Sparta API Testing Project

GitHub

Contents

Setup Instructions

The following setup instructions are for Windows operating systems.

- Clone the repository to you local machine
- Open `/Test-Framework/` in your IDE and build the Maven project.

For greater repeatability of tests, I advise you run the tests against a containerised version of the API. To do this

- Download and unzip `SpartaAcademyApp.zip`
- Follow the instructions in `SpartaAcademyApp/README.md` to set up the Docker image.

The framework is set to run using the Docker image unless it is running on a Linux OS. Thi is so that the tests run against the live API during GitHub action workflow runs. The code responsible for this is in `Test-Framework/src/test/java/com/sparta/ojg/SharedState.java` .

User Guide

The tests can either be run by scenario/feature from the .feature files in `Test-Framework/src/test/resources/features` or all tests can be run by running `Test-Framework/src/test/java/com/sparta/ojg/TestRunner.java` . Running the tests from the TestRunner class has the benefit of generating a Cucumber report.

Tests are organised using tags by:

Feature

- `@Feature1`
- `@Feature2`
- `@Feature3`
- `@Feature4`

- `@Feature5`
- `@Feature6`
- `@Feature7`
- `@Feature8`

Type of HTTP request

- `@Get`
- `@Post`
- `@Delete`
- `@Put`

Happy or sad path

- `@Happy`
- `@Sad`

Whether it is testing the status code, the message in the response body, or that the requested operation has been successful or not

- `@StatusCode`
- `@Message`
- `@Function`

Tests can be run or not by adding/removing the relevant tags from the `@IncludeTags` annotation in the TestRunner class, or to run combinations of tags, add them to the Constants.FILTER_TAGS_PROPERTY_NAME configuration parameter.

For example, to run all tests annotated with **both** the `@Get` and `@Happy` tags, your TestRunner class should look like this

```
package com.sparta.ojg;

import io.cucumber.junit.platform.engine.Constants;
import org.junit.platform.suite.api.*;

@Suite
@IncludeTags({})
@SelectClasspathResource("features")
@ConfigurationParameter(key = Constants.GLUE_PROPERTY_NAME,value = "com.sparta.ojg.stepdefs")
@ConfigurationParameter(key = Constants.FILTER_TAGS_PROPERTY_NAME, value = "@Get and @Happy")
public class TestRunner {

}
```

API Documentation

Here

Testing Strategy

Before writing any test cases/user stories, a 1hr exploratory testing session was conducted to quickly identify obvious defects and to familiarize myself with the Sparta API, the test charter can be found here.

A GitHub project board was used to organise the project. This was divided into the following columns:

- ToDo: The project backlog
- In Progress: Items currently being worked on
- Done: Completed items
- Defects: defects identified during exploratory and automated testing
- Project DoD: The post requisites for the project to be considered completed.

An item was created for each feature to be tested, with these correlating to each endpoint of the API, and added to the ToDo column, and moved between columns appropriately. The tests for each feature were split into scenarios, written in Gherkin syntax.

The tests were written to assert correct functionality, response status codes, descriptive messages in response bodies, and input validation. Additionally, although the Swagger documentation suggests no authentication is required for any API calls, some tests were written to assert whether a valid bearer token is required for successful data read/writes. Each feature was given a priority value from P0 to P2 based on how critical the feature is, for instance get single Spartan was given a higher priority than get all Spartans, since so long as the former works correctly it can be used to substitute the latter. The priorities were also based on industry regulations, for example delete Spartan was given priority P0 since GDPR regulations stipulate that a person's data must be deleted when it is no longer needed.
The more specific test strategy for each feature is outlined below:

Obtain bearer token: /Auth/login (POST)

Verify that a token can be obtained from this endpoint when the request body contains a correctly formatted, valid username and password, but not if username or password are invalid.

View all courses: /api/courses (GET) and View all Spartans: /api/spartans (GET)

This endpoint does not take any parameters, so tests were just written to assert that the request is successful if a valid bearer token is provided in the request, and not otherwise.

View a specific course: /api/courses/{courseId} (GET) and View a specific Spartan: /api/spartans/{spartanId} (GET)

- Check that data can't be accessed without providing a valid bearer token with the request.
- Check for appropriate response when requesting a non-existing course.
- Check for appropriate response when request contains invalid parameter.
    - Only one test is written for this, where the `courseId` / `id` parameter is given as a string.

