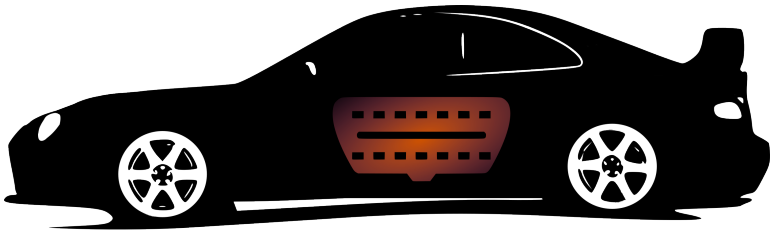
PTI

Car Tracking



Sergio Sanz
Ferran Fuentes
Ricard Medina
Marcel Sánchez
Adriana López

FIB (UPC), 1 de junio de 2023

Índice

1. Introducción

CarTracking es un proyecto que tiene el objetivo de dar al usuario un método fácil y sencillo de consultar los datos de la centralita de su coche. Esto le permite conocer consumos, velocidad media, kilómetros recorridos ente otros datos que el coche tiene pero este no muestra.

Gracias a un lector OBD2 [30] Bluetooth genérico, la aplicación móvil de nuestro sistema y nuestra aplicación web, el usuario puede cómodamente acceder a los datos que recolecta su coche. Además damos la posibilidad de certificar mediante blockchain el kilometraje del vehículo.

1.1. Soluciones actuales y otros proyectos

En el mercado existen sistemas similares que hacen uso del conector OBD2 para proporcionar datos de la centralita del coche; sin embargo muchos de estos sólo permiten consultar los datos en un teléfono móvil en el mismo momento y, aún que muchos permiten el guardado de los datos, no dan una forma cómoda de visualizarlos.

Además anteriormente en esta asignatura, se han desarrollado dos proyectos que guardan ciertas similitudes con el nuestro:

- Clear Mileage [18]: Es un proyecto que tiene como objetivo certificar por blockchain el kilometraje de un coche de cara a la compraventa de coches de segunda mano.
- Car Locator [17]: Es un proyecto que crea un dispositivo para monitorear y rastrear un vehículo.

Ambos proyectos, al igual que el nuestro hace uso del puerto OBD2

1.2. Posibles usos comerciales

El punto fuerte y distintivo de CarTracking es el sistema de validación por blockchain de los Km recorridos por un vehículo. Esto podría servir para darle un valor añadido a esos coches que se van a vender de segunda mano ya que se estaría asegurando que los kilómetros realizados por ese vehículo no han sido modificados aportando un extra de confianza a esos usuarios que van hacer la compra.

Dado la gran cantidad de marcas dedicadas a la creación de dispositivos

OBD2, un posible uso comercial interesante sería intentar tener un acuerdo de exclusividad con alguna de estas marcas de manera que sea un aliciente para el usuario comprar ese OBD2 y no otro.

2. Diseño

El diseño por el hemos optado finalmente es el siguiente:

- Aplicación móvil para recoger datos del coche.
- Frontend para ver los datos extraídos del coche (requiere ser usuario de la aplicación).
- Frontend para consultar la cantidad de kilómetros hechos por un coche con cierta matrícula (abierto a todo el mundo).
- Backend para recibir los datos de la aplicación, procesarlos y guardarlos.
- Blockchain para certificar el kilometraje.
- Cliente de blockchain, sobre este servicio recaen los costes de certificar en la blockchain.

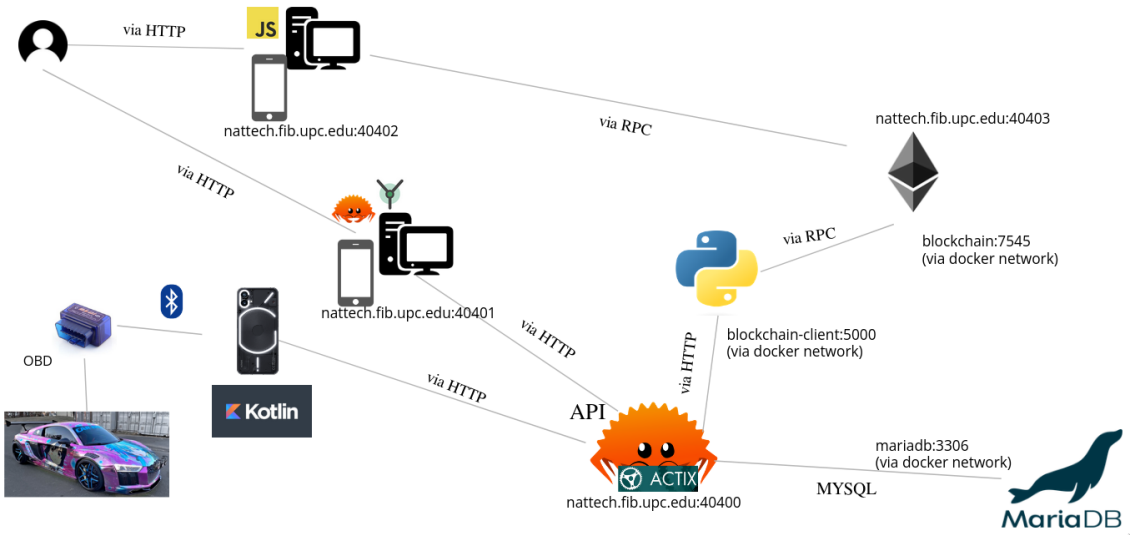


Figura 1: Esquema final

3. Arquitectura de la aplicación

3.1. Backend

Para el desarrollo del backend hemos escogido el lenguaje de programación *Rust*. No había ninguna razón técnica para hacerlo pero de todas las opciones que teníamos para hacerlo nos pareció más interesante que por ejemplo hacerlo con un típico *node* o *java*. Como veremos más adelante, el hecho de también hacer el frontend con *Rust* nos permite compartir librerías y structs comunes.

3.1.1. Objetivo

El objetivo del backend es asegurar la intercomunicación y interoperabilidad entre los diferentes servicios que tenemos.

Tiene que ofrecer una API que permita:

- Registrar/logear usuarios.
- Recoger los datos del coche (enviados mediante la aplicación móvil).
- Guardar estos datos en una base de datos.
- Certificar el cuentakilómetros del coche en una blockchain de etherum.
- Poder pedir los datos del coche desde un frontend web.

Estas funcionalidades han sido implementadas sobre HTTP y son reutilizables desde nuevas implementaciones de frontends mientras se sigan los parámetros definidos en la documentación de más adelante.

3.1.2. Tecnologías utilizadas

En concreto se ha utilizado el framework Actix-web [29] para facilitar todo lo que es servir el backend y sus diferentes rutas con comodidad.

Para la comunicación con la base de datos se ha usado el ORM que proporciona diesel [9] y para la gestión de esta (migraciones...) diesel-cli [8].

3.1.3. Especificación de la API

A continuación veremos un resumen de la documentación de los endpoints que se puede encontrar en:

`https://github.com/s4izh/car-tracking-fullstack/blob/main/backend/test/README.md`.

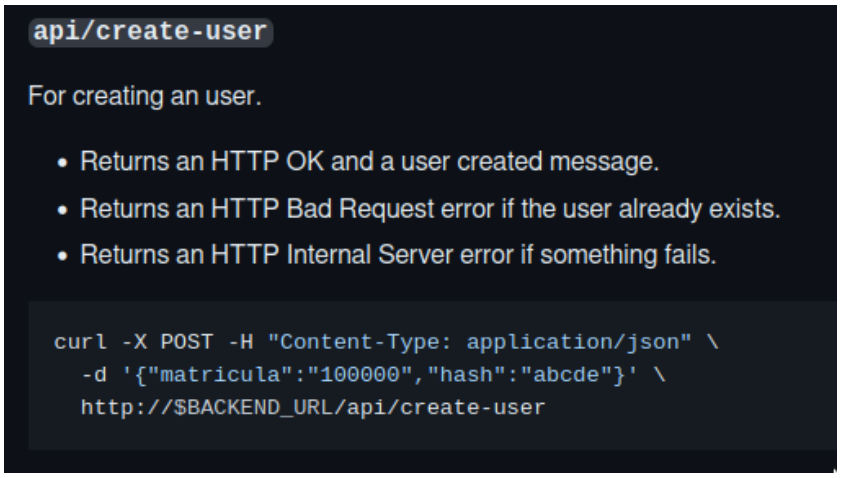


Figura 2: Endpoint para crear usuarios

El sistema de autenticación es muy simple, solo comprueba que el hash enviado sea el correcto (el que se ha guarda cuando se crea el usuario).

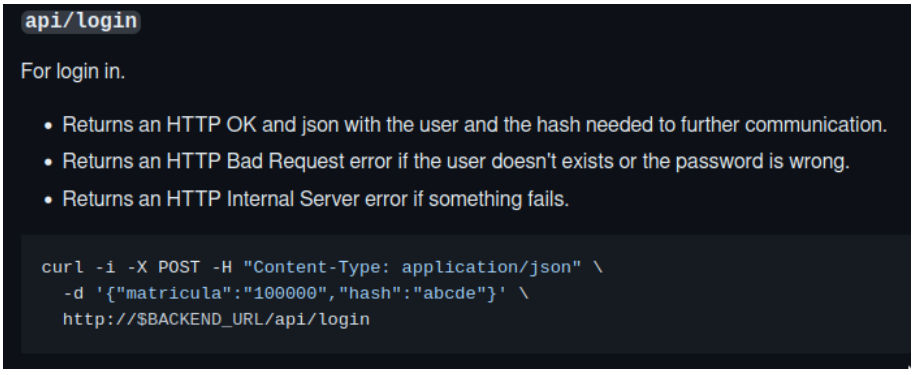


Figura 3: Endpoint para logear usuarios

El siguiente endpoint está pensado para utilizarse desde nuestra aplicación móvil, la cual es la que recoge los datos, los agrupa por viajes y posteriormente los envía aquí. El formato del JSON enviado lo veremos más adelante en el apartado de la aplicación.

