

COMP2005 Assessment 2: Report

Assessment Materials

D2 GitHub Repository:

<https://github.com/Plymouth-University/comp2005-assessment2-corey-richardson>

D3 YouTube Link:

https://youtu.be/ADZzaU-w2ik?si=_rCqnM0IDvDz5qd0

API Endpoints

Table 1: API Endpoints

F1	/api/Patients/never-admitted
F2	/api/Patients/readmitted-within-7-days
F3	/api/Admissions/month-with-most
F4	/api/Patients/multiple-staff

Unit and Integration Testing for Part A

Each layer of the application was tested according to its role in the application architecture. Classes and service methods were tested using Unit Tests to verify their behaviour in isolation from the API returns. Services were tested using mocked dependencies and returns to ensure that the implementations matched expected business logic. Controllers were tested using Integration Tests to verify the API's behaviour.

Table 2: Unit and Integration Test Files and the Application Layer they test.

Test File	Layer Tested	Test Types
AdmissionControllerTest.java	Controller	Integration Tests
AdmissionServiceTest.java	Service	Unit Tests
AdmissionTest.java	Class	Unit Tests
AllocationTest.java	Class	Unit Tests
ApiHelperTest.java	Service	Unit Tests
Comp2005ApiApplicationTests.java	Application Context	Smoke Test
EmployeeTest.java	Class	Unit Tests
PatientControllerTest.java	Controller	Integration Tests
PatientServiceTest.java	Service	Unit Tests
PatientTest.java	Class	Unit Tests

Tables containing detailed information about every implemented test can be found in Appendix A: Unit and Integration Test Tables.

Tests were implemented using the **JUnit** framework. Test classes often contained **'@Nested'** sub-classes, each focusing on a different portion of a service, to ensure that the test suites remained logically separated and readable. **'@BeforeEach'** decorators were used to control test set-up and configurations, and **'@Test'** is used to decorate test methods. JUnit uses these decorations to manage the discovery, lifecycles and execution of the tests.

```
1 @Test
2 void testBoundaryBehaviour() {
3     // Arrange
4     Patient patientOne = new Patient(1, "LESS", "THAN", "06235959");
5     Patient patientTwo = new Patient(2, "EXACTLY", "SEVEN", "07000000");
6     Patient patientThree = new Patient(3, "MORE", "THAN", "07000001");
7
8     Admission firstAdmission_patientOne = new Admission(1, 1, "2024-12-31T00:00:00", "2025-01-01T00:00:00");
9     Admission firstAdmission_patientTwo = new Admission(2, 2, "2024-12-31T00:00:00", "2025-01-01T00:00:00");
10    Admission firstAdmission_patientThree = new Admission(3, 3, "2024-12-31T00:00:00", "2025-01-01T00:00:00");
11
12    Admission readmitWithinSevenDays = new Admission(4, 1, "2025-01-07T23:59:59", null); // < 7 days, INCLUDE
13    Admission readmitExactlySevenDays = new Admission(5, 2, "2025-01-08T00:00:00", null); // = 7 days, INCLUDE
14    Admission readmitAfterSevenDays = new Admission(6, 3, "2025-01-08T00:00:01", null); // > 7 days, EXCLUDE
15
16    mockPatients = new Patient[] { patientOne, patientTwo, patientThree };
17    mockAdmissions = new Admission[] { firstAdmission_patientOne, firstAdmission_patientTwo, firstAdmission_patientThree, readmitWithinSevenDays, readmitExactlySevenDays, readmitAfterSevenDays };
18
19    when(apiHelper.getAllPatients()).thenReturn(mockPatients);
20    when(apiHelper.getAllAdmissions()).thenReturn(mockAdmissions);
21
22    // Act
23    List<Patient> patients = service.getPatientsReadmittedSevenDays();
24
25    // Assert
26    assertNotNull(patients);
27    assertEquals(2, patients.size());
28    assertEquals(1, patients.get(0).getId());
29    assertEquals(2, patients.get(1).getId());
30
31 }
```

Figure 1: Test Function 'testBoundaryBehaviour' from *PatientServiceTests::ReadmittedWithinSevenDaysTests::testBoundaryBehaviour()*

This test is used to verify the behaviour of the **'PatientService::getPatientsReadmittedSevenDays()'** method at the boundary value of 7 days between a discharge and a readmission; this covers an edge case.

The designed behaviour of the method is to be inclusive of admissions beginning exactly 7 days after a previous discharge date. I decided that this was the best behaviour to follow because in real-world practice, users may input admission times using only the hour (XX:00:00), potentially omitting the 'unimportant' precision in time. This covers a common Human Computer Interaction aspect and user heuristic where times are rounded. This decision should be discussed with a client to confirm it matches their intended business logic.

Comments are used in the figure above to specify the sections of the Arrange, Act, Assert process of testing each code segment corresponds to:

Lines 3:20 - Arrange

- These lines set up mock objects and inject them using the **'when(...).thenReturn(...)'** syntax. This isolates the method being tested from their dependencies.

Line 23 - Act

- This line calls the tested method. The Arrange process defines returns the mocked data.

Lines 26:29 - Assert

- These lines confirm that the expected patients were in the returned patient list.

```

1 // https://stackoverflow.com/questions/450807/how-do-i-make-the-method-return-type-generic
2 private <T> T handleRequest(String url, Class<T> responseType) {
3     try {
4         return restTemplate.getForObject(url, responseType);
5     } catch (HttpClientErrorException e) {
6         if (e.getStatusCode() == HttpStatus.NOT_FOUND) {
7             return null;
8         }
9         throw e;
10    }
11 }

```

```

1 @SpringBootTest
2 class ApiHelperTest
3 {
4
5     @MockitoBean
6     private RestTemplate restTemplate;
7
8     @Autowired
9     private ApiHelper apiHelper;
10
11     private Admission mockAdmission = new Admission();
12     private Allocation mockAllocation = new Allocation();
13     private Employee mockEmployee = new Employee();
14     private Patient mockPatient = new Patient();
15
16     private Admission[] mockAdmissions;
17     private Allocation[] mockAllocations;
18     private Employee[] mockEmployees;
19     private Patient[] mockPatients;
20
21     @Nested
22     class HandleRequestTests {
23
24         // https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/HttpClientErrorException.html
25         @Test
26         void propagateExceptionWhenNot404Error() {
27             when(restTemplate.getForObject("https://web.socem.plymouth.ac.uk/COMP2005/api/Patients/1", Patient.class))
28                 .thenReturn(new HttpClientErrorException(HttpStatus.BAD_REQUEST));
29
30             HttpClientErrorException e = assertThrows(HttpClientErrorException.class,
31                 () -> apiHelper.getPatientById(1)
32             );
33
34             assertEquals(HttpStatus.BAD_REQUEST, e.getStatusCode());
35         }
36     }
37
38     // ...
39 }
40 }
41

```

Figure 2: Example of a Test Class, Sub-class and Method, *ApiHelperTest::HandleRequestTests::propagateExceptionWhenNot404Error()*

The above test class is used to verify the behaviour of the helper/utility method **'ApiHelper::handleRequest()'**. This method handles HTTP 404 responses gracefully by returning a null object, but if the API returns a status code of anything other than 200 (SUCCESS) or 404 (NOT FOUND), it should fail loudly and propagate this exception to the ApiHelper method it is being called by.

