COMP2005 Assessment 2: Report

Assessment Materials

D2 GitHub Repository:	https://github.com/Plymouth-University/comp2005-assessment2-corey-richardson
D3 YouTube Link:	

Unit and Integration Testing for Part A

Each layer of the application was tests 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.

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

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.

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 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.

```java
// https://stackoverflow.com/questions/450807/how-do-i-make-the-method-return-type-generic
private <T> T handleRequest(String url, Class<T> responseType) {
    try {
        return restTemplate.getForObject(url, responseType);
    } catch (HttpClientErrorException e) {
        if (e.getStatusCode() == HttpStatus.NOT_FOUND) {
            return null;
        }
        throw e;
    }
}
```

```java
@SpringBootTest
class ApiHelperTest
{

    @MockitoBean
    private RestTemplate restTemplate;

    @Autowired
    private ApiHelper apiHelper;

    private Admission mockAdmission = new Admission();
    private Allocation mockAllocation = new Allocation();
    private Employee mockEmployee = new Employee();
    private Patient mockPatient = new Patient();

    private Admission[] mockAdmissions;
    private Allocation[] mockAllocations;
    private Employee[] mockEmployees;
    private Patient[] mockPatients;

    @Nested
    class HandleRequestTests {

        // https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/HttpClientErrorException.html
        @Test
        void propagateExceptionWhenNot404Error() {
            when(restTemplate.getForObject("https://web.socem.plymouth.ac.uk/COMP2005/api/Patients/1", Patient.class))
                    .thenThrow(new HttpClientErrorException(HttpStatus.BAD_REQUEST));

            HttpClientErrorException e = assertThrows(HttpClientErrorException.class,
                    () -> apiHelper.getPatientById(1)
            );

            assertEquals(HttpStatus.BAD_REQUEST, e.getStatusCode());
        }
    }

    // ...

}
```

Figure 1: Example of a Test Class, Sub-class and Method, ApiHelperTest::HandleRequestTests::propagateExceptionWhenNot404Error()

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 tests, 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.

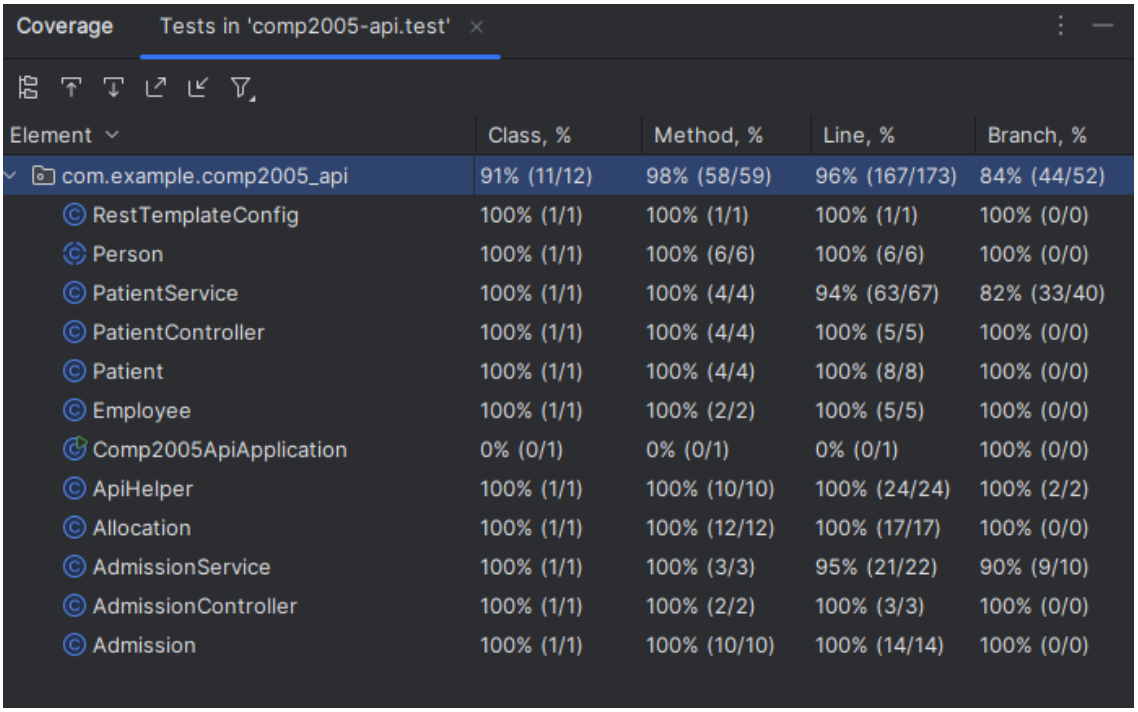
To assist with the implementation of F2, F3 and F4 for Part A of the task, I used a Test Driven Development (TDD) approach, where I wrote failing tests for each method before implementing the methods. The TDD workflow consists of 3 stages:

- 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 the "works correctly".

