

University of Wales Trinity Saint David

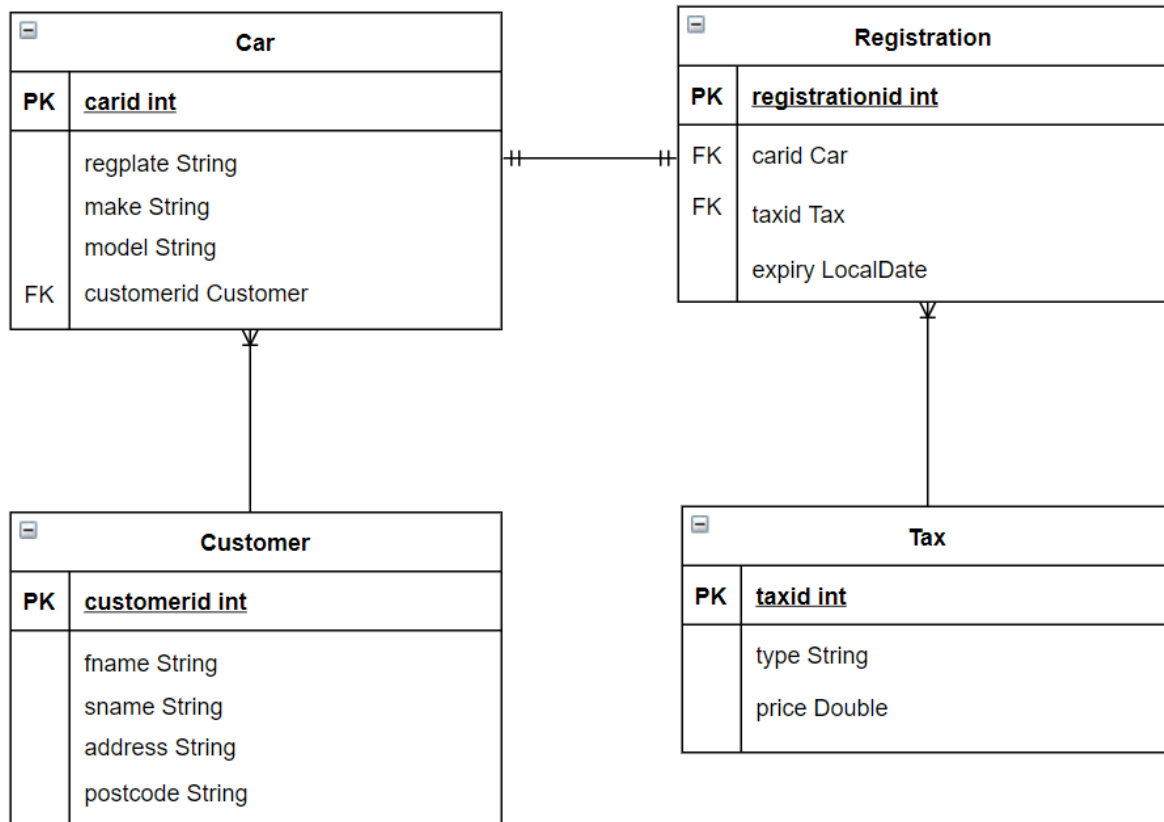
Web Services

ACCA7005

Elliot Short - 1702797

You should write a report of approximately 1000 words which outlines and describes your design and implementation choices.

When designing a RESTful web service database principles must be taken into consideration. In such a web service a user must be able to create a vehicle and a profile for themselves. After this the user must be able to tax this vehicle with the appropriate information. This led to the decision to have four separate classes in order to fulfill the project's criteria. This ERD shows what the layout of this design choice looks like. Customers would include basic customer



information to identify who the owner of the next table is, car. The car table includes registration plate and the type of car it is as well as having customer ID as a foreign key. This choice comes from the fact that in the UK a vehicle can only have one registered owner. This led to the choice to have a one to many relationship between customer and car. This allows the customer to own multiple vehicles but for those vehicles to only have one owner. Next is the registration process in which the customer can tax their vehicle. Before a vehicle can be registered and taxed the Tax table is queried and depending on the type of vehicle the price will vary. For example, a hatchback would have a different rate of tax compared to an estate. Once the tax type is specified all the data up to that point is input into a registration table. This table stores the car that has been registered, the tax id so that the price and type of tax is known, and finally the expiry date when the vehicle's tax runs out. These four tables cover the criteria of a road tax management system, while also adhering to database principles. One of the principles that this

