

Web-Service Assignment 2

REST WEBSERVICE

NATHAN SATHANANDAN

1.0 Table Contents

Table of Contents

1.0 Table Contents	1
2.0 Table Figures	1
3.0 Introduction	2
4.0 Design	2
4.1 Entities	2
4.2 DTO's	3
4.3 Documentation	6
4.4 Security Configuration	7
4.5 Testing	9
4.6 Docker	10
5.0 Conclusion	11

2.0 Table Figures

Figure 4.1.1: Database entities	2
Figure 4.1.2: Code for entity relationships	3
Figure 4.2.1: OwnerDTO class	4
Figure 4.2.2: OwnerDetailsRespository class	4
Figure 4.2.3: OwnerDTOService class	5
Figure 4.2.4: TaxDTO class	5
Figure 4.2.5: TaxDetailsRespository class	6
Figure 4.2.6: TaxDTOService class	6
Figure 4.3.1: Swagger dependency	6
Figure 4.3.2: Swagger web config	7
Figure 4.3.3: Swagger Documentation	7
Figure 4.4.1: Realm role convertor	8
Figure 4.4.2: Key cloak configuration	8
Figure 4.5.1: functions to retrieve access tokens	9
Figure 4.5.2: Test post request	9
Figure 4.5.3: Test get request	10
Figure 4.6.1: Docker compose file	10
Figure 4.6.2: Docker file	11

3.0 Introduction

This report investigates the design and implementation of choices made for a backend restful web service using Spring Boot. The aim was to create a backend service for a road tax management system which contained a car registration, car owners, tax information and tax payment entities. As well as containing the entities the system needed to contain two DTO's, OpenAPI documentation, a security configuration, automated tests and a docker file which deployed the service.

4.0 Design

4.1 Entities

Within the Spring Boot application there was a total of 8 entities created, with two of them being DTO's, these entities were the Address, Car, Owner, OwnerDTO, Sorn, TaxDTO, TaxInfo and TaxPayment. Figure 4.1.1 below shows what the structure of the 6 main entities within the database.

