Building an API

Introduction

I had returned to the reference data team, for whom a brand new service was on the horizon. They had already built and were now maintaining Professional Reference and User Profile micro-services that I was fairly familiar with from my previous work with the team. This new service would be similar to an extent, in that it will manage the profiles of public sector workers, but this time focusing on employees within a very specific field, and would deal with information particular to those in this field. I was keen to get involved at the ground level to see how a micro-service comes together. I asked our Business Analyst a lot of questions throughout the requirements gathering process, and while there was a lot that was yet to be finalised, we were able to benefit as a team from our BA's helpful explanations of what we knew so far. The benefit proved mutual, as this also created a conversation around technical requirements, or information that would impact technical requirements, which our BA could pursue wherever there were gaps. I requested the chance to build an API for the service when the opportunity arose- and was given the first one. As of time of writing this ticket is ongoing, but it has already proved such a valuable learning experience, I have chosen to document it here.

The Development Process

The first step was to familiarise myself with the requirements specific to my ticket. Details as always could be found in Jira and Confluence. Figure 1 shows the user story for my ticket.

```
1  As as user with the appopriate rights
2  I want to be able to retrieve list of all [redacted] Roles
3  so that I can look for [redacted] users matching specific roles and allocate work accordingly
```

Figure 1 – User story.

In essence this would be a simple GET endpoint, requiring no input, which would yield a list of role types. These role types comprise the entire contents of a particular table in the database, and each role type has an ID, an English language description and a Welsh language description. Details of the API structure were also provided (figure 2) including the path which would be useful for request mapping and the structure of the response body, which here would be an array of JSON objects.

API	Response Body

GET:
/refdata/v1/[redacted]/roles
	[
	{
	"roleID": "string",
	"roleDescEn": "string",
	"roleDescCy": "string"
	},
	{
	"roleID": "string",
	"roleDescEn": "string",
	"roleDescCy": "string"
	}
	]

Figure 2 – API structure.

A GitHub repository had already been set up. I spoke to our Scrum Master to arrange access for me, and once that was done I was able to make a remote branch named after the Ticket number and clone the repository on my local machine. Looking through, I could see this was a fairly empty shell. A template of a microservice, with just a skeleton folder structure (figure 3).

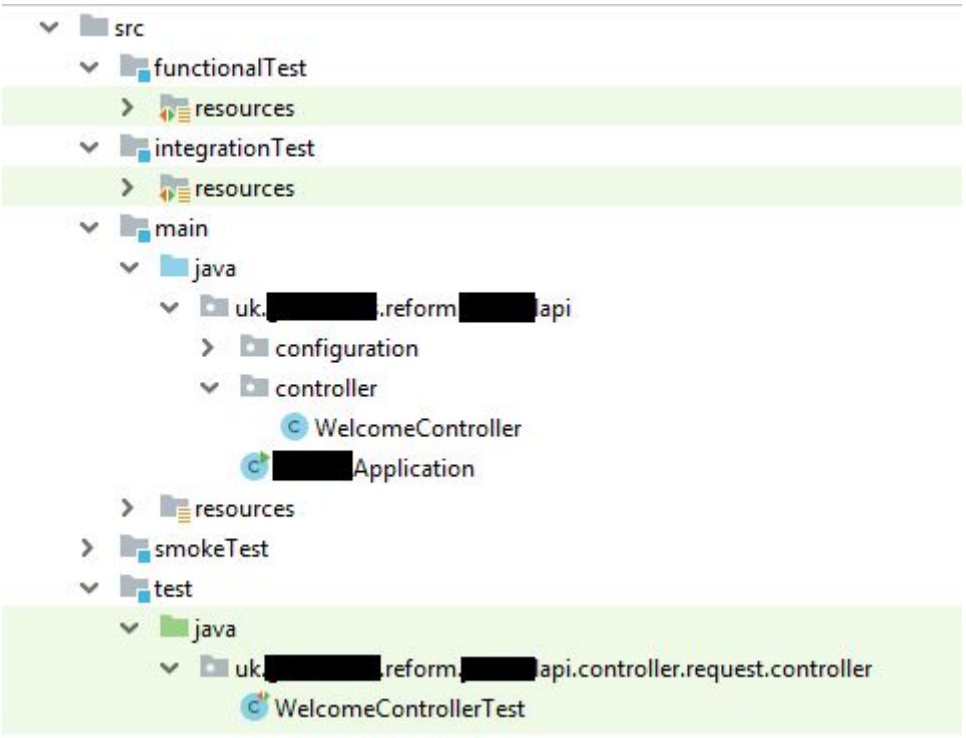


Figure 3 – Skeleton folder structure.

One of the first things to input would be the database SQL. There was, in fact, another repository related to this service that would take care of data loading- taking data from the legacy system and using it to populate the database that had been designed for this. A senior developer advised that it wouldn't do to have the database exist in both repositories, but to set it up in my microservice for now to have something to build on- and later questions around this aspect of the architecture would be more thoroughly discussed and designs finalised.

I spoke to the colleague who had created the database SQL, and he kindly sent me a link to the JIRA where the SQL file had been uploaded. Figure 4 shows a snippet of it.

V1_1__init_tables.sql

DB Connections

```
    CONSTRAINT jud_auth_pk PRIMARY KEY ("[redacted]_office_auth_Id")
     WITH (FILLFACTOR = 10),
    CONSTRAINT jud_auth_jur_unique UNIQUE (jurisdiction_id)
     WITH (FILLFACTOR = 10)

);
CREATE TABLE public.authorisation_type(
    "authorisation_Id" varchar(64) NOT NULL,
    authorisation_desc_en varchar(256) NOT NULL,
    authorisation_desc_cy varchar(256),
    "jurisdiction_Id" varchar(64),
    jurisdiction_desc_en varchar(256),
    jurisdiction_desc_cy varchar(256),
    CONSTRAINT "authorisation_Id" PRIMARY KEY ("authorisation_Id")

);

CREATE TABLE public.[redacted]_role_type(
    role_id varchar(64) NOT NULL,
    role_desc_en varchar(256) NOT NULL,
    role_desc_cy varchar(256),
    CONSTRAINT role_id PRIMARY KEY (role_id)
```

Figure 4 – Database SQL snippet.