Create a new Spartan: /api/spartans (POST)

- Requests with valid Spartan data should be successful (200 status code response, informative message in response, Spartan data can be accessed in subsequent GET requests to Spartans endpoint).
- Requests with invalid Spartan data should be successful (400 status code response, informative message in response, Spartan data does not appear in subsequent GET requests to Spartans endpoint).

> The framework currently only has 3 test cases for invalid data, but more can easily be created by writing the invalid data as a JSON file, and adding to path to the file to the table's in `Test-Framework/src/test/resources/features/createSpartan.feature`

Update an existing Spartan: /api/spartans/{spartanId} (PUT)

The tests for this feature were similar to those for creating a new Spartan, with the addition of tests to ensure that a user cannot update an existing Spartan's ID to an ID that is already taken by another Spartan.

Delete an existing Spartan: /api/spartans/{spartanId} (DELETE)

- Verify delete Spartan correctly affects database and receives response with appropriate status code and informative error message.
- Verify attempts to delete non-existing Spartan don't affect database and receives response with appropriate status code and informative error message.

The full list of tests, with links to the respective items on the project board, is in the table below:

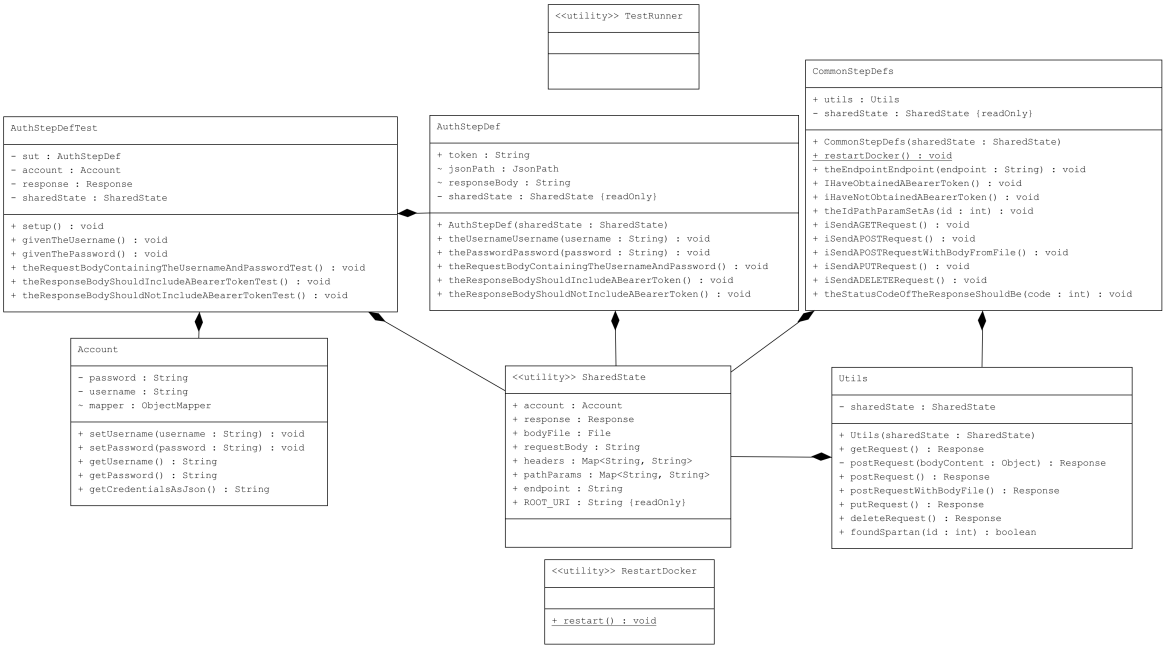
Features, Scenarios, and Defects

Feature	Scenarios	Defects
Feature 01: Obtain bearer token	**Scenario 01.01** Request bearer token with correct username and password and verify status code of response	**Defect 01.01.01** Incorrect status code for attempted login with invalid credentials
	Scenario 01.02 Request bearer token with correct username and password and verify response body contains token	
	Scenario 01.03 Request bearer token with incorrect username and verify status code of response	
	Scenario 01.04 Scenario: Request bearer token with incorrect username and verify response body doesn't contain token	
	Scenario 01.05 Request bearer token with incorrect password and verify status code of response	
	Scenario 01.06 Request bearer token with incorrect password and verify response body doesn't contain token	
Feature 02: Get all courses	**Scenario 02.01** Get all courses with auth	