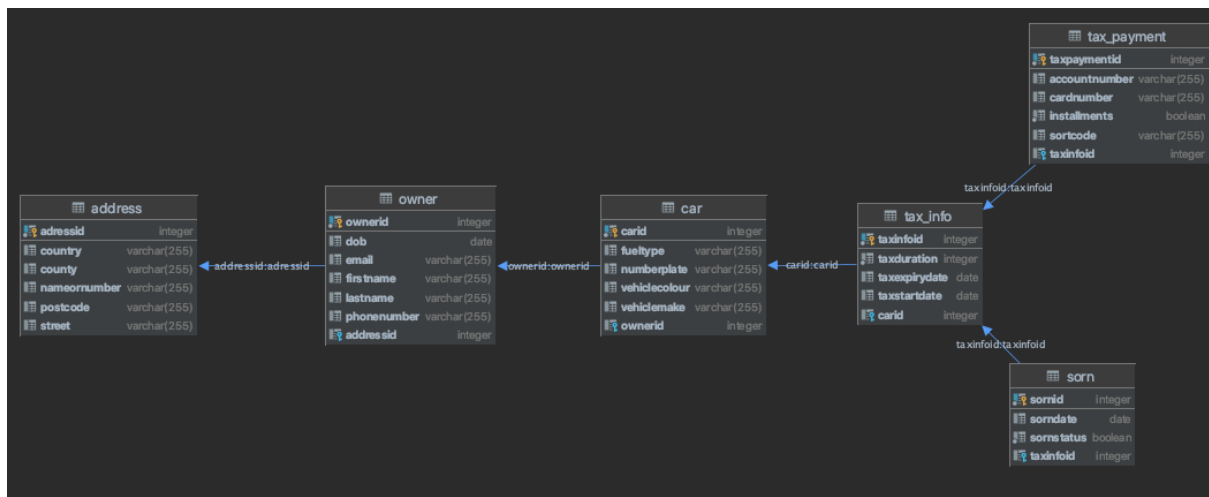


Figure 4.1.1: Database entities

The relationship between the address and owner was a one-to-many relationship because an address can have many people however a person can only have one address. The relationship between the car entity and the owner entity was a many-to-one relationship because a car can only have one owner, however, an owner can have multiple cars. The remaining entities all had one-to-one relationships because a car can only have one set of tax information and tax information can only have one set of sorn and tax payment information. Figure 4.1.2 below shows the code for the relationships in each class.

```

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class TaxInfo implements java.io.Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int taxinfoid;
    private int taxduration;
    private LocalDate taxstartdate;
    private LocalDate taxexpirydate;

    @OneToOne
    @JoinColumn(name = "carid", referencedColumnName = "carid")
    private Car carid;

    @OneToOne(mappedBy = "taxinfoid")
    private Sorn sornid;

    @OneToOne(mappedBy = "taxinfoid")
    private TaxPayment taxpaymentid;
}

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class Car implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int carid;
    private String numberplate;
    private String vehiclemake;
    private String vehiclecolour;
    private String fueltype;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ownerid")
    private Owner ownerid;

    @OneToOne(mappedBy = "carid")
    private TaxInfo taxinfoid;
}

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class TaxPayment implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int taxpaymentid;
    private boolean instalments;
    private String cardnumber;
    private String sortcode;
    private String accountnumber;

    @OneToOne
    @JoinColumn(name = "taxinfoid", referencedColumnName = "taxinfoid")
    private TaxInfo taxinfoid;
}

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class Sorn implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int sornid;
    private LocalDate sorndate;
    private boolean sornstatus;

    @OneToOne
    @JoinColumn(name = "taxinfoid", referencedColumnName = "taxinfoid")
    private TaxInfo taxinfoid;
}

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class Owner implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int ownerid;
    private String firstname;
    private String lastname;
    private LocalDate dob;
    private String phonenumber;
    private String email;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "addressid")
    private Address addressid;

    @OneToMany(fetch = FetchType.LAZY, targetEntity = Car.class, mappedBy = "ownerid")
    private List<Car> carid;
}

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class Address implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int addressid;
    private String nameornumber;
    private String street;
    private String postcode;
    private String county;
    private String country;

    @OneToMany(fetch = FetchType.LAZY, targetEntity = Owner.class, mappedBy = "addressid")
    private List<Owner> ownerid;
}

```

Figure 4.1.2: Code for entity relationships

4.2 DTO's

Within this service two DTO's were created, these were the OwnerDTO and the TaxDTO. The OwnerDTO allowed a user to grab all their details from the address entity, owner entity and car entity. This was created to allow users to quickly access any of their personal information rather than having to make 3 different get requests to the server to receive their personal information. Figure 4.2.1, 4.2.2 and 4.2.3 shows the code for the OwnerDTO class, OwnerDetailsRepository class and the OwnerDTOService class. The OwnerDTO also allowed the user to search for their owner information by putting in their owner id to give them instance results.

```

public class OwnerDTO {
    private int ownerId;
    private String firstname;
    private String lastname;
    private LocalDate dob;
    private String phonenumber;
    private String email;
    private String numberplate;
    private String nameornumber;
    private String street;
    private String postcode;
    private String county;
    private String country;
    private String vehiclemake;
    private String vehiclecolour;
    private String fueltype;

    public OwnerDTO() {
    }

    public OwnerDTO(String firstname, String lastname, LocalDate dob, String phonenumber, String email, String numberplate, String nameornumber, String street, String postcode, String county, String country, String vehiclemake, String vehiclecolour, String fueltype) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.dob = dob;
        this.phonenumber = phonenumber;
        this.email = email;
        this.numberplate = numberplate;
        this.nameornumber = nameornumber;
        this.street = street;
        this.postcode = postcode;
        this.county = county;
        this.country = country;
        this.vehiclemake = vehiclemake;
        this.vehiclecolour = vehiclecolour;
        this.fueltype = fueltype;
    }

    public OwnerDTO(String firstname, String lastname, String dob, String phonenumber, String email, String numberplate, String nameornumber, String street, String postcode, String county, String country, String vehiclemake, String vehiclecolour, String fueltype) {
        this(firstname, lastname, LocalDate.parse(dob), phonenumber, email, numberplate, nameornumber, street, postcode, county, country, vehiclemake, vehiclecolour, fueltype);
    }
}

