RETO BACKEND

CREACIÓN DE UNA API PARA SEGUIMIENTOS DE PEDIDOS

Juan Manuel Porrero Almansa

Fecha: 6 de mayo de 2023

Contenido

1. Modelo de datos

Se ha usado MongoDB como base de datos para almacenar la información y el siguiente modelo de datos:

Colección de Usuarios

_id: Identificador único del usuario, su correo (ObjectId, clave primaria)

nombre: Nombre completo del usuario (cadena de caracteres)

contraseña: Contraseña del usuario (cadena de caracteres)

Colección de Vehículos

_id: Identificador único del vehículo, su matrícula (ObjectId, clave primaria)

modelo: Modelo del vehículo (cadena de caracteres)

capacidad: Capacidad de carga del vehículo en kg (entero)

Colección de Pedidos

_id: Identificador único del pedido (ObjectId, clave primaria)

cliente: Identificador del usuario que realizó el pedido (ObjectId, clave foránea hacia la colección de Usuarios)

fecha_creacion: Fecha y hora en que se creó el pedido (fecha y hora)

fecha_entrega: Fecha y hora en que se debe entregar el pedido (fecha y hora)

direccion_origen: Dirección de origen del pedido (cadena de caracteres)

direccion_destino: Dirección de destino del pedido (cadena de caracteres)

peso: Peso del paquete en kg (entero)

Colección de Seguimiento de Pedidos

_id: Identificador único de seguimiento (ObjectId, clave primaria)

pedido: representa al objeto pedido que está asociado a este seguimiento.

vehiculo: representa al objeto vehículo que está asociado al seguimiento.

fecha_hora: Fecha y hora en que se actualizó la información de seguimiento (fecha y hora)

ubicacion: Ubicación actual del vehículo, un objeto con campos "type" (string) y "coordinates" (array de dos números decimales)

A continuación se adjunta un ejemplo de documento de la colección “tracking” que es la que “resume” todo:


```
_id: ObjectId('6453df8325b81e42b4519ea0')
trackingNumber: "Prueba"
▾ order: Object
    orderNumber: "CT789"
  ▾ client: Object
      _id: "juanmaporreroalmansa@gmail.com"
      name: "Juanma Porrero"
      password: "anVhbm1hMTI="
    creationDate: 2023-05-03T20:28:28.382+00:00
    deliveryDate: 2023-05-03T20:28:28.382+00:00
    originAddress: "Ciudad Real"
    deliveryAddress: "Madrid"
    weight: 897
▾ vehicle: Object
    _id: "8967LMR"
    model: "BMW X6"
    capacity: 898
  lastUpdate: 2023-05-04T16:37:45.341+00:00
▾ featureCollection: Object
    type: "FeatureCollection"
  ▾ features: Array
    ▾ 0: Object
        type: "Feature"
      ▸ properties: Object
      ▾ geometry: Object
          type: "LineString"
        ▾ coordinates: Array
          ▾ 0: Array
              0: 78.78667
              1: 89.8977
          ▾ 1: Array
              0: 89.6777
              1: -67.9777
          ▾ 2: Array
              0: 6.5563
              1: -32.7645
_class: "retoBackendOrenes.RetoBackend.orderTracking.domain.OrderTracking"
```


Este modelo de datos representa una aplicación de entregas o logística que permite a los usuarios realizar pedidos y realizar su seguimiento. La aplicación cuenta con cuatro colecciones: Usuarios, Vehículos, Pedidos y Seguimiento de Pedidos.

La Colección de **Usuarios** contiene información sobre los usuarios registrados en la aplicación, donde el campo "_id" es un identificador único generado por la aplicación, que utiliza el correo electrónico del usuario como clave primaria. El campo "nombre" almacena el nombre completo del usuario y el campo "contraseña" almacena la contraseña del usuario.

La Colección de **Vehículos** contiene información sobre los vehículos utilizados para realizar las entregas, donde el campo "_id" es un identificador único generado por la aplicación, que

utiliza la matrícula del vehículo como clave primaria. El campo "modelo" almacena el modelo del vehículo y el campo "capacidad" almacena la capacidad de carga del vehículo en kilogramos.