Lines 27:28 creates a mock response for when the API is called, to mock an exception; this is the 'Arrange' stages of the AAA process. Lines 30:32 cover both the Act and Assert stages using the lambda function to not only detect the exception with **'assertThrows'**, but to also store the returned status code for use in Line 34, which covers only the Assert stage.

Mock objects were used to isolate units of code using a top-down manner to emulate the behaviour of dependent methods and API calls. By replacing dependencies with mocks, tests can concentrate on the logic of the method they are designed to test, rather than external factors.

The **'Mockito'** library was largely used to create mock instances of service class methods, which were subsequently injected into the controller layer using **'@Mock'** and **'@InjectMock'** decorators. Method returns were controlled using **'when(<method>).thenReturn(<data>')** logic to ensure consistency in the behaviour of dependencies during test runs.

I ran my tests using IntelliJ IDEA Community Edition (2024.2.2). To do this, open the **'part-a-web-service-api'** Spring Boot project and navigate to the **'src/test/java'** directory in the **Project** explorer window. To run all tests right click on the **'com.example.comp2005_api'** package or to run individual test classes right click on the respective file (for example **'ApiHelperTest.java'** and select the option for **'Run Tests in com.example.comp2005_api'** or **'Run ApiHelperTest'** (example).

Any selected tests will run, and the test results will be displayed in the terminal section of the IDE.

Tests can also be run from the terminal section using the command **'./gradlew clean test'**.

Use of the Test-Driven Development Approach

The implementation of F1 followed a more traditional development methodology where code was written before tests were designed. This was necessary because I first needed to understand the process for implementing these endpoints using this framework and tech stack.

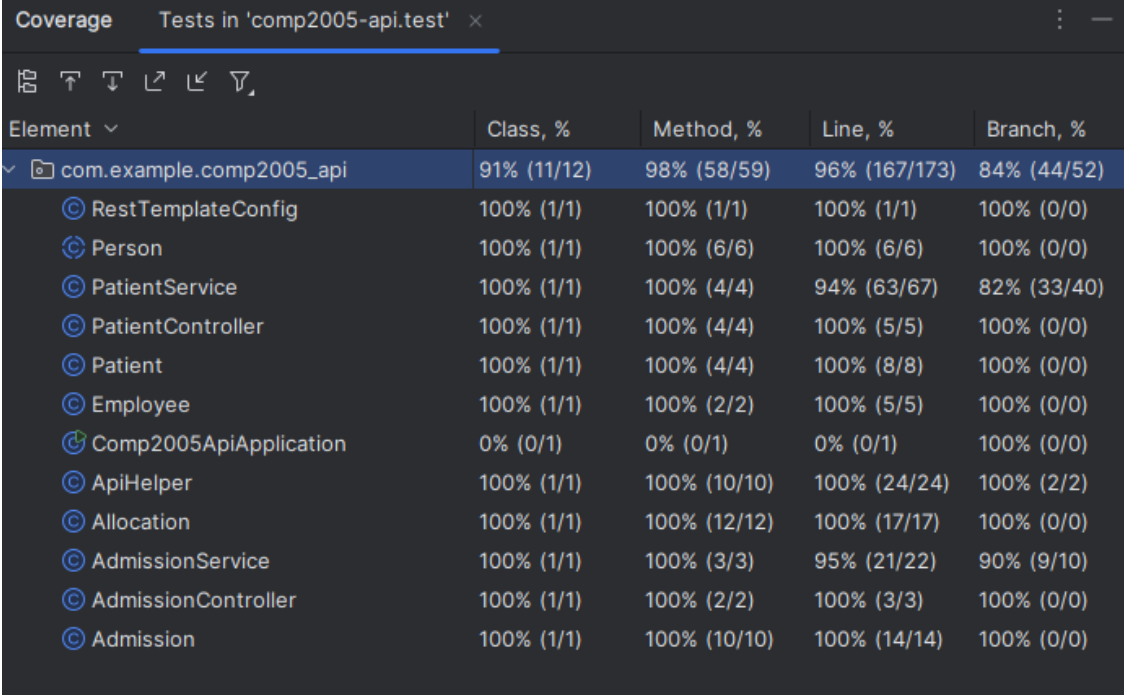
To assist with the implementation of F2, F3 and F4 for Part A of the task, I used a Test-Driven Development (TDD) approach, where I wrote failing tests for each method before implementing the methods. The TDD workflow consists of 3 stages:

- Red: Write a failing test.
- Green: Write code to make the test pass.
- Refactor: Refactor and tidy the code, without changing its behaviour.

This approach helped me to ensure that the behaviour of methods and return values matched the expected behaviour. This approach shifts the focus from writing code that “works” to writing code that “works correctly”.

The use of this approach is evidenced by commits ``11a2b7e``, ``b8812a0`` and ``8cc565b``.

Metrics and Code Coverage



The screenshot shows the 'Coverage' window in IntelliJ IDE, titled 'Tests in 'comp2005-api.test''. It displays a table of code coverage metrics for various classes and methods. The table has five columns: 'Element', 'Class, %', 'Method, %', 'Line, %', and 'Branch, %'. The 'Element' column lists classes and methods, some with expand/collapse icons. The other columns show the percentage of coverage and the number of items covered/total items.

Element	Class, %	Method, %	Line, %	Branch, %
com.example.comp2005_api	91% (11/12)	98% (58/59)	96% (167/173)	84% (44/52)
RestTemplateConfig	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
Person	100% (1/1)	100% (6/6)	100% (6/6)	100% (0/0)
PatientService	100% (1/1)	100% (4/4)	94% (63/67)	82% (33/40)
PatientController	100% (1/1)	100% (4/4)	100% (5/5)	100% (0/0)
Patient	100% (1/1)	100% (4/4)	100% (8/8)	100% (0/0)
Employee	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
Comp2005ApiApplication	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
ApiHelper	100% (1/1)	100% (10/10)	100% (24/24)	100% (2/2)
Allocation	100% (1/1)	100% (12/12)	100% (17/17)	100% (0/0)
AdmissionService	100% (1/1)	100% (3/3)	95% (21/22)	90% (9/10)
AdmissionController	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
Admission	100% (1/1)	100% (10/10)	100% (14/14)	100% (0/0)

Figure 3: Code Coverage Metrics from IntelliJ IDE

Test Report: [part-a-web-service-api/htmlReport/index.html](#)

A large proportion of the application has been covered by the unit and integration tests. This should result in better-tested and more robust code, reducing the number of bugs likely to be found in a non-development environment.

