Writing Functional Tests

Introduction

Our client requested that all teams adopt the Serenity framework for functional tests. From the client perspective this would yield a number of benefits. The framework produces illustrative test reports that serve technical and non-technical team members alike. Also this would ensure consistent reporting across all projects being developed for the client. Moreover it equips projects with the ability to potentially adopt a Behaviour Driven Development (BDD) approach- a natural extension of the Test driven approach already in use- moving forward.

For my team, after the process of integrating the Serenity framework into our project was done, we had accrued a backlog of functional tests to write, so we split the work between us and worked collaboratively to complete it.

The Development Process

We started with the customary Three Amigos agile ceremony. However it differed slightly to a normal Three Amigos meeting in that the scenarios and corresponding acceptance criteria had already been written for previous stories, we just needed to harvest them from those various tickets. We had decided to split the work by API and the relevant past tickets tended to have acceptance criteria referencing multiple APIs, so we created a spreadsheet and meticulously went through each ticket splitting out the scenarios according to API (see figures 1 and 2).

As an AM developer
I want to have a complete suite of functional tests for the filter resource API endpoint
so that the functionality of the endpoint meets the functional requirements

Figure 1 – Filter resource functional tests user story.

~~AM-328~~
Scenario8: FilterResource to check both ResourceId and ResourceType for explicit access records
GIVEN there is an explicit access record for a resource
WHEN filterResource method or equivalent API is called
THEN for the explicit access record to be used to provide access to the resource, both the resourceId and resourceType must match, in addition to the accessor

~~AM-329~~
Scenario9: Resource Filter Having Wild Card permission
Given When User wants Filter for Resource
When I call Filter Resource API
Then I can get Filter Envelope with wild card

Scenario10: Resource Filter Having Wild Card permission and Resource having Explicit Permission
Given When User wants Filter for Resource
When I call Filter Resource API
Then I can get Filter Envelope with wild card & Explicit permissions merged

Scenario11:Filter Resource for any user once Wild card access has been revoked
Given When User wants Filter wild card access to resource but wildcard access revoked
When I call Get access Filter API
Then Then I can get Filter Envelope should be empty

Figure 2 – Some acceptance criteria pertaining to the Filter Resource API.

In the codebase, one class had previously contained the functional test suite in its entirety, but because we were intending to expand it considerably, in order to make it more scalable, it was refactored and split into separate classes- one for each API (see figure 3). I was tasked with the functional tests for the Grant Access API, and then some of the tests for the Filter Resource API. For the sake of brevity I will focus here on the tests I wrote for the latter.

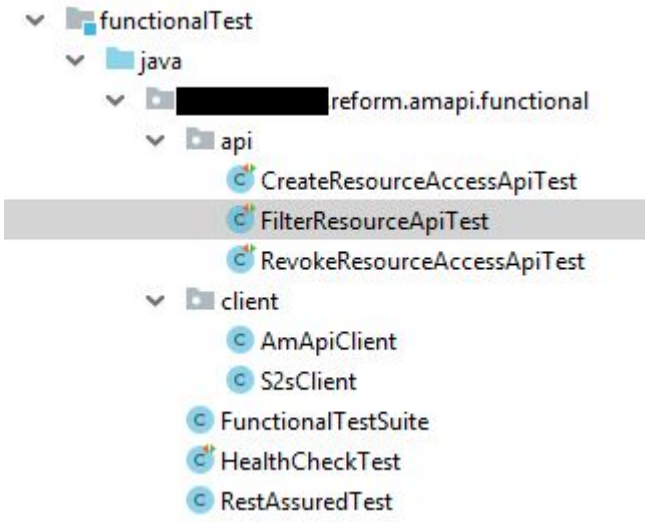

Figure 3 – Functional test folder structure after refactor.

To begin with I read through the relevant classes to understand how they work and see how existing examples of functional tests were written. Each of the API test classes inherited from a parent class (see figure 4) that contained a series of reusable variables and methods.

```
public class FilterResourceApiTest extends FunctionalTestSuite {
```