La Colección de **Pedidos** contiene información sobre los pedidos realizados por los usuarios, donde el campo "_id" es un identificador único generado por la aplicación. El campo "cliente" representa el usuario que realizó el pedido. Los campos "fecha_creacion" y "fecha_entrega" almacenan las fechas y horas en que se creó el pedido y cuando se debe entregar el pedido. Los campos "direccion_origen" y "direccion_destino" almacenan las direcciones de origen y destino del pedido. El campo "peso" almacena el peso del paquete en kilogramos.

La Colección de **Seguimiento de Pedidos** contiene información sobre el seguimiento de los pedidos realizados, donde el campo "_id" es un identificador único generado por la aplicación. El campo "pedido" representa el pedido que se está siguiendo. El campo "vehículo" representa el vehículo asociado al seguimiento. El campo "fecha_hora" almacena la fecha y hora en que se actualizó la información de seguimiento. El campo "ubicacion" almacena la ubicación actual del vehículo, que es un objeto con campos "type" (tipo de ubicación) y "coordinates" (coordenadas de ubicación).

En general, este modelo de datos permite a la aplicación gestionar y hacer seguimiento de pedidos y vehículos de manera efectiva, brindando a los usuarios una experiencia de entrega más transparente y confiable.

Las coordenadas se han almacenado de esa forma por un motivo que se explicará más adelante.

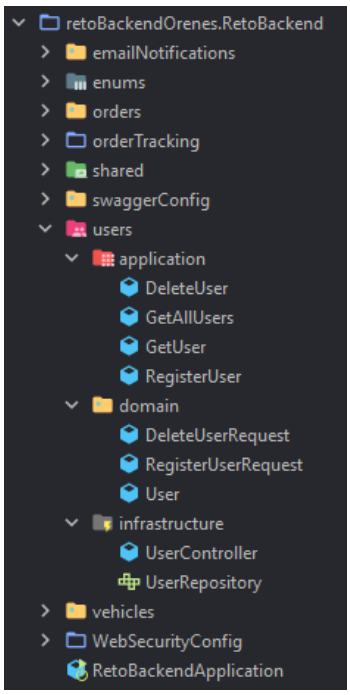
2. API desarrollada

El lenguaje de programación que se ha escogido es JAVA (versión 17) y el framework Spring Boot (versión 3).

2.1. Patrón de diseño empleado: Arquitectura hexagonal

La arquitectura hexagonal (también conocida como "puertos y adaptadores") es un patrón de arquitectura de software que se centra en separar la lógica de negocio de la implementación técnica. La idea principal es que la lógica de negocio se encuentra en el centro de la arquitectura y está protegida de los detalles técnicos de la implementación, que se encuentran en los bordes de la arquitectura.

En resumen, la arquitectura hexagonal divide la aplicación en componentes distintos, con responsabilidades bien definidas. Esto permite que la lógica de negocio esté protegida y sea independiente de los detalles técnicos de la implementación, lo que facilita el mantenimiento y la evolución de la aplicación a lo largo del tiempo.



Como se puede observar en la imagen, se ha decido separa por colecciones los componentes de la API.

En cada uno de ellos, en este caso “users” como se puede ver, tendremos 3 directorios que serán application, domain e infrastructure:

- En application, estarán las funciones externalizadas que funcionarán cuando se les llame desde el endpoint, de esta forma si se quiere cambiar o actualizar algo, únicamente tendremos que dirigirnos a esas clases.

```java
@Component
public class GetUser  {

    @Autowired
    UserRepository userRepository;

    public ResponseWrapper<User> getUser (String username){
        ResponseWrapper<User> wrapper = new ResponseWrapper<>();

        Optional<User> user = userRepository.findByUsername(username);

        if(user.isPresent()){
            wrapper.setResponse(user.get());
            wrapper.setCode(ResponseCodes.SUCCESS.getResponseCode());
            wrapper.setMessage(new ResponseMessage(ResponseCodes.SUCCESS.getDescription()));

        }else{
            wrapper.setResponse(null);
            wrapper.setCode(ResponseCodes.GENERIC_ERROR.getResponseCode());
            wrapper.setMessage(new ResponseMessage("No existe ningún usuario con ese email o usuario"));
        }

        return wrapper;
    }
}
```

- En domain, se define la representación del objeto principal, así como la forma que tendrán las peticiones que nos lleguen del cliente para poder pasar de JSON a Objeto JAVA mediante su mapeado.