There are some minor gaps in coverage, primarily to do with the **'Comp2005ApiApplication.java'** file, which only contains the main function which acts as an entry point to the application; even then, this is covered by a **'contextLoad'** test which confirms that the application successfully begins. There is also a gap in coverage on the two service files, relating to catching exceptions and date parsing errors: branch coverage.

Usability Testing for Part B

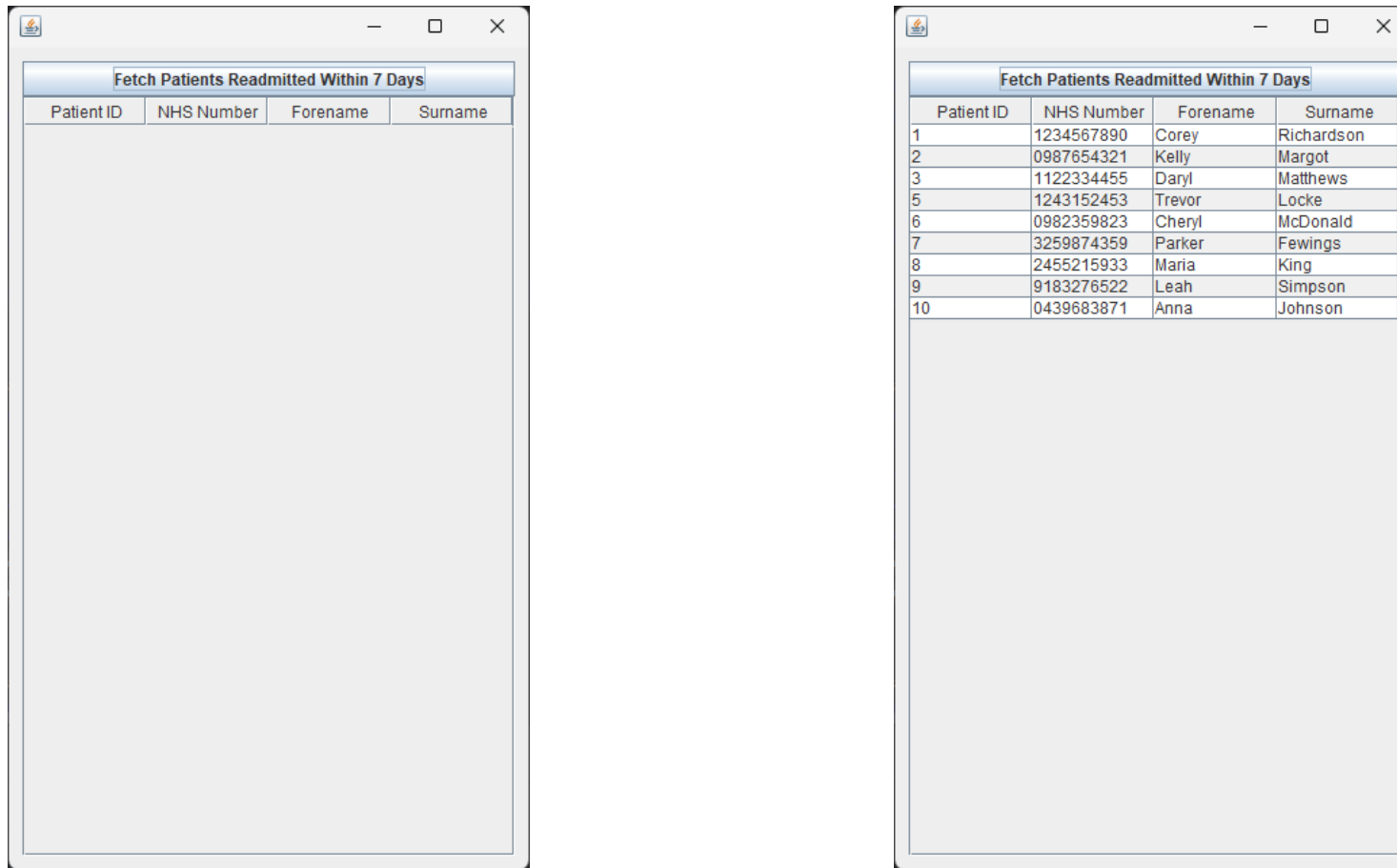


Figure 4: GUI Application before Usability Testing

Note: Additional detail about UI considerations made before the Usability Testing can be found in Appendix B: Before Usability Test UI Design Considerations.

COMP2005: Maternity Swing UI Usability Test

Thank you for participating in this Usability Test. The purpose of this survey is to gather feedback on a desktop application that fetches and displays patient information for patients readmitted within a seven day window.

Your responses to this survey may be used to identify weaknesses in the User Interface and design of the application, as well as the User Experience.

- Ⓢ There are no incorrect answers. We are testing the usability of the application, we are NOT testing your technical abilities as an end-user. Your feedback is useful in the development of this project, whether it is positive or negative.

All responses are anonymous; we do not collect any personal information. Your participation is voluntary and you may exit the survey at any time.

Thank you!

Figure 5: Usability Test Survey Title and Description

The survey responses were mostly positive to the initial design, however, they did provide guidance as to 4 enhancements to make to the UI/UX:

- “I do believe a title would go a long way for people like me who cannot figure things out implicitly”
- “Make the table cells read-only”
- “font size bigger (my eyes are really bad so this is a personal one)”
- “if the api response != 200 then show an error message”

In response to these suggestions:

- I added a title to the frame and added a title label which displays above the ‘Fetch Patients Readmitted Within 7 Days’ button.
- I increased the width of the frame to 600px and created font variables ‘titleFont’ and ‘largerFont’. These fonts are used across components to provide a font size larger than Swing’s default font size. The frame is resizable, however this does not scale font size responsively.
- I override the ‘isCellEditable’ method of the ‘DefaultTableModel’ to ensure that cells were not editable.
- I added a ‘JOptionPane’ message dialog that displays when the HTTP request throws an exception. As the API silently fails and simply returns an empty list if it catches an exception, this will only get hit if ‘parsePatients()’ fails to successfully parse the JSON response. This is likely a limitation with my API’s designed behaviour.