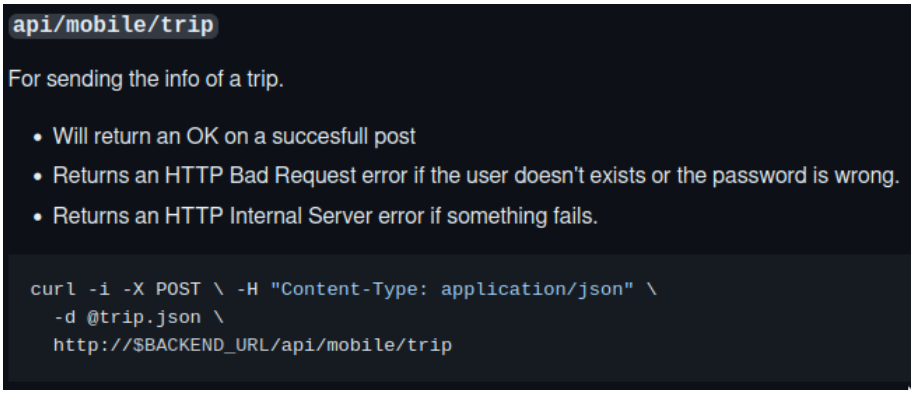


Figura 4: Endpoint para enviar los datos recogidos en el coche

El endpoint *get-trips* es el que llama el frontend para obtener los datos de los viajes de un usuario.

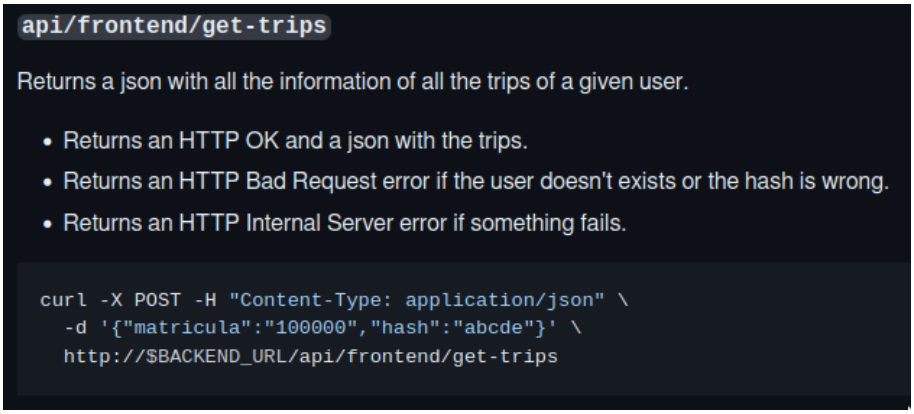


Figura 5: Endpoint para obtener los datos de todos los viajes de un usuario

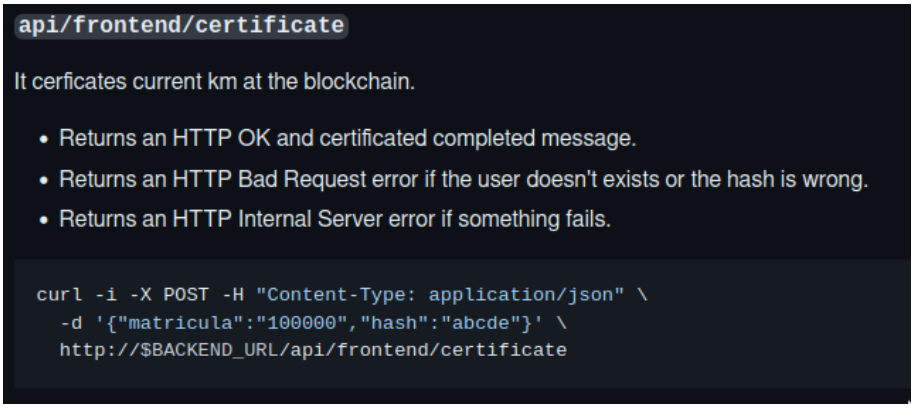


Figura 6: Endpoint para certificar el kilometraje actual

Como podemos ver es necesario enviar las credenciales en cada petición para asegurarnos de que el usuario es el auténtico, esto es algo que nos gustaría cambiar a algún sistema de autenticación más robusto.

3.1.4. Extractos del código

Esta es la función main del backend en la cual se obtiene una pool de conexiones con la base de datos y se montan las diferentes rutas.

```rust
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    env_logger::init();

    // set DATABASE_URL and SERVER_PORT
    let database_url = env::var("DATABASE_URL").expect("DATABASE_URL mu

    let db_pool = {
        let manager: ConnectionManager<MysqlConnection> =
            ConnectionManager::<MysqlConnection>::new(&database_url);
        Pool::builder()
            .build(manager)
            .expect("Error building a connection pool")
    };
    let backend_port: u16 = env::var("BACKEND_PORT")
        .expect("BACKEND_PORT must be set")
        .parse()
        .expect("BACKEND_PORT must be a number");

    HttpServer::new(move || {
        let logger = actix_web::middleware::Logger::default();
        App::new()
            .wrap(logger)
            .wrap(Cors::permissive())
            .app_data(web::Data::new(db_pool.clone()))
            .service(
                scope("/api")
                    .service(
                        scope("/frontend")
                            .service(frontend::test)
                            .service(frontend::certificate)
                            .service(frontend::get_trips)
                    )
                    .service(
                        scope("/mobile")
                            .service(mobile::add_trip)
                    )
                    .service(common::create_user)
                    .service(common::login),
            )
            .default_service(web::route().to(not_found))
    })
    .bind(("0.0.0.0", backend_port))?
    .run()
    .await
}
```

Figura 7: Función main

Y lo siguiente es un ejemplo de una query a la base de datos utilizando el ORM de diesel.

```
let result = schema::users::dsl::users
    .filter(matricula.eq(&user.matricula))
    .filter(hash.eq(&user.hash))
    .first::<models::BdUser>(&mut *conn);
match result {
    Err(diesel::NotFound) => {
        return Ok(HttpResponse::BadRequest().body("User doesn't exists or bad password"))
    }
    Ok(_) => (),
    Err(_) => {
        return Ok(HttpResponse::InternalServerError().body(format!("Error finding user")))
    }
}
```

Figura 8: Extracto de código que verifica un usuario

3.2. Base de datos

3.2.1. Motivación

Debido a la naturaleza de nuestro trabajo, tener un buen sistema de almacenamiento, control y gestión de datos es imprescindible. Buscábamos un sistema de gestión que fuera funcional, compatible con el resto de nuestro proyecto y fácil de migrar y escogimos MariaDB.

Dada la cantidad de datos a procesar y la variedad entre ellos, inicialmente organizamos todo en varias tablas clasificando por categorías las diferentes funcionalidades que pretendíamos ofrecer. Todo esto se implementó de forma local con intención de conectarlo con el resto del proyecto y exportar las tablas con alguna herramienta. Finalmente cambiamos todo, desde el esquema de organización de datos, tanto las tablas y el contenido de cada una de ellas, como la localización de las mismas, obteniendo lo que hemos considerado como el esquema más óptimo para almacenar toda la información que guardamos en nuestra base de datos.

3.2.2. Organización

La base de datos está formada por tres tablas distintas, que comparten campos e información, de forma que están todas comunicadas entre sí, actualizándose en paralelo al insertar algo en cualquiera de ellas. Las tres tablas son: *car-data*, *users* y *trips*. En cada una de ellas podemos almacenar los siguientes datos:

Figura 9: Relación entre las 3 tablas

```
MariaDB [db]> describe car_data;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| id                  | int(11)     | NO   | PRI | NULL    |       |
| trip                | int(11)     | NO   |     | NULL    |       |
| timestamp           | varchar(255)| NO   | PRI | NULL    |       |
| speed               | int(11)     | NO   |     | NULL    |       |
| rpm                 | int(11)     | NO   |     | NULL    |       |
| throttle            | double      | NO   |     | NULL    |       |
| engine_load         | double      | NO   |     | NULL    |       |
| engine_coolant_temp | double      | NO   |     | NULL    |       |
| oil_temp            | double      | NO   |     | NULL    |       |
| fuel_level          | double      | NO   |     | NULL    |       |
| fuel_consumption    | double      | NO   |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
11 rows in set (0.005 sec)
```

Figura 10: Contenido de la tabla car-data

```
MariaDB [db]> describe users;
+--------------+--------------+------+-----+---------------------+----------------+
| Field        | Type         | Null | Key | Default             | Extra          |
+--------------+--------------+------+-----+---------------------+----------------+
| id           | int(11)      | NO   | PRI | NULL                | auto_increment |
| matricula    | varchar(255) | NO   | UNI | NULL                |                |
| hash         | varchar(255) | NO   |     | NULL                |                |
| total_km     | int(11)      | NO   |     | 0                   |                |
| trip         | int(11)      | NO   |     | 0                   |                |
| date_created | timestamp    | NO   |     | current_timestamp() |                |
+--------------+--------------+------+-----+---------------------+----------------+
6 rows in set (0.001 sec)
```

Figura 11: Contenido de la tabla users

```
MariaDB [db]> describe trips;
+-----------------+-------------+------+-----+---------------------+-------+
| Field           | Type        | Null | Key | Default             | Extra |
+-----------------+-------------+------+-----+---------------------+-------+
| id              | int(11)     | NO   | PRI | NULL                |       |
| trip            | int(11)     | NO   | PRI | NULL                |       |
| km              | int(11)     | NO   |     | NULL                |       |
| max_speed       | int(11)     | NO   |     | NULL                |       |
| avg_speed       | double      | NO   |     | NULL                |       |
| fuel_percentage | double      | NO   |     | NULL                |       |
| duration        | int(11)     | NO   |     | NULL                |       |
| trouble_codes   | varchar(255) | NO   |     | NULL                |       |
| date_created    | timestamp   | NO   |     | current_timestamp() |       |
+-----------------+-------------+------+-----+---------------------+-------+
9 rows in set (0.001 sec)
```

Figura 12: Contenido de la tabla trips

3.2.3. Diesel CLI

Una herramienta clave para la gestión y administración de nuestra base de datos ha sido Diesel CLI. Diesel CLI es una herramienta de línea de comandos que complementa al framework de Diesel y proporciona funcionalidades adicionales para el desarrollo de aplicaciones en Rust. Está pensado principalmente para aplicaciones que interactúan con bases de datos relacionales, como es el caso de nuestro proyecto.

