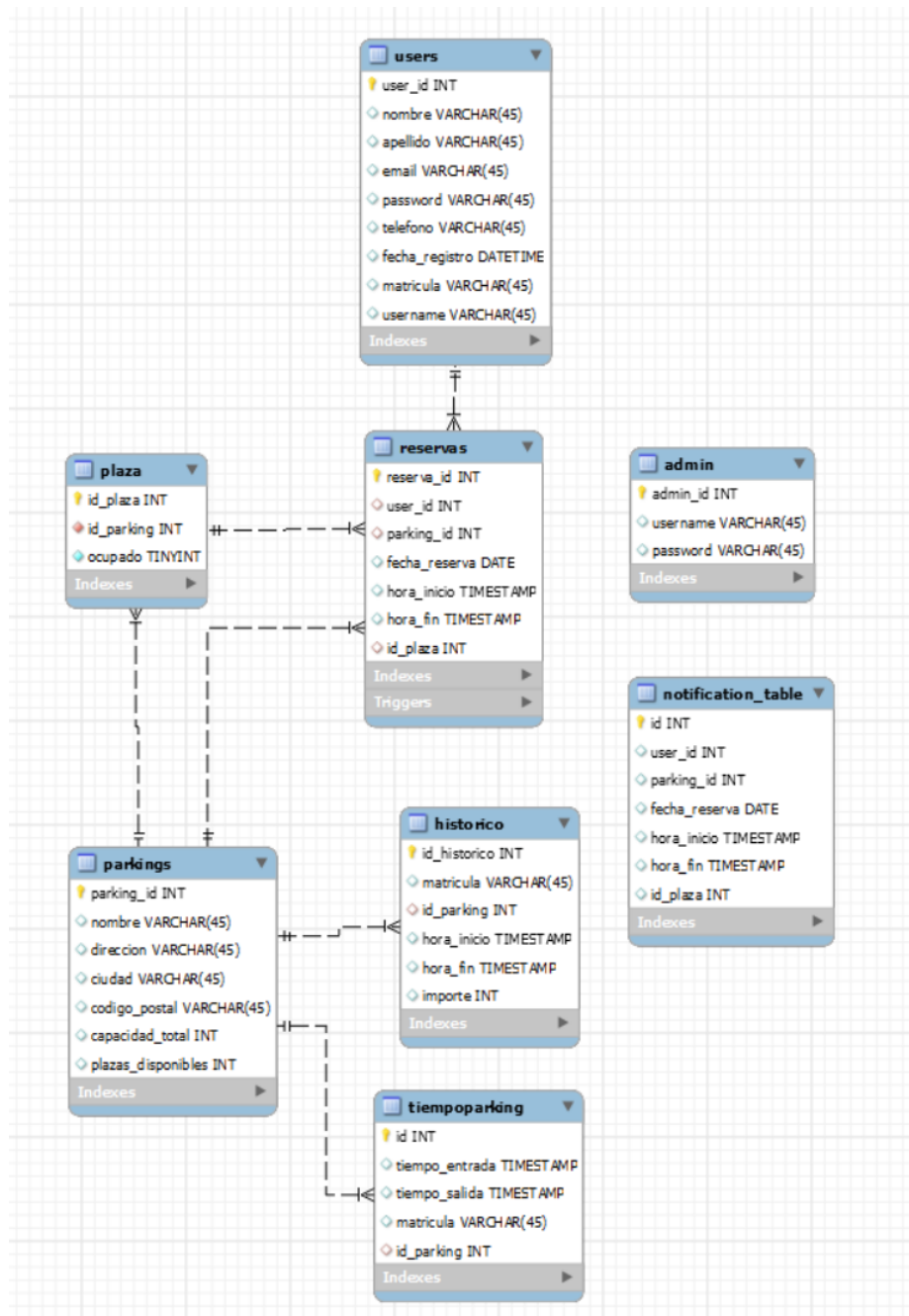


Figura 29. Esquema E/R de nuestra BBDD

Tablas utilizadas:

- **users:** Tabla para almacenar los datos de los usuarios incluida la matrícula de su coche.
- **admin:** Tabla para almacenar el usuario y contraseña del administrador para que pueda acceder a funciones especiales desde la web.
- **parkings:** Almacena información de los parkings como el nombre, capacidad total, plazas libres, etc...
- **plaza:** Representa una plaza del parking y en función de si *'ocupado'* es 0 o 1 si está libre o no, también tiene una llave foránea *'id_parking'* para saber a qué parking pertenece.
- **reservas:** Almacena la reserva de un usuario, y tiene varias llaves foráneas: *'user_id'* para identificar qué usuario ha hecho la reserva, *'parking_id'* para saber en qué parking está la reserva e *'id_plaza'* para saber qué plaza del parking se ha reservado.
- **tiempoparking:** Tabla para calcular con la matrícula el tiempo que ha estado un usuario en el parking y calcular el importe, tiene la llave foránea *'id_parking'* para saber en qué parking estamos.
- **historico:** Almacena la hora en la que un coche entra y sale de que parking gracias a una llave foránea *'id_parking'* con su respectivo importe.
- **notification_table:** Tabla en la que se copian los datos de las reservas cuando entra una nueva para notificar al Broker MQTT de que ha habido una nueva inserción en dicha tabla.

5.2.4. Capa de aplicación

El objetivo de la capa de aplicación en el desarrollo de software es manejar la lógica de negocio y coordinar las interacciones entre el usuario y los datos almacenados. En una aplicación web, esta capa es responsable de procesar las solicitudes del usuario, ejecutar la lógica de negocio necesaria y devolver la respuesta adecuada al cliente (el navegador web). Utilizando servlets de Java, esta capa se descompone en varios elementos clave que colaboran para proporcionar una funcionalidad completa y eficiente.

Aplicación Web

Nuestra aplicación se puede dividir en:

Cliente (Frontend)

Es la parte de la aplicación con la que interactúan los usuarios finales. En nuestro caso está compuesta por HTML, CSS y JavaScript. Hemos dividido en carpetas el proyecto, en el directorio raíz del proyecto encontramos los .html a los que podrá acceder un usuario normal, y en la carpeta 'css' y 'js' los archivos de configuración de CSS y JavaScript correspondientes a los .html, en la carpeta 'img' tenemos las imágenes que utilizamos en la web, también tenemos el config.json que lo utilizamos para llamar a la API de Google Maps (Más adelante veremos un esquema de cómo se ven estos archivos en el navegador).

