

Web Services Assignment

ACCA7005

Liam Samuel Roberts

May 14, 2021

1 Introduction

This report accompanies the implementation of a rest service designed to fulfil the backend requirements of a vehicle tax management service. The service must have entities which include car registration, car owner, tax information and tax payments. The service had to include two data transfer objects, DTOs, and have automatically generated supporting documentation with suitable test coverage. The service also required the implementation of a security technology. Finally, the service needed to be deployed using Docker.

2 Design, Testing and Documentation

2.1 Entity-Relationship Design

Typically, when storing owners is mentioned, an address table usually follows and as such the design process focused on the four entities outlined in the brief and the addition of an address entity. The process began by listing all the necessary attributes associated with the entities, which were then refined into tables. Moving forward, the goal was to consider and define the relationships between these tables. For example, one car's tax cost can be paid through one payment and as such a one-to-one relationship exists between the two entities. The same is true for the relationship between car and car tax. One owner may own many cars; however, one car cannot have multiple owners, therefore, a one-to-many relationship is used between the owner and car tables. Finally, a one-to-many relationship is used between address and owner, allowing for multiple owners to have occupy the same address. A database diagram showcasing this design can be seen in Fig. 1.

2.2 Data Transfer Objects

From the design of the database, it was decided that the most suitable form of creating two DTOs would be to have one DTO contain data stored on the owner and their address and another which focused on the data associated with a vehicle, its tax information, and the corresponding payment. This saw the creation of "ownerDTO" and "taxDTO", respectively. "OwnerDTO" contained the first name, surname and entire address details of the owner, opting

