



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

University of Wales Trinity Saint David

DVLA: MSc Software Engineering (Web Development)

10/05/2023

Assignment 2

ACCA7005 – Web Services

By Rania Abdalla - 2222148

Table of Contents

<i>Introduction</i>	<i>3</i>
<i>Entities</i>	<i>3</i>
<i>Repositories.....</i>	<i>4</i>
<i>DTOs.....</i>	<i>4</i>
<i>Security Configuration</i>	<i>7</i>
<i>Areas for improvement.....</i>	<i>8</i>

Introduction

This report describes the design and implementation choices for my road tax management application through Spring Boot. The application is made up of entities, repositories, data transfer objects (DTOs). The data for these is accessible through Http requests to various endpoints that are secured using Keycloak to determine which users/roles have access to them. The application uses a PostgreSQL database to store all data.

Entities

The application consists of 4 entities Car, Owner, Taxinfo and Payment

Each entity has an automatically generated ID when a record has been created using “@GeneratedValue (strategy = GenerationType.IDENTITY)”

Relationships between entities

- Car
 - Contains entity properties for a car such as registration, make, model and VIN.
 - Has a many-to-one relationship with owner as one owner can have many cars and one car can have only one owner. The owner ID is added to each car by using @JoinColumn annotation to generate a column for the owner in the car entity.
 - One-to-many relationship with Taxinfo, as one car can have many tax information details considering tax information expires and needs to be renewed.
- Owner
 - Contains properties of a car owners' details such name and date of birth
 - An owner has a one-to-many relationship with car so for each owner the cars they own are stored in a list of type Car.
- Taxinfo
 - Contains properties for tax information for a car such as amount, reference, and expiry.
 - It has a many to one relationship with car as a car may have many tax details as it includes start data, expiry etc.
 - Taxinfo also has a one-to-one relationship with payments as for each tax information there will be payment made.
- Payment
 - Contains properties for payment details such as method and duration.
 - It has a one-to-one relationship with tax information as each payment should only be associated with one tax information details.

Repositories

The repositories package contains repository interfaces for each entity to be able to define a set of operations for persisting and retrieving entity objects from a database. Each interface extends CRUDRepositories which allows the use of generic CRUD operations on the entities.

DTOs

Because of relationships between entities in my application, Data transfer objects (DTOs) are required to allow a user to get details of a specific record in one request.

I have created three DTOs for this application for Car, Payment and Taxinfo. Each of these follows a similar pattern to get the information required. That is through creating a DTO class that declares variables for the details required, a DTO service file that converts an object to a DTO object and two functions that can be called to retrieve a list of DTOs or a specific DTO using a unique identifier and a DTO details repository to create endpoints for retrieving the DTOs.

Below is a description for the car DTO only, I followed a similar approach for the PaymentDTO and TaxinfoDTO.

For a car to be able to retrieve its details and its owner's details with one get request I did the following.

- Created a DTO class that declared variables for all the entity properties that a user might want to retrieve.
- Created a CarDTOService file, which is annotated using @Service and is autowired to CarRepository to achieve dependency injection. The service file has a method for converting a Car object to a CarDTO object. This method takes in a Car object and instantiates a new CarDTO object. It uses the getter and setter methods in the CarDTO class to set the variables specified in it with the details retrieved from the car object that is passed in. An example of this is shown in figure 1 below:

```

1 usage
private CarDTO convertToCarDTO(Car car)
{
    CarDTO cDTO = new CarDTO();
    cDTO.setCarid(car.getCarid());
    cDTO.setCarreg(car.getCarreg());
    cDTO.setMake(car.getMake());
    cDTO.setModel(car.getModel());
    cDTO.setVin(car.getVin());

    cDTO.setOwnerid(car.getOwner().getOwnerid());
    cDTO.setName(car.getOwner().getName());
    cDTO.setDob(car.getOwner().getDob());
    cDTO.setAddress(car.getOwner().getAddress());

    return cDTO;
}

```

Figure 1: Convert to a Car DTO in the CarDTOService file.