Maternity Readmissions

— □ ×

Maternity Ward Patients Readmitted Within 7 Days

Fetch Patients Readmitted Within 7 Days

Patient ID	NHS Number	Forename	Surname
------------	------------	----------	---------

Maternity Readmissions

— □ ×

Maternity Ward Patients Readmitted Within 7 Days

Fetch Patients Readmitted Within 7 Days

Patient ID	NHS Number	Forename	Surname
1	1234567890	Corey	Richardson
2	0987654321	Kelly	Margot
3	1122334455	Daryl	Matthews
5	1243152453	Trevor	Locke
6	0982359823	Cheryl	McDonald
7	3259874359	Parker	Fewings
8	2455215933	Maria	King
9	9183276522	Leah	Simpson
10	0439683871	Anna	Johnson

Figure 6: GUI Application after Usability Testing

Note: Examples are using mock data to populate the tables.

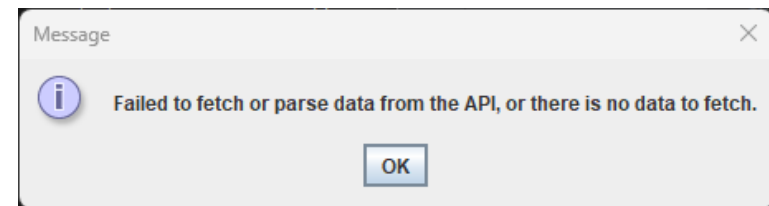
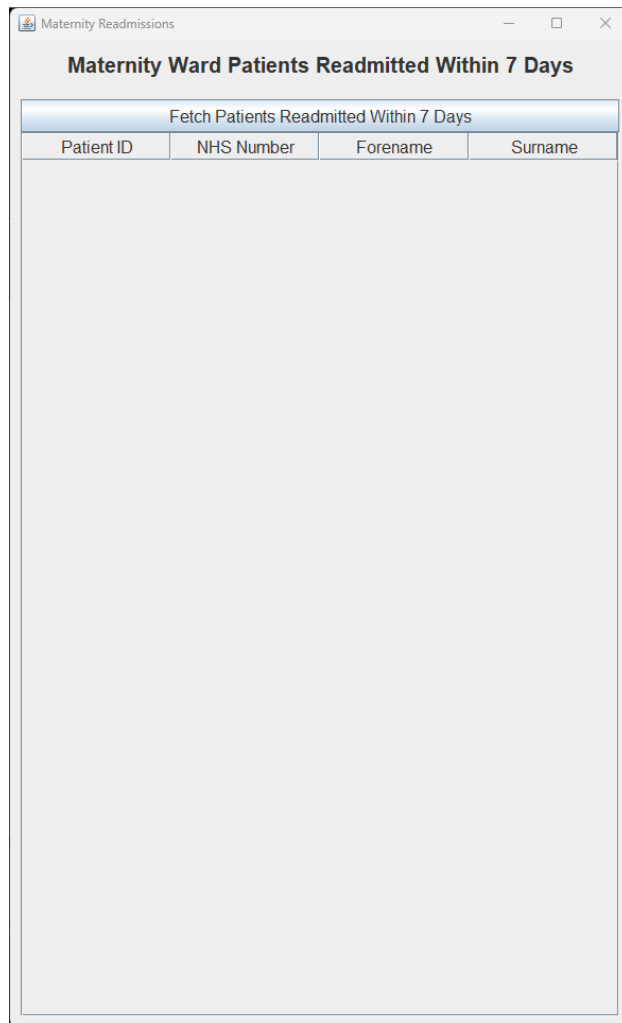


Figure 7: GUI Application Error Message Dialog



```
1 private static List<PatientModel> parsePatients(String jsonResponse) {  
2     // throw new RuntimeException("Mock parsing failure"); // Used to test Error Message Dialog  
3  
4     Gson gson = new Gson();  
5     Type patientListType = new TypeToken<List<PatientModel>>() {}.getType();  
6     return gson.fromJson(jsonResponse, patientListType);  
7 }
```

Figure 8: Modification made to parsePatients() to test Error Message Dialog Box

Note: Could've just stopped the API server...

Jisc Online Survey

<https://app.onlinesurveys.jisc.ac.uk/s/plymouth/comp2005-maternity-swing-ui-usability-test>

JSON Export of Survey

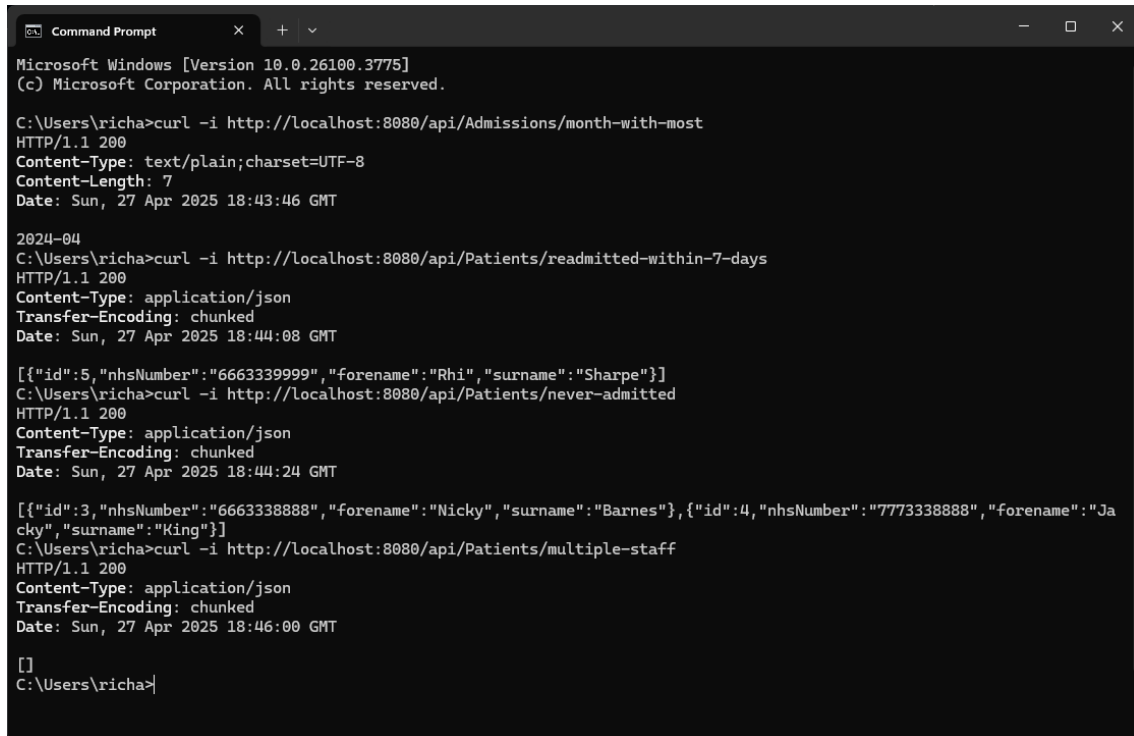
https://github.com/Plymouth-University/comp2005-assessment2-corey-richardson/blob/main/survey/COMP2005_MaternitySwingUiUsabilityTest.json