Before rushing to begin building the endpoint functionality, I wanted to make sure the database was in working order. In GitBash, I connected to the database and listed the tables (figure 5).

```
764108@LTGBO6897 MINGW64 ~/Documents/JRD
$ winpty docker exec -it rd-        -db psql -U postgres

psql (9.6.16)
Type "help" for help.

postgres=#
postgres=# \c dbjuddata
You are now connected to database "dbjuddata" as user "postgres".
dbjuddata=# \dt
                  List of relations
 Schema |             Name              | Type  |   Owner
--------+-------------------------------+-------+-----------
 public | authorisation_type            | table | dbjuddata
 public | base_location_type            | table | dbjuddata
 public | contract_type                 | table | dbjuddata
 public | flyway_schema_history         | table | dbjuddata
 public |         _office_appointment   | table | dbjuddata
 public |         _office_authorisation | table | dbjuddata
 public |         _role_type            | table | dbjuddata
 public |         _user_profile         | table | dbjuddata
 public | region_type                   | table | dbjuddata
(9 rows)
```

Figure 5 – Connecting to the database.

I then ran a couple of simple CRUD statements to create a series of rows in the table my ticket concerns itself with, and then to read the contents of the table afterwards (figure 6).

```
dbjuddata=# INSERT INTO _       _role_type (role_id, role_desc_en) VALUES
dbjuddata-# ('1', '          '),
dbjuddata-# ('2', 'Advisory Committee Member -          '),
dbjuddata-# ('3', 'Advisory Committee Member - Non      _       :'),
dbjuddata-# ('4', 'Chairman of the Advisory Committee'),
dbjuddata-# ('5', 'MAGS - AC Admin User'),
dbjuddata-# ('6', 'Acting President'),
dbjuddata-# ('7', '          ');
INSERT 0 7
dbjuddata=# select * from         _role_type;
 role_id |               role_desc_en                | role_desc_cy
---------+--------------------------------------------+--------------
 1       |                                            |
 2       | Advisory Committee Member -                |
 3       | Advisory Committee Member - Non            |
 4       | Chairman of the Advisory Committee         |
 5       | MAGS - AC Admin User                       |
 6       | Acting President                           |
 7       |                                            |
(7 rows)
```

Figure 6 – Insert into and then select all from the roles table.

Satisfied that all was well with the database, I could move on to developing my endpoint. A colleague advised that the approach generally taken to TDD here deals with functional tests first, then the functionality, then integration tests, then unit tests. I therefore followed this path outlined for me.

Functional tests

As figure 3 showed, the functional test directory sat completely empty, so the entire functional test suite required setting up. I created sub directories and packages according to convention established in existing services, with

configuration components in their own package, the idam (identity access management) client and its attendant classes in a separate package, and the S2S and API clients also in their own appropriately named package (figure 7).

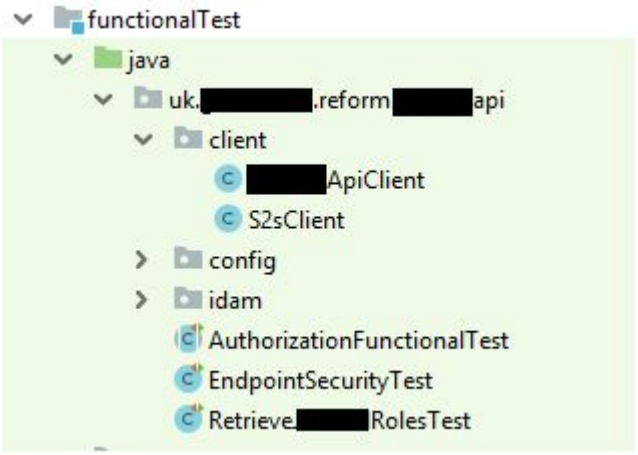


Figure 7 – Functional test suite folder structure after set up.

This follows the pattern set by other services. The S2sClient class contains a method with sends the microservice name, and a time-based one time password (TOTP) generated by google authenticator, to the s2s service which then returns a JSON web token (JWT) to attach to request headers for the sake of service to service authentication. This is very much like the process described in detail in the Demo UI project. The idam package contains functionality which similarly interacts with the Identity Access Management service to return bearer tokens. These are useful when it comes to authenticating users, and for my purposes, ensuring access to an endpoint for users with only certain roles. There's a lot in this test suite already, but for brevity I will focus on the ApiClient class, the AuthorizationFunctionalTest abstract class and the RetrieveRoles class.

AuthorizationFunctionalTest is the parent class that all subsequent test classes (including my own RetrieveRoles) extend. It contains a selection of reusable declared variables, each given the protected access modifier so that they can be used throughout, but only within, the functional test suite. It also contains a 'setUp()' method with a @Before annotation which ensures that the method runs prior to the remainder of the class, and of course child classes. Within this 'setUp()' method (figure 8) an ApiClient object is instantiated, it's constructor taking a url, an s2s token and an idam object as parameters- and this is then stored in a variable for later use.

```
IdamOpenIdClient idamOpenIdClient = new IdamOpenIdClient(configProperties);
log.info("idamOpenIdClient: " + idamOpenIdClient);

String s2sToken = new S2sClient(s2sUrl, s2sName, s2sSecret).signIntoS2S();

[redacted]ApiClient = new [redacted]ApiClient([redacted]ApiUrl, s2sToken, idamOpenIdClient);
```

Figure 8 – Inside the AuthorizationFunctionalTest class' setup method.

The ApiClient class contains a couple of useful reusable methods. First withUnauthenticatedRequest() returns a request specification built with a series of methods provided by the Serenity framework (figure 9). It specifies relaxed https validation so that all hosts are trusted, sets a base URI and adds a couple of headers.

```
60      private RequestSpecification withUnauthenticatedRequest() {
61          return SerenityRest.given()
62                  .relaxedHTTPSValidation()
63                  .baseUri([redacted]ApiUrl)
64                  .header( headerName: "Content-Type", APPLICATION_JSON_UTF8_VALUE)
65                  .header( headerName: "Accepts", APPLICATION_JSON_UTF8_VALUE);
66      }
```

Figure 9 – A flexible and reusable request specification.