En nuestro caso utilizaremos esta herramienta para generar tanto las migraciones de las bases de datos como para generar el código Rust necesario para interactuar con la base de datos a partir del esquema de la base de datos.

3.2.4. Migraciones

Las migraciones en Diesel CLI permiten gestionar los cambios en las estructuras de la base de datos a medida que la aplicación a la que pertenecen evoluciona.

Nos permiten crear, modificar y eliminar tablas, columnas, índices y otros elementos dentro del esquema de la base de datos.

Para crear una migración como tal, debemos utilizar el comando "diesel migration generate", con esto se creará un directorio llamado */migrations* en el que tendremos otro directorio que contendrá 2 archivos *".sql"*:

- up.sql
- down.sql

Es dentro de estos archivos que debemos instanciar nuestro SQL para crear las tablas de la base de datos. En *"up.sql"* instanciaremos todo el SQL necesario para la creación de nuestras tablas, en nuestro caso las 3 instancias create table para generar *users, car-data y trips.*

En "down.sql" revertiremos todo lo que hayamos hecho en up.sql, por lo que haremos un drop de las 3 tablas que hemos instanciado el otro fichero.

```sql
-- Your SQL goes here
CREATE TABLE users (
    id              INT             NOT NULL AUTO_INCREMENT,
    matricula       VARCHAR(255)    NOT NULL UNIQUE,
    hash            VARCHAR(255)    NOT NULL,
    total_km        INT             NOT NULL DEFAULT 0,
    trip            INT             NOT NULL DEFAULT 0,
    date_created    TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
);

CREATE TABLE trips (
    id              INT             NOT NULL,
    trip            INT             NOT NULL,
    km              INT             NOT NULL,
    max_speed       INT             NOT NULL,
    avg_speed       INT           NOT NULL,
    fuel_percentage INT           NOT NULL,
    duration        INT             NOT NULL,
    trouble_codes   VARCHAR(255)    NOT NULL,
    date_created          TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id, trip)
);

CREATE TABLE car_data (
  id                INT             NOT NULL,
  trip              INT             NOT NULL,
  timestamp         VARCHAR(255)    NOT NULL,
  speed             INT             NOT NULL,
  rpm               INT             NOT NULL,
  throttle          INT           NOT NULL,
  engine_load       INT           NOT NULL,
  engine_coolant_temp INT         NOT NULL,
  oil_temp          INT           NOT NULL,
  fuel_level        INT           NOT NULL,
  fuel_consumption  INT           NOT NULL,
  PRIMARY KEY (id, trip, timestamp)
);
```

Figura 13: Contenido fichero *up.sql*

```sql
-- This file should undo anything in `up.sql`
DROP TABLE users;
DROP TABLE trips;
DROP TABLE car_data;
```

Figura 14: Contenido fichero *down.sql*

Esquema de la base de datos

Tal y como ya hemos explicado, Diesel CLI también nos genera de forma automática un esquema de la base de datos "schema.rs". Este archivo contiene la definición en código Rust de las tablas, columnas, relaciones y elementos que tenemos en nuestra base de datos. Utiliza una sintaxis específica de Diesel para definir este esquema. Dispone de varias secciones y cada una de ellas representa una de las tablas de la base de datos.

Es un archivo muy importante para el funcionamiento de Diesel ya que es lo que utiliza para generar las consultas SQL de forma segura, proporcionando métodos y estructuras que facilitan la interacción entre la base de datos y el resto de puntos de nuestro sistema, haciendo que la inserción de datos funcione sin problemas y que tanto las peticiones como las consultas sean factibles desde el *frontend* y a la hora de certificar con *blockchain* el kilometraje.

3.3. Blockchain

3.3.1. Utilidad

El uso de una blockchain en este proyecto nos aporta un extra de utilidad al mismo a más de ofrecer una gran herramienta para darle valor a la plataforma. Con la blockchain implementada se le permite al usuario la posibilidad de certificar el kilometraje de su vehículo de forma gratuita y sin necesidad de tener ninguna cuenta Ethereum ni conocimiento alguno sobre el ámbito. Esto se debe a que el encargado de tramitar las inserciones de transacciones en la blockchain es el back-end de la aplicación y, a ojos del usuario, él solo tiene que pulsar un botón de certificar en la web de la aplicación. Aparte de esto, también se ofrece otro servicio web abierto para todos los usuarios que no pide acceso mediante login para poder consultar el kilometraje de un vehículo y la fecha de la última certificación. Para consultar el kilometraje de un vehículo lo único que se debe hacer es introducir la matrícula del vehículo y clicar en un botón para aceptar. Con este servicio ofrecemos la posibilidad al dueño del vehículo de asegurarle a terceros cuál es el kilometraje de su vehículo y cuando fue la última vez que se certificó. Dicho servicio puede ser de gran utilidad para dar información verificada en la venta de vehículos de segunda mano o incluso para poder informar correctamente del estado del vehículo en una revisión. No hay acceso mediante login a este servicio para que cualquier usuario pueda ver el kilometraje certificado de un vehículo, sea usuario de nuestra aplicación o no.

3.3.2. Resumen del funcionamiento

Como ya se ha explicado, en nuestro proyecto recopilamos diversa información sobre el estado y condiciones del vehículo. Uno de los valores medidos es la velocidad media, la cual utilizamos para hacer una media y obtener el kilometraje recorrido por el vehículo en un trayecto. Posteriormente se suma el kilometraje obtenido con el que ya se tenía del propio vehículo en la BD para obtener el kilometraje total del coche.

- **Consulta de datos**
  Para consultar los datos se ha creado un front-end propio, gestionado con un archivo en JavaScript encargado de comunicarse con la blockchain de Ganache para hacer las consultas pertinentes. El front-end desarrollado consta de un único campo de inserción en el que se indica la matrícula que se quiere consultar y mediante la pulsación de un botón se hace la request. Posteriormente se rellenan los campos pertinentes al kilometraje y la fecha en la que se hizo la última certificación. En caso de no existir la matrícula que se quiere consultar se lanza una excepción avisando del error.
  Este archivo JavaScript encargado de manejar los request se conecta directamente a la blockchain desplegada con Ganache en la que consulta os datos. Al tratarse de solo consultas este proceso no provoca gastos de ETH.

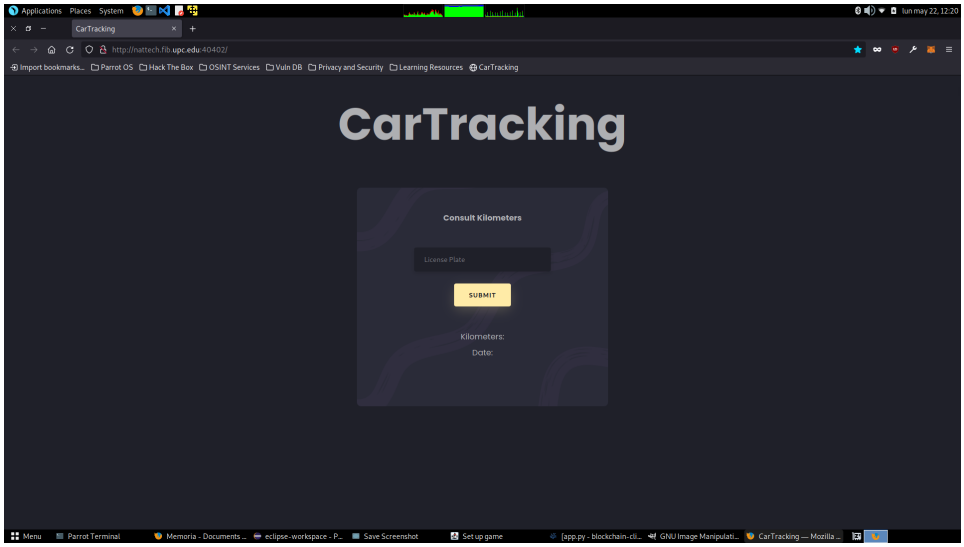


Figura 15: Diseño final del front-end para la blockchain.