system adheres to is the concept of reducing repeating data. This ERD shows that there is no unnecessary data replication across the four tables, this principle is important because as a service expands and takes in more and more data, even just one field of repeated data can severely slow query times. By reducing this it ensures this RESTful service will stay as quick as possible.

When deciding what information was important to bundle together to form a DTO it was important to consider who would be trying to access the information and for what reason. For example, if a user of the service wanted to find out who was the owner of a car you want a DTO that gives you all the relevant information that you would need there. In this first example is the customer DTO which contains all the information that the service contains on a customer and what their car is. By turning all this information into a DTO it is easy to see who is the owner of a certain car. This information allows the process of registering a vehicle to be quicker, when registering you must know what type of vehicle it is you own as that will change the price of tax. This DTO allows the user to quickly access this information by seeing what model of car it is they are trying to tax. On the front end of a web service, this information could be used in the user's profile, when they want to see what information of theirs is being stored on the

system. This information could also be used to see what vehicles that a customer owns. If a customer own's multiple vehicles this DTO can be used to output all the vehicles that customer owns, otherwise the user has to look up each vehicle manually.

The second DTO acts as almost an extension and clarification of the registration table. When querying the registration table the data that is returned can be hard to interpret as it doesn't give back what the vehicle is and what the tax type is, rather it returns what the ID is for both of these fields. What the registration DTO does is display all this information allowing for easier access than what the web service requires at a base level. The second image shows that the DTO displays the tax type, price of tax, expiry date and the registration plate of the

```
public class CustomerDTO
{
    private int customerid;
    private String fname;
    private String sname;
    private String address;
    private String postcode;
    private String regplate;
    private String make;
    private String model;
```

```
public class RegistrationDTO
{
    private int registrationid;
    private String type;
    private double price;
    private LocalDate expiry;
    private String regplate;
```

vehicle in question. On the front end of a web service this information would be useful being bundled together, whether it's a customer querying information about their registration or an automated system determining how long a certain vehicle has left until they need to be taxed again.

When building the authentication for the service the types of users querying the service was a key consideration. What different types of users would be querying the service, what roles would they have and what permissions would they need. Two users were decided upon, and admin and a default user. No more were needed as there wouldn't be any significant variance in the type of customer that would use this service, so there wasn't any need to give them specific roles. The default user would need to be able to add their own customer data, car data and be able to get their own customer and car data. The customer would also be able to get the tax and registration tables, the tax table would allow the customer to see what price they may have to pay depending on their vehicle while the registration data would be able to show them when they need to renew their road tax. The admin role would have the power to change anything in the service. This is so there is a role that can change data if anything breaks on the front end,

```
.antMatchers(HttpMethod.GET, ...antPatterns: "/*").hasRole("cartaxuser")
.antMatchers(HttpMethod.POST, ...antPatterns: "/car/").hasRole("cartaxuser")
.antMatchers(HttpMethod.POST, ...antPatterns: "/customer/").hasRole("cartaxuser")
.antMatchers(HttpMethod.POST, ...antPatterns: "/*").hasRole("cartaxadmin")
.antMatchers(HttpMethod.DELETE, ...antPatterns: "/*").hasRole("cartaxadmin")
.antMatchers(HttpMethod.PATCH, ...antPatterns: "/*").hasRole("cartaxadmin")
.antMatchers(HttpMethod.PUT, ...antPatterns: "/*").hasRole("cartaxadmin")
```

having a role that can have power over the service also allows for easier debugging as you don't need to create new authentication tokens depending on what you need to do with the service.

Overall I'm happy with my design decisions and my implementation of these decisions. If I had to make a front end based off these decisions I would be happy that the back end has sufficient functionality that I wouldn't have to go back and rethink my choices.