

A Reflection On The Road Tax Management RESTful API

With A Focus On Design And Implementation

Finchley A. Harwood

School of Applied Computing

University of Wales Trinity St. David

May 2021

Abstract

The Road Tax Management Service is an API used for interacting with a Postgres database for storing tax information on vehicles. The service uses Keycloak to provide OAuth 2.0 security and Swagger Codegen to provide documentation inline with OpenAPI specification. The following report is an overview of the design decisions behind the author's implementation of the Road Tax Management Service RESTful web service.

Contents

1	Introduction	3
2	Design	3
2.1	High-level Design	3
2.2	Database design	4
2.3	RESTful Service Design	5
3	Implementation	6
3.1	Users	6
3.2	Testing	6
3.3	Exceptions	6
3.4	Documentation	7
4	Summary Reflection And Future Work	7

1 Introduction

The Road Tax Management System is a Dockerised RESTful Spring Boot Application Programming Interface (API). The API allows the user to interact with a Postgres database that contains information on drivers, vehicles and tax payments for vehicles. The system uses Keycloak to provide different access levels and security. The following report is a brief discussion of the design and implementation choices behind the Road Tax Management System.

2 Design

2.1 High-level Design

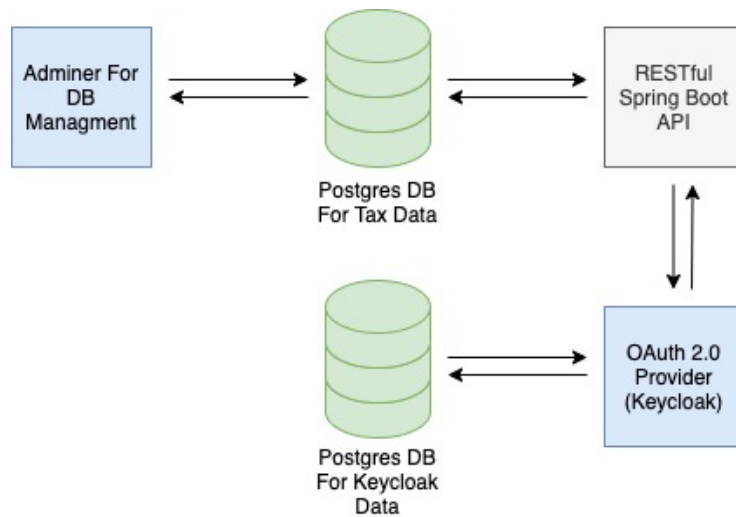


Figure 1: A diagram showing the high-level design for the system. This design is reflected in the `docker-compose.yml` file, which is used for deployment.

The design process began by mapping out the different components that would make up the system. The diagram of Figure 1 is an overview of the high-level design that resulted from mapping out these different components. Two separate Postgres databases are used for persisting both tax data and Keycloak data. Adminer is used for directly interacting with the database, and Keycloak is used to provide security. This design facilitates the requirement for the system to be dockerisable and secure. In addition, since it is modular, the system is somewhat future-proof in that it is easily expandable.

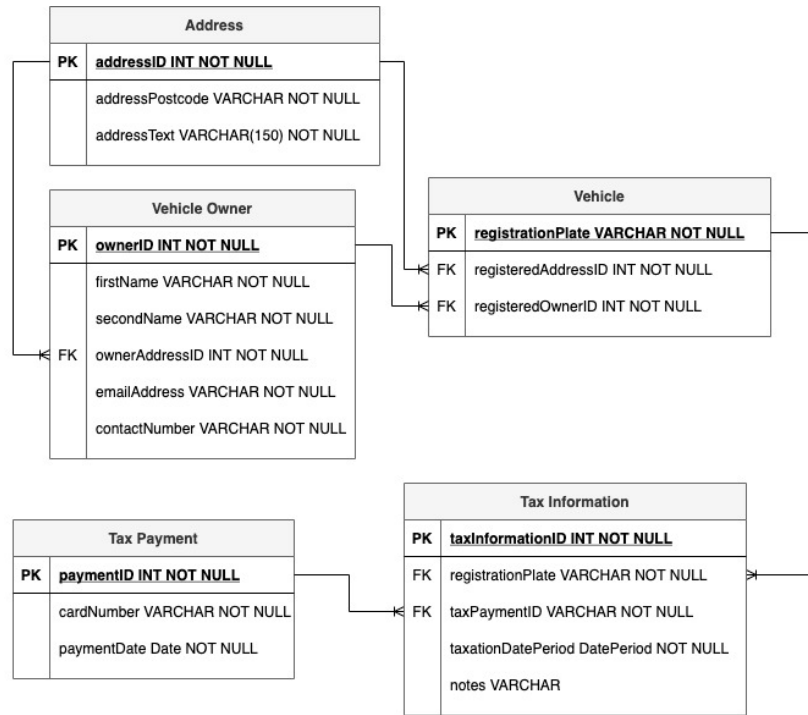


Figure 2: A diagram showing the design for the Postgres Database. In an attempt to keep things simple, the database is designed to be compact but functional.

2.2 Database design

The diagram of Figure 2 summarises the design for the relational Postgres database for storing data on taxes. The Address entity is intended to be used as a lookup table for storing different addresses, and in an attempt to keep things simple has only two fields. The Vehicle and Vehicle Owner entities capture the minimum amount of information required to describe them. A Vehicle Owner is capable of owning many vehicles, and a vehicle and a driver can be registered to a different address, allowing for flexibility for businesses.

