

Evaluation Report for The Implementation of a Tax Management Web Service

LINK TO CODE: <https://github.com/CroucherR/taxmanagement>

Author: Riley Croucher

Student Number: 2013055

Degree Programme: Software Engineering (Web Development) MSc

Lecturer: Dr Tim Bashford

1. Service Design

1.1 Road Tax

Road Tax also known as Vehicle Tax is a tax paid by vehicle owners that is paid into a government fund to be used on infrastructure projects such as road work and maintenance. Each vehicle owner pays the rates of vehicle tax for their vehicle tax band. To determine vehicle tax bands a document (V149) is released by the DVLA each year which denotes the different variables which affect the cost of road tax. One of the larger variables is the date in which the vehicle was registered.

Table: Showing the Tax Band determination based on registration date.

Registration Date	Tax Band Determination
Before 1 st of March 2001	Engine size: Lower tax band: Engine size <1550cc Higher tax band: Engine size >=1550cc
Between the 1 st of March 2001 and the 31 st of March 2017	CO2 Emission and Alternative Fuels: Lowest tax band: Up To 100g/km
After the 31 st of March 2017	CO2 Emission and Alternative Fuels: These are stricter than the previous tier Lowest tax band: 0g/km

There are also other factors that determine Vehicle tax such as whether the vehicle is classed as a classic car, or whether the vehicle has a list price value of over £40,000 on first registration. A vehicle that was built more than 40 years ago is determined as a Classic vehicle and thus does not need to pay vehicle tax and a vehicle with a list price greater than £40,000 on first registration has to pay an additional rate of road.

1.2 ERD Design

Following the research as described in section 1.1 to implement a realistic road tax management service I had to incorporate the different variables of a vehicle which are deterministic of its tax band and thus overall tax cost. Figure 1 below shows the finished ERD diagram in which the entities and their variables are depicted along with the relationships between them. I decided to implement five entities: Vehicle, Owner, VehicleRegistration, TaxInformation and TaxPayment. Within each of these entities I described the variables they would need in order to meet the realistic road tax band determination, within vehicle I have a Year variable which would allow the system to check if it counts as a classic vehicle, I also have an EngineStats String variable which allows for the entry of either the size of the engine in CCs for Vehicles registered before March 2001 or for the input of CO2 emissions for vehicles after 2001. The Boolean variables in the TaxInformation Entity (IsClassic and IsValue) determine whether the

vehicle classes as a classic vehicle based on the year stored in Vehicle and also whether the vehicle is liable to extra road tax as its list price value (stored in value within vehicle) is over £40,000. The relationships between each of these entities are ONE-ONE expect for the Owner to VehicleRegistration relationship which is defined as a ONE-MANY as one Owner can have many vehicle registrations.

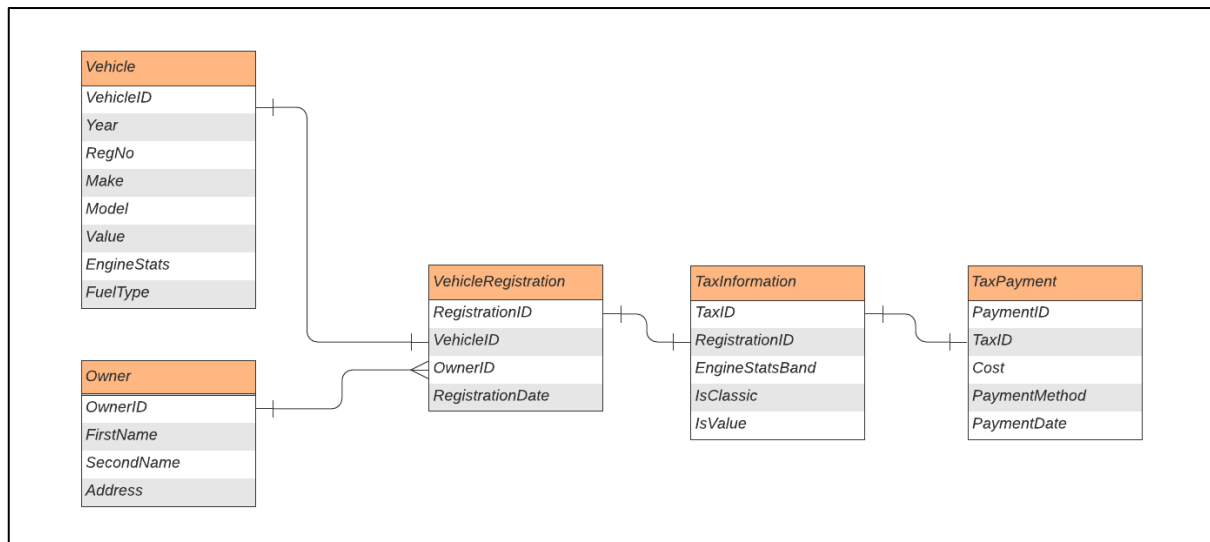


Figure 1: Showing the ERD diagram for the service.

2. Service Implementation

2.1 Models and Repositories

Each of the entities defined in figure 1 were created as model entities using Java Classes which initialised the variables stored within them using getters and setters. Figure 2 below shows a screenshot of the Owner model entity Class. Within these entity classes the relationships with other entities are also defined as can be seen in figure 2 which defined a ONE-MANY relationship with the VehicleRegistration.class.