- **Smart Contracts**
  Para desarrollar nuestro smart contract decidimos utilizar solidity. En él creamos un map en el que guardar la totalidad de los vehículos y creamos un struct para cada vehículo en el que se guarda la información esencial para ser certificada. Aunque en el smart contract ofrecemos la posibilidad de hacer la certificación de kilometraje y timestamp de forma separada la función que finalmente se ha optado por usar y que el backend llama en el momento de crear una transacción es Certify-Kilometer. En esta función se certifican el kilometraje y el timestamp a la vez.

```
function certifyKilometer(string memory licensePlate,
    uint256 currentKilometer) public {
    if (bytes(cars[licensePlate].licensePlate).length == 0)
    {
        cars[licensePlate] = Car(licensePlate,
        currentKilometer, block.timestamp);
    }
    else {
        require(currentKilometer >=
```

```
        cars[licensePlate].kilometer, "New kilometer value
        should not be less than previous value");
        cars[licensePlate].kilometer = currentKilometer;
        cars[licensePlate].certifiedTimestamp = block.timestamp;
    }
}
```

Para obtener los datos de la blockchain se llama a la función getCetrification que devuelve el valor del kilometraje y el timestamp correspondiente a la matrícula pasada por parámetro.

```
function getCarCertification(string memory _licensePlate)
public view returns (uint256, uint256) {
        return (cars[_licensePlate].kilometer,
 cars[_licensePlate].certifiedTimestamp);
}
```

- **Despliegue de la Blockchain**
  Para este proyecto, debido a que no va a lanzarse de forma comercial, se ha decidido utilizar Ganache de Truffle, que simula una blockchain de Ethereum sin la necesidad de hacer transacciones reales.
  El comando utilizado para inicializar Ganache en el que se desplegarán nuestros smart contracts es "ganache-cli -p 7545 -i 5777 -h 0.0.0.0 -m "tu frase semilla de ganache"". En él especificamos que usamos el puerto 7545 para la comunicación, que usamos la red de development(5777), que se escucharán todas las solicitudes de red entrantes que lleguen a cualquier dirección IP disponible en la máquina y que la frase semilla será "tu frase semilla de ganache". Esta frase semilla ha sido de gran utilidad, ya que al inicializar Ganache siempre con esta semilla los contratos se despliegan en la misma dirección y con el mismo abi, lo que nos permite comunicarnos con el contrato desde el docker donde está el servidor de inserción de datos mediante web3. Sin esta semilla no sería posible insertar los datos desde el código python debido a que la dirección no se podría hardcodear en el código y no tendríamos cómo referirnos a el smart contract desplegado.

3.4. Blockchain Client

Para proporcionar una aplicación más user firendly decidimos que el usuario no tuviera que encargarse de ningún aspecto mínimamente complejo rela-

cionado con la inserción de datos en la blockchain, por esta razón creamos el servidor intermediario para insertar datos y hacemos que se realice la inserción con un click. Con esta metodología nuestra blockchain tiene un único usuario capaz de insertar valores, que es el servidor python. Para ello tenemos también hardcodeada en el código python una de las direcciones que se genera en Ganache para poder identificar al cliente en las inserciones.

3.4.1. Funcionamiento

Para poder insertar los datos correctamente se utiliza la librería web3, que nos permite insertar los datos en una blockchain que no está en local. Para hacerlo utilizamos una semilla al iniciar el cliente de Ganache, para asegurarnos de que el smart contract se despliega siempre en la misma dirección. De esta forma en el código de python está hardcodeada la dirección y el abi del contrato desplegado, para que con web3 nos conectemos a la instancia correcta.
Entonces, el usuario, después de loguearse en la página principal, tiene la posibilidad de hacer click en un botón de certificar mediante el cuál hace un request para que el valor que hay guardado en la BD sobre su kilometraje se guarde en la blockchain. Esta petición se enruta a través del backend y mediante el servidor que hace de cliente se hace el tratamiento de esta request. Para hacerlo se dispone de un código en python que recibe los valores de la matrícula y kilometraje que hay que certificar y se conecta a la blockchain de Ganache local utilizada e introduce la transacción deseada. En la transacción se accede a la función encargada de la inserción de datos del archivo en solidity desplegado en Ganache.

3.4.2. Futuras optimizaciones

En cuanto al gasto de ETH, todos los gastos son generados por el cliente que proporcionamos, por ende, no se repercuten en el cliente. Ya que nuestro trabajo no es de uso comercial y usamos Ganache donde no gastamos ETH reales esto no supone un problema, pero pensando en una aplicabilidad real se debería introducir algún método para limitar la cantidad de certificaciones posibles o hacer que el cliente pagara por ellas.

3.5. Frontend web

El frontend web tiene como objetivo mostrar la información de los trayectos recogida por el OBD2, además de dar la posibilidad de certificar por blockchain el kilometraje. También es el medio por el cual se puede registrar un nuevo usuario a la plataforma de Car Tracking.

3.5.1. Resumen del funcionamiento

La vista principal de la web es una página simple donde se muestra el nombre de la plataforma y un link al front-end para comprobar si una matricula está o no certificada. En la cabecera de la web inicial encontramos unos botones de Login o SingUp, que como su nombre indica sirven para acceder a las vistas para iniciar sesión o darse de alta a la plataforma.

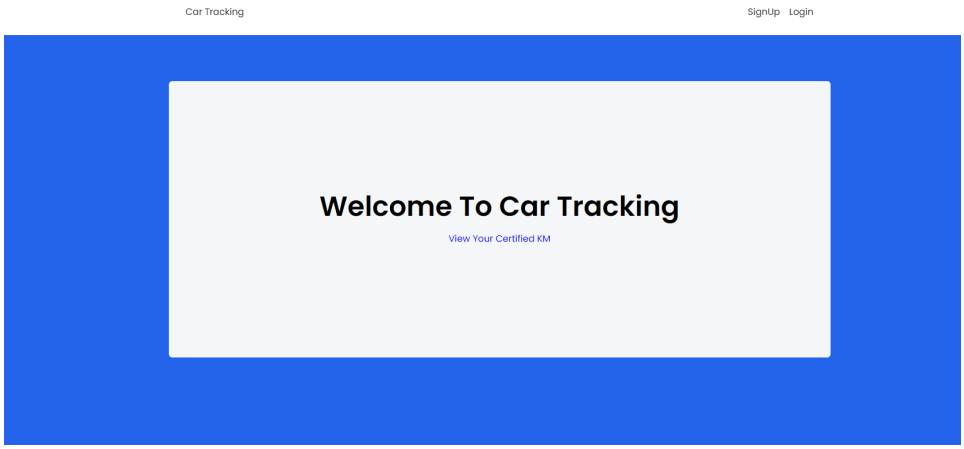


Figura 16: Vista principal de la web.

Una vez el usuario ha introducido sus credenciales pasa a ver (si hay datos) la información de sus trayectos.

Tambien al iniciar sesión en la cabecera aparece una nueva opción que es la de certificar, la cual lleva a una vista en la cual el usuario tiene que volver a introducir sus credenciales y si son correctas se lleva a cabo el certificado en blockchain.

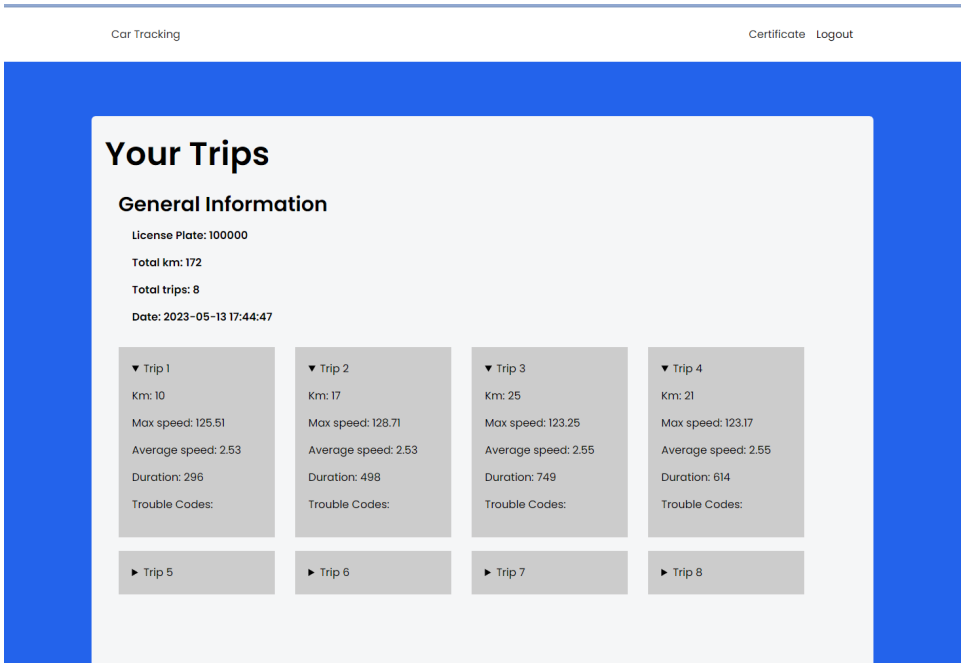


Figura 17: Vista información trayectos.

3.5.2. Tecnologias Utilizadas

Para el desarrollo del front-end web se ha usado Yew, un framework basado en Rust, el motivo de esta elección ha sido porque el backend al estar desarrollado también en Rust se pueden hacer cosas interesantes como compartir structs comunes entre el front y el back y así facilitar la comunicación. Otro motivo ha sido porque era una buena oportunidad de descubrir el funcionamiento de otras tecnologías e intentar implementar algo que no fuera lo estándar.

3.5.3. Lógica del funcionamiento

El frontend sigue la siguiente estructura de ficheros: Un fichero Cargo.toml que es el archivo de configuración principal en el ecosistema de desarrollo de Rust, dónde se define el nombre del proyecto, las dependencias que són necesarias... etc. Después dispone de una carpeta src/ dónde se encuentra ya la estructura de la web, esta carpeta incluye los siguientes directorios:

- api/: Incluye todo lo relacionado con las llamadas a las API para poder

comunicarse con el backend, así como las estructuras comunas entre el front y el back.

- components/: Incluye todos los componentes relacionados con la web, como los botones, los pop-up, los input text... Se define la lógica que usan.
- pages/: Contiene un archivo por cada vista de la web. Dentro de los archivos va el código relacionado con la lógica y la estructura que sigue cada vista.
- app.rs: Archivo que se usa como main.
- router.rs: Archivo que define las rutas que sigue la web.
- store.rs: Archivo que se encarga de centralizar toda la lógica del estado de la apliación.

3.6. Aplicación móvil

La aplicación móvil se encarga de recolectar los datos del dispositivo OBD2 que se conecta al puerto del mismo nombre en el coche. Esta recolección se consigue mediante una conexión bluetooth con dicho dispositivo, por lo tanto esta es sólo compatible con aquellos lectores OBD2 que tengan la posibilidad de conectarse mediante bluetooth.

La aplicación se distribuye en tres vistas, cada una con una funcionalidad distinta pero con interacción entre ellas. Las vistas son: Home, Bluetooth, Settings.