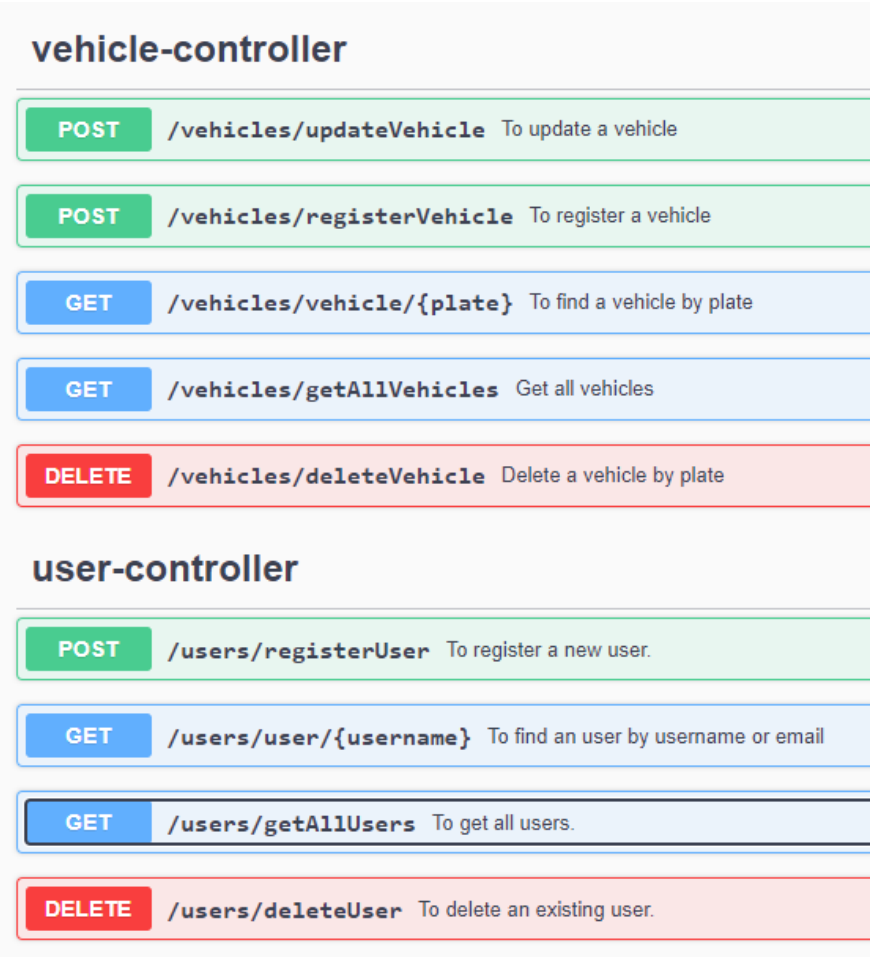
```java
@Data
public class RegisterUserRequest {

    private String username;
    private String name;
    private String password;
```