It also has the method `getAllCarDTO()` which returns a list of all CarDTOs as shown in figure 2 below. Where “`carRepository.findAll`” retrieves all the car objects from the database using the `carRepository` interface that is autowired. “`(List<Car>)`” casts the results of `findAll` to a List of type `Car`. “`stream`” converts the `List<Car>` to a stream to allow using stream operations on the list. “`map(this::convertToCarDTO)`” applies a mapping function to each element in the stream and calling the `convertToCarDTO` function to convert each car object to a `CarDTO` object. “`collect(Collectors.toList())`” collects the stream of `CarDTO` objects into a List of `CarDTO`.

```

//Return all cars in the Car DTO format
2 usages
public List<CarDTO> getAllCarDTO()
{
    return((List<Car>)carRepository.findAll()).stream().map(this::convertToCarDTO).collect(Collectors.toList());
}

```

Figure 2: Method to return all CarDTOs.

- Lastly, I created a `CarDetailsRepository` class which is a `RestController` for creating endpoints to retrieve the car details. This is done by issuing a GET http request to the endpoint “`/cardetails`” or “`cardetails/{a specific id}`” by passing a car ID in the URL when issuing a request as shown in figure 3 below.

```

13
14 @RestController
15 public class CarDetailsRepository {
16     @Autowired
17     CarDTOService carDTOService;
18
19
20     @GetMapping("/cardetails")
21     public List<CarDTO> getAllCarDetails() { return carDTOService.getAllCarDTO(); }
22
23
24     @GetMapping("/cardetails/{id}")
25     public CarDTO getCarByID(@PathVariable("id") Integer carid)
26     {
27         CarDTO cDTO = carDTOService.getCarDTOById(carid);
28
29         if(cDTO != null)
30         {
31             return cDTO;
32         }
33         throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Car ID not found");
34     }
35 }
36
37
38

```

Figure 3: Rest Controller for mapping endpoints for retrieving car details in DTO format.

Security Configuration

I used Keycloak to create a realm and created a client, roles, and users to allow secured access to various endpoints in my application. I created an admin, user, and member roles. An admin can make any HTTP request to all endpoints, a user is only able to make a POST request a member can only view and modify the endpoints for their specific records which I have included for demonstration purposes as shown in figure 4 below:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{

    http.authorizeHttpRequests()

        //roadtaxservicemember allows specific user to modify "their" details
        .requestMatchers(HttpMethod.GET, ...patterns: "/paymentdetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.GET, ...patterns: "taxinfodetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.GET, ...patterns: "cardetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.GET, ...patterns: "owner/2").hasRole("roadtaxservicemember")

        .requestMatchers(HttpMethod.PUT, ...patterns: "/paymentdetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.PUT, ...patterns: "taxinfodetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.PUT, ...patterns: "cardetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.PUT, ...patterns: "owner/2").hasRole("roadtaxservicemember")

        .requestMatchers(HttpMethod.DELETE, ...patterns: "/paymentdetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.DELETE, ...patterns: "taxinfodetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.DELETE, ...patterns: "cardetails/1").hasRole("roadtaxservicemember")
        .requestMatchers(HttpMethod.DELETE, ...patterns: "owner/2").hasRole("roadtaxservicemember")

        //only an admin can view/modify all details
        .requestMatchers(HttpMethod.DELETE, ...patterns: "/*").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.GET, ...patterns: "/*").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.PUT, ...patterns: "/*").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.PATCH, ...patterns: "/*").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.GET, ...patterns: "/cardetails").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.GET, ...patterns: "/taxinfodetails").hasRole("roadtaxserviceadmin")
        .requestMatchers(HttpMethod.GET, ...patterns: "/paymentdetails").hasRole("roadtaxserviceadmin")

        //anyone can create a record
        .requestMatchers(HttpMethod.POST, ...patterns: "/*").hasRole("roadtaxserviceuser")

        .anyRequest().authenticated();
}
```

Figure 4: Restricting endpoints to certain realm roles.

Because of the security configuration and endpoint restrictions when using Postman to make requests if not authenticated properly then requests will be unauthorised if the user I created has not signed in or forbidden if the user does not have the right privileges with the HTTP status codes 401 and 403 respectively.

Areas for improvement

Although I can use this service to perform various tasks I have come across an issue once I created a Docker file to deploy my service particularly when it comes to Keycloak. Although I tried, I was unable to download the Keycloak image within my project to be able to include in the Docker file and to be able to use Postman or the command line to authorise users and make requests. Instead to be able to issue requests a version of Keycloak must be running on localhost:8090 to match the configuration in my application.properties file and the service to be running in an IDE. I would have also liked to spend more time on testing my service.