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

Metrics and Code Coverage



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 2: 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

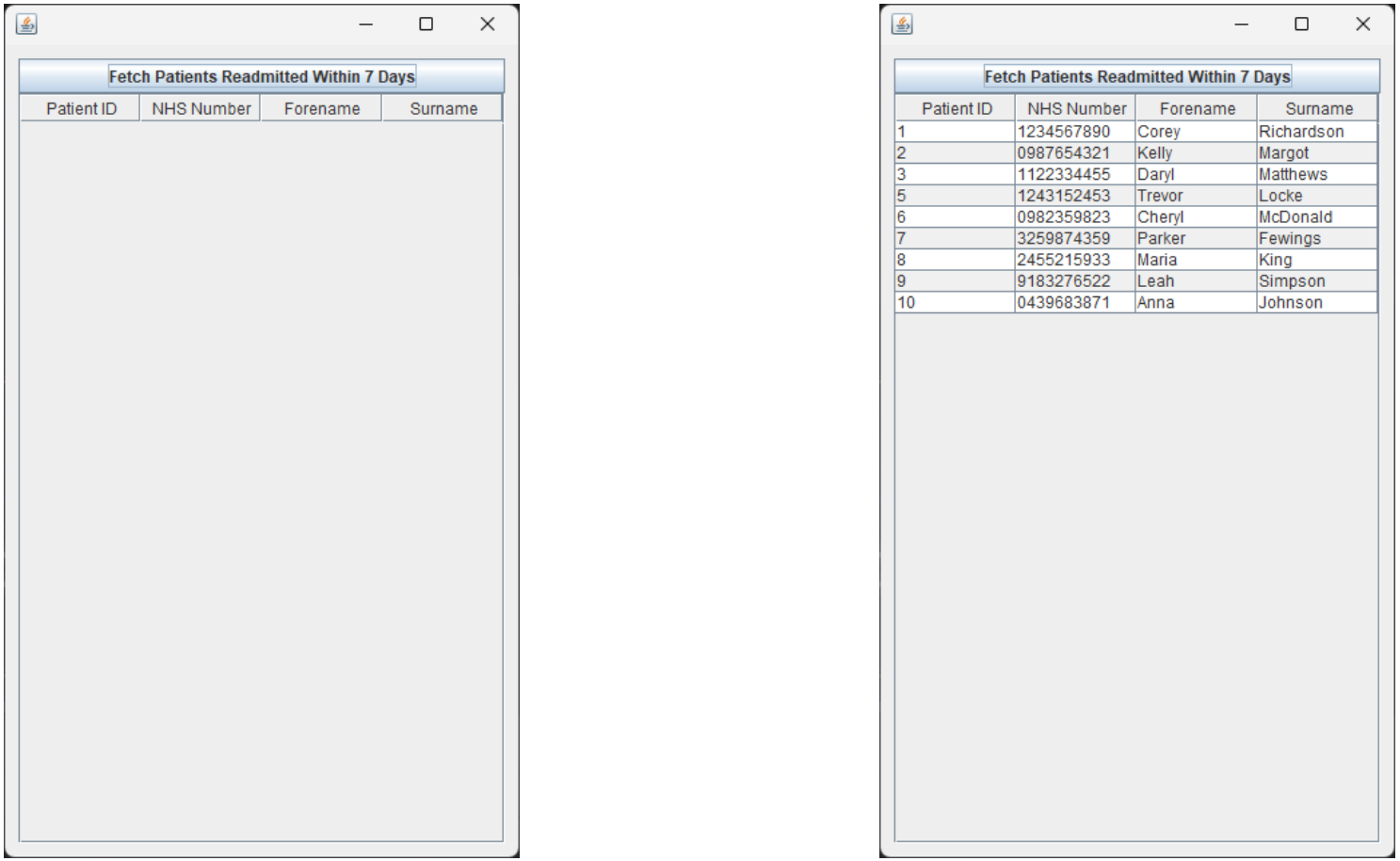


Figure 3: GUI Application before Usability Testing

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 `(int) (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.