Then retrieveAll[...]Roles() builds on the aforementioned to carry out a GET request on the path I'm working on, storing the response in an aptly named variable, then asserting that the response has the expected http status code (which is provided as an argument) before returning the response body (figure 10).

```
public Map<String, Object> retrieveAll[redacted]Roles(String roleOfAccessor, HttpStatus expectedStatus) {
    Response response = withUnauthenticatedRequest()
            .get( path: "/refdata/v1/[redacted]/roles")
            .andReturn();

    response.then()
            .assertThat()
            .statusCode(expectedStatus.value());

    return response.body().as(Map.class);
}
```

Figure 10 – A method for testing the functionality I will build.

The syntax here mirrors the acceptance criteria, with the request specification being the Given, the first half of the method in figure 10 being the When, and the remainder being Then.

My test class, which inherits this method, then calls on it to test the sunny day scenario and rainy day scenario respectively (figures 11 and 12).

```
@Test
public void rdcc_739_ac1_user_with_appropriate_rights_can_retrieve_list_of_all_[redacted]_roles() {
    //Given I am a user with the appropriate rights to retrieve list of all [redacted] Roles
    //When I search for the list of all [redacted] Roles
    //Then I should be able to see the list of [redacted] roles
    Map<String, Object> [redacted]RoleTypesResponse = [redacted]ApiClient.retrieveAll[redacted]Roles(caseworker, HttpStatus.OK);
    List<HashMap> [redacted]Roles = (List<HashMap>) [redacted]RoleTypesResponse.get("[redacted]Roles");
    [redacted]Roles.stream().forEach(role -> {
        assertThat( actual: "roleId").isNotNull();
        if (role.get("roleId").equals("1")) {
            assertThat(role.get("roleDescEn")).isEqualTo("[redacted]");
        } else if (role.get("roleId").equals("2")) {
            assertThat(role.get("roleDescEn")).isEqualTo("Advisory Committee Member - [redacted]");
        }
    });
```

Figure 11 – Test for the positive.

```
@Test
public void rdcc_739_ac2_user_without_appropriate_rights_cannot_retrieve_list_of_all_[redacted]_roles() {
    //Given I am a user without the appropriate rights to retrieve list of all [redacted] Roles
    //When I search for the list of all [redacted] Roles
    //Then I should get a failure outcome as I do not have the appropriate rights for accessing role information
    Map<String, Object> response = [redacted]ApiClient.retrieveAll[redacted]Roles(puiOrgManager, HttpStatus.FORBIDDEN);
    log.info("response::::::" + response);
    assertThat(response.get("errorMessage")).isNotNull();
    assertThat(response.get("errorMessage")).isEqualTo("9 : Access Denied");
}
```

Figure 12 – Test for the negative.

As per convention, the test name includes the Jira number, acceptance criteria number and a descriptive name. The acceptance criteria is included as comments as well.

Being that this is TDD, these tests are expected to fail at this point (see figures 13 and 14).

```
TEST STARTED: rdcc_739_ac1_user_with_appropriate_rights_can_retrieve_list_of_all_[redacted]_roles
------------------------------------------------------------------------
2020-01-18 10:24:04.476  INFO   --- [    Test worker] net.serenitybdd.core.Serenity           :
2020-01-18 10:24:04.487  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest   :
2020-01-18 10:24:04.488  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest   :
2020-01-18 10:24:04.489  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest   :

1 expectation failed.
Expected status code <200> but was <404>.
```

Figure 13 – Test for the positive failure result.

```
TEST STARTED: rdcc_739_ac2_user_without_appropriate_rights_cannot_retrieve_list_of_all_[redacted]_roles
------------------------------------------------------------
2020-01-18 10:23:49.567  INFO   --- [    Test worker] net.serenitybdd.core.Serenity           : TEST
2020-01-18 10:23:54.134  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest    : Confi
2020-01-18 10:23:54.136  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest    : Confi
2020-01-18 10:23:54.136  INFO   --- [    Test worker] u.g.h.r.j.AuthorizationFunctionalTest    : Confi

1 expectation failed.
Expected status code <403> but was <404>.
```

Figure 14 – Test for the negative failure result.

The functionality

Turning my attention next to the functionality, I started by creating a domain package and within it defining a JPA entity. This class represents the data that can be persisted to a table in the database- any one instance of this class would represent a row. I ended up creating entities to represent all of the tables in the database to be able to account for the relationships between them, but here I will focus on the entity that relates to my functionality (figure 15).

```
16  @Entity(name = "[redacted]_role_type")
17  @NoArgsConstructor
18  @Getter
19  @Component
20  public class [redacted]RoleType {
21
22      @Id
23      @Size(max = 64)
24      private String roleId;
25
26      @Column(name = "ROLE_DESC_EN")
27      @Size(max = 256)
28      private String roleDescEn;
29
30      @Column(name = "ROLE_DESC_CY")
31      @Size(max = 256)
32      private String roleDescCy;
33
34      @OneToMany(mappedBy = "[redacted]RoleType")
35      private List<[redacted]OfficeAppointment> [redacted]OfficeAppointments = new ArrayList<>();
36
37      public [redacted]RoleType(String roleId, String roleDescEn, String roleDescCy) {
38          this.roleId = roleId;
39          this.roleDescEn = roleDescEn;
40          this.roleDescCy = roleDescCy;
41      }
```

Figure 15 – Role types entity.

On line 16 the @Entity annotation defines this class as an entity so that JPA (Java Persistence API) is aware of it. I've given it the same name as the table it represents. Below that are a couple of Lombok annotations that serve in lieu of boilerplate code. The @Id annotation defines the primary key, the @Column annotation provides details of the table column these variables correspond to- in this case the name, and @Size specifies the maximum length of the field. The last field, OfficeAppointments, represents another entity that this entity has a one to many relationship to, as the annotation implies. The 'mapped by' attribute establishes a bidirectional association between these entities.

The response class (figure 16) looks fairly similar to the entity class, but it is not part of the domain package and concerns itself with a lot less. It simply represents the response object. The @JsonProperty annotation defines the property used in serialization and deserialization of JSON.

```java
@Getter
@AllArgsConstructor
public class [redacted]RoleTypeResponse {

    @JsonProperty
    private String roleId;
    @JsonProperty
    private String roleDescEn;
    @JsonProperty
    private String roleDescCy;

    public [redacted]RoleTypeResponse([redacted]RoleType [redacted]RoleType) {
        this.roleId = [redacted]RoleType.getRoleId();
        this.roleDescEn = [redacted]RoleType.getRoleDescEn();
        this.roleDescCy = [redacted]RoleType.getRoleDescCy();
    }
```

Figure 16 – Role type response class.