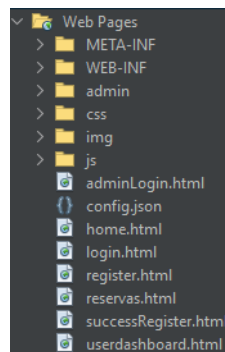


Figura 30. Carpetas front

Dentro de la carpeta 'admin' encontraremos los archivos .html relacionados con las vistas a las que puede acceder el usuario con rol administrador y dentro de la carpeta 'css' y 'js' encontraremos una sub-carpeta con los archivos correspondientes a estas vistas.

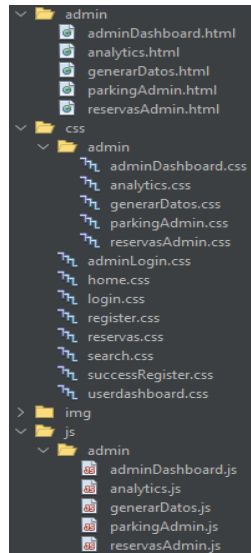


Figura 31. Carpetas back

Servidor (Backend)

El servidor es donde reside la lógica de la aplicación, hemos utilizado Servlets que son componentes Java que responden a las solicitudes HTTP. Actúan como controladores que procesan las solicitudes, ejecutan la lógica de negocio y generan respuestas. Dentro del directorio raíz del proyecto encontramos la carpeta 'Source Packages' que dentro de ella encontramos las siguientes carpetas:

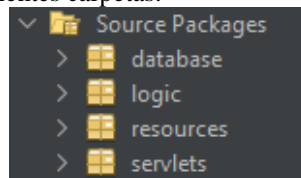


Figura 32. Carpetas Java

Database

Conecta nuestra aplicación web con la BBDD, dentro de ella encontramos una clase 'ConnectionDDBB', encargada de gestionar la conexión de la BBDD en Azure con la web, utiliza un conector JDBC, configuramos el Pool de Conexiones en el archivo 'Web Pages/META-INF/context.xml'

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/">
  <Resource name="jdbc/Vendify"
    auth="Container"
    type="javax.sql.DataSource"
    username="davinccx"
    password="Vendify2024"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://vendifydb.mysql.database.azure.com:3306/parking"
    maxTotal="100"
    maxIdle="30"
    maxWaitMillis="10000"/>
</Context>
```

Figura 33. Pool de conexiones

También encontramos alguna función que retorna la ejecución de una query en SQL como:

```
public static PreparedStatement getUsers(Connection con)
{
    return getStatement(con,"SELECT * FROM users");
}
```

Figura 34. Ejemplo prepared statement

Dentro de la carpeta database encontramos constructores de todas las tablas de la BBDD:

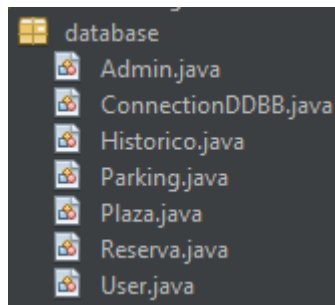


Figura 35. Carpetas database

Logic

Aquí almacenamos funciones que son independientes de la BBDD y los servlets, dentro de ella encontramos:

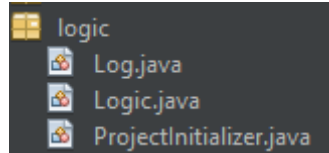


Figura 36. Carpetas logic

'Log', clase encargada de inicializar los logs, hablaremos más tarde de dónde y cómo se configuran estos logs.

'Logic', clase que da nombre a la carpeta, tenemos funciones como 'getParkingsFromDB()' que se encarga de retornar en formato ArrayList los parkings de la base de datos para poder trabajar con ellos desde Java.

```

public static ArrayList<Parking> getParkingsFromDB() {
    ArrayList<Parking> parkings = new ArrayList<Parking>();
    ConnectionDDBB conector = new ConnectionDDBB();
    Connection con = null;

    try {
        con = conector.obtainConnection(true);
        Log.log.debug("DataBase connected");
        PreparedStatement ps = ConnectionDDBB.getParkings(con);
        ResultSet rs = ps.executeQuery();
        Log.log.info("Query=> {}", ps.toString());
        while (rs.next()) {
            Parking p = new Parking();
            p.setParking_id(rs.getInt("parking_id"));
            p.setNombre(rs.getString("nombre"));
            p.setDireccion(rs.getString("direccion"));
            p.setCiudad(rs.getString("ciudad"));
            p.setC_postal(rs.getString("codigo_postal"));
            p.setCapacidad_total(rs.getInt("capacidad_total"));
            p.setPlazas_disponibles(rs.getInt("plazas_disponibles"));
            parkings.add(p);
        }
    } catch (SQLException e) {

        parkings = new ArrayList<Parking>();
        Log.log.error("Error: {}", e);

    } catch (NullPointerException e) {

        parkings = new ArrayList<Parking>();
        Log.log.error("Error: {}", e);

    } catch (Exception e) {

        parkings = new ArrayList<Parking>();
        Log.log.error("Error: {}", e);

    } finally {
        conector.closeConnection(con);
        Log.log.debug("DataBase disconnected");
    }

    return parkings;
}

```

Figura 37. Función de logic

'*ProjectInitializer*', implementa la interfaz **ServletContextListener** y está anotada con **@WebListener**, lo que indica que debe ser registrada como un listener en el contexto del servlet. Los listeners de contexto son componentes especiales en las aplicaciones web Java que permiten ejecutar código cuando el contexto de la aplicación se inicializa y destruye

```

package logic;

import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

@WebListener
public class ProjectInitializer implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {

        Log.log.info("-->Inicia la aplicación<--");

    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {

    }

}

```

Figura 38. Project Initializer

Resources

Dentro de esta carpeta encontramos el archivo 'log4j2.xml', es el archivo de configuración de los logs para implementar los logs hemos tenido que importar la dependencia **log4j** en nuestro proyecto. En este archivo de configuración debemos especificar la ruta donde queremos que nuestros logs se guardan y el formato.