Figure 4 – All functional test classes inherited from this parent class

One method I made use of a lot was 'createExplicitGrantForFilterCase' (figure 5). This method takes five parameters, with which it then builds a custom body for a request to the Grant Access API. This could fulfil the Given part of the acceptance criteria, without requiring several lines of code to be repeated throughout the class.

```
protected void createExplicitGrantForFilterCase(String resourceId, String accessorId, AccessorType accessorType,
                                                String relationship, Permission permission) {
    ExplicitAccessGrant explicitAccessGrant = ExplicitAccessGrant.builder()
        .resourceId(resourceId)
        .resourceDefinition(ResourceDefinition.builder()
            .serviceName(serviceName)
            .resourceName(resourceName)
            .resourceType(resourceType)
            .lastUpdate(Instant.now())
            .build())
        .accessorIds(ImmutableSet.of(accessorId))
        .accessorType(accessorType)
        .relationship(relationship)
        .attributePermissions(ImmutableMap.of(JsonPointer.valueOf(""), ImmutableSet.of(permission)))
        .lastUpdate(Instant.now())
        .build();

    amApiClient.createResourceAccess(explicitAccessGrant).post(
        path: amApiClient.getAccessUrl() + "api/" + version + "/access-resource");
}
```

Figure 5 – A reusable method inherited in my test class.

Another helpful method was 'createGenericFilterResourceMetadata' (figure 6). This too proved useful in all of my tests, this time fulfilling the When statements. The method similarly creates a custom body for requests to the filter resource API. I amended the method slightly to include a JSON field and value so that I could assert that the contents of the returned resource were as expected.

```
protected FilterResource createGenericFilterResourceMetadata(
    String accessorIdCustom, String resourceIdCustom, String relationshipCustom) {
    if (accessorIdCustom != null) {
        accessorId = accessorIdCustom;
        relationship = relationshipCustom;
        resourceId = resourceIdCustom;
    }
    return FilterResource.builder()
        .userId(accessorId)
        .userRoles(ImmutableSet.of(relationship))
        .resource(Resource.builder()
            .id(resourceId)
            .definition(ResourceDefinition.builder()
                .serviceName(serviceName)
                .resourceName(resourceName)
                .resourceType(resourceType)
                .lastUpdate(Instant.now())
                .build())
            .data(JsonNodeFactory.instance.objectNode().put( fieldName: "name",  v: "test"))
            .build())
        .attributeSecurityClassification(ImmutableMap.of(JsonPointer.valueOf(""), PUBLIC))
        .build();
}
```

Figure 6 – Another inherited reusable method inherited in my test class.

I also researched Serenity to better my understanding of its capabilities. I learned it could be used with '@Step' annotations that take a String parameter. The Strings passed would typically be the Given When Then statements of the acceptance criteria. This could make for a more readable and accessible report produced, and support the adoption of Behaviour Driven Development. I perceived that utilising the BDD functionality of Serenity could therefore yield an improved quality of output over the basic usage currently required of us, so I raised the suggestion with our QA. I received feedback that it was a good idea, but I would not be required to include it in my development work presently, as it was not yet the client's company standard to use the BDD features of Serenity. The client had begun hosting BDD workshops however so it could well be implemented in future when they fully adopt BDD.

Having researched the framework and the pre-existing code, and gathered which of the parent class members at my disposal would be useful to me, I started work on adding the missing functional tests. In the first instance I wrote shells for the tests, and copied in the Given When Then statements of the acceptance criteria to use as a guide (see figure 7).

```
    @Test
    public void verifyFilterResourceWithWildCardPermission() {
//        Given When User wants Filter for Resource


//        When I call Filter Resource API


//        Then I can get Filter Envelope with wild card

    }
```

Figure 7 – Tests started out as empty shells save for the acceptance criteria for guidance.

The functionality under test here, wildcard, is a feature that was added so that if a resource had wildcard access applied to it, it was essentially accessible to all users. So here (figure 8), I am defining a resource with wildcard access and then calling the Filter Resource API with a generic accessor id (one of the reusable members of the parent class). Finally I'm asserting that the API returns this resource, regardless of the accessor not having explicit access, because the resource has wildcard access.

```
+     @Test
+     public void verifyFilterResourceWithWildCardPermission() {
+ //        Given When User wants Filter for Resource
+         createExplicitGrantForFilterCase(resourceId, "*", DEFAULT, null, READ);
+
+ //        When I call Filter Resource API
+         FilterResource filterResourceMetadata = createGenericFilterResourceMetadata(accessorId, resourceId, relationship);
+         Response response = amApiClient.filterResource(filterResourceMetadata)
+             .post(amApiClient.getAccessUrl() + api + version + filterResouce);
+
+ //        Then I can get Filter Envelope with wild card
+         response.then()
+             .assertThat()
+             .statusCode(HttpStatus.OK.value())
+             .body("resource.data.name", Matchers.equalTo("test"))
+             .body("access.permissions.values()[0][0]", Matchers.equalTo("READ"))
+             .log();
+     }
+
```

Figure 8 – The complete test verifying Filter Resource returned resources with wildcard access applied.