Following this I set up the persistence package and created a RoleTypeRepository interface which extends the JPA specific repository interface (figure 17). This is a fairly simple set up owing to Spring Data's work behind the scenes rendering boilerplate DAO implementation unnecessary. It is given the entity type and the primary key type, and a @Repository annotation. As this was so simple, I went ahead and set up repository interfaces for each of the remaining domain entities as well.

```java
@Repository
public interface [redacted]RoleTypeRepository extends JpaRepository<[redacted]RoleType, String> {
}
```

Figure 17 – Repository interface.

Next was the service layer and the controller- I'll describe the latter first. All services being built for our client make use of Swagger, a tool that auto-generates documentation for RESTful APIs. In the controller class it is now standard practise to include Swagger annotations that declare and manipulate the output of the documentation (figure 18).

```java
@RequestMapping(path = "refdata/v1/[redacted]")
@RestController
@Slf4j
@NoArgsConstructor
public class [redacted]Controller {

    @Autowired
    protected [redacted]RoleTypeService [redacted]RoleTypeService;

    @ApiOperation(
            value = "Retrieves all [redacted] roles"
    )

    @ApiResponses({
            @ApiResponse(
                    code = 200,
                    message = "List of [redacted] role types",
                    response = [redacted]RoleTypeEntityResponse.class
            ),
            @ApiResponse(
                    code = 403,
                    message = "Forbidden Error: Access denied"
            ),
            @ApiResponse(
                    code = 500,
                    message = "Server Error",
                    response = String.class
            )
```

Figure 18 – Swagger annotations.

A note on the response type declared in the success @ApiResponse. It had originally been a 'List.class' as the method in the controller and in the service implementation were returning a java list of type RoleTypeResponse. However, while this worked correctly I received a review comment on my pull request from a colleague (figure 19) asking for it to be amended.

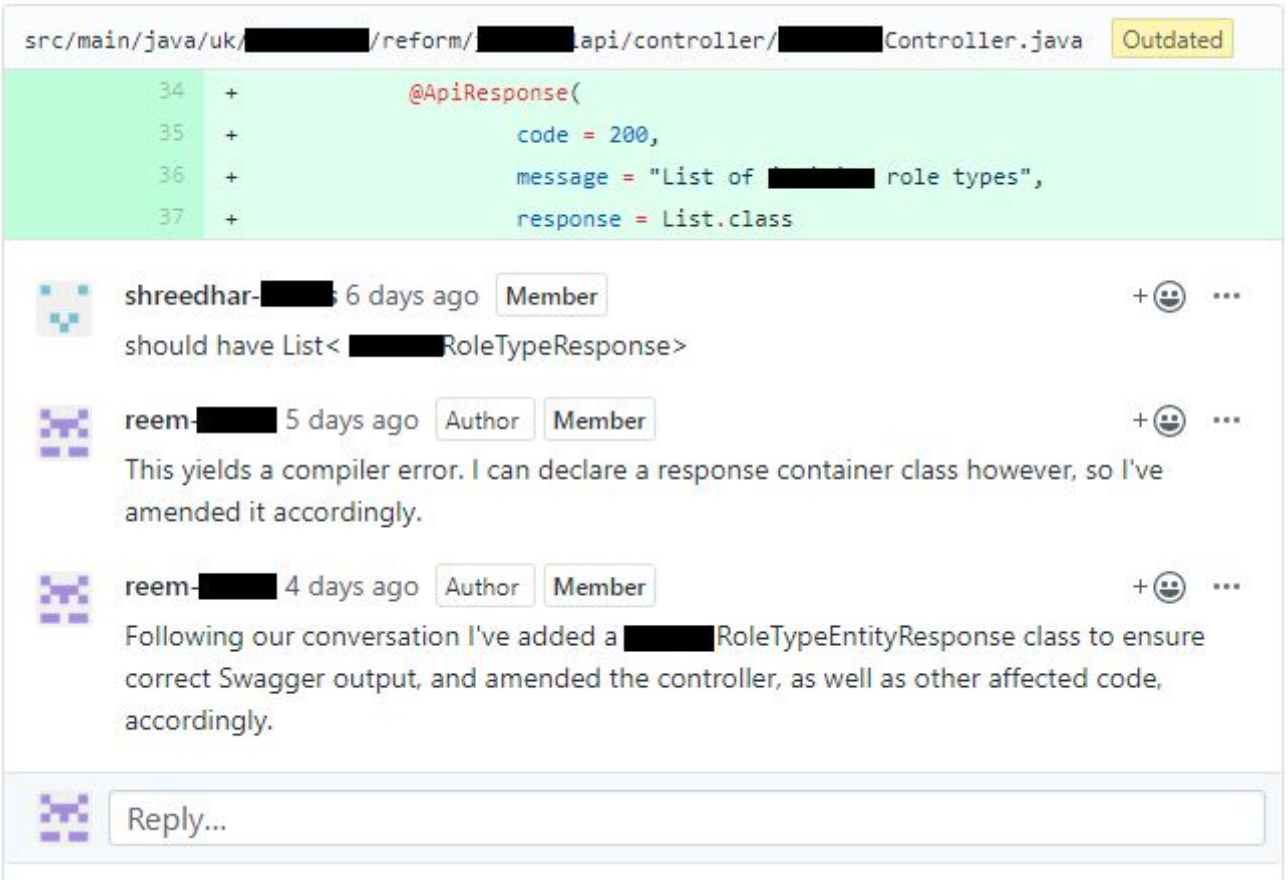


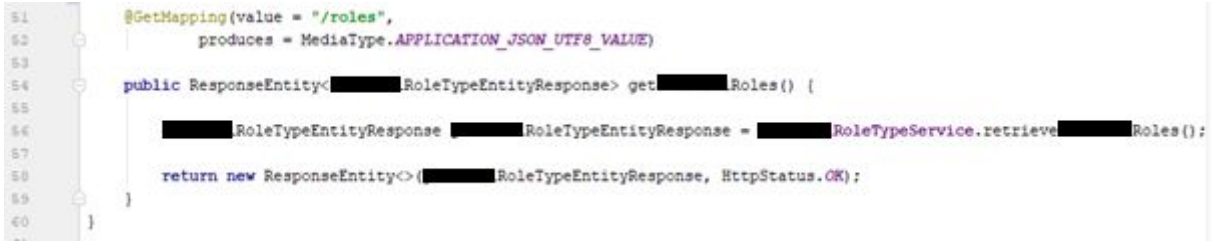
Figure 19 – Pull request review comment.