```

Figure 4.2.1: OwnerDTO class

```

@Api(tags = "Address Entity")
@RestController
@CrossOrigin
public class OwnerDetailsRepository {

    @Autowired
    private OwnerDTOService ownerDTOService;

    @GetMapping("/ownerdetails")
    public List<OwnerDTO> getAllOwnerDetails() { return ownerDTOService.getAllOwnerDTO(); }

    @GetMapping("/ownerdetails/{id}")
    public OwnerDTO getOwnerById(@PathVariable("id") int id){
        OwnerDTO ownerDTO = ownerDTOService.getOwnerDTOById(id);
        if (ownerDTO != null){
            return ownerDTO;
        }
        throw new RuntimeException(HttpStatus.NOT_FOUND, "owner id not found");
    }
}

```

Figure 4.2.2: OwnerDetailsRepository class

```

@Service
public class OwnerDTOService {
    @Autowired
    CarRepository carRepository;

    private OwnerDTO convertToOwnerDTO(Car car){
        OwnerDTO ownerDTO = new OwnerDTO();
        ownerDTO.setOwnerId(car.getOwnerId().getOwnerId());
        ownerDTO.setNumberPlate(car.getNumberPlate());
        ownerDTO.setFirstname(car.getOwnerId().getFirstname());
        ownerDTO.setLastname(car.getOwnerId().getLastname());
        ownerDTO.setDob(car.getOwnerId().getDob());
        ownerDTO.setPhoneNumber(car.getOwnerId().getPhoneNumber());
        ownerDTO.setEmail(car.getOwnerId().getEmail());
        ownerDTO.setVehicleMake(car.getVehicleMake());
        ownerDTO.setVehicleColour(car.getVehicleColour());
        ownerDTO.setFuelType(car.getFuelType());
        ownerDTO.setNameOrNumber(car.getOwnerId().getAddress().getNameOrNumber());
        ownerDTO.setStreet(car.getOwnerId().getAddress().getStreet());
        ownerDTO.setPostcode(car.getOwnerId().getAddress().getPostcode());
        ownerDTO.setCountry(car.getOwnerId().getAddress().getCountry());
        ownerDTO.setCountry(car.getOwnerId().getAddress().getCountry());

        return ownerDTO;
    }

    public List<OwnerDTO> getAllOwnerDTO(){
        return ((List<Car>)carRepository.findAll()).stream().map(this::convertToOwnerDTO).collect(Collectors.toList());
    }

    public OwnerDTO getOwnerDTOByOwnerId(int id){
        List<OwnerDTO> ownerDTOList = getAllOwnerDTO();
        for (OwnerDTO ownerDTO : ownerDTOList){
            if (ownerDTO.getOwnerId() == id){
                return ownerDTO;
            }
        }
        return null;
    }
}

```

Figure 4.2.3: OwnerDTOService class

The second DTO within the Spring Boot application was the TaxDTO. This DTO allowed the user to access all the information regarding the tax of their vehicle by accessing the Owner entity, Car entity, TaxInfo entity and TaxPayment Entity. This was created to allow users to quickly access any of their tax related information rather than having to make multiple get requests to the server to receive their tax information. The TaxDTO also allowed the user to search for their tax information by putting in their owner id to give them instance results. Figure 4.2.4, 4.2.5 and 4.2.6 shows the code for the TaxDTO class, TaxDetailsRespository class and the TaxDTOService class.

```

public class TaxDTO {
    private int ownerId;
    private String numberplate;
    private int taxduration;
    private LocalDate taxstartdate;
    private LocalDate taxexpirydate;
    private boolean installments;

    public TaxDTO() {
    }

    public TaxDTO(String numberplate, int taxduration, LocalDate taxstartdate, LocalDate taxexpirydate, boolean installments) {
        this.numberplate = numberplate;
        this.taxduration = taxduration;
        this.taxstartdate = taxstartdate;
        this.taxexpirydate = taxexpirydate;
        this.installments = installments;
    }

    public TaxDTO(String numberplate, int taxduration, String taxstartdate, String taxexpirydate, boolean installments){
        this(numberplate, taxduration, LocalDate.parse(taxstartdate), LocalDate.parse(taxexpirydate), installments);
    }
}