I decided to leave the commented Given When Then statements as I felt they improved readability at the very least, and could prove useful should the client request the '@Step' annotations be used in future.

The next test (figure 9) was fairly simple to write as it followed the same pattern. It required one extra line in the Given section as the test was going to assert that the Filter Resource API would handle resources with two types of access applied to them correctly.

```
+     @Test
+     public void verifyFilterResourceWithWildCardAndExplicitPermission() {
+ //        Given When User wants Filter for Resource
+         createExplicitGrantForFilterCase(resourceId, "*", DEFAULT, null, READ);
+         createExplicitGrantForFilterCase(resourceId, accessorId, USER, relationship, UPDATE);
+
+ //        When I call Filter Resource API
+         FilterResource filterResourceMetadata = createGenericFilterResourceMetadata(accessorId, resourceId, relationship);
+         Response response = amApiClient.filterResource(filterResourceMetadata)
+             .post(amApiClient.getAccessUrl() + api + version + filterResouce);
+
+ //        Then I can get Filter Envelope with wild card & Explicit permissions merged
+         response.then()
+             .assertThat()
+             .statusCode(HttpStatus.OK.value())
+             .body("resource.data.name", Matchers.equalTo("test"))
+             .body("access.permissions.values()[0].size()", Matchers.equalTo(2))
+             .body("access.permissions.values()[0][0]", Matchers.equalTo("UPDATE"))
+             .body("access.permissions.values()[0][1]", Matchers.equalTo("READ"))
+             .log();
+     }
+
```

Figure 9 – The completed test verifying Filter Resource returned resources with both types of access in one response.

I created the wildcard access to the resource with a 'READ' permission and the explicit access with an 'UPDATE' permission, so that I could assert that both permissions were returned when the Filter Resource API was called.

The next test required me to deviate from this established pattern slightly. Here I was testing the Filter Resource API would behave correctly once wildcard access on a given resource was revoked. So the first line here (figure 10) was

the same as before, applying wildcard access to a resource, but it then needed to be revoked to finish setting up the premise of this test.

```
+     @Test
+     public void verifyFilterResourceWhenWildCardPermissionRevoked() {
+ //        Given When User wants Filter wild card access to resource but wildcard access revoked
+         createExplicitGrantForFilterCase(resourceId, "*", DEFAULT, null, READ);
+         ExplicitAccessMetadata explicitAccessMetadata = ExplicitAccessMetadata.builder()
+             .resourceId(resourceId)
+             .accessorId("*")
+             .accessorType(DEFAULT)
+             .attribute(attribute)
+             .serviceName(serviceName)
+             .resourceName(resourceName)
+             .resourceType(resourceType)
+             .build();
+         amApiClient.revokeResourceAccess(explicitAccessMetadata);
+
+ //         When I call Get access Filter API
+         FilterResource filterResourceMetadata = createGenericFilterResourceMetadata(accessorId, resourceId, relationship);
+         Response response = amApiClient.filterResource(filterResourceMetadata)
+             .post(amApiClient.getAccessUrl() + api + version + filterResouce);
+
+ //         Then I can get Filter Envelope should be empty
+         response.then()
+             .assertThat()
+             .statusCode(HttpStatus.OK.value())
+             .contentType(Matchers.isEmptyOrNullString())
+             .log();
+     }
```

Figure 10 – The completed test verifying Filter Resource would return no permissions once wildcard access was revoked.

I found that the AmApiClient class contained a ‘revokeResourceAccess’ method I could use, but it took ExplicitAccessMetaData as an argument which would need to be built in the first instance, which was fairly simple to do but added several extra lines of code. I considered housing the ExplicitAccessMetaData builder in a method, putting this in the FunctionalTestSuite class and calling on it in the same way as I do the ‘createGenericFilterResourceMetadata’ method. Ultimately I discerned this would just displace the bulk of the lines of code elsewhere, and doing this would only save me from writing unnecessary lines of code if this were a method I could use again in other tests- which is not the case here.