PDF of Survey Responses

https://github.com/Plymouth-University/comp2005-assessment2-corey-richardson/blob/main/survey/PrintSurveyAnalysis_OnlineSurveys.pdf

Additional System Testing

I used 'curl' CLI requests and a Postman collection to test the endpoints of the API.



```
Microsoft Windows [Version 10.0.26100.3775]
(c) Microsoft Corporation. All rights reserved.

C:\Users\richa>curl -i http://localhost:8080/api/Admissions/month-with-most
HTTP/1.1 200
Content-Type: text/plain; charset=UTF-8
Content-Length: 7
Date: Sun, 27 Apr 2025 18:43:46 GMT

2024-04

C:\Users\richa>curl -i http://localhost:8080/api/Patients/readmitted-within-7-days
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 27 Apr 2025 18:44:08 GMT

[{"id":5,"nhsNumber":"6663339999","forename":"Rhi","surname":"Sharpe"}]
C:\Users\richa>curl -i http://localhost:8080/api/Patients/never-admitted
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 27 Apr 2025 18:44:24 GMT

[{"id":3,"nhsNumber":"6663338888","forename":"Nicky","surname":"Barnes"}, {"id":4,"nhsNumber":"7773338888","forename":"Jacky","surname":"King"}]
C:\Users\richa>curl -i http://localhost:8080/api/Patients/multiple-staff
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 27 Apr 2025 18:46:00 GMT

[]
C:\Users\richa>
```

Figure 9: curl Requests to the API Endpoints

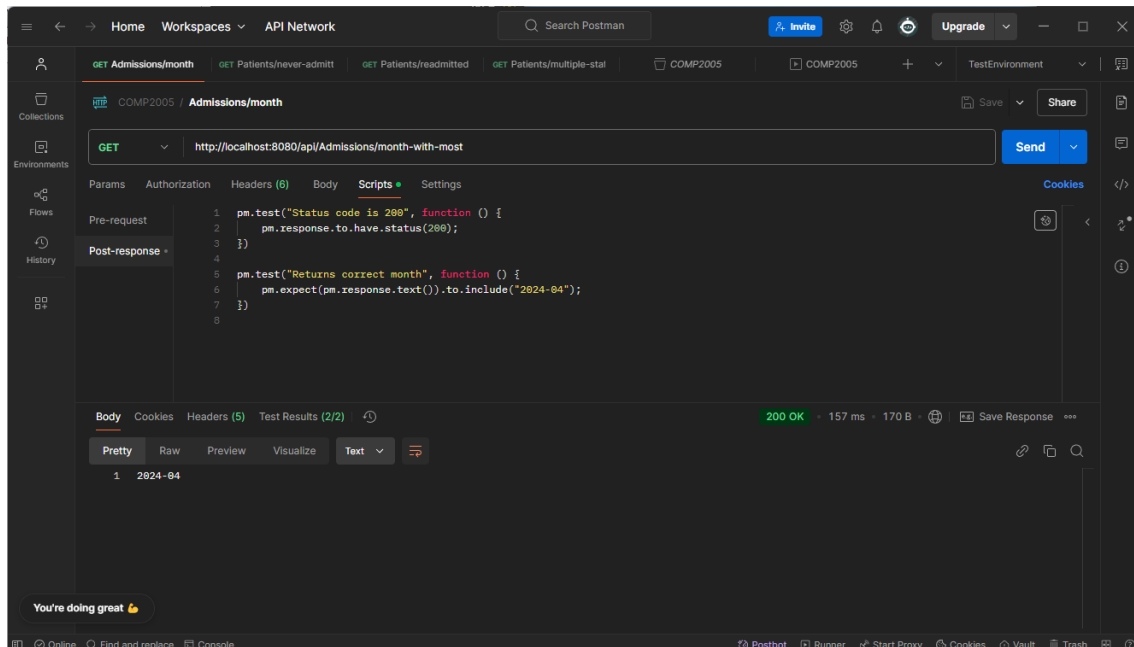


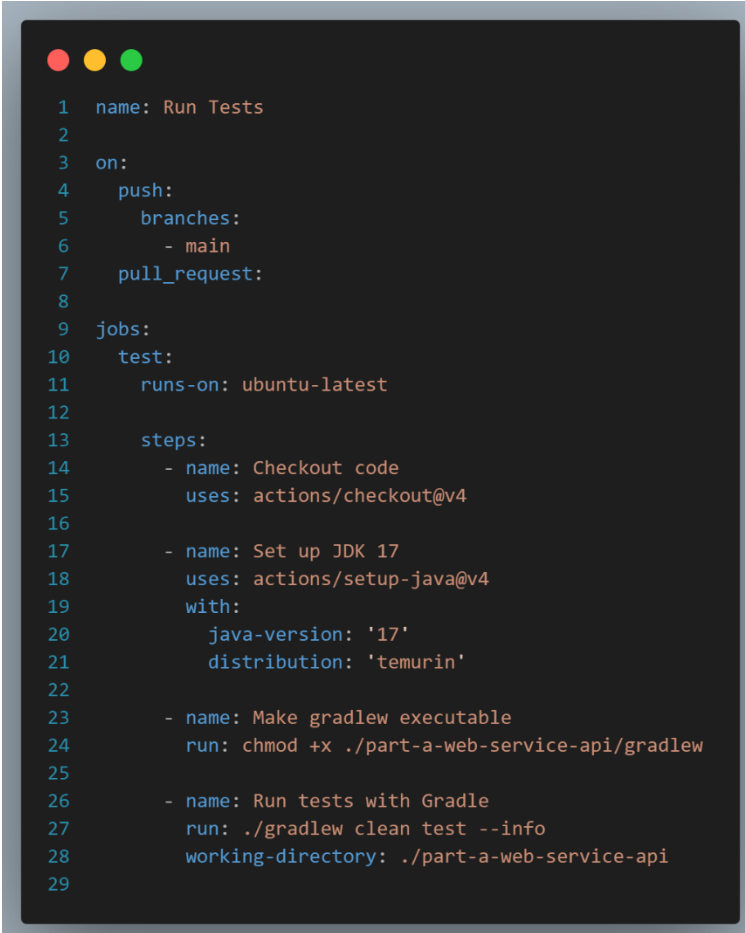
Figure 10: Postman Request to the '/month-with-most' Endpoint and Testing

Tools and Practices

GitHub Actions, CI/CD

I set up a GitHub Actions workflow to automatically run the full test suite on every push to the main branch of the GitHub repository.