```

Figure 4.2.4: TaxDTO class

```

@Api(tags = "Address Entity")
@RestController
@CrossOrigin
public class TaxDetailsRepository {
    @Autowired
    private TaxDTOService taxDTOService;

    @GetMapping("/taxdetails")
    public List<TaxDTO> getAllTaxDetails() { return taxDTOService.getAllTaxDTO(); }

    @GetMapping("/taxdetails/{id}")
    public TaxDTO getOwnerById(@PathVariable("id") int id){
        TaxDTO taxDTO = taxDTOService.getTaxDTOByTaxID(id);
        if (taxDTO != null){
            return taxDTO;
        }
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "owner id not found");
    }
}

```

Figure 4.2.5: TaxDetailsRepository class

```

@Service
public class TaxDTOService {
    @Autowired
    TaxPaymentRepository taxPaymentRepository;

    private TaxDTO convertToTaxDTO(TaxPayment taxPayment){
        TaxDTO taxDTO = new TaxDTO();
        taxDTO.setOwnerId(taxPayment.getTaxinfo().getCarid().getOwnerId().getOwnerId());
        taxDTO.setNumberplate(taxPayment.getTaxinfo().getCarid().getNumberplate());
        taxDTO.setTaxduration(taxPayment.getTaxinfo().getTaxduration());
        taxDTO.setInstallments(taxPayment.isInstallments());
        taxDTO.setTaxstartdate(taxPayment.getTaxinfo().getTaxstartdate());
        taxDTO.setTaxexpirydate(taxPayment.getTaxinfo().getTaxexpirydate());
        return taxDTO;
    }

    public List<TaxDTO> getAllTaxDTO(){
        return ((List<TaxPayment>)taxPaymentRepository.findAll()).stream().map(this::convertToTaxDTO).collect(Collectors.toList());
    }

    public TaxDTO getTaxDTOByTaxID(int id){
        List<TaxDTO> taxDTOList = getAllTaxDTO();
        for (TaxDTO taxDTO: taxDTOList){
            if (taxDTO.getOwnerId() == id){
                return taxDTO;
            }
        }
        return null;
    }
}

```

Figure 4.2.6: TaxDTOService class

4.3 Documentation

The car tax management application used what is known as swagger to generate OpenAPI documentation. Swagger was implemented by adding a dependency to the maven file and by also adding a swagger config class to the application as well. Both the dependency and the config file can be seen in figures 4.3.1 and 4.3.2 below.

```

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-boot-starter</artifactId>
  <version>3.0.0</version>
</dependency>

```

Figure 4.3.1: Swagger dependency

```

@Configuration
public class SwaggerConfig {
    @Bean
    public Docket api(){
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}

```

Figure 4.3.2: Swagger web config

When the application is running, the documentation is accessible through the <http://localhost:8080/swagger-ui/#/> URL. Figure 4.3.3 below shows the webpage of the swagger documentation. This documentation shows what can requests are available for each entity so that people know what requests can be made to the backend service.