A similar decision needed to be made for my last functional test for the Filter Resource API (figure 11).

```
    @Test
    public void verifyFilterResourceChecksBothResourceIdAndResourceType() {

//        GIVEN there is an explicit access record for a resource
        createExplicitGrantForFilterCase(resourceId, accessorId, accessorType, relationship, READ);
        ExplicitAccessGrant.builder()
            .resourceId(resourceId)
            .resourceDefinition(ResourceDefinition.builder()
                .serviceName(serviceName)
                .resourceName(resourceName)
                .resourceType(otherResourceType)
                .lastUpdate(Instant.now())
                .build())
            .accessorIds(ImmutableSet.of(accessorId))
            .accessorType(accessorType)
            .relationship(relationship)
            .attributePermissions(ImmutableMap.of(JsonPointer.valueOf(""), ImmutableSet.of(UPDATE)))
            .lastUpdate(Instant.now())
            .build();
```

Figure 11 – Snippet of test verifying the API checks both resource id and type.

Here I needed to create the explicit access record twice- once with a variant resource type. This would allow me to later assert that the Filter Resource API would only return one of the permissions, having checked both resource id and type. To create the second explicit access record the generic reusable method couldn't serve my purpose as it didn't allow for a custom resource type to be set, so again the entire builder pattern was required. As with before I decided against housing this inside another helper method in the FunctionalTestSuite class at this time, as I only needed this variation once. In order to avoid hard coding the resource type variant I added an 'otherResourceType' to the list of protected variables (figure 12) in the FunctionalTestSuite class.

```
    public class FunctionalTestSuite {

@@ -66,6 +72,7 @@
        protected final JsonPointer attribute = JsonPointer.valueOf("");
        protected static String resourceName = "claim-test";
        protected static String resourceType = "case-test";
+       protected static String otherResourceType = "other-case-test";
```

Figure 12 – My addition to the list of protected variables.

In order to run these functional tests Docker had to be running first. Then, when running the tests in Intellij, Serenity produces a detailed breakdown of the progress, and success or failure, of each test (figure 13), along with stack traces.

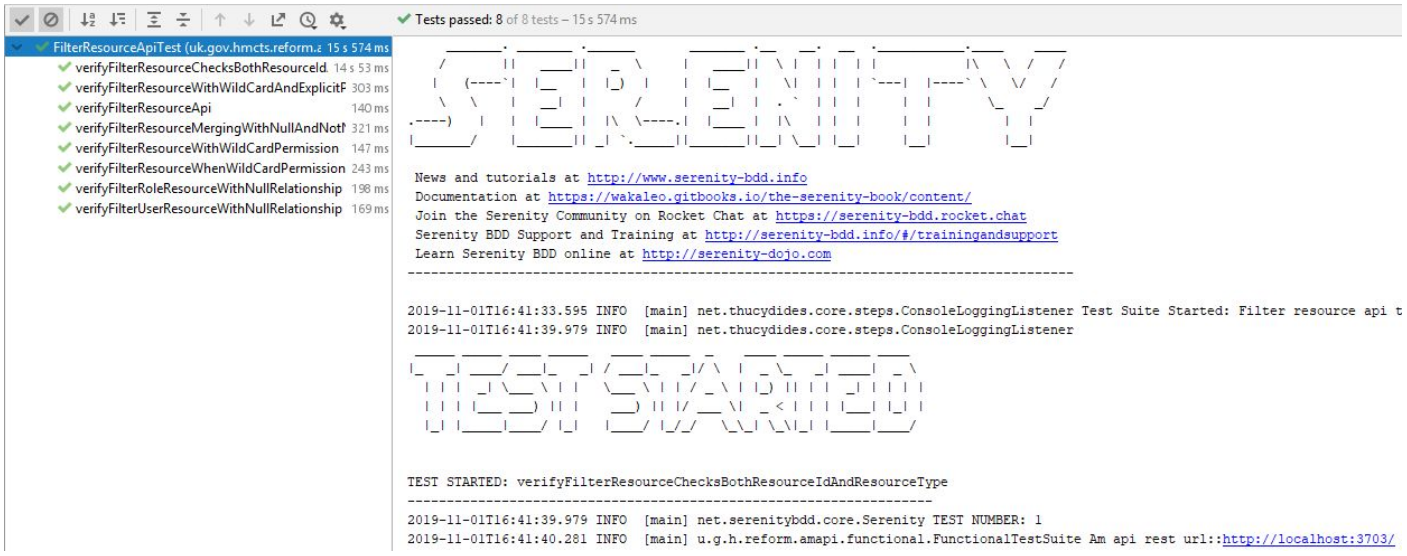

Figure 13 – Serenity's in-IDE test report.