System Testing

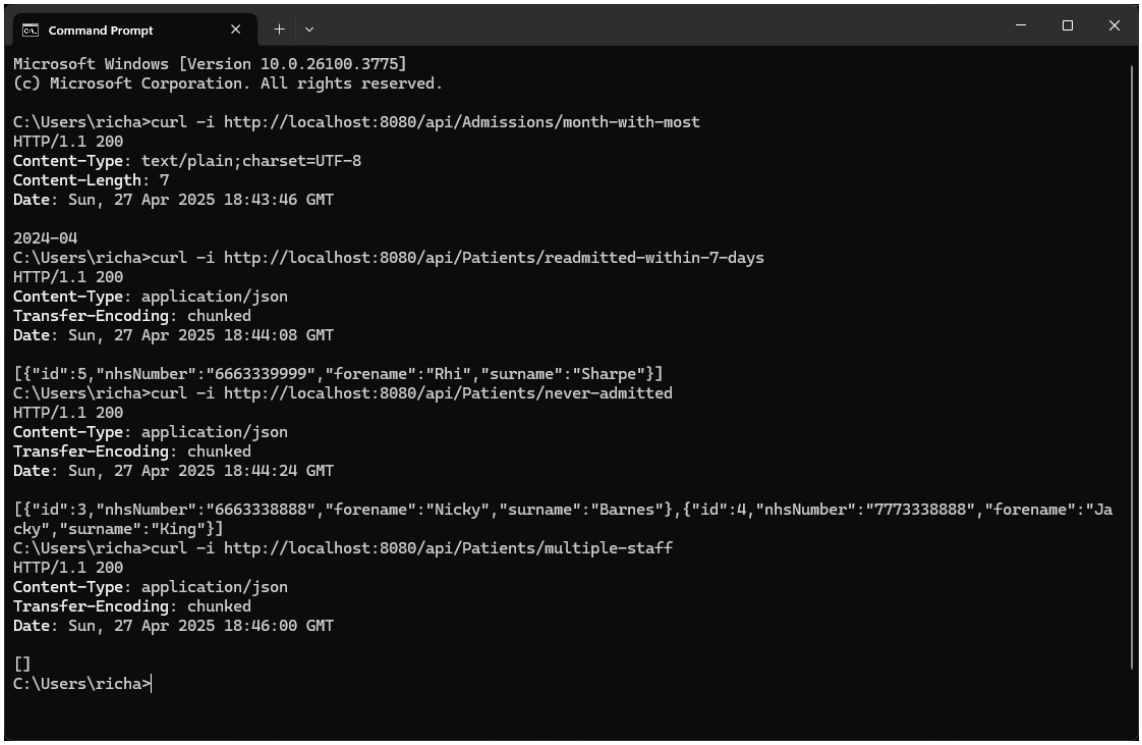


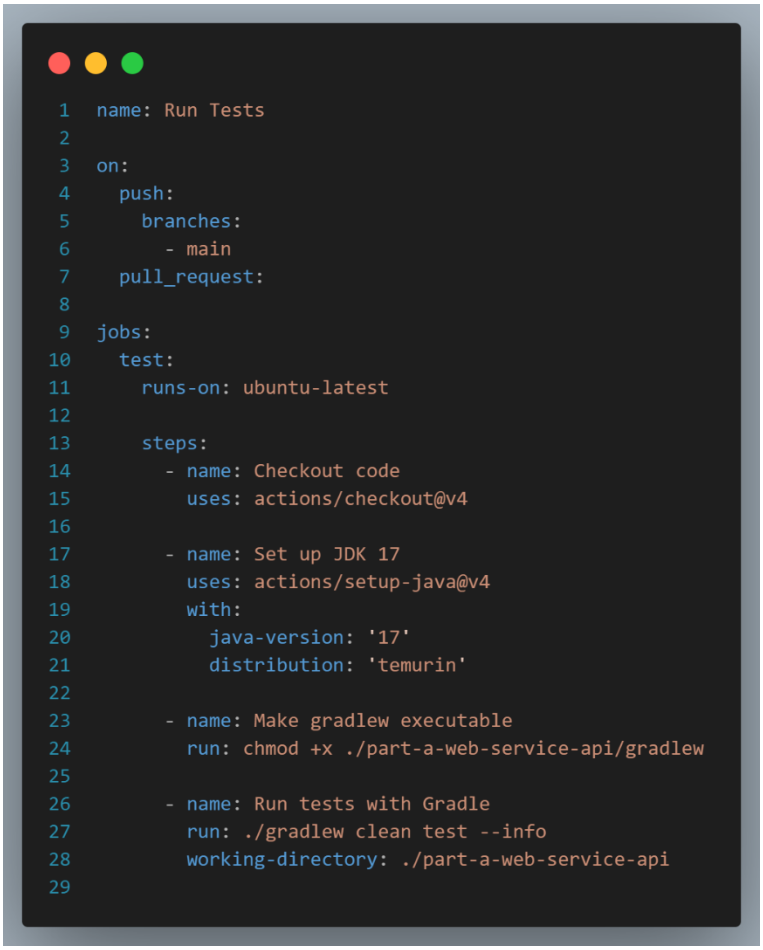
Figure 4: curl Requests to the API Endpoints