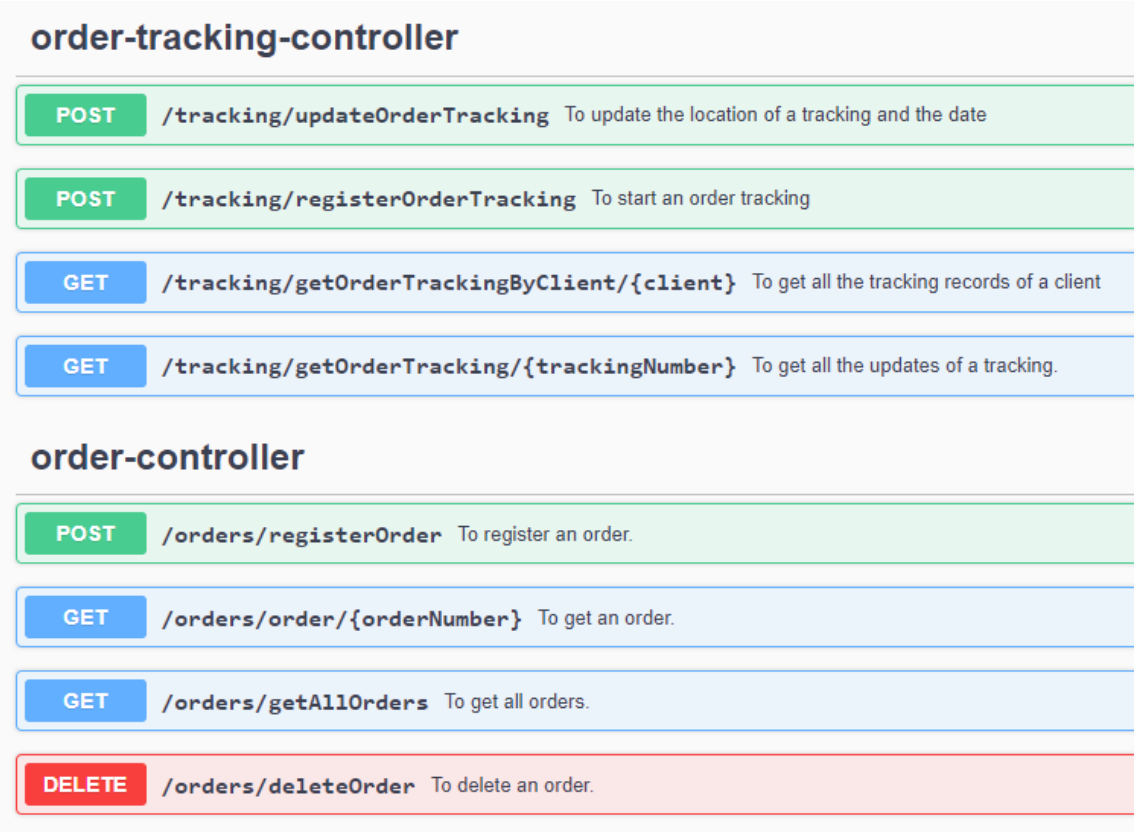
- En infrastructure, tenemos la interfaz JAVA que representa el repositorio de ese objeto en la Mongo, donde se encuentran las querys para comunicarnos con ella. También se ecuentra el controller, que gestiona los endspoints.

```java
@GetMapping("/getAllUsers")
public ResponseWrapper<List<User>> allUsers () { return getAllUsers.getAllUsers(); }
```

2.2. Endpoints de la API

La API se ha documentado mediante **Swagger**, ya que resulta muy útil a la hora de probar y ver qué hace cada endpoint de la aplicación. De esta forma, se adjuntarán capturas para que se vea la función de cada uno.





2.3. Funcionalidades de la API

A modo de resumen, se hará una descripción de cómo funciona la API.

1. Lo primero que tendremos que hacer será crear un cliente y un vehículo con sus respectivos endpoints y parámetros necesarios (también se cuenta con validaciones para que los datos sean los correctos).

2. Después, podremos crear un pedido, que, necesariamente tiene que hacer un cliente que exista, por lo tanto, si metemos un cliente que no está registrado, nos dará un error.

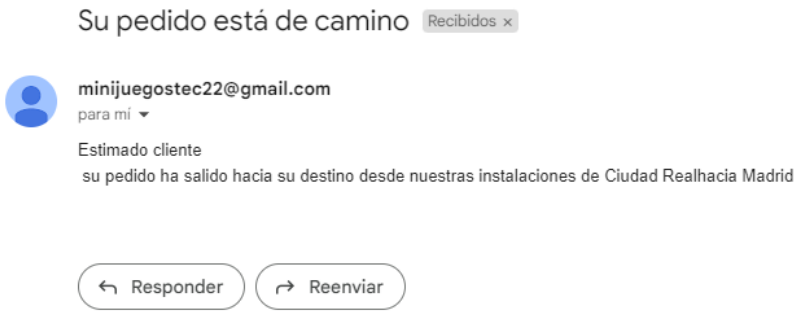
3. Una vez creado el pedido, podremos crear un seguimiento, introduciendo entre sus parámetros el vehículo asociado a ese seguimiento, que, también tiene que estar registrado, y además un pedido existente.