This workflow is defined in `‘.github\workflows\run-tests.yaml’`.



```
1 name: Run Tests
2
3 on:
4   push:
5     branches:
6       - main
7   pull_request:
8
9 jobs:
10  test:
11    runs-on: ubuntu-latest
12
13    steps:
14      - name: Checkout code
15        uses: actions/checkout@v4
16
17      - name: Set up JDK 17
18        uses: actions/setup-java@v4
19        with:
20          java-version: '17'
21          distribution: 'temurin'
22
23      - name: Make gradlew executable
24        run: chmod +x ./part-a-web-service-api/gradlew
25
26      - name: Run tests with Gradle
27        run: ./gradlew clean test --info
28        working-directory: ./part-a-web-service-api
29
```

Figure 11: GitHub Actions Workflow

The workflow ensures that it is testing the latest pushed code (Lines 14:15) before setting up a Java 17 environment (Lines 17:21) and ensuring that the Gradle Wrapper file is executable (Lines 23:24). It then runs the test suite (Lines 26:28) and reports the results on the Actions section of the GitHub repository; test failures are also notified by email.

The automated workflow ensures that tests are executed regularly on changes in an independent and consistent environment, identifying integration, build and regression errors early in the change process.

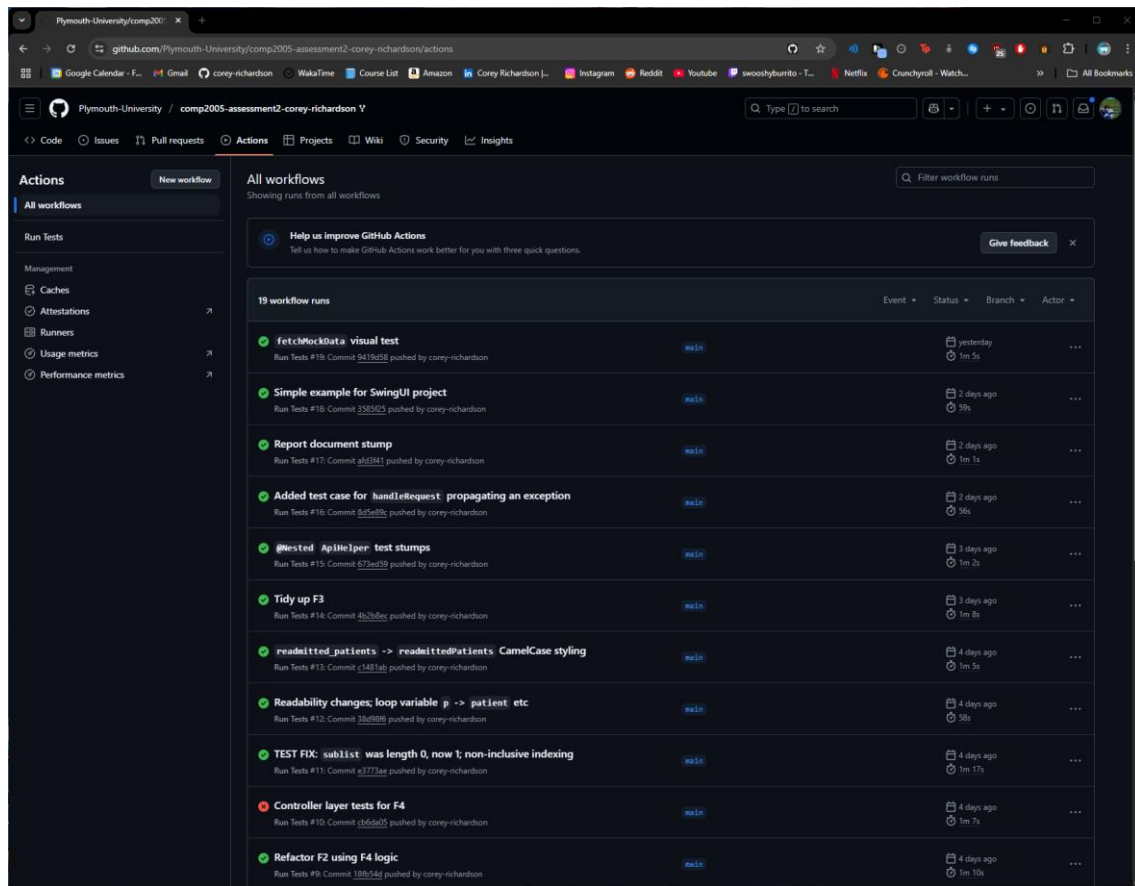


Figure 12: GitHub Actions Page

<https://github.com/Plymouth-University/comp2005-assessment2-corey-richardson/actions>

To test the workflow I had set up, I used the **'GitHub Local Actions'** extension for VS Code to run the workflow locally before pushing new code. This extension utilises the **Docker Engine** to create containerised environments like those used by the GitHub action runners, and **'nektos/act'** to locally run the workflow.

This local testing helped me to reduce the development time for the CI/CD pipeline and ensured that the workflow would work as intended before integrating it into the main GitHub repository.

