

UWTSD

DVLA: MSc Software Engineering (Web Development)

Web Services (ACCA7005), Individual Assignment

Callan Birch-Williams, 2222124

Table of Contents

Introduction.....	3
System Design	3
Implementation.....	4
Documentation.....	5
Testing	5
Security.....	7
Docker	8
Bibliography	8
Figure 1: Entity relationship diagram.....	3
Figure 2: Repository detection strategy	4
Figure 3: Controller exception handling and response.....	4
Figure 4: Vehicle DTO	5
Figure 5: Service unit testing.....	6
Figure 6: Controller integration testing	6
Figure 7: Security filter chain	7

Introduction

The purpose of this document is to outline and discuss some of the design decisions and implementation details of a RESTful web service for the backend of a road tax management system (<https://github.com/CB2222124/rtns>).

System Design

The fundamental entities established for this application are as follows:

- An owner within this application is an individual uniquely identifiable by their username. A sensible addition to this application in the future would be the ability for the owner to be an organisation.
- A tax class within this application is a unique entity that holds a description and tax rate. The tax rate for a specific vehicle within this application is determined by its tax class. This is an appropriate simplification of real-world VED rate calculations.
- A vehicle within this application is uniquely identifiable by its registration. It is strictly associated with a tax class and an owner. Ownership can be transferred, but a fundamental design choice for this application is that a vehicle must have an associated owner to exist as an entity. This abstraction represents the real-world association between a vehicle and its registered keeper, who is required to handle taxation and is the target of reminders.

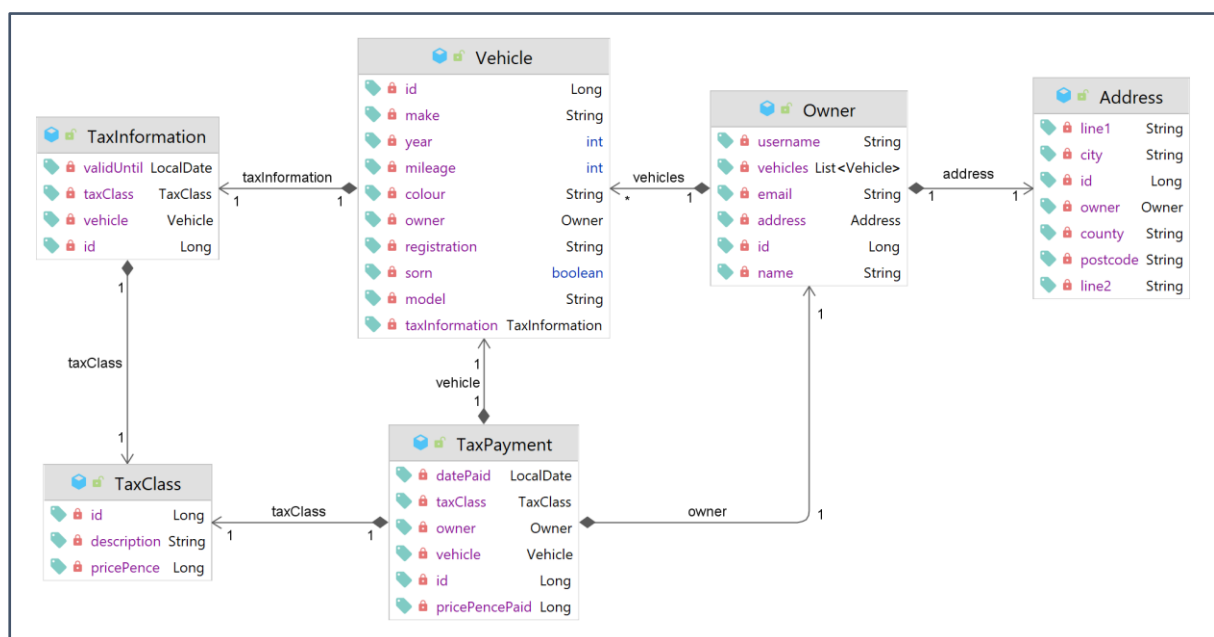


FIGURE 1: ENTITY RELATIONSHIP DIAGRAM

Implementation

The structure of the application has been designed to achieve a strong separation of concerns. At the package level, the application is split into logical components. Furthermore, separate controllers, services and repositories exist for the entities that they concern. Application repositories are configured to not be served as rest resources by default, such that service consumer interaction with the data layer is hidden behind a controller, service and then repository. This guarantees that each main application component is responsible for serving the API, business logic and data access layer respectively.

```
@Configuration
public class RepositoryConfig implements RepositoryRestConfigurer {
    @Override
    public void configureRepositoryRestConfiguration(...) {
        config.setRepositoryDetectionStrategy(
            RepositoryDetectionStrategy.RepositoryDetectionStrategies.ANNOTATED);
    }
}
```

FIGURE 2: REPOSITORY DETECTION STRATEGY

Services within this application will only throw expected custom exceptions that the controller layer can provide appropriate responses for. While not necessary for this application, the controller exception handlers could easily be modified to capture the exception message as a parameter, facilitating more fine-grained responses if required.

```
@ExceptionHandler(VehicleNotFoundException.class)
public ResponseEntity<String> handleVehicleNotFoundException() {
    return ResponseEntity.status(HttpStatus.NOT_FOUND)
        .body("Vehicle not found");
}
```