4. El usuario puede ir actualizando las coordenadas del seguimiento y la fecha a la que se actualizó.

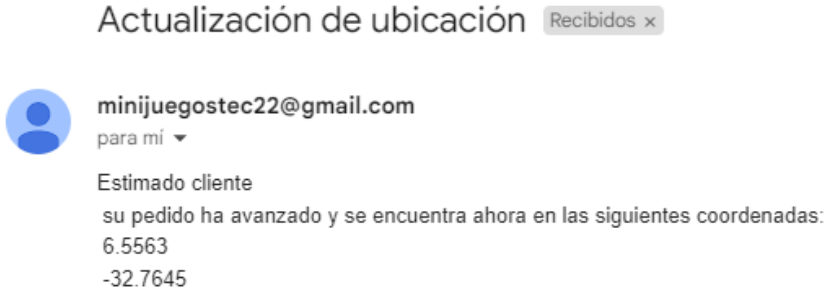
Después se puede hacer uso de los diferentes endpoints para modificar lo que se pueda, borrar o recuperar el dato que se desee.

2.4. Aspectos extra

Cuando se empieza un seguimiento se notificará por correo electrónico al cliente, con un mensaje tal que así.



Y cuando se actualice el estado (las coordenadas) se notificará también al cliente.



Para la **seguridad de la API** se ha decidido usar el módulo Spring Security que sirve para proteger las URLs y para autenticación de usuarios.

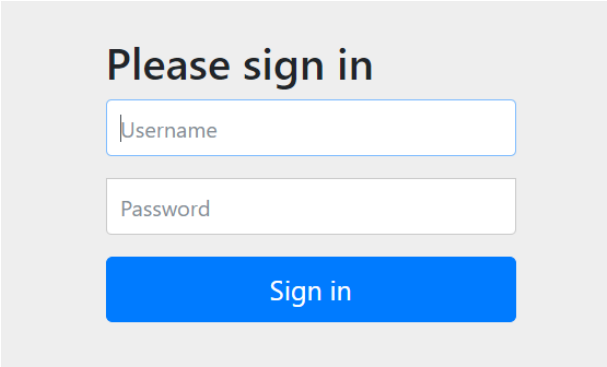
Todas las rutas quedan protegidas por lo tanto y para conseguir acceso se debe introducir tanto en el usuario y contraseña “admin”.

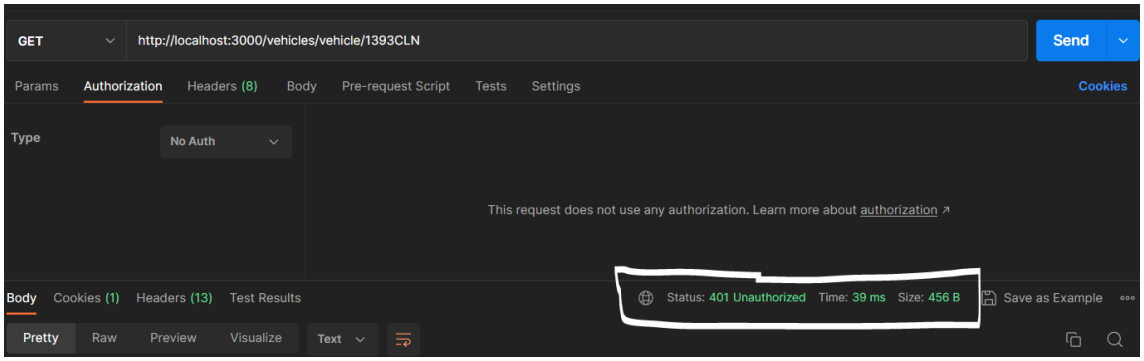
Como nota he de decir que esto no pasa cuando la aplicación se ejecuta desde un Docker, ya que, al tratarse de una autenticación básica de usuario y contraseña, y no por Token de acceso, Spring Security no interpreta bien de dónde proviene la autenticación y no la pedirá.