```

<?xml version="1.0" encoding="UTF-8" ?>
<Configuration status="info">
    <Appenders>
        <!-- LOG -->
        <RollingFile name="LogFile" fileName="C:\\Users\\David\\Desktop\\pruebaUbicua\\logs\\log.log"
            filePattern="C:\\Users\\David\\Desktop\\pruebaUbicua\\logs\\Ubipark-%d{yyyy}-%i.log.gz">
            <PatternLayout>
                pattern="%nd{yyyy-MM-dd HH:mm:ss.SSS} %-5level %-30l %logger(36) ### %msg"
                header="-- UBIPARK LOGS --" />
            <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="10 MB" />
            </Policies>
            <DefaultRolloverStrategy max="20"/>
        </RollingFile>
        <!-- LOGDB -->
        <RollingFile name="LogFileDB" fileName="C:\\Users\\David\\Desktop\\pruebaUbicua\\logs\\logdb.log"
            filePattern="C:\\Users\\David\\Desktop\\pruebaUbicua\\logs\\Ubipark-%d{yyyy}-%i.logdb.gz">
            <PatternLayout>
                pattern="%nd{yyyy-MM-dd HH:mm:ss.SSS} %-5level %-30l %logger(36) ### %msg"
                header="-- UBIPARK DATABASE LOGS --" />
            <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="10 MB" />
            </Policies>
            <DefaultRolloverStrategy max="20"/>
        </RollingFile>
    </Appenders>
    <Loggers>
        <!-- LOG -->
        <Logger name="log" additivity="FALSE" level="INFO">
            <AppenderRef ref="LogFile" level="INFO" />
        </Logger>
        <!-- LOGDB -->
        <Logger name="logdb" additivity="FALSE" level="INFO">
            <AppenderRef ref="LogFileDB" level="INFO" />
        </Logger>
    </Loggers>
</Configuration>

```

Figura 39. Configuración Logs

Servlets

No hemos insertado código ya que es demasiado largo

Una de las carpetas más importantes de nuestro proyecto, disponemos de 23 servlets para la interacción con la BBDD, podemos dividir los servlets en diferentes tipos:

Get Servlet

Petición GET para obtener por ejemplo todo el histórico de nuestra aplicación, utilizamos la anotación `@WebServlet` para mapear el servlet, una vez obtenemos los datos procedentes de la BBDD mandamos un json con éstos a la web y los mostramos. También tenemos una modificación de este servlet que se les pasa un parámetro por ejemplo el username de un usuario y el servlet te devuelve la información de sólo es usuario.

Register Servlet

Petición de tipo POST con los datos que queremos registrar, por ejemplo en el formulario de registro enviamos los datos del usuario y en este servlet va obteniendo los parámetros del json y hace un INSERT en la BBDD, también funciona así con las reservas.

Update Servlet

Similar al servlet Register, también es una petición HTTP POST pero en vez de hacer un INSERT hace un UPDATE en la BBDD.

Generate Servlet

También es de tipo POST pero primero generamos los datos aleatorios con nuestra clase de Java y luego los insertamos en la base de datos y por último los enviamos a la web en formato JSON para mostrarlos.

Login Servlet

Utilizados para el inicio de sesión del usuario y el administrador, mandamos los campos al servlet y hace una comprobación en la BBDD de que el usuario/e-mail y la contraseña coincidan, en caso afirmativo redireccionará al usuario a la página principal, en caso contrario saltará un error.

Delete Servlet

Petición de tipo DELETE, por ejemplo, en caso de querer eliminar un usuario mandamos su id al servlet y éste hace un DELETE en la BBDD, en caso de eliminar un parking sus plazas también se eliminarán, si eliminamos una reserva la plaza se liberará.

Otras carpetas interesantes

La carpeta Dependencias contiene todas las dependencias necesarias para ejecutar nuestra aplicación como gson, jakarta, log4j, etc...

Por último, en la carpeta Project Files tenemos el archivo pom.xml, es el archivo de configuración principal de un proyecto Maven. Define las configuraciones, dependencias, plugins, y otras propiedades que Maven necesita para construir y gestionar el proyecto.

Esquema de Ventanas

A continuación, explicaremos el flujo de la aplicación mediante unos mockups simples, cuando el usuario accede se le mostrará la ventana Home donde puede ver ir a Login Register o Administrator

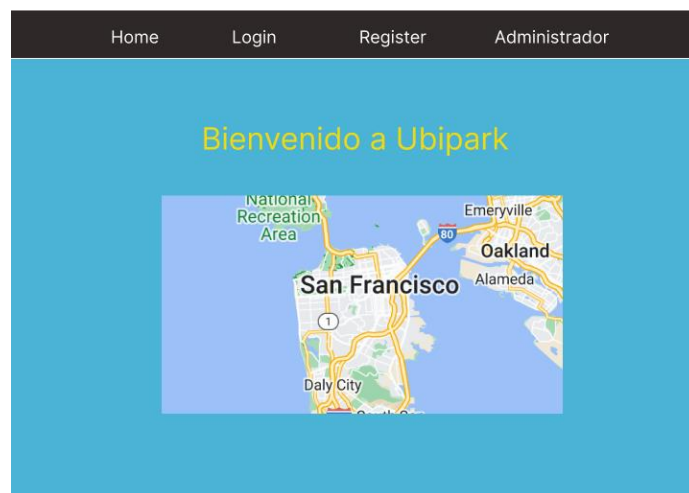
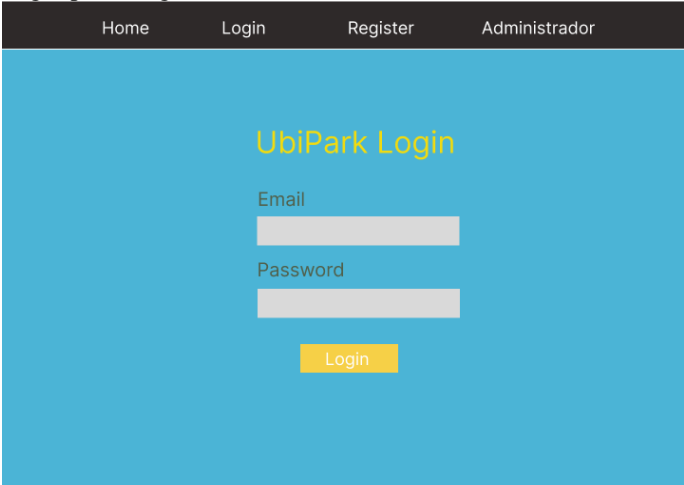


Figura 40. Esquema de Home

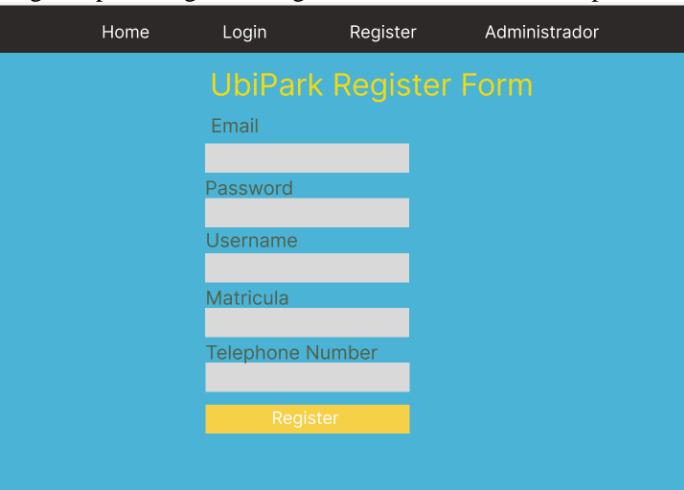
Si clicas en Login podrás loguearse como usuario nominal



The image shows a web interface for logging in. At the top, there is a dark navigation bar with four links: 'Home', 'Login', 'Register', and 'Administrador'. Below this, the main area has a light blue background. Centered on the page is the title 'UbiPark Login' in yellow. Underneath the title are two input fields: 'Email' and 'Password', both with light gray borders. Below these fields is a yellow button with the text 'Login' in black.