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**.



```yaml
name: Run Tests

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Make gradlew executable
        run: chmod +x ./part-a-web-service-api/gradlew

      - name: Run tests with Gradle
        run: ./gradlew clean test --info
        working-directory: ./part-a-web-service-api

```

Figure 5: GitHub Actions Workflow

The workflow ensures that it is testing the latest pushed code before setting up a Java 17 environment and ensuring that the Gradle Wrapper file is executable. It then runs the test suite 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.

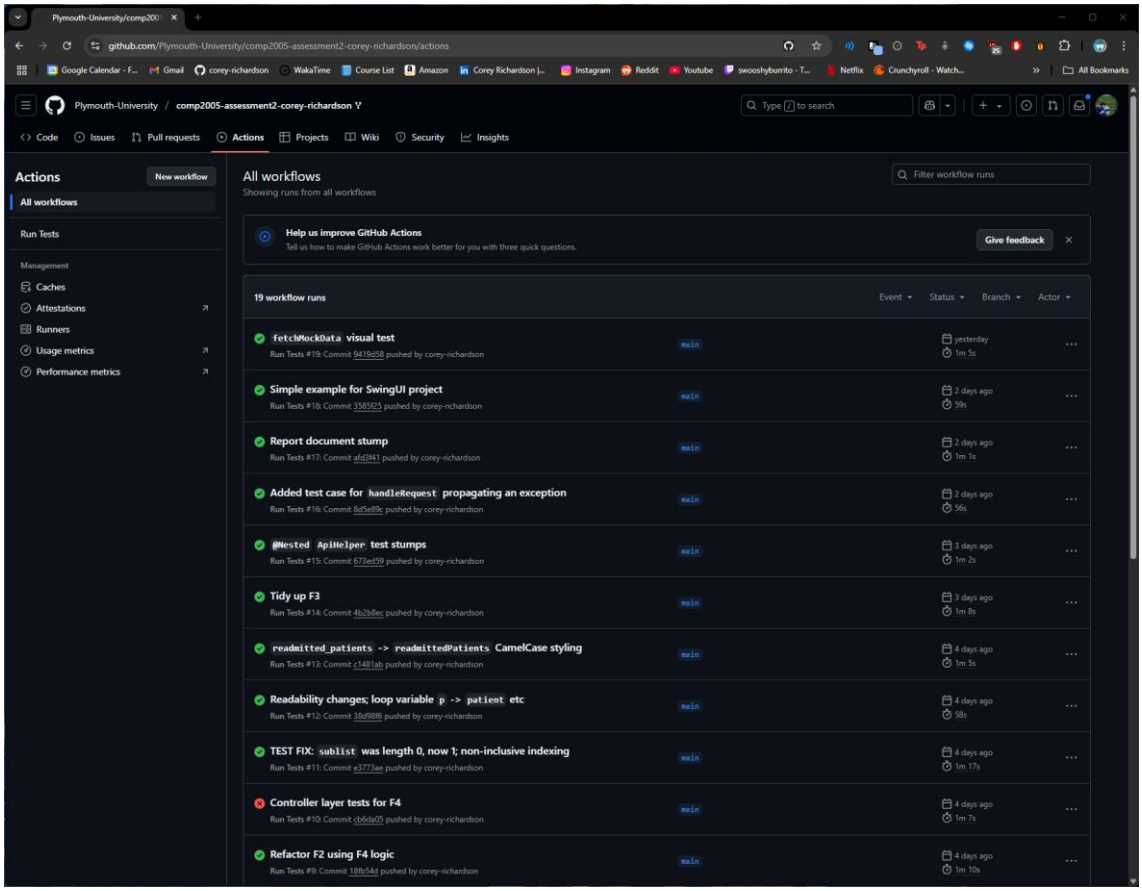


Figure 6: GitHub Actions Page

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 ‘nektons/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.