Repositories for each of the models were then created. I decided to implement CRUD repositories for the Vehicle and Owner models as I believe these models will only need to use the CRUD operations whereas I used PaginationandSorting repositories for the TaxInformation, VehicleRegistration and TaxPayment repositories as these may need functions with a higher complexity.

```

/**
 * OWNER MODEL -
 * Creates the owner entity and initialises the variables using getters and setters.
 */

@Entity
public class Owner implements java.io.Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int ownerid;
    private String firstname;
    private String secondname;
    private String address;
    @OneToMany(fetch = FetchType.LAZY, targetEntity = VehicleRegistration.class, mappedBy = "owner")
    private List<VehicleRegistration> owner;

    public Owner() {

    }

    public Owner(Owner owner, String firstname, String secondname, String address) {
        this.firstname = firstname;
        this.address = address;
        this.secondname = secondname;
    }

    public int getOwnerid() { return ownerid; }

    public void setOwnerid(int ownerid) { this.ownerid = ownerid; }

    public String getFirstname() { return firstname; }

    public void setFirstname(String firstname) { this.firstname = firstname; }

    public String getSecondname() { return secondname; }

    public void setSecondname(String secondname) { this.secondname = secondname; }

    public String getAddress() { return address; }

    public void setAddress(String address) { this.address = address; }
}

```

Figure 2: Showing the Owner Model Entity.

2.2 DTOs

Within my service I have three Data Transfer Objects (DTOs), these DTOs enable data to be transferred from other entities which has a relationship with the entity DTO and thus produce more readable outputs. I created a VehicleRegistrationDTO, a TaxInformationDTO and a TaxPaymentDTO, I chose to implement these DTOs as these original entities did not show a degree of detail to which a user would be able to understand. For example figure 3 and figure 4 below show the VehicleRegistration output compared to the VehicleRegistrationDTO. Within the standard output the only information that can be gathered is the registration date and what entities relate to it whereas the DTO output shows the name of the owner along with vehicle details such as the regNo , make and model.

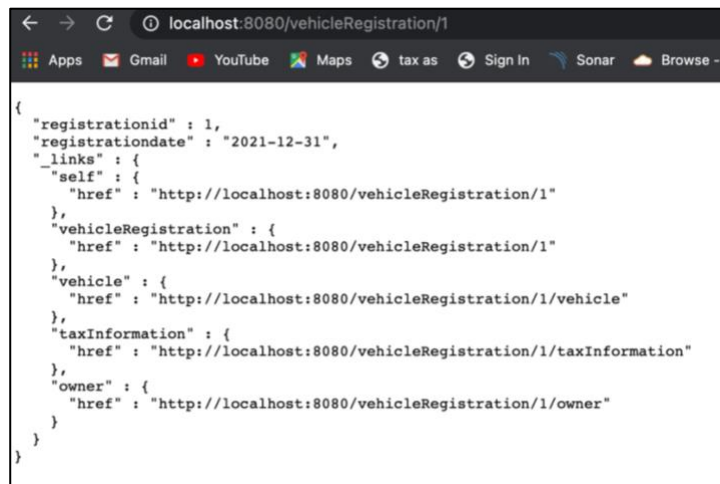


Figure 3: Showing the VehicleRegistration output.



Figure 4: Showing the VehicleRegistrationDTO output.

To implement these DTOs I created similar models but also had to build services for each which would contain the methods for converting the original entity data into the DTO as well as getting and setting the correct variables. The repositories implemented for the DTOs consisted of mapping the correct service method to a url to display either all the DTOs for that entity or finding a DTO by ID number.

2.3 Swagger

Swagger was implemented to allow for full documentation of the service. To do this a SwaggerConfig class was created which uses a docket api method to build and return the documentation of the service. Each of the repositories were then exposed to the swagger service using a @Api "Address Entity" tag. This tag exposes the repositories using the created RepositoryCong which overrides the RestConfiguration for each of the entities.

2.4 Testing

Within my application I implemented a series of Junit tests in order to test whether incorrect GET requests, successful POST requests, successful GET requests and successful DELETE requests worked correctly for each of the endpoints. These tests implemented a H2 in memory database in order to ensure that data from the main postgres database was no effected. To set up and run the H2 database a separate testing application.properties was created within the test section of the project.

2.5 Keycloak

The security implemented in this management service is Keycloak. Keycloak allows for authentication and authorisation by creating a Realm in which users can be assigned certain roles. These roles can then receive an access token when posting their details to the Keycloak server. This access token will then allow them to authenticate HTTP requests made to the tax management service. Authorisation of each of the roles is implemented within the `TaxManagementServiceConfig` and denotes that a user can only make Get requests while an admin can make all requests.

To allow for swagger and any other urls I implemented a Whitelist method in the `TaxManagementServiceConfig`. The urls within the Whitelist are permitted by keycloak no matter which role or user submits the request.

2.6 Docker

Docker has also been implemented in order to run the base service through a docker container. I was unable to implement the Keycloak server into Docker but I have included instructions in order to run the service with or without keycloak and have attached the exported realms to the project.