FIGURE 3: CONTROLLER EXCEPTION HANDLING AND RESPONSE

Data transfer objects (DTOs) are used throughout this application. By wrapping requests and responses in DTOs, the application is less likely to expose internals. Furthermore, DTOs are useful for returning custom views of data that combine entities. A good example of both advantages is the DTO used to provide tax information about a vehicle. Details about the owner are not exposed and tax information from other entities (Tax class and renewal rate) is included.

```
public record VehicleDTO(  
    String registration,  
    String make,  
    String model,  
    int year,  
    int mileage,  
    String colour,  
    boolean sorn,  
    LocalDate taxValidUntil,  
    Long taxClass,  
    Long renewalPricePence)  
{
```

FIGURE 4: VEHICLE DTO

Documentation

To provide API documentation, *springdoc-openapi* [1] is used to deploy *swagger-ui* to infer API semantics based on the annotations of mapped rest resources (In this context of this application these are all controllers). As API documentation is served at runtime, *swagger-editor* [2] is used to generate static HTML documentation.

Additionally, conventional Javadoc documentation is used at the service layer to provide full documentation coverage of the application.

Testing

To provide a suitable degree of test coverage, two main types of tests were conducted. At the application service level, comprehensive unit tests covering all functionality are used to ensure correct behaviour. At a controller level (The level at which a user would consume this service), integration testing is conducted to ensure that for every possible request, the application produces the correct response type.

To facilitate unit testing, *Mockito* [3] is used to mock any dependencies that a service may have on repositories. On the other hand, the Spring Boot testing framework is used to facilitate integration testing through bean mocking so that the interaction between controller and service can be tested. Rather than invoking the controller methods, the exposed resources are tested themselves by calling them using the Spring testing frameworks *MockMvc*.

```

@ExtendWith(MockitoExtension.class)
public class TaxClassServiceTest {

    @Mock
    private TaxClassRepository taxClassRepository;

    @InjectMocks
    private TaxClassService taxClassService;

    ...

```

FIGURE 5: SERVICE UNIT TESTING

```

@WebMvcTest(controllers = OwnerController.class)
@AutoConfigureMockMvc(addFilters = false)
@ExtendWith(MockitoExtension.class)
public class OwnerControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private OwnerService ownerService;

    ...

```

FIGURE 6: CONTROLLER INTEGRATION TESTING

To test the application further, end-to-end testing could be conducted making use of an in-memory database such as *H2* or by using *Testcontainers* [4], which is useful for cases where there are dependencies on a specific database that in-memory options cannot fulfil. Furthermore, considerations to the testing of endpoint security should be considered.

Security

This application makes use of Keycloak to provide a security layer. A realm import file is used to provide a keycloak docker container with a realm, roles, and a default admin user (Covered in the next section of this document).

A security configuration bean is used to handle incoming requests and protect application endpoints depending on user authorization and role.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests()
        //Documentation.
        .requestMatchers(HttpMethod.GET, "/v3/api-docs/**").permitAll()
        .requestMatchers(HttpMethod.GET, "/swagger-ui/**").permitAll()
        //Token requests.
        .requestMatchers(HttpMethod.POST, "/auth/token").permitAll()
        //Allow anyone to register.
        .requestMatchers(HttpMethod.POST, "/owners").permitAll()
        //Tax class requests.
        .requestMatchers(HttpMethod.GET, "/taxClasses/**").permitAll()
        .requestMatchers(HttpMethod.PUT, "/taxClasses").hasRole(ADMIN)
        .requestMatchers(HttpMethod.POST, "/taxClasses").hasRole(ADMIN)
        //Tax request.
        .requestMatchers(HttpMethod.POST, "/tax").hasRole(OWNER)
        ...
}
```

FIGURE 7: SECURITY FILTER CHAIN

The application provides functionality for an unauthorized user to either get a bearer token through authorization or create a new account. This functionality is achieved by providing endpoints which connect to the Keycloak instance through its API.

One of the key improvements that could be made to this applications security configuration would be method-level authorization. For example, exposing the vehicles owned by an owner only to the owner themselves (And administrators). This is something that was experimented with but not fulfilled within the time allocated. Furthermore, user creation at the Keycloak layer makes use of the super admin credentials when interacting with the API. It would be more appropriate to use client credential grants, though this required specific realm configuration that was unable to be achieved in this iteration of the application.

Docker

To deploy this service, a docker-based approach has been used. A multi-stage *Dockerfile* has been created to first build the application, then run it. This removes the dependency on the Java build process by providing a standalone build that only requires Docker on the target platform. This approach also has an advantage over those that use a build tool image (Such as maven) as the resulting container only contains a runtime environment and the built application, making it lightweight.

A *docker-compose.yml* file is used to run the backend service, keycloak and a Postgres database. As the application is dependent on a working database instance, it waits for the database service health check to report healthy. Furthermore, the backend service also depends on keycloak. The keycloak image is provided with admin credentials and an application realm configured with roles and one demonstrational admin user.

Bibliography

- [1] N. Badr, "springdoc-openapi v2.1.0," [Online]. Available: <https://springdoc.org/v2/>. [Accessed 05 2023].
- [2] S. Software, "API Editor - Download or Try it in the Cloud," [Online]. Available: <https://editor.swagger.io/>. [Accessed 05 2023].
- [3] S. Faber, "Mockito framework site," [Online]. Available: <https://site.mockito.org/>. [Accessed 05 2023].
- [4] R. North, "Testcontainers for Java," [Online]. Available: <https://www.testcontainers.org/>. [Accessed 05 2023].