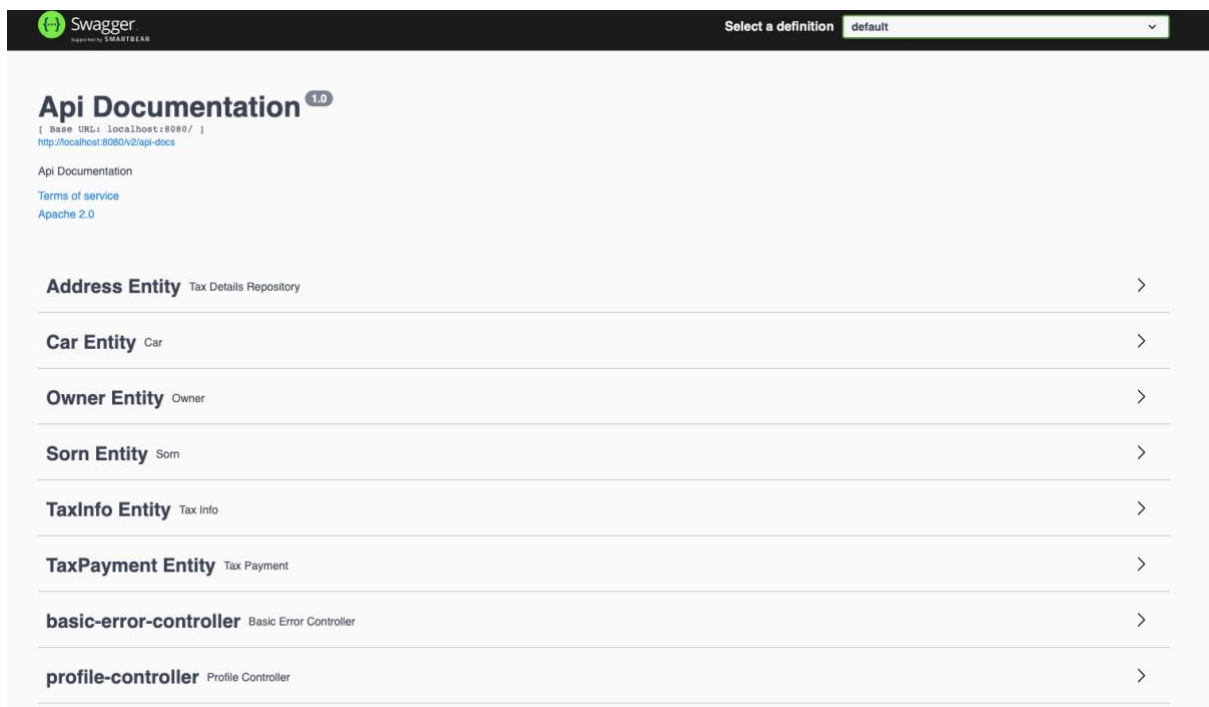


Figure 4.3.3: Swagger Documentation

4.4 Security Configuration

The security for the application was handled by key clock. Within key cloak two roles were created, the admin user role and the basic user role. These roles meant that basic users couldn't make any post, patch, or delete requests to the backend application and that admin users could make any type of request. Within the application there are two classes, one class takes a response token and uses it to extract the available roles on key cloak. The other class is used to determine what endpoints each role has access. The two classes can be seen in Figures 4.4.1 and 4.4.2 below.

```

public class KeycloakRealmRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {
    @Override
    public Collection<GrantedAuthority> convert(Jwt jwt) {
        final Map<String, Object> realmAccess = (Map<String, Object>) jwt.getClaims().get("realm_access");
        return ((List<String>) realmAccess.get("roles")).stream()
            .map(roleName -> "ROLE_" + roleName)
            .map(SimpleGrantedAuthority::new)
            .collect(Collectors.toList());
    }
}