Figura 41. Esquema de Login

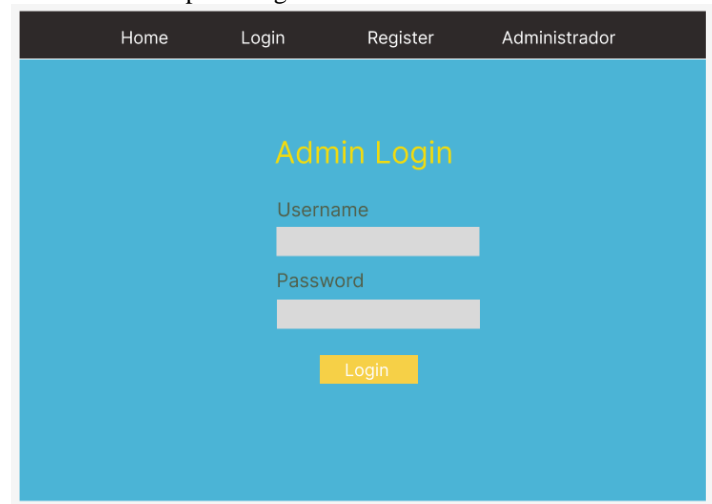
Si clicas en Register podrás loguearse registrarse e iniciar sesión después.



The image shows a web interface for registering. At the top, there is a dark navigation bar with four links: 'Home', 'Login', 'Register', and 'Administrador'. Below this, the main area has a light blue background. Centered on the page is the title 'UbiPark Register Form' in yellow. Underneath the title are six input fields: 'Email', 'Password', 'Username', 'Matricula', and 'Telephone Number', all with light gray borders. Below these fields is a yellow button with the text 'Register' in black.

Figura 42. Esquema de Register

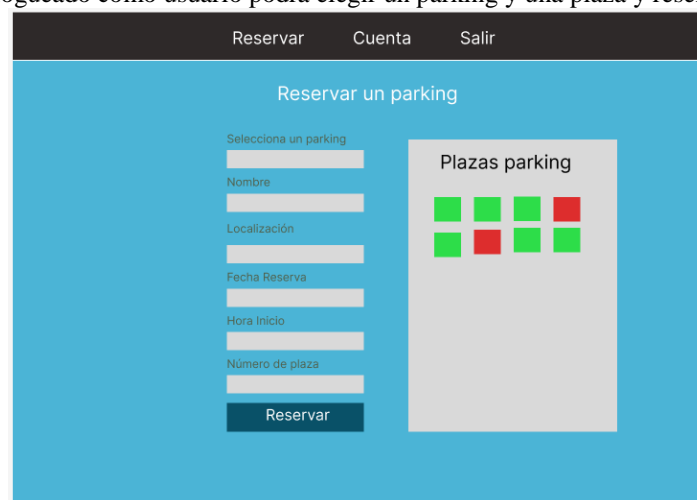
Si clicas en Administrador podrá loguearse como usuario administrador.



The image shows a web interface for an administrator login. At the top, there is a dark navigation bar with four links: 'Home', 'Login', 'Register', and 'Administrador'. The main content area has a light blue background. In the center, the text 'Admin Login' is displayed in yellow. Below this, there are two input fields: 'Username' and 'Password', both with light gray borders. A yellow 'Login' button is positioned below the password field.

Figura 43. Esquema de Register Admin

Una vez logueado como usuario podrá elegir un parking y una plaza y reservar.



The image shows a web interface for reserving a parking space. At the top, there is a dark navigation bar with three links: 'Reservar', 'Cuenta', and 'Salir'. The main content area has a light blue background. In the center, the text 'Reservar un parking' is displayed in white. Below this, there are several input fields: 'Selecciona un parking', 'Nombre', 'Localización', 'Fecha Reserva', 'Hora Inicio', and 'Número de plaza'. A dark blue 'Reservar' button is positioned below the 'Número de plaza' field. To the right of the input fields, there is a gray box titled 'Plazas parking' containing a 2x4 grid of colored squares. The top row has three green squares followed by one red square. The bottom row has one green square, one red square, one green square, and one green square.

Figura 44. Esquema de Reservar una plaza

O podrá dirigirse a ver sus datos y las reservas que ha hecho.



The image shows a user account management interface. At the top, there is a dark navigation bar with three links: "Reservar", "Cuenta", and "Salir". Below this, the main content area has a light blue background. On the left, under the heading "Account Information", there are six input fields arranged in two columns: "Nombre", "Apellido", "Username", "E-mail", "Password", and "Teléfono". Below these fields is a dark blue button labeled "Editar". To the right of the form is a grey rectangular box with the heading "Tus reservas".

Figura 45. Esquema de las reservas del usuario

Si se loguea como administrador podrá ver todos los datos de Reservas, Usuarios y Parkings



The image shows an administrator dashboard interface. At the top, there is a dark navigation bar with six links: "Reservas", "Usuarios", "Parkings", "Generar Datos", "Ubipark Analytics", and "Salir". Below this, the main content area has a light blue background with the word "INFORMACIÓN" centered in the middle.

Figura 46. Esquema de Información

En generar datos al darle al botón se generarán datos de cada tipo

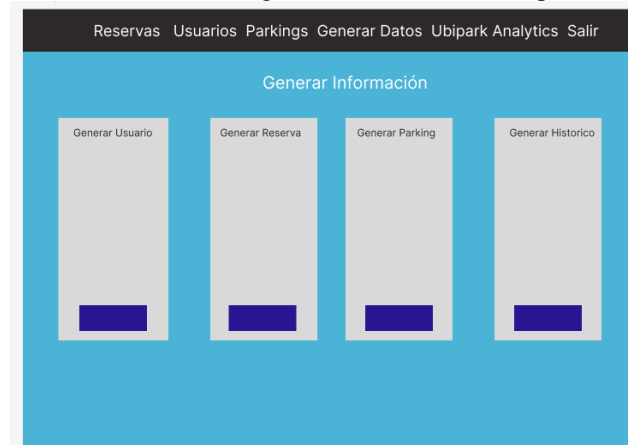


Figura 47. Esquema de información generada

Y por último podrá ver las estadísticas de la web en Ubipark Analytics

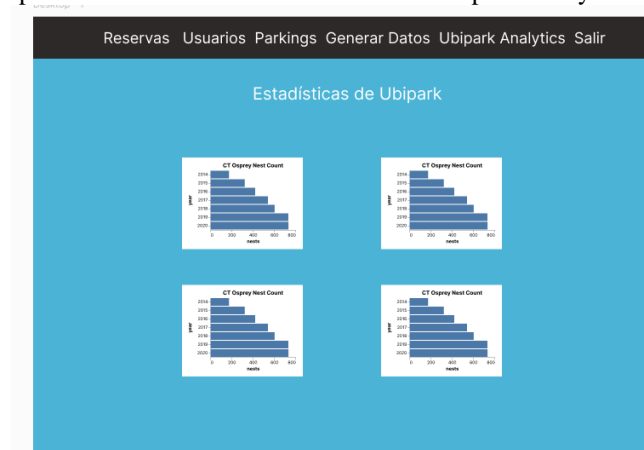


Figura 48. Esquema de estadísticas de Ubipark

5.3. Plan de desarrollo

METODOLOGÍA DE DESARROLLO:

Para este proyecto hemos dispuesto de un tiempo de aproximadamente dos semanas desde el final del segundo semestre y la fecha de entrega. Durante estas dos semanas hemos trabajado con una metodología SCRUM, con dos sprints de una semana cada uno, con reuniones diarias, revisiones al final de cada sprint y retrospectivas para mejorar continuamente el proceso.

FASES DEL PROYECTO Y DETALLE DE TAREAS:

- Sprint 1: Planificación, Diseño y Desarrollo de la capa de percepción.
 - Día 1-2: Investigación y Análisis de requisitos:
 - Análisis del mercado y competencia (Todos los miembros)
 - Compra de los materiales necesarios (Todos los miembros)
 - Día 3-7: Diseño de la Arquitectura del proyecto y desarrollo de la capa de percepción:
 - Diseño de la arquitectura del proyecto (Todos los miembros)
 - Realización de los programas necesarios para la capa de percepción, así como el la creación de la maqueta (Javier José Guzmán Rubio, Carlos Javier Prado Vázquez)