Should a test fail it would furnish you with details of the specific expectation that was not met (figure 14). This provided a helpful head start when it came to solving failing tests.

```
java.lang.AssertionError: 1 expectation failed.
JSON path access.permissions.values()[0].size() doesn't match.
Expected: <3>
  Actual: 2
```

Figure 14 – Details of a failed test.

Once completed functional tests were peer reviewed and a pull request made to merge with master, the Jenkins pipeline build at this point- and all subsequent Jenkins builds- produces illustrative test reports, found in the build artefacts, as per the client's requirements (see figures 15 and 16).

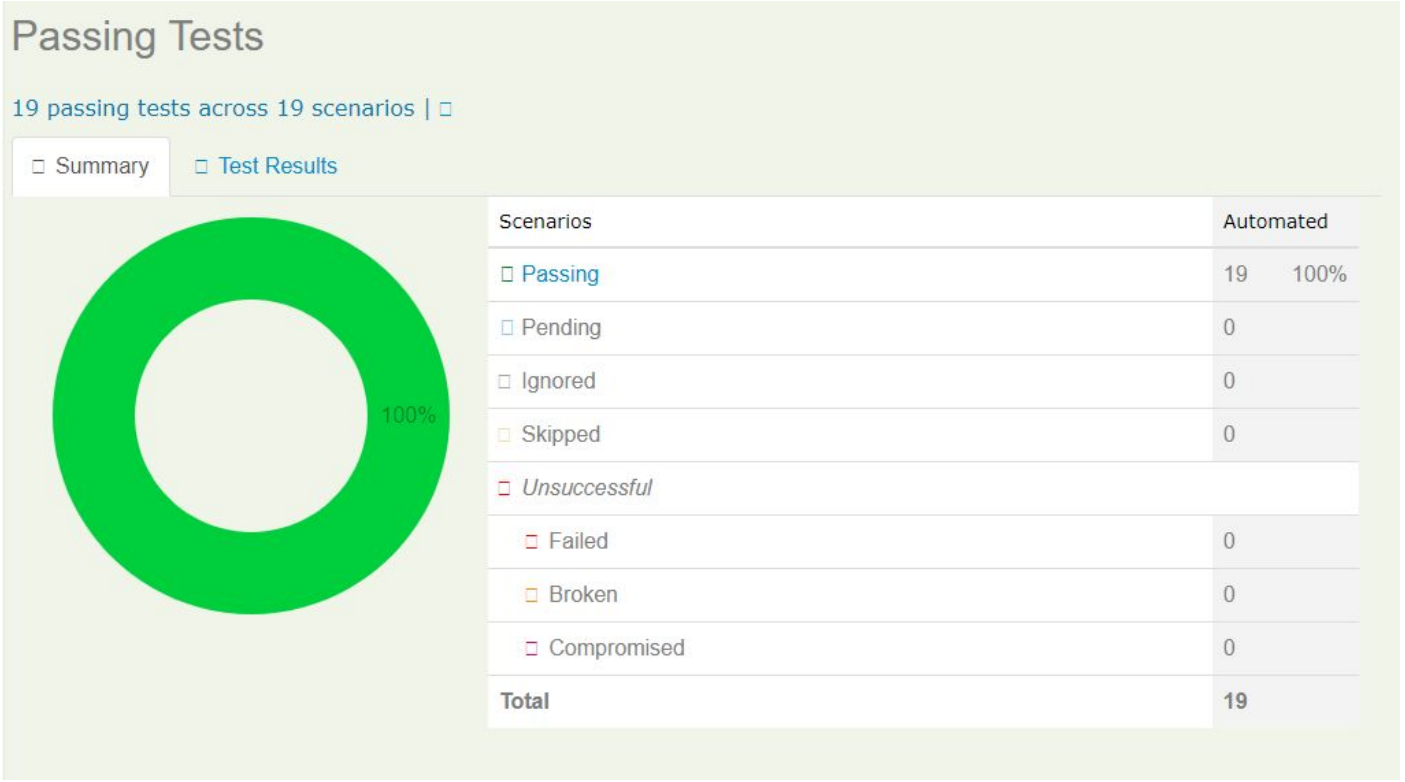

Figure 15 – Serenity report main page.