We had a conversation about it and it turned out that this issue with returning lists of response types had surfaced before. Swagger apparently doesn't display them correctly, and so they had worked around it by creating a sort of response wrapper class that returned a list, so I followed suit (figure 20).

```
public class [redacted]RoleTypeEntityResponse {

    private List<[redacted]RoleTypeResponse> [redacted]RoleTypeResponses;

    public [redacted]RoleTypeEntityResponse(List<[redacted]RoleType> [redacted]RoleTypes) {
        this.[redacted]RoleTypeResponses = [redacted]RoleTypes.stream()
                .map([redacted]RoleType -> new [redacted].RoleTypeResponse([redacted]RoleType))
                .collect(Collectors.toList());
    }

    @JsonGetter("judicialRoles")
    public List<[redacted]RoleTypeResponse> get[redacted]RoleTypes() { return [redacted]RoleTypeResponses; }
}
```

Figure 20 – This response class creates a list of role type responses.

Back to the controller, after the Swagger annotations, there's a @GetMapping annotation (figure 21) which here specifies in its attributes the path, beyond the base URI on line 19 (figure 18), and the content produced defaults to JSON.

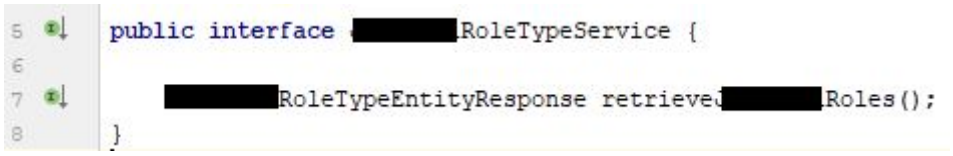


```
@GetMapping(value = "/roles",
        produces = MediaType.APPLICATION_JSON_UTF8_VALUE)

public ResponseEntity<[redacted]RoleTypeEntityResponse> get[redacted]Roles() {

    [redacted]RoleTypeEntityResponse [redacted]RoleTypeEntityResponse = [redacted]RoleTypeService.retrieve[redacted]Roles();

    return new ResponseEntity<>([redacted]RoleTypeEntityResponse, HttpStatus.OK);
}
}
```

Figure 21 – Controller method.

Following this is a method to get all the roles which returns a response entity- the type being the response class I'd recently added. In the body of the method the logic is delegated to the service and the returned value is stored in a variable is of type RoleTypeEntityResponse.

The service layer consists of two parts: an interface and an implementation class that extends this interface (figures 22 and 23).



```
public interface [redacted]RoleTypeService {

    [redacted]RoleTypeEntityResponse retrieve[redacted]Roles();
}
```

Figure 22 – Role type service interface.

```
@Service
@Slf4j
public class [redacted]RoleTypeServiceImpl implements [redacted]RoleTypeService {

    @Autowired
    [redacted]RoleTypeRepository [redacted]RoleTypeRepository;

    public [redacted]RoleTypeEntityResponse retrieve[redacted]Roles() {
        List<[redacted]RoleType> [redacted]RoleTypes = [redacted]RoleTypeRepository.findAll();

        if ([redacted]RoleTypes.isEmpty()) {
            throw new ResourceNotFoundException("4 : Resource not found");
        }

        return new [redacted]RoleTypeEntityResponse([redacted]RoleTypes);
    }

}
```

Figure 23 – Role type service implementation.

Having a service layer that the controller delegates to for all its business logic needs is an example of the separation of concerns design principle. This way the controller need only handle requests, the persistence layer need only concern itself with the repository, and the service layer can handle any logic that may have to occur in between. This also makes for better security as it adds an additional barrier to accessing the database.

In the service implementation here, the retrieve roles method calls the findAll() method which the jpa repository interface provides, and stores the returned values in a variable. A conditional then checks if the variable is empty, and, if so, throws a resource not found exception. Otherwise the code moves on to return a new RoleTypeEntityResponse containing the returned values.

The functional tests again

Having completed the functionality I ran my functional tests again and had a realisation. The role types table currently sat empty. And unlike the Professional Reference service where a test organisation could be created and persisted ahead of testing it's retrieval by whatever search input- the role types table should be pre-populated. I therefore needed to find a way to load test data ahead of the functional tests.

I wrote some simple CRUD statements. A couple of lines inserting rows into the table (figure 24), and a line to delete them afterwards (figure 25), and saved both in files that I put in the resources directory (figure 26).

```
INSERT INTO [redacted]_role_type (role_id, role_desc_en, role_desc_cy) VALUES ('1', [redacted], 'test');

INSERT INTO [redacted]_role_type (role_id, role_desc_en, role_desc_cy) VALUES ('2', 'Advisory Committee Member -[redacted]', 'test');
```

Figure 24 – Creating rows in the table.

```
DELETE FROM [redacted]_role_type WHERE role_desc_cy = 'test';
```

Figure 25 – Deleting rows in the table.

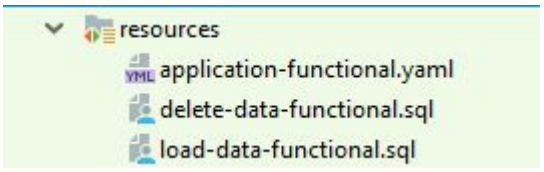


Figure 26 – The resources directory.