```

Figure 4.4.1: Realm role convertor

```

@Configuration
@EnableWebSecurity
public class CarTaxManagementSecurityConfig extends WebSecurityConfigurerAdapter {

    String jwkSetUri = "http://localhost:8090/auth/realms/VehicleTaxManagementRealm/protocol/openid-connect/certs";

    private Converter<Jwt, ? extends AbstractAuthenticationToken> jwtAuthenticationConverter(){
        JwtAuthenticationConverter jwtConverter = new JwtAuthenticationConverter();
        jwtConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRealmRoleConverter());
        return jwtConverter;
    }

    @Bean
    JwtDecoder jwtDecoder(){
        return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri).build();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().authorizeRequests()
            .antMatchers(HttpMethod.GET, "/").permitAll()
            .antMatchers(HttpMethod.GET, "/v2/api-docs").permitAll()
            .antMatchers(HttpMethod.GET, "/swagger-resources").permitAll()
            .antMatchers(HttpMethod.GET, "/swagger-resources/**").permitAll()
            .antMatchers(HttpMethod.GET, "/v3/api-docs/**").permitAll()
            .antMatchers(HttpMethod.GET, "/configuration/ui").permitAll()
            .antMatchers(HttpMethod.GET, "/configuration/security").permitAll()
            .antMatchers(HttpMethod.GET, "/swagger-ui.html").permitAll()
            .antMatchers(HttpMethod.GET, "/swagger-ui/*").permitAll()
            .antMatchers(HttpMethod.GET, "/webjars/**").permitAll()
            .antMatchers(HttpMethod.GET, "/address/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.GET, "/sorn/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.GET, "/owner/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.GET, "/taxinfo/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.GET, "/taxpayment/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.GET, "/profile/*").hasRole("taxmanagementuser")
            .antMatchers(HttpMethod.POST, "/**").hasRole("taxmanagementadmin")
            .antMatchers(HttpMethod.PUT, "/**").hasRole("taxmanagementadmin")
            .antMatchers(HttpMethod.PATCH, "/**").hasRole("taxmanagementadmin")
            .antMatchers(HttpMethod.DELETE, "/**").hasRole("taxmanagementadmin")
            .anyRequest().authenticated().and().oauth2ResourceServer().jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()));
    }
}

```

Figure 4.4.2: Key cloak configuration

4.5 Testing

Spring Boot is a very robust framework and handles problems very well, therefore the JUnit tests created for the application test the correct authorization, post, and get http codes are returned when making requests. However, to make the tests work the access token needed to be retrieved from the key cloak server so that all the requests were authorised to have access to the application. Figure 4.5.1 below shows the code that was implemented to receive the access tokens for the basic and admin users.

```
public String RetrieveAdminJwtToken() throws UnirestException {
    HttpResponse<String> response = Unirest.post( url: "http://localhost:8080/auth/realms/VehicleTaxManagementRealm/protocol/openid-connect/token") .HttpRequestWithBody()
        .header( name: "Content-Type", value: "application/x-www-form-urlencoded")
        .header( name: "Cookie", value: "JSESSIONID=EA37A32E1060B7684C51EF2E082B4391.v0019481; JSESSIONID=406C6853E3769CF5E41C3912C1895528")
        .field( name: "client_id", value: "tax-management") .MultipartBody()
        .field( name: "username", value: "adminuser")
        .field( name: "password", value: "123456")
        .field( name: "grant_type", value: "password")
        .asString();
    String[] responseInfo = response.getBody().split( regex: "\\n");
    String JwtAccessToken = responseInfo[3];
    return JwtAccessToken;
}

public String RetrieveBasicUserJwtToken() throws UnirestException {
    HttpResponse<String> response = Unirest.post( url: "http://localhost:8080/auth/realms/VehicleTaxManagementRealm/protocol/openid-connect/token") .HttpRequestWithBody()
        .header( name: "Content-Type", value: "application/x-www-form-urlencoded")
        .header( name: "Cookie", value: "JSESSIONID=EA37A32E1060B7684C51EF2E082B4391.v0019481; JSESSIONID=406C6853E3769CF5E41C3912C1895528")
        .field( name: "client_id", value: "tax-management") .MultipartBody()
        .field( name: "username", value: "basicuser")
        .field( name: "password", value: "123456")
        .field( name: "grant_type", value: "password")
        .asString();
    String[] responseInfo = response.getBody().split( regex: "\\n");
    String JwtAccessToken = responseInfo[3];
    return JwtAccessToken;
}
```

Figure 4.5.1: functions to retrieve access tokens

The tests in the application were also ran in a specific order because before a get request can be tested that data needs to already exist, therefore this means that the post request need to be ran first. Figures 4.5.2 and 4.5.3 below show examples of a get and post request.

```
@Test
@Order(1)
public void TestAddressTypePostSuccessfulHttpCode() throws URISyntaxException, UnirestException {

    URI uri = new URI(GetURL( ...params: "address"));

    Address address = new Address();
    address.setCountry("England");
    address.setCounty("Staines");
    address.setPostcode("TW197AF");
    address.setNameornumber("34");
    address.setStreet("Osborne Avenue");

    HttpHeaders headers = new HttpHeaders();
    headers.set("X-COM-PERSIST", "true");
    headers.add( headerName: "Authorization", headerValue: "Bearer "+RetrieveAdminJwtToken());

    HttpEntity<Address> request = new HttpEntity<>(address, headers);
    ResponseEntity<String> response = restTemplate.postForEntity(uri, request, String.class);

    Assert.assertEquals(HttpStatus.SC_CREATED, response.getStatusCodeValue());
}
```