The Tax Information entity is used to relate vehicles and tax payments to one another. This entity uses a so-called ‘DatePeriod’ type which encapsulates a period of time between a start date and end date, i.e. the period for which a taxation payment is valid. All of the entities in the Database design are reflected in the `@Entity` objects of the REST service¹.

¹Note that the final implementation is a *reflection* of this design and not an exact copy of it.

2.3 RESTful Service Design

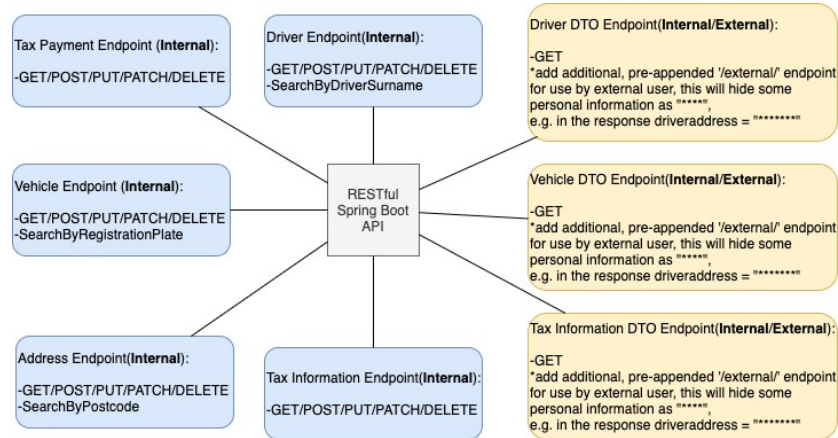


Figure 3: A diagram showing the different endpoints for the Spring Boot service. Note how different end points are designated as either being **Internal** or **Internal/External**. If an endpoint is external then it can be accessed by a user who is from outside the organisation.

The diagram of Figure 3 was created to summarise the different endpoints that are in the REST service. Endpoints are classified as either **Internal** or **External**. This designation allows for different amounts of information to be returned during a query from a user. The idea behind this being that the API might be used by internal (DVLA) staff or external operators, such as insurance companies that may need to have restricted access to certain data fields. The actual implementation of this is arbitrary, certain data is hidden which may in reality need to be given to external users, but it works as a proof of concept.

All of the externally accessible endpoints are Data Transfer Objects (DTOs). DTOs contain a clear response, allowing external users to access precisely the information they need, without having confusing information such as links to other endpoints.

There are three different search endpoints for internal use only, these being search by registration plate, postcode and surname. This allows internal staff to search for different records with only partial information.

3 Implementation

3.1 Users

There are three different user-types specified in the Keycloak service (admin, basic and external). The level of access granted to an external user is a subset of the level of access granted to a basic user, and similarly an admin user is a superset of a basic user. External users only have access to endpoints which are pre-appended with “/external/”. This allows for a different function to be used which can hide sensitive user information.

Basic users can access the external DTO endpoints, the non-external DTO endpoints, search endpoints and all *GET* requests on internal endpoints. The admin user can do everything that a basic user can do, but also has access to the *POST*, *PUT*, *PATCH* and *DELETE* HTTP verbs.

To obtain an access token, a user must first post to the Keycloak endpoint specified in the README.md file. Upon obtaining an access token a user can interact with the endpoints that the token grants them access to. The author recommends using ‘Postman’ for interaction with the API.

3.2 Testing

There are number of different test files in the project. As discussed in the README.md file, before running the tests the developer must first run `docker-compose up` in the parent directory of the project. This is due the fact that the tests rely on Keycloak to be running in the background.

The `DatePeriodTest` class tests the `DatePeriod` class discussed in Section 2.2. The tests check that the methods `parse()` and `toString()` belonging to the class are working correctly.

The `TestParent` class contains methods and parameters which are used by classes that directly test the REST service. The `KeyCloakAuthTest` and `RoadTaxManagementServiceApplicationTests` classes extend this `TestParent` class. The `KeyCloakAuthTest` class tests the Keycloak authorisation tokens and levels of access. The `RoadTaxManagementServiceApplicationTests` class tests the different endpoints of the API.

3.3 Exceptions

The `RestEntityExceptionHandler` class handles three different types of exception, and returns a valid error message to the user of the API. The most notable of these exceptions is the `DatePeriodFormatException`, which occurs when the user has failed to format a `DatePeriod` properly in a request. Since a `DatePeriodFormatException` occurs due to a `DateParseException`, the code

always returns the error message for a `DatePeriodFormatException` if it exists in the stack trace.

3.4 Documentation

OpenAPI documentation generated using Swagger Codegen can be found at the URL specified in the `README.md` file of the project. To ensure ease of access, the `Security Config` class, in the configuration package, permits access to all of the endpoints related to Swagger Codegen for any user. This means that the user need only enter the correct URL into a web browser to view the documentation.

4 Summary Reflection And Future Work

The Road Tax Management System is a RESTful web service written in Spring Boot. The project consists five different components: two Postgres databases, an Keycloak service, a Restful web service and Adminer. These are run as containers using `docker-compose`. The services work together to provide a secure and documented API with different levels of access. An attempt at providing a decent level of test coverage has been made, but this is no guarantee of the system being free of errors.

Future work may include expanding on the exception handling, allowing for more precise errors to be returned to the user. In addition, creating a client service for the API would be a useful exercise and could form the basis for an internal or external application.