- Sprint 2: Desarrollo de las capas de Procesamiento y aplicación, y documentación.
 - Días 8-12: Desarrollo de las conexiones MQTT (Adrián Ajo)
 - Días 8-12: Desarrollo de la base de datos (David Fernandez, Adrián Ajo)
 - Días 8-12: Desarrollo de la Aplicación Web (David Fernandez)
 - Realización de la memoria del proyecto (Todos los miembros)

Siendo esta la distribución seguida para nuestro proyecto, debido al escaso tiempo disponible, todos los miembros han ayudado en las partes de los demás para que el proyecto avanzara lo más rápidamente posible en los momentos necesarios.

6. Conclusiones

Hemos trabajado con una variedad de hardware y software a lo largo de este proyecto para abordar conceptos importantes para el futuro, especialmente en el ámbito del Internet de las Cosas (IoT). Nuestras habilidades se han mejorado gracias a las herramientas y métodos que este proyecto nos ha brindado. Hemos cubierto una amplia gama de conocimientos en hardware y software, desde la implementación de sistemas con placas de desarrollo y sensores hasta la configuración de servidores, el almacenamiento de bases de datos y la creación de aplicaciones web.

La capacidad de crear un sistema inteligente como el nuestro, que facilita tareas cotidianas como la gestión de estacionamientos mediante el uso de lectores RFID y la comunicación MQTT, es notable. Este sistema no se limita a la detección y gestión de estacionamientos; puede adaptarse a una variedad de situaciones diarias. La posibilidad de integrar y automatizar estos procedimientos permite la innovación en una variedad de campos de manera simple y eficiente.

Un sistema similar al que hemos desarrollado podría ayudar a muchos conductores a resolver problemas de manera activa (encontrar rápidamente espacios de aparcamiento disponibles) y pasiva (ahorrar tiempo, evitar congestiones, reducir la incertidumbre sobre la disponibilidad de espacios de aparcamiento, prevenir la circulación innecesaria), lo que aumenta la comodidad y el bienestar.

7. Bibliografía

Referencias bibliográficas consultadas para la realización del trabajo. Esta bibliografía deberá estar en formato IEEE y referenciar, en caso necesario, en los distintos apartados del documento.

<https://www.youtube.com/watch?v=4V-wLR35HO0>

Anexo I – Manual de instalación

ARDUINO:

Para poder trabajar con las placas ESP32 y cargarles el código necesario tendrás que seguir estos pasos:

Descarga el Arduino IDE:

1. Visita la página de descargas del Arduino IDE
(<https://www.arduino.cc/en/software>)

2. Selecciona el sistema operativo que estás utilizando (Windows, macOS, Linux) y descarga la versión correspondiente.

Configurar el Arduino IDE para Soportar Placas ESP32

Agregar URL del Gestor de Placas

("(https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json)").

Es un enlace al archivo de configuración JSON que contiene la información necesaria para añadir soporte para las placas ESP32 en el Arduino IDE.

Instalar las Placas ESP32:

- Ve a **Herramientas > Placa > Gestor de placas**.
- En la ventana del Gestor de Placas, escribe **esp32** en el campo de búsqueda. Selecciona **esp32** by Espressif Systems y haz clic en **Instalar**.
- Espera a que finalice la instalación.

Seleccionar la Placa ESP32:

- Una vez instalada, ve a **Herramientas > Placa** y selecciona el modelo de tu placa ESP32 de la lista. Por ejemplo, si tienes una ESP32 Dev Module, selecciona **ESP32 Dev Module**.

Configurar el Puerto Serial:

- Conecta tu placa ESP32 a tu ordenador mediante un cable USB.
- Ve a **Herramientas > Puerto** y selecciona el puerto serial al que está conectada tu placa ESP32. (Muchas veces requiere instalar el driver ch340 este driver para poder utilizar este puerto)

Instalar el código necesario en cada ESP32:

- Dependiendo de qué placa estés configurando deberás cargar un código u otro ya sea una placa para la gestión de sensores o una de las placas de la maquina de entrada y salida

MOSQUITTO:

Para nuestro proyecto hemos utilizado Mosquitto como broker mqtt. Para poder conectar diferentes dispositivos se deben seguir los siguientes pasos:

1. Descargar Mosquitto. Para ello nos vamos a la página oficial y descargamos la versión correspondiente. En nuestro caso la siguiente:

Windows

- [mosquitto-2.0.18a-install-windows-x64.exe](#) (64-bit build, Windows Vista and up, built with Visual Studio Community 2019)

2. Una vez que lo tenemos descargado debemos cambiar el archivo .conf para permitir conexiones de distintos dispositivos. Para ello realizaremos una copia del archivo en el escritorio, y en la parte final, dentro del apartado de PSK based SSL/TLS support, añadiremos las siguientes dos líneas, las cuales nos permitirán recibir conexiones a través del puerto 1883.