3.6.1. Especificaciones

La aplicación se ha desarrollado en Kotlin [12] que desde el año 2019 es el lenguaje de programación preferido por Google para el desarrollo de aplicaciones Android [14].

La aplicación sigue la estructura de ficheros típica. Existe un fichero MainActivity.kt que contiene el "main"de la aplicación entre otras funciones. Luego cada vista cuenta con sus ficheros de funciones [Vista]Fragment.kt y ficheros xml para especificar la interficie. Otros ficheros se han añadido para incorporar otras funcionalidades.

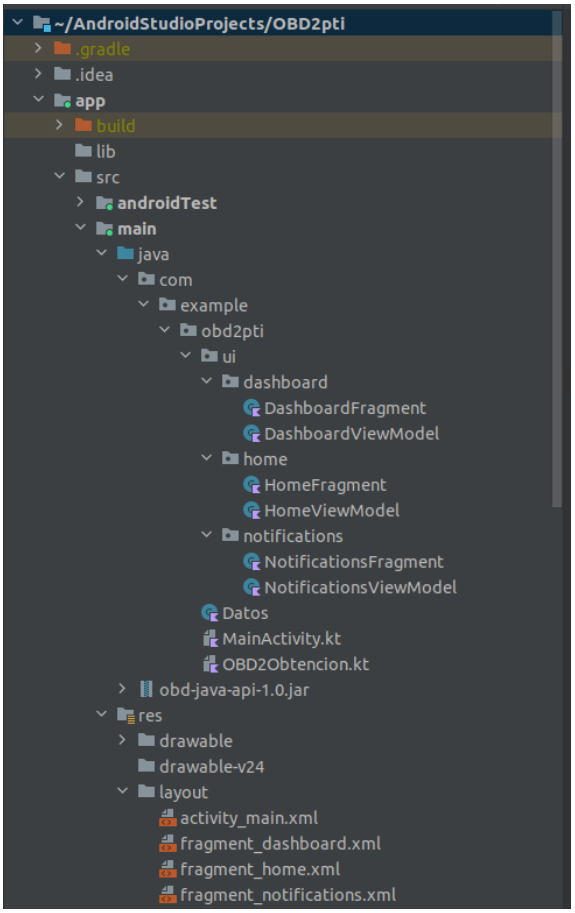


Figura 18: Estructura de ficheros de la App.

3.6.2. Conectividad Bluetooth

Sin duda alguna la parte más crítica de esta aplicación es la posibilidad de poder conectarse al lector OBD2 mediante bluetooth. Para poder implementar la búsqueda de dispositivos y conexión bluetooth he consultado los manuales oficiales disponibles [2]. Aún que estos están incompletos ya que no especifican que en las últimas versiones de Android se debe hacer una comprobación de permisos cada vez que se requiere del uso de alguna de estas funciones.

Primero de todo se debe declarar en el manifiesto de la aplicación (AndroidManifest.xml) la voluntad de utilizar las funcionalidades bluetooth:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN"
        android:usesPermissionFlags="neverForLocation"
        tools:targetApi="s" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
```

Como se puede observar hace falta pedir permisos para toda funcionalidad básica bluetooth que se quiera utilizar, pero la peor parte viene cuando se quiere utilizar alguna de estas. Como he mencionado antes, hace falta comprobar los permisos en todas y cada una de las ocasiones que se utilicen las funcionalidades, en este ejemplo de código para obtener los dispositivos bluetooth disponibles se puede observar como se hace:

```
fun startDiscovery() {
    val isLocationPermissionGranted = hasPermission(Manifest.permission.
    ACCESS_FINE_LOCATION)
    if (bluetoothAdapter?.isEnabled == false) {
        val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M &&
        !isLocationPermissionGranted) {
            requestLocationPermission()
        } else {
            if (ActivityCompat.checkSelfPermission(
                    this,
                    Manifest.permission.BLUETOOTH_SCAN
                ) != PackageManager.PERMISSION_GRANTED
            ) {

                val myNewVal = ActivityCompat.checkSelfPermission(
                    this,
                    Manifest.permission.BLUETOOTH_SCAN
                )
                val myManifest = Manifest.permission.BLUETOOTH_SCAN
                Log.d("console", "before crash")
                Log.d("console", "Manifest.permission.BLUETOOTH_SCAN:
                $myManifest")
                Log.d("console", "My val: $myNewVal")
                Log.d(
```

```
                    "console",
                    "PackageManager.PERMISSION_GRANTED:
                    ${PackageManager.PERMISSION_GRANTED}"
                )
                Log.d(
                    "console",
                    "PackageManager.PERMISSION_GRANTED:
                    ${Manifest.permission.BLUETOOTH_SCAN}"
                )


                return
            }
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT)
        }
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        requestMultiplePermissions.launch(arrayOf(
            Manifest.permission.BLUETOOTH_SCAN,
            Manifest.permission.BLUETOOTH_CONNECT))
    }
    else{
        val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
        requestBluetooth.launch(enableBtIntent)
    }
    val filter = IntentFilter(BluetoothDevice.ACTION_FOUND)
    registerReceiver(receiver, filter)

    //Start discovery
    bluetoothAdapter?.startDiscovery()
    return
}
```

Mucha parte del código de la aplicación se reduce a comprobación de permisos.

Resumidamente, el proceso por el cual debe pasar el usuario para conectarse al dispositivo OBD2 es:

1. Obtener la lista de dispositivos, ya sea de los dispositivos previamente emparejados con el móvil o bien la de dispositivos bluetooth disponibles en los alrededores. Estas se pueden conseguir en la vista Bluetooth.

2. Identificar el dispositivo OBD2 por su nombre en la lista.

3. Conectarse pulsando sobre el nombre en la lista.

Una vez se selecciona el dispositivo la aplicación ejecuta las funciones pertinentes para conectarse a él. Cuando se consigue realizar la conexión se empieza con la recolección de datos.

3.6.3. Recolección de datos

Para la recolección de datos inicialmente había elegido una api totalmente escrita en Kotlin [28], pero esta resultó tener problemas que el desarrollador aún no había resuelto; por lo tanto decidí cambiar a una api escrita en java [20]. Como Kotlin está diseñado para interoperar con Java, no ocurrió ningún problema de incompatibilidad.

Para ejecutar cualquier comando primero hay que declararlo en una variable, o bien se puede declarar al momento de ejecutarse. Cada comando se ejecuta con la función .run(), a la cual se le tiene que pasar un inputstream y un outputstream de un socket conectado al dispositivo OBD2 (este socket puede ser bluetooth o Wi-Fi).

Aquí un ejemplo de cómo declaramos y ejecutamos un comando:

```
var speedcomm:SpeedCommand = SpeedCommand()
 try {
     speedcomm.run(socket.inputStream, socket.outputStream)
} catch (e: Exception) {
    //println("Error al leer velocidad");
}
```

Los comandos pueden lanzar una excepción en caso de que nuestro dispositivo no lo soporte, por lo tanto hay que ejecutarlos utilizando try/catch.

Antes de empezar a enviar comandos al dispositivo hace falta prepararlo para ello, estos comandos especifican entre otras cosas el tiempo de timeout para un comando, el protocolo a usar etc.

```
EchoOffCommand().run(socket.inputStream, socket.outputStream)
LineFeedOffCommand().run(socket.inputStream, socket.outputStream)
TimeoutCommand(500).run(socket.inputStream, socket.outputStream)
SelectProtocolCommand(com.github.pires.obd.enums.ObdProtocols.AUTO).run
(socket.inputStream, socket.outputStream)
```

Los datos que recogemos se guardan en una clase dónde luego se escribirán sus valores en un JSON para su posterior envío al backend.

Todo el proceso de recolección corre en una clase que extiende Thread, por lo tanto este se ejecuta en un thread independiente. Esto no solo permite utilizar la aplicación durante la recolección, ya que el thread principal no se bloquea, si no que también hace posible la utilización de otras aplicaciones mientras esta ocurre en un segundo plano.

Una vez finalizado el trayecto, el usuario debe detener la recolección de los datos usando un botón especifico para ello en la vista Home, en pulsar ese botón la recolección se detendrá y se escribirán todos los datos en un archivo JSON.

3.6.4. Organización y envio de datos

Como bien se ha explicado en el apartado anterior, en ejecutar un comando guardamos su resultado en una variable de la clase Datos. Esta clase contiene una variable por cada dato que recopilamos:

```
var currentTime : String = ""
var speed: Int = 0
var rpm: Int = 0
var throttlePosition : Float = 0.0f
var engineLoad :Float = 0.0f
var coolantTemp : Float = 0.0f
var oilTemp: Float = 0.0f
var fuelLevel: Float = 0.0f
var fuelConsumption : Float = 0.0f
```

Una vez de finaliza la recolección, estos datos se escriben en un fichero JSON utilizando JsonWriter [3]. Los ficheros JSON siguen la siguiente estructura:

```
{
  "matricula": "5038LML",
  "hash": "password1234",
```

```
  "km": 28,
  "max_speed": 125,
  "speed_average": 27.31177829099307,
  "fuel_percentage": 6.2745094,
  "duration": 3800,
  "trouble_codes": "C0300\n",
  "data": [
    {
      "timestamp": "2023-05-07T13:13:47.195158",
      "speed": 0,
      "rpm": 0,
      "throttle": 0,
      "engine_load": 0,
      "engine_coolant_temp": 72,
      "oil_temp": 0,
      "fuel_level": 63.92157,
      "fuel_consumption": 0
    }, ....
    ]
}
```


Cada JSON tiene: la matrícula del vehículo, un hash de la contraseña del usuario, los km recorridos en el viaje, la velocidad máxima alcanzada, la velocidad media del trayecto, el porcentaje del tanque de combustible consumido, la duración del viaje en segundos, los códigos de error que se hayan podido encontrar y un vector de datos. Estos datos consisten en muestras de diferentes datos que se obtienen en un intervalo de 750ms, estos son: un timestamp, la velocidad actual, las revoluciones del motor, la posición del acelerador, la carga del motor, la temperatura del refrigerante, la temperatura del aceite, el nivel de combustible en tanto por ciento y el consumo en L/100km.

Estos datos se utilizan para calcular los datos que se muestran al principio del JSON, como los km recorridos y la velocidad media etc.

Este formato de JSON se ha estructurado de acuerdo a lo que el backend espera leer, por lo tanto cualquier cambio en este también se debe ver reflejado en él o bien la subida de datos no sera posible.

Para enviar los datos el usuario debe pulsar el botón dispuesto en la vista Home. El envío de datos al backend se hace mediante http y usando una

api para Kotlin llamada Fuel [26]. Esta api es muy simple y fácil de utilizar, para este caso de uso en concreto es ideal debido a que solo necesitamos hacer un http post por cada fichero que queremos subir. La documentación de esta api [27] es muy extensa y útil.

En más detalle, la aplicación itera por cada archivo JSON que encuentra en la carpeta donde se almacenan. Por cada archivo que encuentra, realiza un post hacia el backend; si este es satisfactorio, el nombre del archivo se añade a una lista para su posterior eliminación, en caso contrario el archivo no se añade a esta lista y se intentará subir la próxima vez que el usuario ejecute el proceso.

3.6.5. Interfaz

La interfaz se organiza en tres vistas o secciones: Home, Bluetooth y Settings; cada una con una función concreta.

Vista Home La vista Home contiene dos lineas de texto, una indica el estado de la conexión Bluetooth y otra el estado de la recolección de datos. Luego hay dos botones: uno detiene la recolección para grabar los datos en un archivo y el otro activa la subida de dichos archivos al backend.

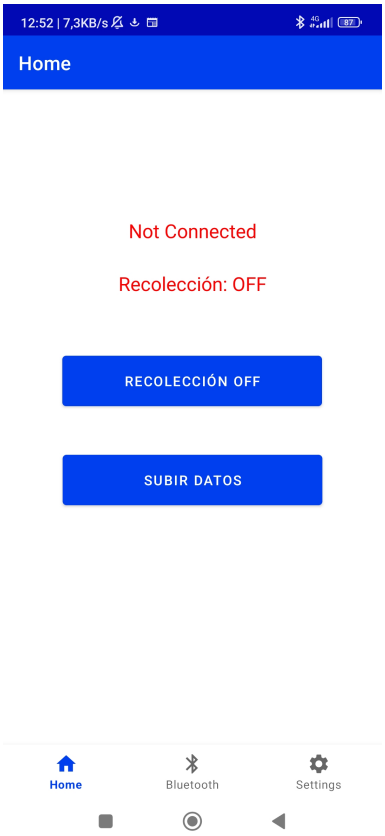


Figura 19: Vista Home

Vista Bluetooth La vista de Bluetooth consiste en dos botones que actualizan una lista de dispositivos sobre ellos. Un botón mostrará en la lista los dispositivos ya emparejados con el teléfono, el otro mostrará los dispositivos bluetooth disponibles en los alrededores que no están emparejados con el teléfono.

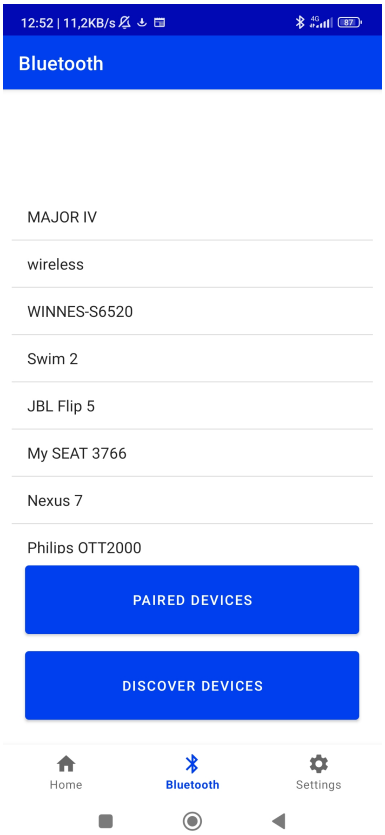


Figura 20: Vista Bluetooth

Vista Settings La vista Settings consta de dos campos de texto para añadir los credenciales que se escriben en los JSONs para que el backend pueda identificar el usuario. Estos campos son matrícula y contraseña. También cuenta con botón para validar e insertar los credenciales especificados y con una checkbox para limitar la subida de datos para que únicamente se lleve a cabo cuando se esté conectado a una red Wi-Fi.

Los iconos de la aplicación también son personalizados y fueron cambiados usando las instrucciones de la documentación especifica de Android [1]. Desde AndroidStudio este es un proceso fácil y sencillo.

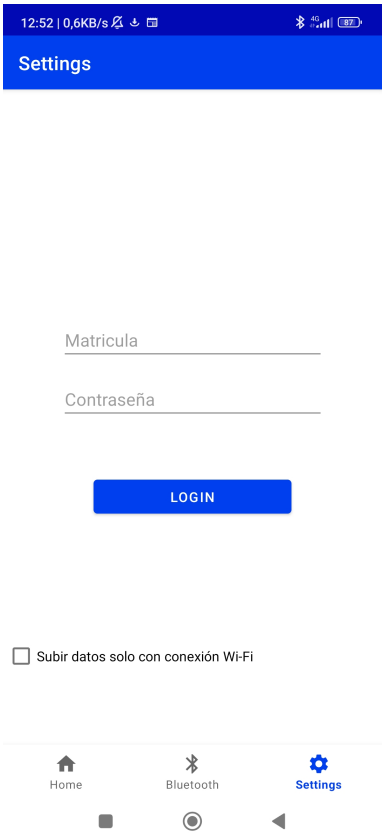


Figura 21: Vista Settings

4. Orquestación y despliegue

Se ha llevado a cabo utilizando *docker-compose* i una *cronjob*.

4.1. Contenedores

Hemos definido varios contenedores, uno para cada servicio exceptuando la aplicación de móvil. También hemos escrito dos ficheros, docker-compose.yml para testear en local y prod.yml para desplegar el sistema en un servidor de producción, concretamente una máquina virtual de la FIB.

- `https://github.com/s4izh/car-tracking-fullstack/blob/main/docker-compose.yml`

- `https://github.com/s4izh/car-tracking-fullstack/blob/main/prod.yml`

En las variables de entorno que definimos como URLs hemos podido usar el nombre de los diferentes contenedores ya que *docker-compose* de actuar como DNS y hacer esta resolución de nombres en tiempo de ejecución.

A continuación veremos fragmentos del archivo *prod.yml*.

4.1.1. MariaDB

```yaml
services:
  mariadb:
    container_name: mariadb
    image: mariadb:10.6.4-focal
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=rust
      - MYSQL_DATABASE=db
      - MYSQL_USER=db
      - MYSQL_PASSWORD=db
    expose:
      - 3306
      - 33060
    volumes:
      - db-data:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "mariadb", "-u", "root", "-prust"]
      interval: 5s
      timeout: 10s
      retries: 3
```

Figura 22: Contenedor de MariaDB

Este primer contenedor esta basado en la imagen oficial de MariaDB y no tiene ninguna modificación.

Se definen las variables de entorno correspondientes a las credenciales en la base de datos y se crea un volumen *db-data* con los contenidos de la base de datos para tener persistencia.

Se exponen también los puertos necesarios para poder comunicarse con ella.

Destacar también el atributo *healthcheck* que define una función para poder testear si la base de datos se levanta correctamente. El contenedor utilizará

esta funcionalidad para esperar a iniciarse a que la base de datos este sana.

4.1.2. Blockchain

```yaml
blockchain:
  container_name: blockchain
  stdin_open: true
  tty: true
  build:
    context: blockchain
  working_dir: /app/blockchain
  volumes:
    - ".:/app"
    - "ganache-data:/app/ganache-data"
  ports:
    - 8084:8545
    - 8083:7545
    - 8082:3000
  expose:
    - 8545
    - 7545
    - 3000
```

Figura 23: Contenedor de la blockchain

Destacar de este contendor que expone los puertos necesarios para servir el frontend y también para poder hacer RPCs directamente a la blockchain. También el volumen *ganache-data* para conseguir persistencia.

```dockerfile
FROM ubuntu:latest

RUN apt update
RUN apt install curl -y
RUN curl -sL https://deb.nodesource.com/setup_14.x | bash -
RUN apt update
RUN apt install nodejs -y
RUN apt install build-essential -y
RUN apt install tmux -y

RUN npm install -g ganache-cli
RUN npm install -g truffle@5.4.23
```

Figura 24: Dockerfile de la blockchain

Tuvimos problemas poniendo en contenedores la blockchain, con lo que al

final usamos una imagen de Ubuntu como base y instalamos todo lo necesario encima.

4.1.3. Cliente de la blockchain

```
blockchain-client:
  container_name: blockchain-client
  stdin_open: true
  tty: true
  build:
    context: blockchain-client
  working_dir: /app/blockchain-client
  volumes:
    - ".:/app"
  environment:
    - BLOCKCHAIN_URL=http://blockchain:7545
  ports:
    - 8085:5000
  expose:
    - 5000
  command: python app.py
```

Figura 25: Contenedor del cliente de la blockchain

Este contenedor expone el puerto 5000, el que necesitará el backend para poder comunicarse con él por HTTP para poder hacer las certificaciones en la blockchain.