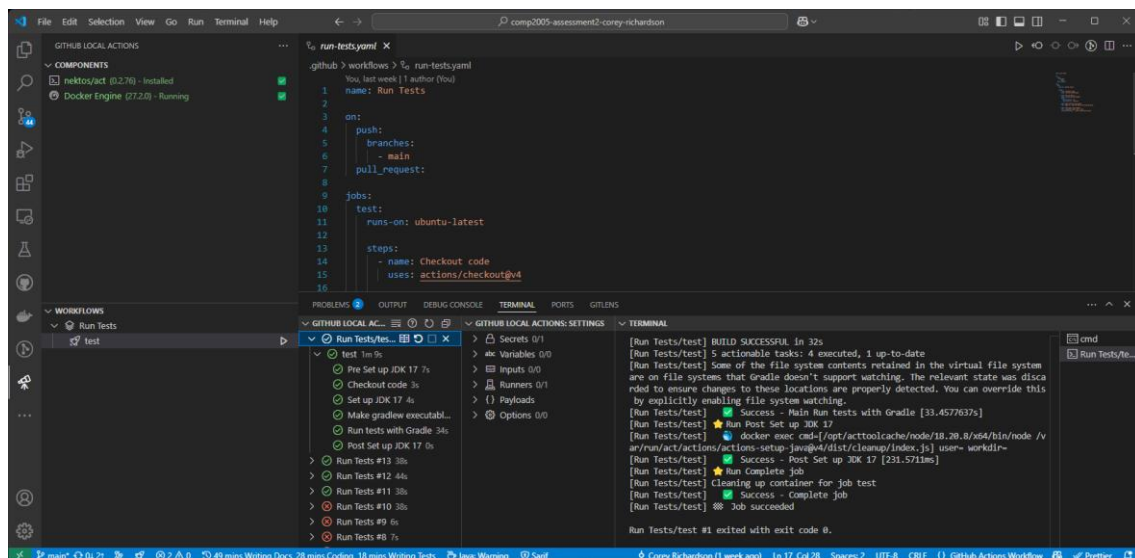
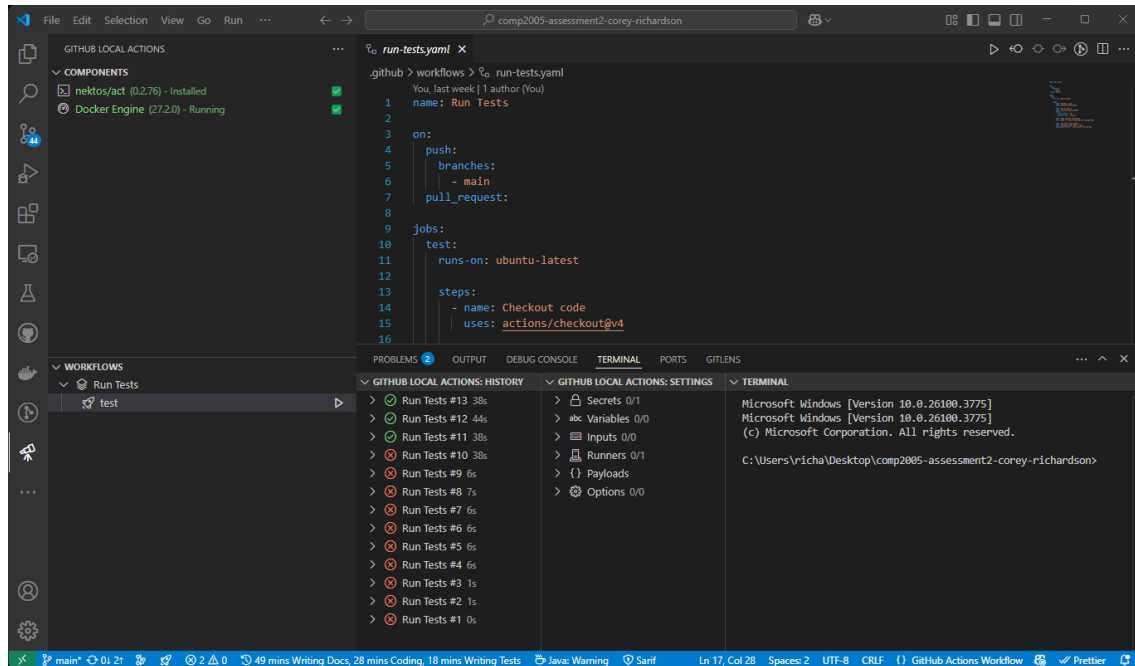


Figure 13: GitHub Local Actions Workflow

GitHub Local
Actions for VS
Code (Sanjula
Ganepola)

<https://marketplace.visualstudio.com/items?itemName=SanjulaGanepola.github-local-actions>

Evaluation

Unit and Integration tests were successfully used here to create a robust product. The use of a TDD approach ensured I wrote code that “works correctly” as opposed to just “works”. The API side of the project can be tested completely automatically, as it is done on every push to the GitHub repository using a GitHub Actions workflow. The GUI application was successfully tested by an external user and advice was given and implemented to improve the User Interface and User Experience. The metrics show that the code is sufficiently covered by tests, reducing the risk of tests missing a potential error, however, as with any software application, it is not possible that all bugs can be accounted for through testing.

I was able to use of range of software tools and frameworks during the development and testing of this project such as IntelliJ IDEA for coding, Docker Engine, nektos/act and GitHub Actions for CI/CD, Mockito for mocking any external dependencies and JUnit for creating and running unit and integration tests. As it was my first time using a few of these tools, I am happy with how they have been used to create this product.

Appendix A: Unit and Integration Test Tables

The following tables list each test case and explain their purpose in testing the application.

AdmissionControllerTest.java	
Layer Tested	Controller
Test Types	Integration Tests
returnsExpectedMonth	Checks that the controller returns the expected month string when admissions exist and are returned by the Service layer.
returnsEmptyWhenNoAdmissions	Ensures that the controller returns a fallback message that is passed by the Service layer if no admissions exist.
handlesErrorsGracefully	Checks that the controller returns a fallback message that is passed by the Service layer if it encounters an error or exception.

AdmissionServiceTest.java	
Layer Tested	Service
Test Types	Unit Tests
MonthWithMostAdmissionsTests	
returnsExpectedMonth	Checks that the service method returns the correct datestring/month when admissions are successfully fetched, in form 'YYYY-MM'.
returnsEmptyWhenNoAdmissions	Checks the service returns a fallback message if it fetches no admissions; loudly fails.
handlesErrorsGracefully	Checks that the service returns an expected fallback message if the API fails to fetch any admissions.

AdmissionTest.java		
Layer Tested	Class	
Test Types	Unit Tests	
testParameterlessNotNull		Tests that the parameterless constructor creates an object.
testConstructedNotNull		Tests that the constructor with parameters creates an object.
testSetAndGetId		Tests for GETTER and SETTER methods.
testSetAndGetPatientId		
testSetAndGetAdmissionDate		
testSetAndGetDischargeDate		
constructedAdmissionTest		Test that the constructor with parameters creates an object and assigns expected values to the attributes.

AllocationTest.java		
Layer Tested	Class	
Test Types	Unit Tests	
testParameterlessNotNull		Tests that the parameterless constructor creates an object.
testConstructedNotNull		Tests that the constructor with parameters creates an object.
testSetAndGetId		Tests for GETTER and SETTER methods.
testSetAndGetAdmissionId		
testSetAndGetEmployeeId		
testSetAndGetStartTime		
testSetAndGetEndTime		
constructedAllocationTest		Test that the constructor with parameters creates an object and assigns expected values to the attributes.