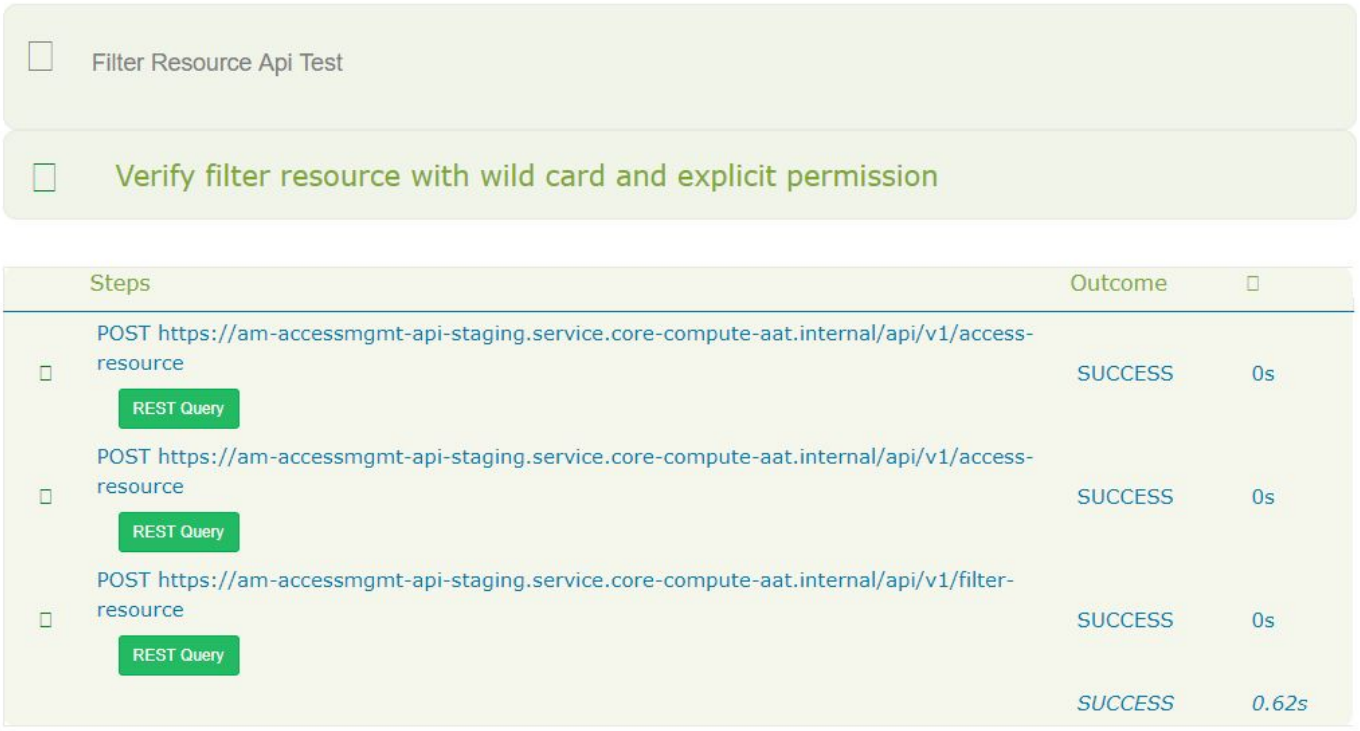

Figure 16 – Serenity report test details.

Reflective Statement

The demands on my time during the course of this project were varying and complex. There was the primary focus of supporting the team's efforts to grow a comprehensive suite of functional tests, but my days were not routine as my attention was split between this and preparation for the Oracle Certified Associate Java 8 exam which was taken off site. I was advised by colleagues to schedule a date for the exam, so as to have a deadline to work towards. While this did increase the pressure I felt, it also sharpened my focus, as I was now planning and monitoring my work to meet two deadlines concurrently. To that end, I spoke with my line manager to request time for self-study, and kept in regular communication regarding the progress of my work.

Another salient factor is that this was again a period of transition as I had just been assigned to a new line manager. My previous manager was several grades of seniority above me, and while having a direct relationship with a colleague at that level of experience is notionally very valuable, their responsibilities of overseeing all of our company's teams at this client site and the escalating workload left them without any residual time. I therefore initiated a conversation and we resolved that another line manager, who had already been informally mentoring me for some time, would be better positioned to support my personal development. This ultimately proved to be a very positive change.

Regarding the OCA Java 8 exam I was able to pass on the first attempt with a score of 77%, and with regards to the functional tests I along with my team mates were successful in fulfilling the client's wishes to have illustrated Serenity reports detailing test coverage of all areas of functionality. Having established this, anyone working on this project can easily build on the functional test suite moving forward.