```
FROM python:3.9

WORKDIR /app/blockchain

RUN pip install flask
RUN pip install web3
```

Figura 26: Dockerfile del cliente de la blockchain

4.1.4. Backend

```yaml
backend:
  container_name: backend
  stdin_open: true
  tty: true
  environment:
    - DATABASE_URL=mysql://db:db@mariadb/db
    - BACKEND_PORT=8080
    - BLOCKCHAIN_CLIENT_URL=http://blockchain-client:5000
    - RUST_LOG=debug
    - CARGO_HOME=/app/.cargo
    - CARGO_TARGET_DIR=/app/target
  build:
    context: backend
  working_dir: /app
  volumes:
    - ".:/app"
    - "cargo:/app/.cargo"
    - "target:/app/target"
  ports:
    - 8080:8080
  expose:
    - 8080
  depends_on:
    mariadb:
      condition: service_healthy
  command: make backend-deploy
```

Figura 27: Contenedor del backend

Se expone el puerto 8080 que será el encargado de servir la API del backend. Destacar también que este contenedor no se lanzará si la base de datos no está sana, condición de *(service healthy)*.

```dockerfile
FROM rust:1.67.1

WORKDIR /app

RUN cargo install diesel_cli --no-default-features --features mysql
RUN cargo install cargo-make
```

Figura 28: Dockerfile del backend

4.1.5. Frontend

```yaml
frontend:
  container_name: frontend
  stdin_open: true
  tty: true
  environment:
    - BACKEND_URL=http://nattech.fib.upc.edu:40400
    - RUST_LOG=debug
    - CARGO_HOME=/app/.cargo
    - CARGO_TARGET_DIR=/app/target
  build:
    context: frontend
  working_dir: /app
  volumes:
    - ".:/app"
    - "cargo:/app/.cargo"
    - "target:/app/target"
    - "rustup:/app/.rustup"
  ports:
    - 8081:8081
  command:
    make frontend-run
```

Figura 29: Contenedor del frontend

El frontend se servirá en el puerto 8081, este contenedor, como el backend, está creado a partir de una imagen de rust y le hemos instalado (entre otras cosas) las dependencias necesarias para poder compilar a *wasm*.

```dockerfile
FROM rust:1.67.1

# Instalar Node.js y npm
RUN curl -sL https://deb.nodesource.com/setup_16.x | bash -
RUN apt-get update && apt-get install -y nodejs

# Instalar npx
RUN npm install -g npm@9.6.5
RUN npm install -g postcss-cli@8.3.1 autoprefixer@10.4.0 tailwindcss@3.1.4
#RUN npm install -g npx


WORKDIR /app

RUN rustup toolchain install stable
RUN rustup target add wasm32-unknown-unknown
RUN cargo install trunk
RUN cargo install wasm-bindgen-cli
```

Figura 30: Dockerfile del frontend

4.1.6. Volúmenes

Estos son los diferentes volúmenes de docker creados con los contenedores anteriores:

```
volumes:
  db-data:
  cargo:
  target:
  rustup:
  ganache-data:
```

Figura 31: Volúmenes de docker

- *db-data*: guarda los datos de la base de datos.
- *cargo*: guarda la cache de cargo, el gestor de paquetes de rust.
- *target*: guarda los binarios compilados de rust para que no se tengan que recompilar otra vez al volver a ejecutar la imagen.
- *rustup*: cache para rustup, el gestor de versiones de rust.
- *ganache-data*: volumen para guardar los datos de la blockchain.

4.2. Mapeo de puertos final

Así es como quedarían los puertos mapeados en (localhost) después de ejecutar:

```
docker-compose -f prod.yml up -d
```

- 8080: estaría el backend escuchando peticiones.
- 8081: está trunk escuchando peticiones GET para servir el frontend.
- 8082: está truffle sirviendo el frontend de la blockchain.
- 8083:8084 los diferentes puertos de ganache necesarios para hacer RPCs.
- 8085: aquí escucha las peticiones HTTP el cliente de la blockchain, está mapeado aquí para poder hacer pruebas pero recomendamos quitarlo.

En el caso de nuestra máquina virtual, por NAT, el puerto 8080 se mapeaba

al 40400 de la dirección `nattech.fib.upc.edu`, y así sucesivamente, el 8081 al 40401...

Con el atributo *expose* se marcan que puertos se quieren exponer a la docker network que forman los diferentes contenedores, con la configuración mostrada conseguiríamos un setup final así:

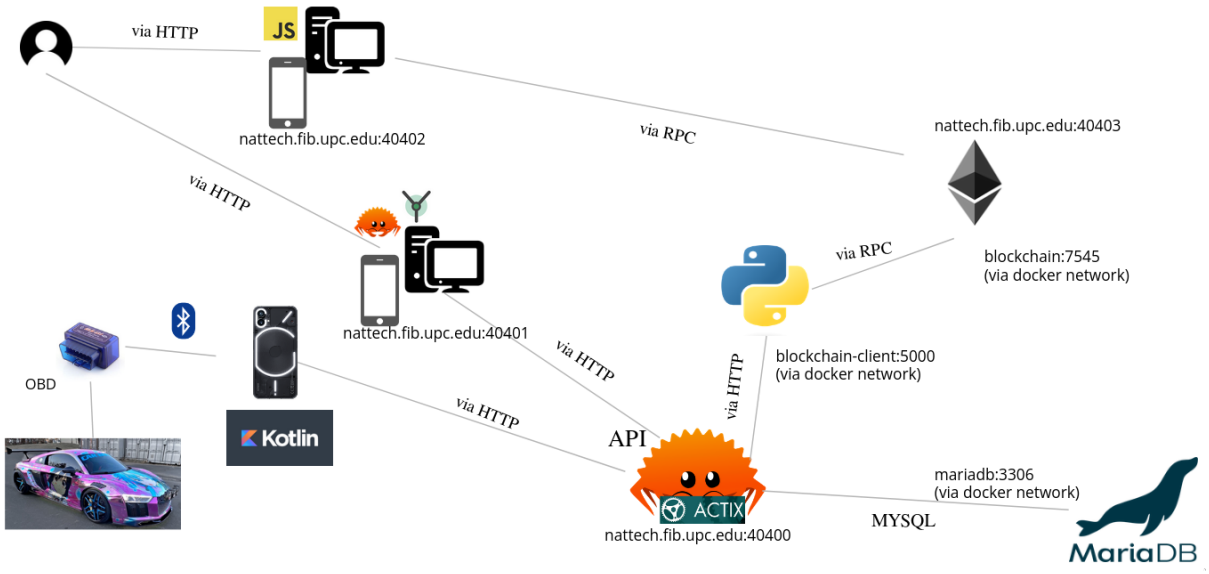


Figura 32: Esquema final

Como vemos en el esquema, tanto las conexiones del backend con la base de datos y el cliente de blockchain como la conexión del cliente a la blockchain se hacen a través de la docker network creada.

4.3. Despliegue

Al principio el despliegue era bastante manual, se llevaba acabo de la siguiente manera:

4.3.1. Primer despliegue

- Entrar a la VPN de virtech
- Conectarse por *ssh* a la máquina virtual

- Hacer un *git clone* de nuestro repositorio
- Construir las imagenes de docker con *docker-compose -f prod.yml build*
- Ejecutarlas con *docker-compose -f prod.yml up*
- Había que ejecutar manualmente ganache con *make ganache* y lo mismo para truffle con *make truffle*

4.3.2. Posteriores Despligues

Para los siguientes despliegues los pasos eran los mismos pero en vez de hacer *git clone* teníamos que ejecutar *git pull* para conseguir los nuevos cambios.

En el README del proyecto se recoge detalladamente más información sobre compilar cada servicio, entrar en cada contenedor, etc.

`https://github.com/s4izh/car-tracking-fullstack/tree/main#readme`

4.4. Mejoras

4.4.1. tmux

tmux es un multiplexador de terminales que permite entre otras cosas tener sesiones persistentes aunque nos desconectemos de la máquina. De esta manera después de volver a conectarnos a la máquina podíamos ejecutar *tmux attach* para volver a nuestra sesión anterior.

En una misma sesión de *tmux* también se pueden tener varias ventanas, aprovechamos esta oportunidad para ejecutar en una *docker-compose -f prod.yml up* en una ventana y en otra ventana diferente, dividiéndola en dos paneles *make ganache* y *make truffle*.

De esta manera podíamos observar los logs en tiempo real de los diferentes servicios que se estaban ejecutando, pudiendo también pararlos, cambiarlos, recompilarlos y volverlos a ejecutar cuando quisiéramos.

4.4.2. Cronjob

Para automatizar el proceso hemos probado a usar una *cronjob* que se ejecute cada 10 minutos.

Esta *cronjob* lo que hará es comprobar si hay commits nuevos en la rama

main de nuestro repositorio.

Con un CURL a la API de github se puede obtener información de los commits de cierta rama, es posible aprovecharse de esto para obtener el hash del commit más reciente y guardarlo en un archivo. La siguiente vez que se ejecute la script, se comprobará si el hash del commit más reciente es igual o no al que se ha guardado en un archivo anteriormente, si es diferente se pullearan los cambios de la rama main de nuestro repositorio y se buildearán y desplegarán nuevamente los diferentes contenedores.