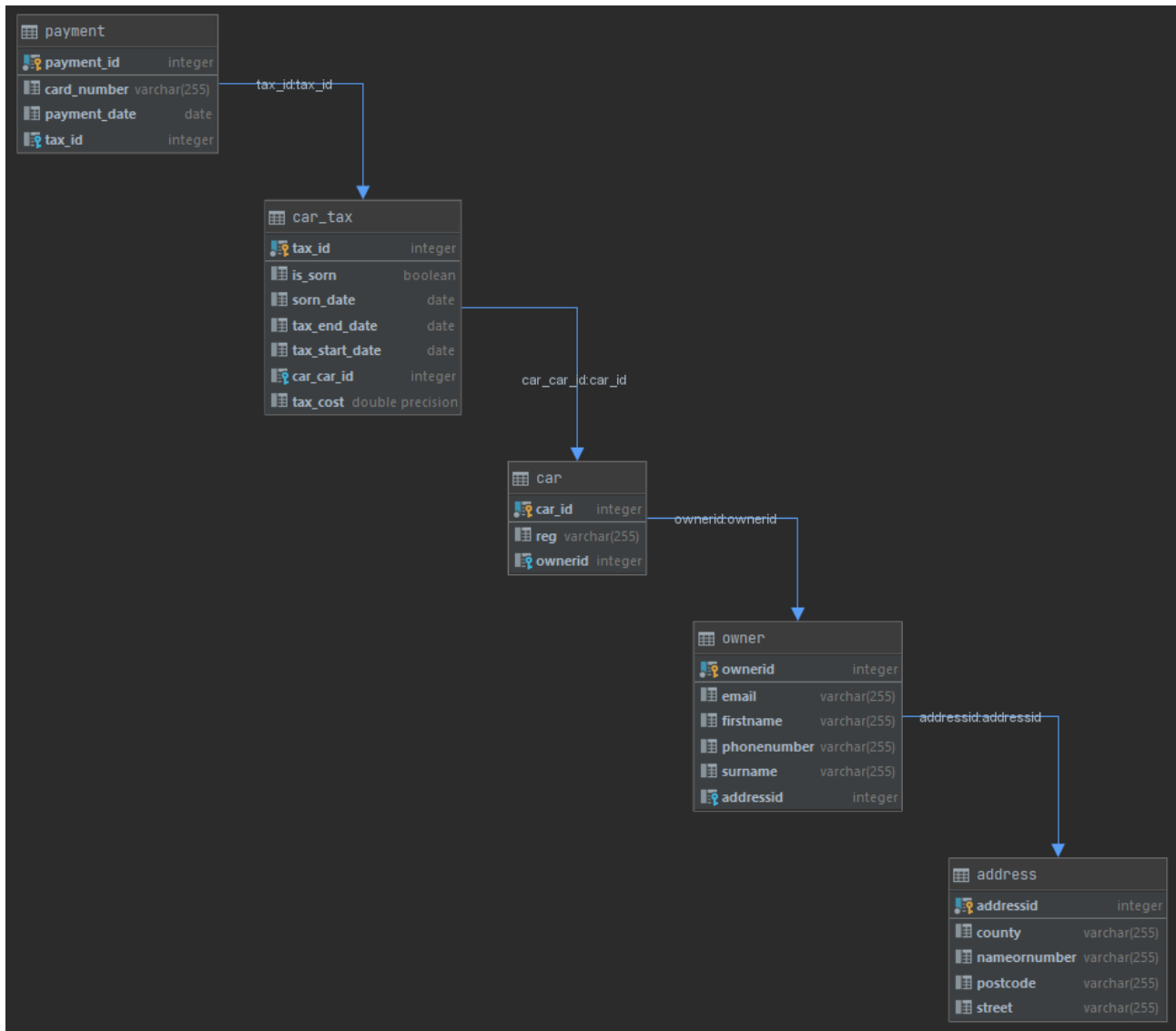


Figure 1: Tax Service Database Diagram

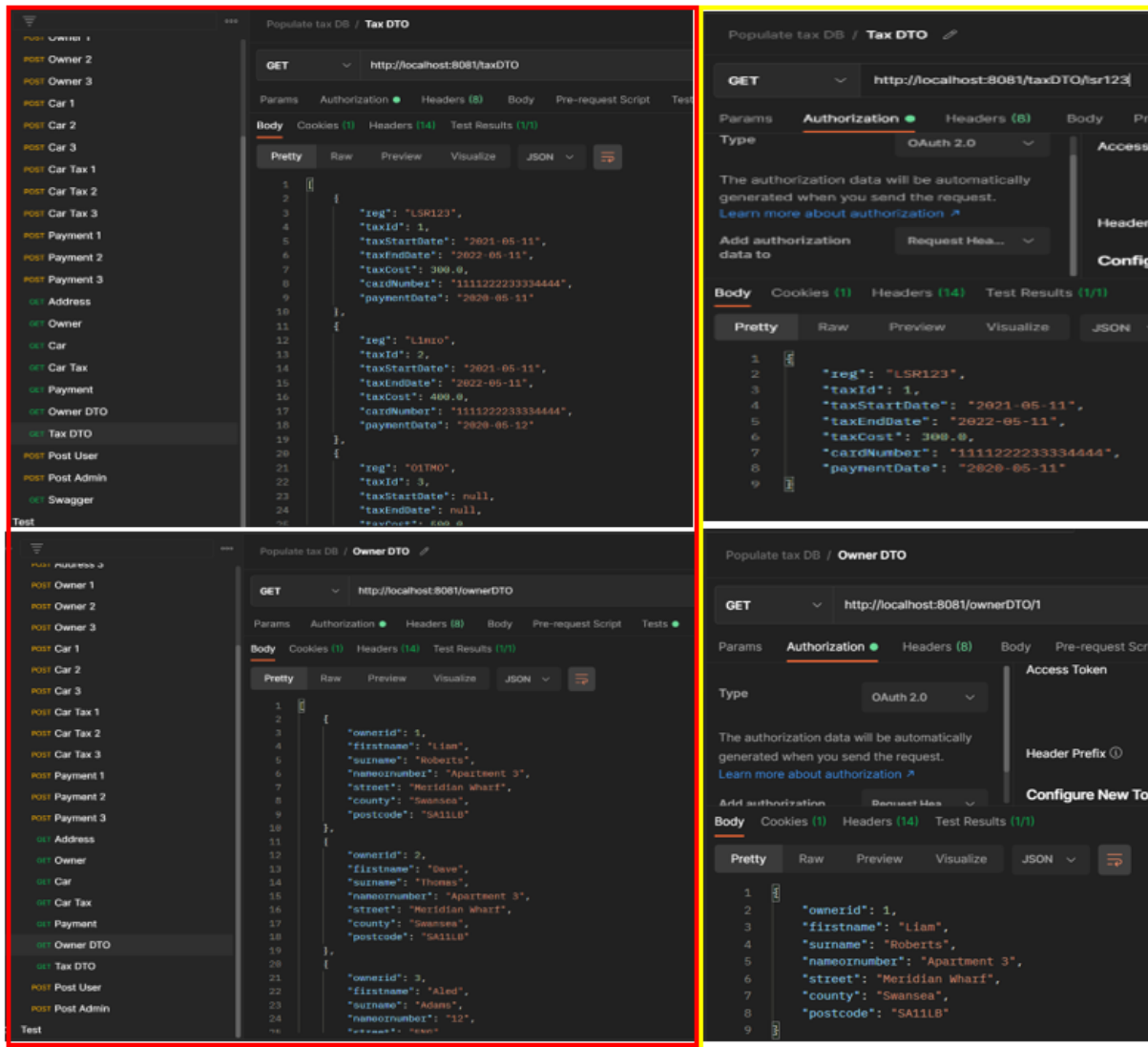


Figure 2: Response of Get Requests sent to each DTO

to omit email and phone number. "TaxDTO" included the vehicle registration, tax id, the start and end dates of its current tax period, tax cost, card number and payment date. Fig. 2 contains responses of get requests featuring the DTOs. In the red box are the get requests targeting the standard end points of the DTOs. In the yellow box are get requests which provide data in the end point. For example, the upper get request includes a specific registration as part of the URL which returns the DTO of that specific registration. The lower get request contains a specified user ID so that only that owner's details are returned. The ability to search via an owner ID or vehicle registration is functionality that a real world implementation of a similar service would require as standard.

2.3 Testing Coverage

To accomplish the suitable test coverage requirement, it was decided that tests should be focused on the get and post functions of each of the end points. Examples of these tests can be seen in Fig. 3. The first test simply tests that a user cannot input a string in place of an ID when forming a get request, checking that a BAD REQUEST response is returned. The second test checks that an OK response code is returned by a get request made to the end point. Finally, the third test checks that data can successfully be posted to the entity by ensuring the

```

@Test
public void TestIncorrectTypeAddressGet() throws UnirestException, URISyntaxException {
    //GET request
    URI uri = new URI ( GetURL ( _params: "address","hello" ));
    Unirest.setTimeouts( connectionTimeout: 0, socketTimeout: 0);
    com.mashape.unirest.http.HttpResponse<String> response = Unirest.get(String.valueOf(uri))
        .header( name: "Authorization", value: "Bearer "+JWTAccessToken)
        .header( name: "Content-Type", value: "application/json")
        .asString();

    //assertion
    Assert.assertEquals(HttpStatus.SC_BAD_REQUEST,response.getStatus());
}

```

1.

```

@Test
public void TestSuccessfulAddressGet() throws UnirestException, URISyntaxException {
    //GET request
    URI uri = new URI ( GetURL ( _params: "address" ));
    Unirest.setTimeouts( connectionTimeout: 0, socketTimeout: 0);
    com.mashape.unirest.http.HttpResponse<String> response = Unirest.get(String.valueOf(uri))
        .header( name: "Authorization", value: "Bearer "+JWTAccessToken)
        .header( name: "Content-Type", value: "application/json")
        .asString();

    //assertion
    Assert.assertEquals(HttpStatus.SC_OK,response.getStatus());
}