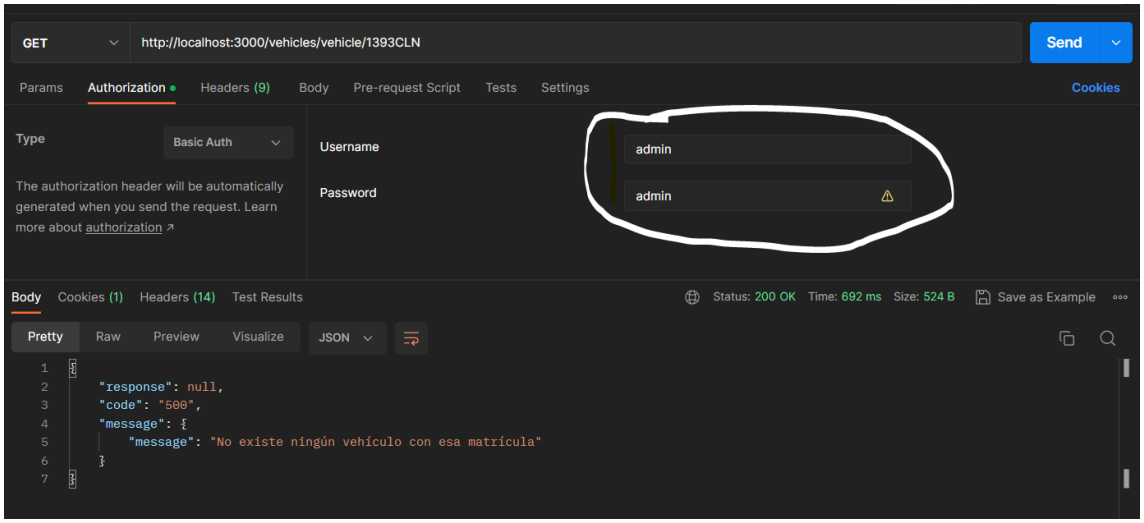
Si se quiere que salte la autenticación se debe de ejecutar en local desde el propio terminal.

POR DEFECTO ESTÁ DESACTIVADA YA QUE A LA HORA DE HACER PETICIONES POST PEDÍA UN TOKEN CSRF QUE COMPLICABA LA CORRECCIÓN. Si se quiere activar, comentar lo siguiente en WebSecurityConfig:

```java
@Bean
@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf().disable();
    return http.build();
}
```







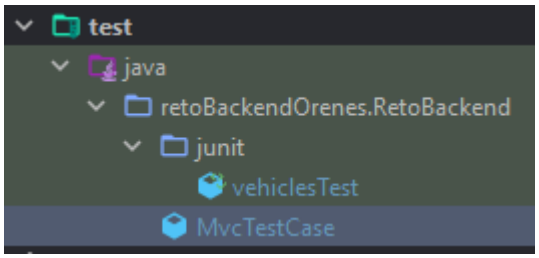
En cuanto al tema de las coordenadas, se ha decidido guardar en un formato específico como el de GeoJson https://geojson.io/#new&map=7.25/38.985/-3.582 de forma que si copiamos la parte correspodiente y la pegamos podremos ver el recorrido del vehículo de la siguiente forma:


```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [
          [
            -3.914463884311914,
            38.96908170549793
          ],
          [
            -4.043809121343827,
            38.71362131742626
          ],
          [
            -4.330216431915204,
            38.645103201471954
          ],
          [
            -4.284021704404154,
            38.384860789884215
          ],
          [
            -4.385650104929113,
            38.02185444013037
          ]
        ],
        "type": "LineString"
      }
    }
  ]
}
```



```
"order": {...},
"vehicle": {...},
"lastUpdate": {...},
"featureCollection": {
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "prop0": "value0"
      },
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [
            78.78667,
            89.8977
          ],
          [
            89.6777,
            -67.9777
          ]
        ]
      }
    }
  ]
```


Se ha realizado también **testing unitario con Junit**, de la parte de Vehículos, ya que las demás serían exactamente igual, se ha utilizado esta de ejemplo.



MvcTestCase define las peticiones que se usarán en los test, es la configuración básica del entorno de pruebas:

```
private final String username = "admin";
private final String password = "admin";

@Before
public void setup() throws Exception {
    this.mvc = MockMvcBuilders
            .webAppContextSetup(context)
            .apply(SecurityMockMvcConfigurers.springSecurity())
            .build();

    // Autenticar al usuario
    this.mvc.perform(formLogin().user(username).password(password));
}
```