```
GITHUB_USER="s4izh"
GITHUB_REPO="car-tracking-fullstack"
LAST_COMMIT_FILE="/tmp/last-commit"

if [ -f "$LAST_COMMIT_FILE" ]; then
  LAST_COMMIT=$(cat "$LAST_COMMIT_FILE")
else
  LAST_COMMIT=""
fi

API_URL="https://api.github.com/repos/$GITHUB_USER/$GITHUB_REPO/commits/main"
NEW_COMMIT=$(curl -s "$API_URL" | grep -m 1 -o '"sha": "[^"]*' | cut -d'"' -f4)

if [ "$LAST_COMMIT" == "$NEW_COMMIT" ]; then
  echo "No new commit found"
else
  echo "New commit found: $NEW_COMMIT, redeploying..." >> /tmp/deploy.log
  echo "$NEW_COMMIT" > "$LAST_COMMIT_FILE"
  cd /app
  git restore .
  git pull
  docker-compose down
  docker-compose -f prod.yml build
  docker-compose -f prod.yml up -d
  sleep 10
  docker exec -ti blockchain make ganache &
  sleep 10
  docker exec -ti blockchain make truffle &
fi
```

Figura 33: Cronjob

5. Pruebas

5.1. Aplicación

Para realizar pruebas en la aplicación ha sido fundamental el uso de un simulador de OBD, ya que probar la aplicación con un dispositivo real requeriría de estar en un coche constantemente lo cual no resulta práctico.

Por suerte existe un simulador de un dispostivo OBD2 llamada OBDSim[4]. Este simulador está un poco limitado en lo que a datos se refiere, ya que solo ofrece unas métricas limitadas pero son más que suficientes para probar la aplicación. Consta de distintos módulos, por ejemplo uno de una interfície donde se puede dar valor a distintas métricas como la velocidad o las revoluciones del motor, otro que también hemos utilizado lanza valores aleatorios por cada lectura realizada.

Para conectarse a este simulador puede utilizarse tanto Wi-Fi como bluetooth, en nuestro caso para utilizar este último hace falta el software com0com [11], que recomienda el mismo creador del simulador para poder conectar el simulador a un puerto COM bluetooth.

Para poder utilizarse antes se debe abrir un puerto COM entrante en el módulo bluetooth del ordenador. Recomendamos utilizar este simulador en Windows, ya que el proceso de preparación es mucho más sencillo.

Una vez hayamos instalado todo el software necesario y abierto el puerto COM podemos ejecutar la aplicación por el siguiente comando en consola:

```
obdsim.exe -g [Módulo] -w [Puerto COM, por ejemplo: COM5]
```

Una vez ejecutado la consola pasa a mostrar los logs de la aplicación, además, cada 10 segundos nos muestra cuantas solicitudes de datos se han realizado y cuantos se han servido.

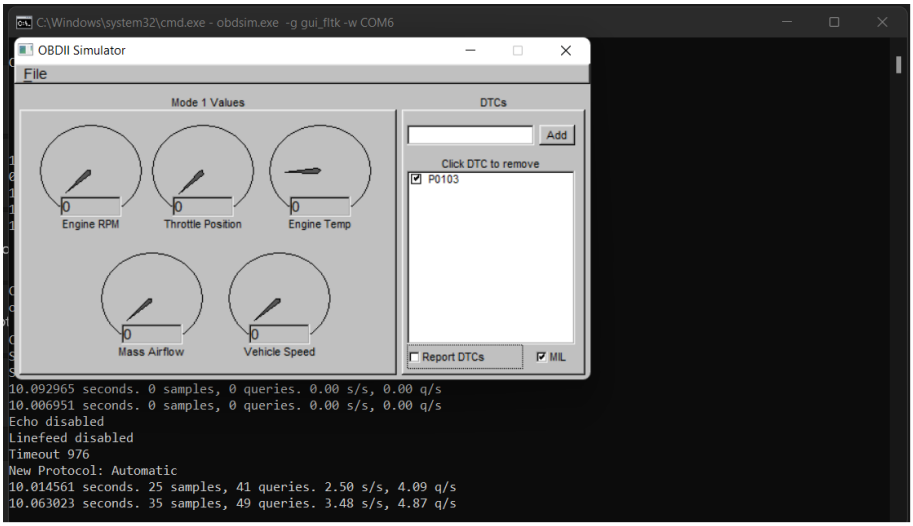


Figura 34: OBDSim en ejecución

5.2. Backend

Para ayudar con el desarollo del backend hay disponibles una serie de scripts en el directorio */backend/tests* diseñadas para llenar la base de datos con diferentes datos

También hay diferentes scripts con *curl* para testear los diferentes endpoints.

`https://github.com/s4izh/car-tracking-fullstack/tree/main/backend/test`

En el anterior enlace se encuentran dichas scripts y la documentación para testear/usar cada endpoint.

6. Complicaciones

Hemos tenido varias:

- La máquina de la FIB se borra de golpe sin avisar (suponemos que inactividad).
- La máquina de la FIB se cierra a partir de las 12, con lo cual no se puede trabajar por la noche.

- Las librerías de gráficos para hacer frontend en Yew no funcionaban con la versión que hemos escogido nosotros.

7. Conclusión

Ha sido una buena experiencia hacer este proyecto, pese a que ninguno de los 5 sabía nada de las tecnologías que decidimos utilizar hemos conseguido sacar el proyecto adelante con una solución funcional.

7.1. Aspectos a mejorar

- Usar un sistema de autenticación mejor (por ejemplo con JSON Web Tokens)
- Un frontend más vistoso que consiguiera sacar a luz todos los datos que tenemos guardados en la BD, por ejemplo con algún tipo de gráfico o tabla.
- Un sistema de CI/CD más seguro que la *cronjob* que hay ahora.

8. Repositorios

Cada uno tiene un mirror del repositorio, probad estos enlaces si los otros no funcionan:

- `https://github.com/s4izh/car-tracking-fullstack`
- `https://github.com/FerranFuentes/car-tracking-fullstack`
- `https://github.com/R-kill-9/car-tracking-fullstack`
- `https://github.com/marcelituu/car-tracking-fullstack`
- `https://github.com/adriroyo/car-tracking-fullstack`

Referencias

[1] Android. *Cómo cambiar el ícono de la app.* https://developer.android.com/codelabs/basic-android-kotlin-training-change-app-icon?hl=es-419. 2021.

[2] Android. *Introducción general a Bluetooth.* https://developer.android.com/guide/topics/connectivity/bluetooth?hl=es-419. 2021.

[3] Android. *JsonWriter.* https://developer.android.com/reference/android/util/JsonWriter. 2023.

[4] Gary 'ChunkyKs' Briggs. *OBDSim a simple OBD/ELM327 simulator.* https://icculus.org/obdgpslogger/obdsim.html. 2011.

[5] Abhishek Chanda. *Build an API in Rust with JWT Authentication.* https://auth0.com/blog/build-an-api-in-rust-with-jwt-authentication-using-actix-web/.

[6] Bocksdin Codes. *Diesel ORM - Rust + Actix Web + PostgreSQL (2022).* https://www.youtube.com/watch?v=EkI0AmHmZT8.

[7] codevoWeb. *Build a Frontend Web App in Rust using the Yew.rs Framework.* https://codevoweb.com/build-frontend-web-app-in-rust-using-yew-framework/.

[8] Diesel. *Configuring diesel-cli.* https://diesel.rs/guides/configuring-diesel-cli.html. 2023.

[9] Diesel. *Diesel is a Safe, Extensible ORM and Query Builder for Rust.* https://diesel.rs/. 2023.

[10] Ethereum. *Set Up Web3.js to Use Ethereum in JavaScript.* https://ethereum.org/fr/developers/tutorials/set-up-web3js-to-use-ethereum-in-javascript/.

[11] Vyacheslav Frolov. *The Null-modem emulator an open source kernel-mode virtual serial port driver for Windows, available freely under GPL license.* https://com0com.sourceforge.net/. 2011.

[12] JetBrains. *Kotlin Programming Language.* https://kotlinlang.org/. 2011.

[13] Maria Barcelo; Ordinas Jose; *Protocolos y Aplicaciones internet.* Barcelona: UOC, 2008.

[14] Fredric Lardinois. *Kotlin is now Google's preferred language for Android app development.* https://tcrn.ch/2vJMMO2. 2019.

[15] MariaDB. *Comandos SQL Básicos - MariaDB Knowledge Base*. https://mariadb.com/kb/es/basic-sql-statements/.

[16] Code To The Moon. *Build A Rust Backend (Really FAST Web Services with Actix Web)*. https://youtu.be/L8tWKqSMKUI.

[17] Miguel Guerrero - Eduard Gonzalez - Javier de Muniategui - Joan Saltó. *Car Locator*. https://mwiki.fib.upc.edu/pti/index.php/Categora:Carlocator.

[18] Javier Canoes y otros. *Clear Mileage*. https://mwiki.fib.upc.edu/pti/index.php/Categora:ClearMileage.

[19] Davide del Pa. *YEW Tutorial: 01 Introduction*. https://dev.to/davidedelpapa/yew-tutorial-01-introduction-13ce.

[20] Pires. *OBD-II Java API*. https://github.com/pires/obd-java-api. 2017.

[21] QuickNode. *How to Connect to the Ethereum Network with Web3.js*. https://www.quicknode.com/guides/ethereum-development/getting-started/connecting-to-blockchains/how-to-connect-to-ethereum-network-with-web3js/.

[22] Ambar Quintana. *Crear una base de datos en MySQL / MariaDB*. https://styde.net/crear-una-base-de-datos-en-mysql-mariadb/.

[23] Salanfe. *Ethereum: Create Raw JSON-RPC Requests with Python for Deploying and Transacting with a Smart Contract*. https://medium.com/hackernoon/ethereum-create-raw-json-rpc-requests-with-python-for-deploying-and-transacting-with-a-smart-7ceafd6790d9.

[24] Stackoverflow. *Stackoverflow.* https://es.stackoverflow.com. 2008.

[25] Truffle Suite. *Pet Shop Tutorial*. https://trufflesuite.com/guides/pet-shop/.

[26] Kittinun Vantasin y Jonathan 'iNoles'. *Fuel, the easiest HTTP networking library for Kotlin backed by Kotlinx Coroutines.* https://github.com/pires/obd-java-api. 2015.

[27] Kittinun Vantasin y Jonathan ïNoles". *Fuel documentation.* https://fuel.gitbook.io/documentation/core/fuel. 2015.

[28] Elton Viana. *A lightweight and developer-driven API to query and parse OBD commands, written in pure Kotlin*. https://github.com/eltonvs/kotlin-obd-api. 2022.

[29] Actix Web. *Actix Web is a powerful, pragmatic, and extremely fast web framework for Rust*. `https://actix.rs/`. 2023.

[30] Wikipedia. *OBD2*. `https://es.wikipedia.org/wiki/OBD`.

[31] Yew. *Tutorial Yew*. `https://yew.rs/docs/tutorial`.

Índice de figuras