ApiHelperTest.java		
Layer Tested	Service	
Test Types	Unit Tests	
HandleRequestTests		
propagateExceptionWhenNot404Error	Checks that if the API call results in a HTTP error other than 404 NOT FOUND, the exception is propagated and the application fails loudly.	
AdmissionTests		
getAllAdmissions_returnsAdmissions	Checks that the method returns a list of Admission objects when the API responds with data.	
getAdmissionById_returnsAdmission	Checks that the method returns a single Admission object when the ID exists.	
getAdmissionById_handles404	Checks that the method returns null rather than throws an exception when an object is not found; 404 NOT FOUND.	
AllocationTests		
getAllAllocations_returnsAllocations	Checks that the method returns a list of Allocation objects when the API responds with data.	
getAllocationById_returnsAllocation	Checks that the method returns a single Allocation object when the ID exists.	
getAllocationById_handles404	Checks that the method returns null rather than throws an exception when an object is not found; 404 NOT FOUND.	
EmployeeTests		
getAllEmployees_returnsEmployees	Checks that the method returns a list of Employee objects when the API responds with data.	
getEmployeeById_returnsEmployee	Checks that the method returns a single Employee object when the ID exists.	
getAllocationById_handles404	Checks that the method returns null rather than throws an exception when an object is not found; 404 NOT FOUND.	
PatientTests		
getAllPatients_returnsPatients	Checks that the method returns a list of Patient objects when the API responds with data.	
getPatientById_returnsPatient	Checks that the method returns a single Patient object when the ID exists.	
getPatientById_handles404	Checks that the method returns null rather than throws an exception when an object is not found; 404 NOT FOUND.	

Comp2005ApiApplication.java		
Layer Tested	Application Context	
Test Types	Smoke Test	
contextLoads	Ensures that the Spring Boot application context starts up correctly.	

EmployeeTest.java		
Layer Tested	Class	
Test Types	Unit Tests	
testParameterlessNotNull		Tests that the parameterless constructor creates an object.
testConstructedNotNull		Tests that the constructor with parameters creates an object.
testSetAndGetId		Tests for GETTER and SETTER methods.
testSetAndGetFirstName		
testSetAndGetLastName		
constructedEmployeeTest		Test that the constructor with parameters creates an object and assigns expected values to the attributes.

PatientControllerTest.java	
Layer Tested	Controller
Test Types	Integration Tests
NeverAdmittedTests	
returnsExpectedPatients	Checks that when the service returns a list of patients, the controller successfully propagates this list.
returnsEmptyListWhenNoPatients	Checks that if the services returns an empty list, the controller layer does so too to ensure no null unexpected behaviour.
handleServiceFailure	Simulates a service failure and ensures that the controller layer handles it gracefully by returning an empty list rather than an exception.
ReadmittedWithinSevenDaysTests	
returnsExpected	Checks that when the service returns a list of patients, the controller successfully propagates this list.
returnsEmptyListWhenNoPatients	Checks that if the services returns an empty list, the controller layer does so too to ensure no null unexpected behaviour.
handleServiceFailure	Simulates a service failure and ensures that the controller layer handles it gracefully by returning an empty list rather than an exception.
MultipleStaffTests	
returnsExpected	Checks that when the service returns a list of patients, the controller successfully propagates this list.
returnsExpectedDifferentAdmissions	Checks that if a patient has multiple admissions with different Employees allocated the method still returns the expected response.
returnsEmptyWhenNoMultiples	Checks that if the services returns an empty list, the controller layer does so too to ensure no null unexpected behaviour.
handleServiceFailure	Simulates a service failure and ensures that the controller layer handles it gracefully by returning an empty list rather than an exception.

PatientServiceTest.java	
Layer Tested	Service
Test Types	Unit Tests
NeverAdmittedTests	
returnsExpected	Checks that the service method returns the expected patients from a list of mocked patients and admissions.
testBoundaryBehaviour	Tests that the actual behaviour matches the expected behaviour at the boundary value of 7 days. Admissions of lengths 6:23:59:59, 7:00:00:00 and 7:00:00:01 are checked - only the patient information related to the first two admission lengths should be returned by the service method. This tests an edge case.
returnsAllWhenNoAdmissions	Checks that the service method returns the expected (all given) patients from a list of mocked patients and an empty list of admissions.
returnsEmptyWhenNoPatients	Checks the service returns a fallback message if it fetches no admissions; loudly fails.
handlesErrorsGracefully	Checks that the service returns an expected fallback message if the API fails to fetch any admissions.
ReadmittedWithinSevenDaysTests	
returnsExpected	Checks that the service method returns the expected patients from a list of mocked patients and admissions.
returnsEmptyWhenNoPatients	Checks the service returns a fallback message if it fetches no admissions; loudly fails.
handlesErrorsGracefully	Checks that the service returns an expected fallback message if the API fails to fetch any admissions.
MultipleStaffTests	
returnsExpected	Checks that the service method returns the expected patients from a list of mocked patients, allocations and admissions.
returnsExpectedDifferentAdmissions	Checks that the service method returns the expected patients from a list of mocked patients, allocations and admissions. In this test, it tests the case where multiple employees are allocated across two separate admissions, whereas the first test only checks a single admission.
returnsEmptyWhenNoMultiples	Checks the service returns a fallback message if it fetches no admissions; loudly fails.
handlesErrorsGracefully	Checks that the service returns an expected fallback message if the API fails to fetch any admissions.

PatientTest.java		
Layer Tested	Class	
Test Types	Unit Tests	
testParameterlessNotNull	Tests that the parameterless constructor creates an object.	
testConstructedNotNull	Tests that the constructor with parameters creates an object.	
testSetAndGetId	Tests for GETTER and SETTER methods.	
testSetAndGetFirstName		
testSetAndGetLastName		
testSetAndGetNhsNumber		
constructedPatientTest	Test that the constructor with parameters creates an object and assigns expected values to the attributes.	

Appendix B: Before Usability Test UI Design Considerations

Before the Usability Testing has taken place, I have already iterated through a couple of options when it comes to the design of the GUI application. These factors have been considered:

- The first draft of the UI used a 'JTextArea' rather than a 'JTable'. This implementation utilised a 'StringBuilder' to append the Patient ID, full name and NHS number, each displayed on a new line. See commit `088ece6`.
- This was later changed to be a 'JTable' in commit `50a5243`. Different columns and column orders were considered:
 - Patient ID, NHS Number, Forename, Surname
 - Patient ID, NHS Number, Full Name
 - Patient ID, NHS Number, Surname, Forename
 - Patient ID, Forename, Surname, NHS Number
 - Naming 'Patient ID' as just 'ID'

I consider the current order to be the best choice as it begins with the two unique columns, followed by name in-order. If sorting was implemented, I may opt for the Surname-Forename option as this would then be in order of most optimal sorting keys for finding a patient.

- The first draft of the UI had a WIDTH:HEIGHT ration of 400:600px. I found that this limited the number of patients that could be displayed on the screen at a time so then opted for a choice that had a HEIGHT greater than the WIDTH – originally 600:400px. I then updated this to instead use the Golden Ratio (1.618). As such, HEIGHT is now calculated as $\text{(int)} (\text{WIDTH} * 1.618)$.
- When first changing to the 'JTable' implementation, all rows were originally the same colour. To improve the readability of the display, I updated this so that alternating rows have a light grey colour, which still provide a clear enough contrast against the text to meet accessibility requirements.