In the AuthorizationFunctionalTest class I added some helpful methods that would be inherited into my test class (figure 27).

```
84      protected static void executeScript(List<Path> scriptFiles) throws SQLException, IOException {
85
86          Log.info("environment script execution started::");
87          try (Connection connection = createDataSource().getConnection()) {
88              try (Statement statement = connection.createStatement()) {
89                  for (Path path : scriptFiles) {
90                      for (String scriptLine : Files.readAllLines(path)) {
91                          statement.addBatch(scriptLine);
92                      }
93                      statement.executeBatch();
94                  }
95              }
96          } catch (Exception exe) {
97              Log.error("FunctionalTestSuite script execution error with script ::" + exe.toString());
98              throw exe;
99          }
100         Log.info("environment script execution completed::");
101     }
102
103     private static DataSource createDataSource() {
104
105         Log.info("DB Host name::" + getValueOrDefault( name: "POSTGRES_HOST",  defaultValue: "localhost"));
106         PGSimpleDataSource dataSource = new PGSimpleDataSource();
107         dataSource.setServerName(getValueOrDefault( name: "POSTGRES_HOST",  defaultValue: "localhost"));
108         dataSource.setPortNumber(Integer.parseInt(getValueOrDefault( name: "POSTGRES_PORT",  defaultValue: "5456")));
109         dataSource.setDatabaseName(getValueOrDefault( name: "POSTGRES_DATABASE", defaultValue: "dbjuddata"));
110         dataSource.setUser("dbjuddata");
111         dataSource.setPassword("dbjuddata");
112         return dataSource;
113     }
```

Figure 27 – methods for setting up test data.

The createDataSource() sets up a connection to the database, and executeScript() method reads the given sql file. In my retrieve roles test class I then made use of these in dbSetup() and dbTearDown() methods (fiures 28 an 29) that execute at the beginning and end of the class' run.

```
22  @RunWith(SpringIntegrationSerenityRunner.class)
23  @ActiveProfiles("functional")
24  @Slf4j
25  public class Retrieve[redacted]RolesTest extends AuthorizationFunctionalTest {
26
27      @BeforeClass
28      public static void dbSetup() throws Exception {
29          String loadFile = ResourceUtils.getFile( resourceLocation: "classpath:load-data-functional.sql").getCanonicalPath();
30          executeScript(ImmutableList.of(Paths.get(loadFile)));
31      }
```

Figure 28 – Loading test data into the database ahead of test execution.

```
@AfterClass
public static void dbTearDown() throws Exception {
    String deleteFile = ResourceUtils.getFile( resourceLocation: "classpath:delete-data-functional.sql").getCanonicalPath();
    executeScript(ImmutableList.of(Paths.get(deleteFile)));
}
```

Figure 29 – Deleting the test data after test execution.

Now when I run the test class, the first test passes and the second fails (figure 30). This meets my expectations. Both return a 200 OK http status because the roles are accessible. The second test should return 403 Forbidden, but securing the endpoint so that it is accessible only to users with certain roles has not been implemented at this stage.

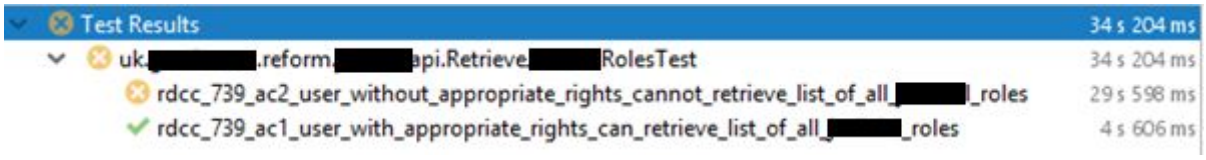


Figure 30 – Functional test results after test data loading set up.

The integration tests

The integration test set up is quite similar to that of the functional tests. There is an abstract parent class that all integration test classes will inherit from, and in that class is a method setting up a connection to the database (figure 31). I used the same SQL file to load the data, putting a copy of it in the resources directory of the integration module (I also later changed the name from functional to integration for good naming's sake).

```
@Before
public void init() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setUrl(url);
    dataSource.setUsername(userName);
    dataSource.setPassword(password);
    // schema init
    Resource initSchema = new ClassPathResource("load-data-functional.sql");
    DatabasePopulator databasePopulator = new ResourceDatabasePopulator(initSchema);
    DatabasePopulatorUtils.execute(databasePopulator, dataSource);
}
```

Figure 31 – Setting up database connection in integration test parent class.

There is also an ApiClient class containing methods such as the below (figure 32).

```
public Map<String,Object> retrieveAll[redacted]RoleTypes(String role) {
    return getRequest( uriPath: APP_BASE_PATH + "/roles", role);
}

/rawtypes, unchecked/
private Map<String, Object> getRequest(String uriPath,String role, Object... params) {

    ResponseEntity<Map> responseEntity;

    try {
        HttpEntity<?> request = new HttpEntity<>(getMultipleAuthHeaders(role));
        responseEntity = restTemplate
                .exchange( url: "http://localhost:" + jrdApiPort + uriPath,
                        HttpMethod.GET,
                        request,
                        Map.class,
                        params);
    } catch (HttpStatusCodeException ex) {
        HashMap<String, Object> statusAndBody = new HashMap<>( initialCapacity: 2);
        statusAndBody.put("http_status", String.valueOf(ex.getRawStatusCode()));
        statusAndBody.put("response_body", ex.getResponseBodyAsString());
        return statusAndBody;
    }
```

Figure 32 – A getRequest() method and a retrieve all role types methods which builds on this.

The getRequest() method here makes use of an instance of RestTemplate (a Synchronous client that performs HTTP requests) to make a GET request to the provided URI and receive a response. Then provided no exceptions occur, a response body and http status are returned. The retrieve all roles method on line 40 builds on this, adding the remainder of the path for this particular endpoint.

The retrieve all roles method is then used in the child test class to test for the sunny day and rainy day scenarios in a fairly similar way to the functional tests (figure 33).

```
@Slf4j
public class Retrieve[redacted]RolesTypesTest extends AuthorizationEnabledIntegrationTest {

    @Test
    public void user_with_caseworker_role_can_retrieve_[redacted]_role_types() {
        Map<String, Object> response = [redacted]ReferenceDataClient.retrieveAll[redacted]RoleTypes(caseworker);
        assertThat(response.get("http_status")).isEqualTo("200 OK");
    }

    //Awaiting complete SIDS and IDAM setup in order to include this test
    public void user_with_non_caseworker_role_cannot_retrieve_judicial_role_types() {
        Map<String, Object> response = [redacted]ReferenceDataClient.retrieveAll[redacted]RoleTypes(puiOrgManager);
        log.info("response::::::" + response);
        assertThat(response.get("http_status")).isEqualTo("403");
    }
}
```

Figure 33 – My integration test class.