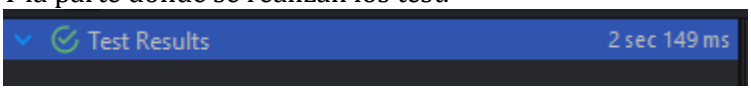
Primero se debe de estar autenticado, común para todos los tests, si no, darían error 401, 403, etc.

```
protected ResultActions doPost(String url, HttpSession session, Object... fieldsAndValues) throws Exception {
    url = normalize(url);

    JSONObject jso = new JSONObject();
    for (int i=0; i<fieldsAndValues.length; i=i+2)
        jso.put(fieldsAndValues[i].toString(), fieldsAndValues[i+1]);

    MockHttpServletRequestBuilder request = MockMvcRequestBuilders.
            post(url).
            contentType("application/json").
            content(jso.toString()).with(csrf());

    if (session!=null)
        request.session((MockHttpSession) session);

    return mvc.perform(request);
}

protected ResultActions doDelete(String url, HttpSession session, Object... fieldsAndValues) throws Exception {
    url = normalize(url);

    JSONObject jso = new JSONObject();
    for (int i = 0; i < fieldsAndValues.length; i = i + 2)
        jso.put(fieldsAndValues[i].toString(), fieldsAndValues[i + 1]);

    MockHttpServletRequestBuilder request = MockMvcRequestBuilders
            .delete(url)
            .contentType("application/json")
            .content(jso.toString()).with(csrf());

    if (session != null)
        request.session((MockHttpSession) session);

    return mvc.perform(request);
}
```

Aquí se define como se harán las peticiones get, delete, put, post.

```java
@Test
@WithUserDetails(value = "ADMIN")
void JupTest() throws Exception{

    MvcResult result;
    String content;
    JSONObject json;

    // Correct Register {Everything is correct}
    result = doPost( url: "/vehicles/registerVehicle", session: null,
            ...fieldsAndValues: "plate", "1234JKL",
            "model", "Audi A7",
            "capacity", 897
    ).andExpect(status().isOk()).andReturn();
    content = result.getResponse().getContentAsString();
    json = new JSONObject(content);
    assertEquals( expected: "200", json.getString( s: "code"));

    // Incorrect {Empty model}
    result = doPost( url: "/vehicles/registerVehicle", session: null,
            ...fieldsAndValues: "plate", "1234JKL",
            "model", "",
            "capacity", 897
    ).andExpect(status().isOk())
            .andReturn();
    content = result.getResponse().getContentAsString();
    json = new JSONObject(content);
    assertEquals( expected: "500", json.getString( s: "code"));


    // Incorrect {Duplicate vehicle plate}
    result = doPost( url: "/vehicles/registerVehicle", session: null,
            ...fieldsAndValues: "plate", "1234JKL",
            "model", "Audi A7",
            "capacity", 897
    ).andExpect(status().isOk())
```

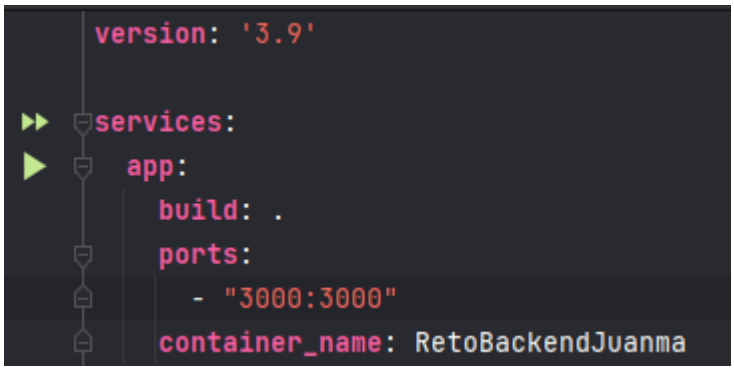
Y la parte donde se realizan los test.



Por último, para facilitar su corrección, se ha dockerizado la aplicación.

```dockerfile
FROM gradle:7.4.2-jdk17-alpine as build
COPY --chown=gradle:gradle . /home/gradle/src
WORKDIR /home/gradle/src
RUN gradle bootjar --no-daemon

FROM amazoncorretto:17.0.3-alpine
WORKDIR /home/app
COPY --from=build /home/gradle/src/build/libs/*.jar /home/app/application.jar
EXPOSE 3000
ENTRYPOINT ["java", "-jar", "/home/app/application.jar"]
```

Archivo dockerfile



Archivo docker-compose.yml

Y para levantar la aplicación basta con ejecutar en el directorio donde se encuentre el archivo dockerfile **docker-compose up.**

NOTA: para acceder a swagger poner en el navegador http://localhost:3000/swagger-ui/index.html#/