Feature	Scenarios	Defects
	Scenario 02.02 Get all courses without auth	**Defect 02.02.01** Get courses works without auth
Feature 03: Get single course	**Scenario 03.01** Request an existing course by ID and verify status code of response	
	Scenario 03.02 Request an existing course by ID and verify request body contains requested data	
	Scenario 03.03 Request a non-existing course by ID	**Defect 03.03.01** GET single course/Spartan incorrect status codes
	Scenario 03.04 Request an existing course by ID without auth	**Defect 03.04.01** Get single course works without auth
	Scenario 03.05 Request a course with invalid ID	
Feature 04: Get all Spartans	**Scenario 04.01** Get all Spartans	
	Scenario 04.02 Get all Spartans without auth	
Feature 05: Create Spartan	**Scenario 05.01** Create Spartan with valid data and verify status code	
	Scenario 05.02 Create Spartan with valid data and verify response body matches request data	
	Scenario 05.03 Create Spartan with valid data and verify it appears in subsequent GET requests	
	Scenario 05.04 Create Spartan with invalid data and verify response status code	
	Scenario 05.05 Create Spartan with invalid data and verify Spartan isn't added to database	
	Scenario 05.06 Create Spartan with invalid data and check response for descriptive error message	
Feature 06: Get single Spartan	**Scenario 06.01** Request existing Spartan by ID and verify status code of response	
	Scenario 06.02 Request existing Spartan by ID and verify response body contains requested data	
	Scenario 06.03 Request a non-existing Spartan by ID	**Defect 06.03.01** Get non-existent spartan gives wrong status code
	Scenario 06.04 Request Spartan with invalid ID	

Feature	Scenarios	Defects
Feature 07: Delete Spartan	**Scenario 07.01** Delete an existing Spartan by ID and verify status code	**Defect 07.01.01**: DELETE Spartan response doesn't provide any information
	Scenario 07.02 Delete an existing Spartan by ID and verify Spartan no longer exists in database	
	Scenario 07.03 Delete a non-existing Spartan	
Feature 08: Update Spartan	**Scenario 08.01** Update existing Spartan with valid data and verify status code of response	**Defect 08.01.01** Incorrect status code for successful update spartan
	Scenario 08.02 Update existing Spartan with valid data and verify database is updated correctly	
	Scenario 08.03 Update existing Spartan with invalid (missing/incorrect data type) data and verify status code of response	**Defect 08.03.01** Update Spartan operation works when data is missing
	Scenario 08.04 Update existing Spartan with invalid (missing/incorrect data type) data and verify database isn't updated	
	Scenario 08.05 Update non-existing Spartan	**Defect 08.05.01** Update non-existing Spartan returns incorrect response code
	Scenario 08.06 Update existing Spartan's ID to already taken ID	

Framework Architecture



The framework is build using JUnit and Cucumber, with features split into scenarios written declaratively in Gherkin syntax, which can be found here. Each step in the scenarios is tied to a method in one of the StepDef classes by the TestRunner class. StepDefs that are used in multiple different scenarios are in `CommonStepDefs.java`, and functionality that is used by multiple different stepdefs are encapsulated into methods in Utils.java. Data is shared between the StepDef classes using a Cucumber Pico Container, SharedState.java. The functionality of all StepDefs were tested independently of their dependencies using Mockito. These tests can be found/ran in `Test-Framework/src/test/java/com/sparta/ojg/stepDefTests`. The data used in the tests is all written in JSON files found here.

CI/CD

A GitHub action workflow was created and can be found here. This runs on pushes and pull requests to `main`, and builds the maven project, and runs the tests, so that the pull request is only successful if all tests pass.

Contribution Guidelines For Future Testers

Future testers should consider:

- Adding more test cases to check correct data validation for path parameters and request bodies for POST and PUT requests. This can easily be achieved by adding them to the data tables in the .feature files.

- The product owner should be contacted to discuss the status codes provided by the API, as in many cases the codes received and the most suitable. Similarly, they may want to consider adding more informative messages to API responses.
- The create Spartan feature needs fixing.
- The Swagger documentation should be updated to correctly provide level of authentication needed for API calls.
- Create POJO objects for Spartans and courses for more thorough comparison of expected and actual data.