The unit tests

For each class I'd created in pursuit of setting up the functionality I wrote a corresponding unit test class. To be concise I'll focus on only the controller unit test here. It's fairly representative as they all use the same frameworks, Junit and Mockito, but it's also perhaps one of the more interesting ones. Figure 34 shows the set up preceding the test, and figure 35 shows the unit test itself.

```
@Slf4j
public class [redacted]ControllerUnitTest {

    @InjectMocks
    private [redacted]Controller [redacted]Controller;

    private [redacted]RoleType [redacted]RoleType1;
    private [redacted]RoleType [redacted]RoleType2;
    private [redacted]RoleTypeService [redacted]RoleTypeServiceMock;
    private [redacted]RoleTypeEntityResponse [redacted]RoleTypeEntityResponse;
    private List<[redacted]RoleType> [redacted]RoleTypeList;

    @Before
    public void setUp() throws Exception {
        [redacted]RoleTypeServiceMock = mock([redacted]RoleTypeService.class);

        [redacted]RoleType1 = new [redacted]RoleType( roleId: "1",  roleDescEn: "testEn",  roleDescCy: "testCy");
        [redacted]RoleType2 = new [redacted]RoleType( roleId: "2",  roleDescEn: "testEn2",  roleDescCy: "testCy2");

        [redacted]RoleTypeList = new ArrayList<>();
        [redacted]RoleTypeList.add([redacted]RoleType1);
        [redacted]RoleTypeList.add([redacted]RoleType2);

        [redacted]RoleTypeEntityResponse = new [redacted]RoleTypeEntityResponse([redacted]RoleTypeList);

        MockitoAnnotations.initMocks( testClass: this);
    }
```

Figure 34 – Unit test set up for the controller class.

First, before even the setUp() method, the @InjectMocks annotation creates an instance of the class under test and injects mocked dependencies, and some variables are declared just below. Then in the setUp() method, a list of roles is created and given as a parameter to a new RoleTypeEnitityResponse object, and the service mock is initialised.

```
@Test
public void testRetrieve[redacted]RoleTypes() {

    final HttpStatus expectedHttpStatus = HttpStatus.OK;

    when([redacted]RoleTypeServiceMock.retrieve[redacted]Roles()).thenReturn([redacted]RoleTypeEntityResponse);

    ResponseEntity<[redacted]RoleTypeEntityResponse> actual = [redacted]Controller.get[redacted]Roles();

    verify([redacted]RoleTypeServiceMock, times( wantedNumberOfInvocations: 1)).retrieve[redacted]Roles();

    assertThat(actual.getStatusCode()).isEqualTo(expectedHttpStatus);
    assertThat(actual.getBody().get[redacted]RoleTypes().toString())
            .contains([redacted]RoleType1.getRoleId(), [redacted]RoleType1.getRoleDescEn(), [redacted]RoleType1.getRoleDescCy());
    assertThat(actual.getBody().get[redacted]RoleTypes().toString())
            .contains([redacted]RoleType2.getRoleId(), [redacted]RoleType2.getRoleDescEn(), [redacted]RoleType2.getRoleDescCy());
}
}
```

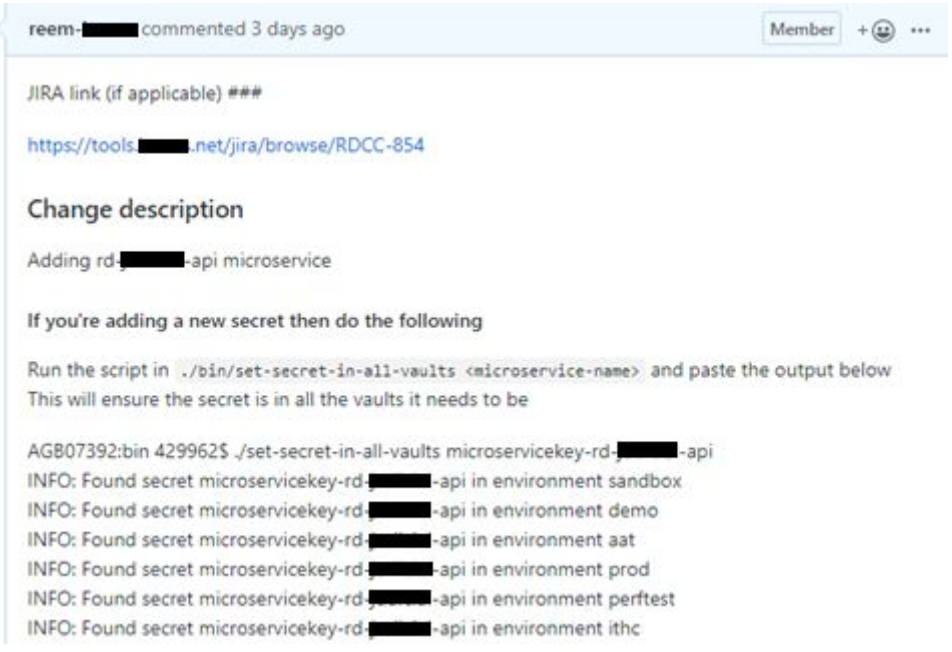
Figure 35 – Unit test for the controller class.

In the test itself, an expected http status variable is declared which will later be used to assert the success of the controller method call. Then on line 53 the behaviour of the service mock is configured using the 'when' and 'thenReturn' combination of Mockito methods. On line 55 the controller method is executed and the return value stored in a variable named 'actual'. Following this is a series of assertions on expectations. First verifying that the controller method call led to the mock service being called on. Then asserting that the http status code of 'actual' is indeed 200 OK. Finally asserting that 'actual' contains the role details I expect to be there.

The build pipeline, S2S and IDAM

Having set up the functionality and the tests, I arrived at a point where I needed to turn my attention towards more general set up for this new microservice. S2S authentication has to be set up in order for this service to be able to interact with other services. Then integration with IDAM so that we can authenticate users by obtaining ID tokens. This would allow me to complete the outstanding part of the functionality that would secure the controller method to be accessible only for users with specific roles. And of course the Jenkins build must be set up. All of this is ongoing as of time of writing, but I will mention here some of what has been accomplished so far.