```
listener 1883
allow_anonymous true
```

Figura 49. Mosquitto config

3. Una vez configurado el archivo debemos entrar en servicios de Windows, buscar el servicio con el nombre “Mosquitto Broker”, accederemos a sus propiedades y elegimos como tipo de inicio “Automático (inicio retrasado)”. Por último iniciamos el servicio, aplicamos y aceptamos:

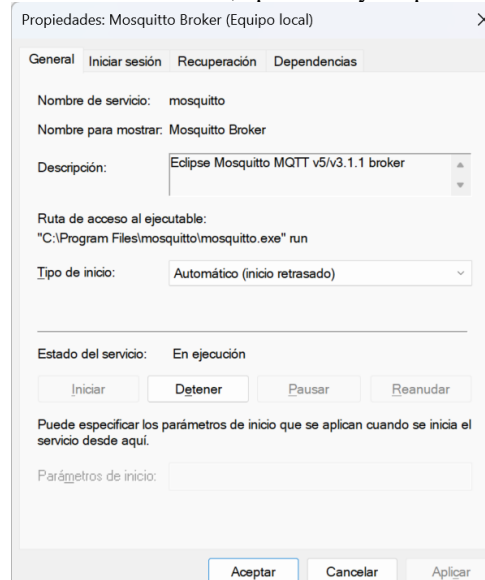


Figura 50. Servicio Mosquitto

4. Por último, debemos comprobar el correcto funcionamiento. Para ello comprobamos la ip a la que se conectarán los dispositivos. Esta ip la encontramos presionando la tecla Windows + R y escribiendo cmd en el menú que aparece:

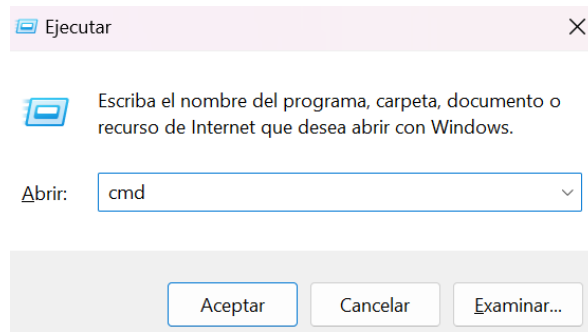


Figura 51. CMD

5. Una vez dentro de la CMD, escribimos ipconfig y copiamos la dirección IPv4.
6. Continuamos dirigiéndonos a la dirección en la que hemos guardado los archivos de Mosquito al descargarlos, y escribiendo en la barra de búsqueda de la carpeta dos veces “cmd”:

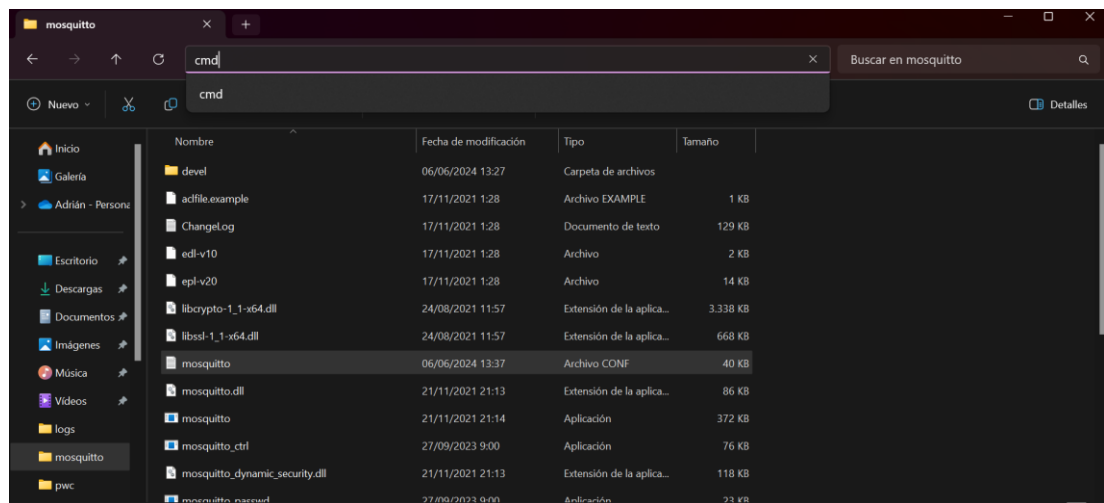


Figura 52. Dirección de los archivos de Mosquitto

7. Una vez tenemos abiertas estas dos CMDs, haremos que una se suscriba a un topic de prueba, y con la otra publicaremos mensajes a ese topic de prueba con los siguientes comandos:

```
C:\Program Files\mosquitto>mosquitto_sub -h 192.168.1.136 -p 1883 -t "Prueba/#" -v
C:\Program Files\mosquitto>mosquitto_pub -h 192.168.1.136 -p 1883 -t "Prueba/Hola" -m "Hola Mundo"
```

8. Por último enviamos el mensaje y vemos si lo recibimos:

```
C:\Program Files\mosquitto>mosquitto_sub -h 192.168.1.136 -p 1883 -t "Prueba/#" -v
Prueba/Hola Hola Mundo
```

Figura 53. Pruebas con Mosquitto

9. En caso de que se produzca un error o el mensaje no llegue, se debe reiniciar el servicio Windows de “Mosquitto Broker”.

Anexo II – Manual de Usuario de la aplicación Web

Este Manual de Usuario te guiará a través de las diferentes funcionalidades de la aplicación y te enseñará cómo utilizarlas de manera efectiva. Comenzarás abriendo el navegador y dirigiéndote a nuestra web, donde podrás ver la pantalla de bienvenida, en la barra superior encontrarás las diferentes secciones de inicio de sesión tanto de usuario nominal como de administrador y el registro. En el pie de la web podrás ver la información sobre todos nuestros parkings.

Bienvenida

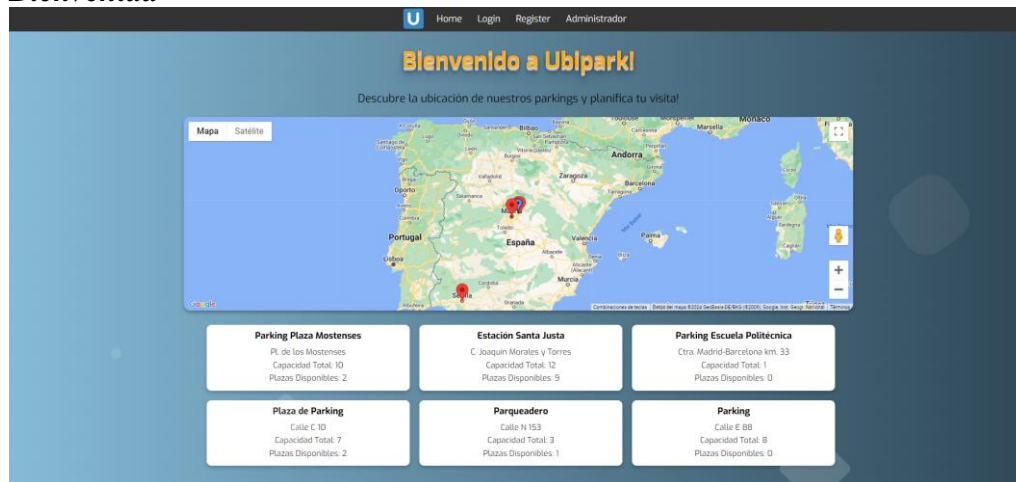


Figura 54. Home.html

Registro de usuario

Ya que no tienes cuentas vamos a proceder a clicar en Register, introduce la información que te pide en los campos correspondientes y pulsa Register!. Si todo ha salido bien debería mostrarte un mensaje como éste.

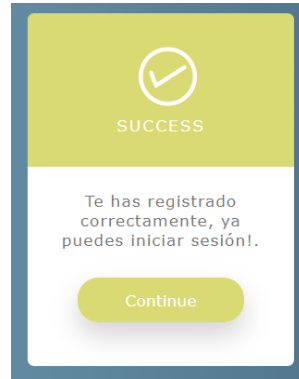


Figura 55. Registro exitoso

Iniciar sesión

A continuación, vamos a Login para iniciar sesión, deberás introducir el correo y la contraseña con la que te registraste anteriormente en el formulario, una vez estés seguro pulsa Login y te redirigirá tu cuenta, en caso fallido te mostrará un aviso de error.

A screenshot of the UbiPark Login form. The title "UbiPark Login" is at the top. Below it are two input fields: "Email" with the placeholder "Enter your email" and "Password" with the placeholder "Enter your password". A green "Login" button is at the bottom.

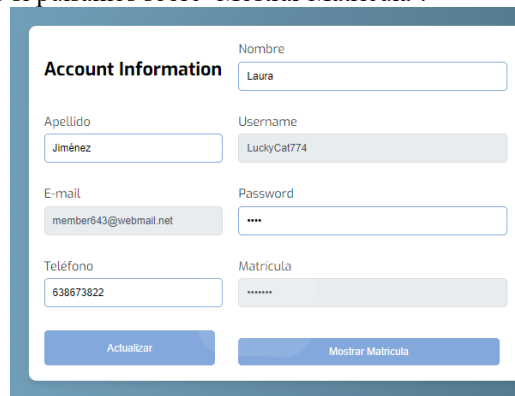
Figura 56. Login

A screenshot of the UbiPark Login form with an error message. The form is identical to the previous one, but with a red error message at the bottom: "❗Error: Your email or password is incorrect. Please try again.".

Figura 57. Failed login

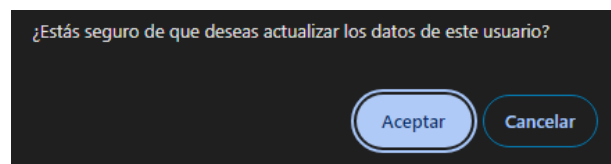
Editar Datos

Una vez hemos iniciado sesión estaremos en la pestaña de información del usuario donde para editar tus datos le daremos al botón azul donde pone editar y cambiaremos los cambios que queramos y le volveremos a dar al botón donde ahora pondrá Actualizar, la web nos pedirá confirmación y aceptaremos. También la matrícula se verá en texto claro si pulsamos sobre ‘Mostrar Matrícula’.



Account Information	
Nombre	<input type="text" value="Laura"/>
Apellido	<input type="text" value="Jiménez"/>
Username	<input type="text" value="LuckyCat774"/>
E-mail	<input type="text" value="member643@webmail.net"/>
Password	<input type="password" value="...."/>
Teléfono	<input type="text" value="638673822"/>
Matrícula	<input type="text" value="....."/>
<input type="button" value="Actualizar"/> <input type="button" value="Mostrar Matrícula"/>	

Figura 58. Account information

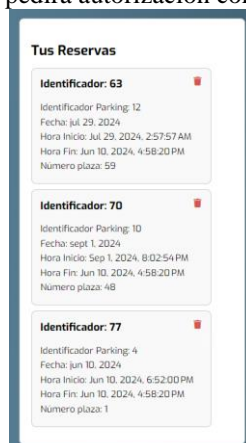


¿Estás seguro de que deseas actualizar los datos de este usuario?

Figura 59. Confirm

Eliminar Reserva

A la derecha de los datos del usuario podemos ver todas las reservas que tiene un usuario, para borrarlo presionamos en el ícono de la basura roja de la reserva que queramos eliminar, la web nos pedirá autorización como anteriormente.



Tus Reservas

Identificador: 63

Identificador Parking: 12
Fecha: jul. 29. 2024
Hora Inicio: Jul. 29. 2024, 2:57:57 AM
Hora Fin: Jun 10. 2024, 4:58:20 PM
Número plaza: 59

Identificador: 70

Identificador Parking: 10
Fecha: sept. 1. 2024
Hora Inicio: Sep 1. 2024, 8:02:54 PM
Hora Fin: Jun 10. 2024, 4:58:20 PM
Número plaza: 48

Identificador: 77

Identificador Parking: 4
Fecha: jun 10. 2024
Hora Inicio: Jun 10. 2024, 6:52:00 PM
Hora Fin: Jun 10. 2024, 4:58:20 PM
Número plaza: 1

Figura 60. Eliminar Reserva

Reservar una plaza de parking

Para reservar una plaza debemos dirigirnos a Reservar en la barra horizontal superior, en el desplegable seleccionamos un parking, esperamos a que carguen las plazas, hacemos clic sobre una que esté libre y rellenamos la fecha de reserva y la hora a la que llegarás. La web mostrará un aviso de cuando la reserva se haya llevado a cabo.

Reserva un Parking

Selección un Parking:

2

Nombre

Estación Santa Justa

Location

C. Joaquín Morales y Torres

Fecha Reserva

14/06/2024

Hora Inicio

12:30

Número de plaza

30

Reservar

Plazas de Parking

Pulsa sobre la plaza que quieras reservar!

24 25 26 27 28

29 30 31 32 33

34 35

Ocupado Libre

Figura 61. Reservar plaza

Iniciar sesión como Administrador

Desde la ventana de bienvenida deberemos darle a Administrador e introducir las credenciales, si es correcto nos redirigirá a otra ventana.

Admin Login

Username

admin

Password

Login

Figura 62. Admin Login

Editar/eliminar datos de reservas, usuarios y parkings

Si nos dirigimos a cualquiera de esas secciones podremos ver todos los datos de la aplicación, en la columna acciones podemos editar si presionamos en el icono del lápiz o eliminar si presionamos sobre el icono de la papelera.

ID Parking	Nombre	Direccion	Ciudad	Codigo Postat	Capacidad Total	Plazas Disponibles	Acciones
1	Parking Plaza Mostenses	Pl. de los Mostenses	Madrid	28015	10	2	 
2	Estación Santa Justa	C. Joaquín Morales y Torres	Sevilla	41003	12	9	 
4	Parking Escuela Politécnica	Ctra. Madrid-Barcelona km. 33	Alcalá de Henares	28805	1	0	 
10	Plaza de Parking	Calle C 10	Murcia	32839	7	2	 
11	Parqueadero	Calle N 153	Sevilla	52734	3	1	 
12	Parking	Calle E 88	Barcelona	05895	8	0	 

Figura 63. Edit Users

Generar datos aleatorios

Nos dirigimos a Generar Datos y si queremos generar un Usuario, por ejemplo, presionamos el botón de ‘Generar User’ y esperamos a que se procese, una vez haya ido todo bien se mostrará en pantalla el usuario generado.

Generar Usuario

Email

support179@test.uah

Nombre

Ana

Apellido

Fernández

Username

LuckyBear419

Password

h%+0YnlYR_g

Teléfono

767032778

Fecha Registro

2024-06-10

Matrícula

9746CCC

Generar User

Figura 64. Generar Usuario

Anexo III – Simulación de datos

Para la creación de datos aleatorios hemos creado una clase en Java llamado `GeneradorDatos.java` donde tenemos varias listas con cadenas aleatorias dependiendo qué datos queremos generar aleatoriamente. Por ejemplo si queremos crear un correo de usuario aleatorio tenemos las siguientes estructuras:

```
private static final String[] USERNAMES = {
    "user", "member", "guest", "contact", "info", "support", "service"
};

private static final String[] DOMAINS = {
    "example", "test", "demo", "ubipark", "webmail"
};

private static final String[] TLDs = {
    "com", "net", "es", "edu", "uah", "gov"
};
```

Figura 65. Random Data

Para la generación del correo tenemos la siguiente función que retorna un String

```
private static final Random RANDOM = new Random();

public static String generarEmail() {
    String username = USERNAMES[RANDOM.nextInt(i: USERNAMES.length)];
    String domain = DOMAINS[RANDOM.nextInt(i: DOMAINS.length)];
    String tld = TLDs[RANDOM.nextInt(i: TLDs.length)];

    int number = RANDOM.nextInt(i: 1000);

    return username + number + "@" + domain + "." + tld;
}
```

Figura 66. Generar email

Por otro lado tenemos un servlet encargado de mandar por una petición POST los datos generados anteriormente, por ejemplo en caso de la generación de un usuario de forma aleatoria:

```

String nombre = GeneradorDatos.generarNombre();
String apellido = GeneradorDatos.generarApellido();
String username = GeneradorDatos.generarUsername();
String email = GeneradorDatos.generarEmail();
String password = GeneradorDatos.generarPassword();
String telefono = GeneradorDatos.generarTelefono();
long millis = System.currentTimeMillis();
Date fechaActual = new Date(millis);
String matricula = Logic.generarMatricula();

ConnectionDDBB conector = new ConnectionDDBB();
con = conector.obtainConnection(true);

String sql = "INSERT INTO users(nombre, apellido, email, password, telefono, fecha_registro, matricula, username) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
statement = con.prepareStatement(sql);
Log.log.info("Query => {}", statement);

statement.setString(1, nombre);
statement.setString(2, apellido);
statement.setString(3, email);
statement.setString(4, password);
statement.setString(5, telefono);
statement.setDate(6, fechaActual);
statement.setString(7, matricula);
statement.setString(8, username);

int result = statement.executeUpdate();

if (result > 0) {
    Log.log.info("Usuario registrado con éxito!");
    JSONObject json = new JSONObject();
    json.put("nombre", nombre);
    json.put("apellido", apellido);
    json.put("email", email);
    json.put("password", password);
    json.put("telefono", telefono);
    json.put("fecha_registro", fechaActual);
    json.put("matricula", matricula);
    json.put("username", username);
    out.print(json.toString());
} else {
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    JSONObject errorJson = new JSONObject();
    errorJson.put("error", "Error al registrar el usuario");
    out.print(errorJson.toString());
    Log.log.error("Error en el registro");
}

```

Figura 67. Código Servlet generarUsuario

Obtenemos los datos aleatorios de nuestra clase `GeneradorDatos`, primero los insertamos en la base de datos y si la inserción ha sido exitosa mandamos un json con los datos generados para que en la sección de la web “Generar Datos” el administrador pueda ver que datos se han generado aleatoriamente.

Hemos seguido este proceso para todos los servlets de generación de datos, es decir, desde la web al darle al botón “Generar X” mandamos una solicitud POST al Servlet correspondiente y que gracias a la clase `GeneradorDatos.java` obtiene los datos que si son correctos los inserta en la BBDD y tras ello los manda a la web en formato JSON y se muestran en el formulario correspondiente.

Anexo V – Hojas de características de los componentes

[ESP32 Datasheet\(PDF\) - ESPRESSIF SYSTEMS \(SHANGHAI\) CO., LTD. \(alldatasheet.com\)](#)

[LCDscreen.PDF \(arduino.cc\)](#)

[Leantec.ES-HC-SR04.pdf](#)

[Módulo semáforo LED compatible con Arduino \(az-delivery.de\)](#)

[Pantalla LCD de 16x2, Display de 16x2, LCD de 16x2 - Winstar Display keypad,teclado,matricial,4x4 \(altronics.cl\)](#)

[<https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf>](#)