```

2.

```

@Test
@Order(4)
public void TestPostAddress() throws URISyntaxException {
    URI uri = new URI(GetURL(_params: "address"));
    Address address = new Address();
    address.setCounty("swansea");
    address.setPostcode("sa15la");
    address.setNameornumber("maelyn");
    address.setStreet("dreogen");
    HttpHeaders headers = new HttpHeaders();
    headers.set("X-COR-PERSIST","true");
    headers.add( headerName: "Authorization", headerValue: "Bearer "+JWTAccessToken);
    HttpEntity<Address> request = new HttpEntity<>(address, headers);
    ResponseEntity<String> response = restTemplate.postForEntity(uri, request, String.class);
    Assert.assertEquals(HttpStatus.SC_CREATED, response.getStatusCodeValue());

    //Test Post Data Landed
    URI uri2 = new URI(GetURL(_params: "address","1"));
    ResponseEntity<Address> response2 = restTemplate.exchange(uri2, HttpMethod.GET, new HttpEntity<>(headers), Address.class);
    Assert.assertEquals(HttpStatus.SC_OK,response2.getStatusCodeValue());
    Assert.assertEquals( expected: 1,response2.getBody().getAddressid());
    Assert.assertEquals( expected: "swansea", response2.getBody().getCounty());
    Assert.assertEquals( expected: "sa15la", response2.getBody().getPostcode());
    Assert.assertEquals( expected: "maelyn", response2.getBody().getNameornumber());
    Assert.assertEquals( expected: "dreogen", response2.getBody().getStreet());
}

```

3.

Figure 3: Examples of the Testing Coverage

CREATED response code is returned. A subsequent get request is made, with the body of the get request being compared to the posted values, ensuring the posted data landed correctly. Tests one and two are also used to test the DTOs end points.

2.4 Documentation

To fulfil the documentation requirement Swagger was used. The documentation can be found at the url "http://localhost:8081/swagger-ui/#/" when the service is running.

3 Security

Keycloak was used to provide OAuth2 authentication for the service. This was done by creating a realm with two roles, one called "TaxServiceUser", and one called "TaxServiceAdmin". Two additional users were created, named "admin" and "user". "admin" had all the available roles associated with it, whilst "user" had only the "TaxServiceUser" role. The REST service would

```

.antMatchers(HttpMethod.GET, ...antPatterns: "/").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/swagger-ui/*").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/swagger-resources/**").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/swagger-ui.html*").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/v2/api-docs*").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/webjars/**").permitAll()
.antMatchers(HttpMethod.GET, ...antPatterns: "/address/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/car/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/owner/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/payment/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/cartax/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/ownerDT0/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.GET, ...antPatterns: "/taxDT0/*").hasRole("TaxServiceUser")
.antMatchers(HttpMethod.POST, ...antPatterns: "/*").hasRole("TaxServiceAdmin")
.antMatchers(HttpMethod.PUT, ...antPatterns: "/*").hasRole("TaxServiceAdmin")
.antMatchers(HttpMethod.PATCH, ...antPatterns: "/*").hasRole("TaxServiceAdmin")
.antMatchers(HttpMethod.DELETE, ...antPatterns: "/*").hasRole("TaxServiceAdmin")

```

Figure 4: Code Snippet Showing the Roles Required for each Request

then use these roles to determine which user could access what. This can be seen in Fig. 4. To summarise Fig. 4, access to the documentation does not require authentication. GET requests made to the service require the "TaxServiceUser" role while any of the other requests require the "TaxServiceAdmin" role.