In order to set up S2S, I cloned the repository for Service Auth Provider, which is an application used to authenticate all services built for the client. In the main.tf file I added a key value pair: the key being the name of the service, the value being the name of the service's Azure Key Vault secret. The secret is a BASE32-encoded sequence of ten random bytes that Azure uses for validating one-time passwords (in the Angular/Express/Node.js project I used an environment variable that represented this key). I then added the same key value pair to the bootstrap.yaml file, following the syntax there, and added the secret key name to the values.yaml file as well. After bumping the Helm chart version I could push my changes and make a pull request (figure 36).



reem-[redacted] commented 3 days ago — Member

JIRA link (if applicable) ###

https://tools.[redacted].net/jira/browse/RDCC-854

Change description

Adding rd-[redacted]-api microservice

If you're adding a new secret then do the following

Run the script in `./bin/set-secret-in-all-vaults <microservice-name>` and paste the output below
This will ensure the secret is in all the vaults it needs to be

```
AGB07392:bin 429962$ ./set-secret-in-all-vaults microservicekey-rd-[redacted]-api
INFO: Found secret microservicekey-rd-[redacted]-api in environment sandbox
INFO: Found secret microservicekey-rd-[redacted]-api in environment demo
INFO: Found secret microservicekey-rd-[redacted]-api in environment aat
INFO: Found secret microservicekey-rd-[redacted]-api in environment prod
INFO: Found secret microservicekey-rd-[redacted]-api in environment perftest
INFO: Found secret microservicekey-rd-[redacted]-api in environment ithc
```

Figure 36 – Pull request and script for setting secret.

The pull request reminded me to run a provided script in Azure-cli to write the secret into all the vaults. The script then checks if it can find the secret in the vaults.

As this is open source software the key vault stores secrets separate from the codebase, and you can put secrets in Azure via the script or the Azure dashboard. All our services use the same vaults- they're split by environment, and stored as key value pairs: secret name and secret value.

When starting up the application, in local or whatever environment, there is configuration (Helm charts and bootstrap.yaml in our own codebase) that retrieves the keys from the vault, and assigns them to an environment variable. The application then uses these environment variable to reference the secrets. These secrets include the PostgreSQL database' password, the app insights key (monitoring tool in azure), as well as the S2S key for the application (figure 37).

```
keyVaults:
  rd:
    secrets:
      - [redacted]-api-POSTGRES-PASS
      - s2s-secret
      - AppInsightsInstrumentationKey
```

Figure 37 – The values.yaml in our codebase is configured to retrieve secrets from the Azure vault.

I also needed to make changes in and a pull request for another service called cnp-flux-config. This one has a quite confusing Readme, but luckily there's an established pattern with previous commits by other new services and with a little research I was able to get along fine.

```
119     k8s/**/common/rd/professional-api.yaml @[redacted]/ref-data
120     k8s/**/common/rd/profile-sync.yaml @[redacted]/ref-data
121     k8s/**/common/rd/user-profile-api.yaml @[redacted]/ref-data
122   + k8s/aat/common/rd/[redacted]-api.yaml @[redacted]/ref-data
123
```

Figure 38 – CODEOWNERS file in cnp-flux-config.

The line added to the CODEOWNERS file (figure 38) essentially gives the ref data team privileges to make changes to the kubernetes configuration for the new service.

```
14  +   chart:
15  +     repository: https://[redacted]public.azurecr.io/helm/v1/repo/
16  +     name: rd-[redacted]-api
17  +     version: 0.0.2
18  +   values:
19  +     java:
20  +       replicas: 2
21  +       useInterpodAntiAffinity: true
22  +       image: [redacted]public.azurecr.io/rd/[redacted]-api:latest
23  +     global:
24  +       environment: aat
25  +       tenantId: "531ff96d-0ae9-462a-8d2d-bec7c0b42082"
26  +       enableKeyVaults: true
```

Figure 39 – Snippet of the yaml file added to cnp-flux-config.

I then needed to add a yaml file in a directory containing my teams other services. The yaml file contains the kubernetes configuration itself. This represents an area of learning for me, but I can see that line 22 refers to the service's docker container, and line 24 is the environment that the kubernetes cluster should be created in.

This brings me to the present moment. Code additions in our own codebase are also required, as well as the Helm charts, such as security configuration classes and changes to the Jenkinsfile_cnp. And I imagine there'll be further interaction with the dev ops and platform engineering teams, as well as the IDAM team. It's all very new right now, but I'm sure by the close of this ticket I will benefit from a much broader understanding of the client's project.

Reflective Statement

Though this ticket is not yet finished, it has already proved to be one of the most interesting, and certainly the most learning intensive, project I've worked on. There's a satisfaction to writing an entire endpoint and TDDing it, rather than testing after the fact to shore up code coverage. This project has helped me fill in some of the gaps in my understanding of how the whole thing works – providing me with the holistic view of things that can only be acquired from personally coding all three (controller, service, persistence) layers, and all three types of tests. And then getting involved with infrastructure set up to boot. The process of delivering this ticket, and researching various aspects of the requisite work has sparked an interest in enterprise application architecture which I intend to pursue in my own time.

I currently find the dev ops and cloud side of things incredibly opaque. This is perhaps not helped by the lack of ownership over that aspect of the work. You don't set things up from scratch here, rather you input lines of code modelling the syntax on an established pattern or set of Readme instructions. I suppose having these things be handled by separate specialist teams is to be expected on such a large project, and this is understandably for the better. However it does prompt me to consider spending some time on self-led learning in this area.

Finally while this project is ongoing, I can already see the benefit this work brings to the client. The new service and its CI build pipeline will be set up, with one of the first endpoints requested ready to expose for consumption whenever this service is scheduled to go live. The database is set up with indexes for the tables to enable faster data retrieval. S2S will also be set up enabling interaction between this service and others including IDAM. And finally I've committed to creating documentation on the entire microservice set up process as it turns out knowledge of the various steps is uneven throughout the team, and no such documentation exists in Confluence. Once this is done, I hope that my team and others working for this client might benefit from it.