Figure 4.5.2: Test post request

```

@Test
@Order(7)
public void TestAddressTypeGetSuccessfulData() throws URISyntaxException, UnirestException {
    URI uri = new URI(GetURL(...params: "address", "1"));

    HttpHeaders headers = new HttpHeaders();
    headers.set("X-COM-PERSIST", "true");
    headers.add(headerName: "Authorization", headerValue: "Bearer "+RetrieveAdminJwtToken());

    ResponseEntity<Address> response = restTemplate.exchange(uri, HttpMethod.GET, new HttpEntity<>(headers), Address.class);
    Assert.assertEquals(HttpStatus.SC_OK, response.getStatusCodeValue());
    Assert.assertEquals(expected: "England", response.getBody().getCountry());
    Assert.assertEquals(expected: "Staines", response.getBody().getCounty());
    Assert.assertEquals(expected: "TW197AF", response.getBody().getPostcode());
    Assert.assertEquals(expected: "34", response.getBody().getNameornumber());
    Assert.assertEquals(expected: "Osborne Avenue", response.getBody().getStreet());
}

```

Figure 4.5.3: Test get request

4.6 Docker

Within the application there is docker file and a docker compose file which is used to run the jar file of the tax management application. Figures 4.6.1 and 4.6.2 below show what the docker file and docker compose file contain.

```

version: "3.6"
services:
  postgres:
    container_name: "POSTGRES_DB"
    image: postgres:latest
    restart: always
    ports:
      - "1000:5432"
    environment:
      POSTGRES_USER: "docker"
      POSTGRES_PASSWORD: "password"
      POSTGRES_DB: "car_tax_management"
    networks:
      - dbnetwork
  adminer:
    image: adminer
    restart: always
    ports:
      - "8080:8081"
    networks:
      - dbnetwork
  keycloak:
    image: quay.io/keycloak/keycloak:latest
    environment:
      - DB_VENDOR=H2
      - KEYCLOAK_USER=docker
      - KEYCLOAK_PASSWORD=docker
    ports:
      - "8090:8080"
    restart:
      always
    networks:
      - dbnetwork
  webservice:
    container_name: springwebservice
    build:
      context: .
      dockerfile: Dockerfile
      target: webservice
    image: node-b-webservice
    restart: always
    ports:
      - "8070:8080"
    networks:
      - dbnetwork
    volumes:
      - ./springwebservice:/var
    depends_on:
      - keycloak
    networks:
      dbnetwork:

```

Figure 4.6.1: Docker compose file

```

#### SHARED DOCKER STAGES #####

# The OS setup
FROM ubuntu as developer-os
MAINTAINER Nathan <2012819@student.uwtsd.ac.uk>
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update --fix-missing && \
apt-get install -y software-properties-common && \
apt-get install -y --no-install-recommends apt-utils && \
apt-get install -y curl \
wget
RUN echo "export SERVER_IP_ADDRESS='0.0.0.0'" >> /etc/profile
RUN apt-get clean

#### THE WEB SERVICE #####

FROM openjdk:8-jdk-alpine as web service
Maintainer Nathan <2012819@student.uwtsd.ac.uk>
Expose 8080
WORKDIR /var
ENTRYPOINT ["java", "-jar", "car_tax_management.jar"]

```

Figure 4.6.2: Docker file

Whilst the docker compose sets up the service and the key cloak server there was a problem when communicating between the service container and the key cloak server. This issue meant that when running the service, the access token provided by the key cloak server doesn't authenticate any of the requests to the spring service however when the services are run locally the authentication works.

5.0 Conclusion

Overall, the application meets the requirements set out by the brief with the exception to the docker file not working with the spring and key cloak containers not communicating properly to authenticate requests.