4 Docker Issues

Unfortunately, Dockerising the service proved problematic. It was easy to successfully create containers for each of the services that would allow them to run. The problems stem from the authorisation. It is possible to send a post request that returns a valid JSON web token, JWT. However there seems to be breakdown in communication between the service and Keycloak containers. Fig. 5 contains the investigation carried out into this issue. Image 1 of Fig. 5 shows each of the containers successfully created. Image 2 shows two attempts at communication between the web service container and the Keycloak container. It was possible to ping the Keycloak container from service container, however it was not possible to ping in the opposite direction. It makes sense that this would lead to UNAUTHORIZED response codes for requests with valid authentication as the web service needs Keycloak to provide it with what roles each user has.

```
Containers: [
  "Containers": [
    {
      "Name": "web-service-docker_postgres_1",
      "Name": "web-service-docker_postgres_1",
      "EndpointID": "658082387590064297a5f289d2250eb220f54e17b6f85d9fc40b46a45df79e9e",
      "MacAddress": "02:42:ac:14:00:04",
      "MacAddress": "02:42:ac:14:00:04",
      "IPV4Address": "172.20.0.4/16",
      "IPV4Address": "172.20.0.4/16",
      "IPV6Address": "",
      "IPV6Address": ""
    },
    {
      "Name": "web-service-docker_keycloak_1",
      "Name": "web-service-docker_keycloak_1",
      "EndpointID": "764204ca87c0ca031f42f16a824c540cabd77e96ae18a8e176470a5c5864f",
      "MacAddress": "02:42:ac:14:00:02",
      "MacAddress": "02:42:ac:14:00:02",
      "IPV4Address": "172.20.0.2/16",
      "IPV4Address": "172.20.0.2/16",
      "IPV6Address": "",
      "IPV6Address": ""
    },
    {
      "Name": "web-service-docker_webservice_1",
      "Name": "web-service-docker_webservice_1",
      "EndpointID": "e087c9562774a911e7105919737dac2cc509deb5f3c9552d006200755b231e0",
      "MacAddress": "02:42:ac:14:00:05",
      "MacAddress": "02:42:ac:14:00:05",
      "IPV4Address": "172.20.0.5/16",
      "IPV4Address": "172.20.0.5/16",
      "IPV6Address": "",
      "IPV6Address": ""
    },
    {
      "Name": "web-service-docker_adminer_1",
      "Name": "web-service-docker_adminer_1",
      "EndpointID": "71bae7be7ee3f77f408cf822cbfbb6721682d44eb0442d2b2002c597aed46839",
      "MacAddress": "02:42:ac:14:00:03",
      "MacAddress": "02:42:ac:14:00:03",
      "IPV4Address": "172.20.0.3/16",
      "IPV4Address": "172.20.0.3/16",
      "IPV6Address": "",
      "IPV6Address": ""
    }
  ]
}

C:\Windows\system32>C:\Windows\system32>docker container exec -it web-service-docker_webservice_1 ping web-service-docker_keycloak_1
PING web-service-docker_keycloak_1 (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.060 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.105 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.206 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.163 ms
64 bytes from 172.20.0.2: seq=4 ttl=64 time=0.274 ms
64 bytes from 172.20.0.2: seq=5 ttl=64 time=0.233 ms
^C
--- web-service-docker_keycloak_1 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss

C:\Windows\system32>C:\Windows\system32>docker container exec -it web-service-docker_keycloak_1 ping web-service-docker_webservice_1
OCI runtime exec failed: exec failed: container_linux.go:367: starting container process caused: exec: "ping": executable file not found in $PATH: unknown
```

Figure 5: Investigation into the Docker Issue

5 Conclusion

The design and implementation of the service is strong as it achieves all but one of the requirements. Keycloak was used to provide suitable user authentication whilst Swagger was used to generate documentation. The test coverage focused on get and post requests. The service had all the required entities and contained two DTOs. However, not being able to get the Keycloak container to communicate with the web service container tarnishes this implementation. It would also have been beneficial to increase the testing coverage to cover patch requests as patch requests are a common occurrence for a service of this kind.