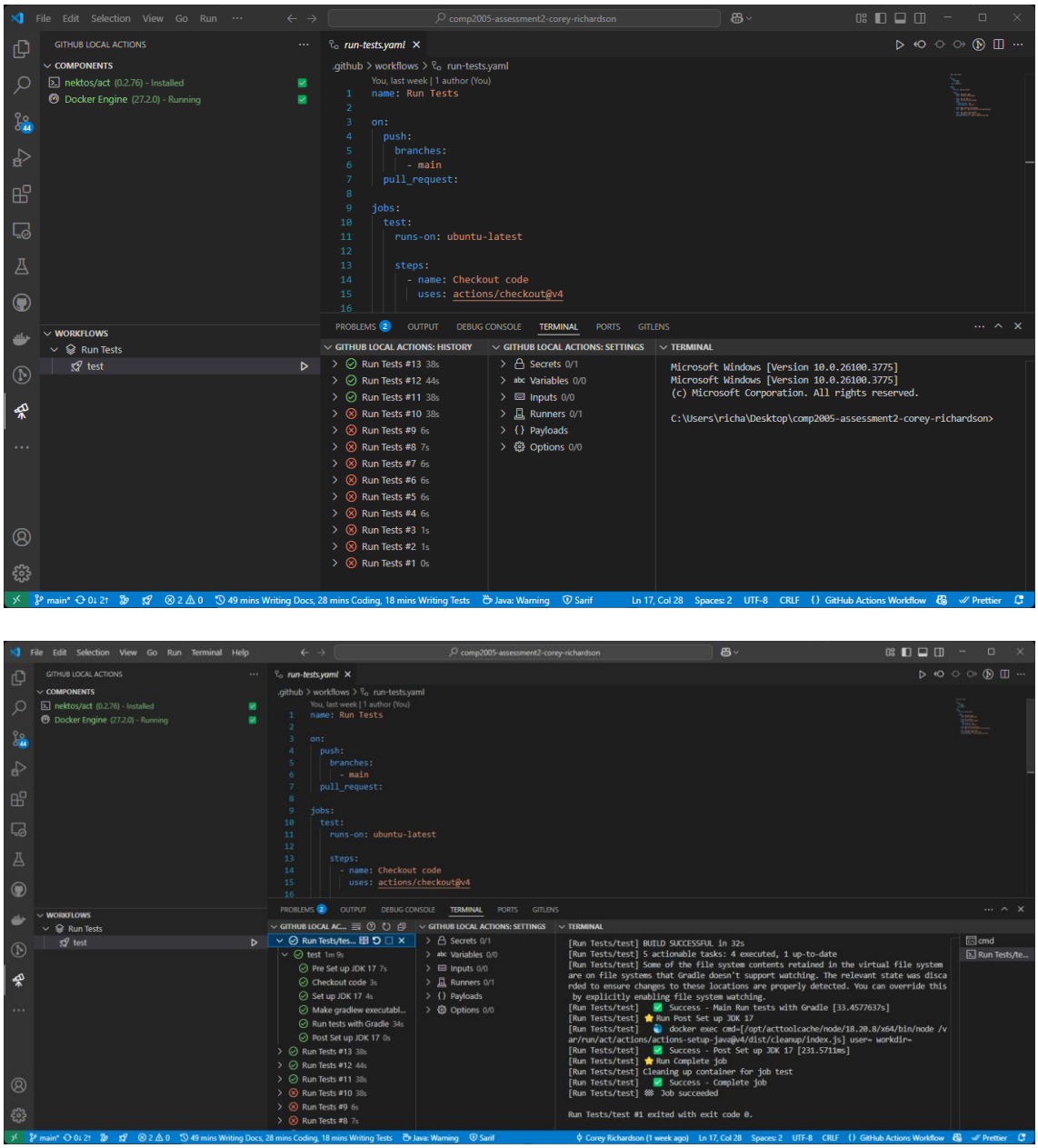


Figure 7: GitHub Local Actions Workflow

GitHub Local Actions for VS Code (Sanjula Ganepola)	https://marketplace.visualstudio.com/items?itemName=SanjulaGanepola.github-local-actions

Evaluation