Testing Strategy

Throughout the testing process, equivalence class testing techniques are applied, be it for unit tests, integration tests, system tests, white-box tests or black-box tests.

White-box tests are used for unit tests of key functions and files which are extensively used by each subsystem, while black-box tests are used for unit testing the user interfaces and integration testing of the application.

White-box tests must be automated to provide statistics for code coverage, while black-box tests are partially automated for integration testing of certain key use cases for easier replication of results. Overall, it is hoped that this testing strategy will bring about a good mix of system robustness and easy test replication.

Frontend Testing

We used Jest and React Testing Library to automatically test the React web app.

Unit Testing

For the unit testing, we performed snapshot testing for the majority of the React components. This is to ensure that the UI produced by the React components doesn't change when we modify the code.

We also mocked the API endpoints to ensure that the React components properly consume the API endpoints.

We also performed additional tests for the teaching and learning platforms to ensure that the editing buttons are only displayed when the teaching content is editable. For example, students are not allowed to edit the teaching content, so everything on the learning platform is not editable. Additionally, if a level is already published and made play-able to students, the learning materials and questions of the level will also no longer be editable.

The above unit tests for each component can be found in the 'index.test.js' files in their respective component folders inside the parent 'src/components' folder.

Integration Testing

For the integration testing, we tested the platforms as a whole to check the interfaces between the components. There are 3 integration tests, one for each platform (teaching, learning and admin). For each integration test, we used jest to mimic a user navigating through all the different pages and hence different components of the system. These tests ensured that the links between the different components were working together seamlessly. For the integration testing, it still used mocked API endpoints to ensure that the React components properly consumed the API endpoints.

Backend Testing

Integration Testing

Integration tests are carried out to test if the different layers of the module are working properly. The layers involved in each module are the controller, service, repository layer and also the backing MySQL database to see if data can be stored and retrieved properly. A series of blackbox testing and basis path testing is carried out. For example, blackbox testing is carried out using equivalence classes of values where a correct password can be provided to test the login module to see if a status code of 400 is returned. The inverse, where the correct password is provided should see the test return a status code of 200 where a token is returned.

A series of basis path testing is also used, where an API endpoint only allows users to modify a resource that is created by them. For example, a user is only allowed to modify their own progress context of a game. This check is performed in the service layer. As such, two integration tests are performed to ensure that a user is able to modify their own progress context, and a separate test will be performed to make sure that no other user other than the owner of the resource is able to modify the progress context.

Each integration test yields an API endpoint documentation of how to use the endpoint using Spring Rest Docs, which combines hand-written documentation written with Asciidoctor and auto-generated snippets produced with Spring MVC Test. This approach helps us to produce documentation that is accurate, concise, and well-structured. The documentation then allows your users to get the information they need with a minimum of fuss.

The use of Spring REST Docs in this Project allows us to create APIs, add behavior tests and generate documentation of it, all in one same process. Given the necessity of an API, we can write the test and its contract in the form of documentation. It would be a way to develop based on tests, hence achieving the goal of test-driven development.

System Testing

Functional Testing

We used Python and Selenium to perform automated functional testing on the entire system. Selenium was used to simulate a user accessing the frontend through an internet browser. The simulated user would then perform a series of tasks, including creating topics and playing the game. These tasks would send requests to the API endpoints of the backend. Hence, this testing would ensure that the whole system could be used as a whole to perform the functionalities listed in the functional requirements of the software requirements specification.

Non-functional Testing

Jmeter is used to create and run load testing on the backend. All ‘fetching’ endpoints are used, starting from fetch topics to game map, learning materials, question and progress. This is done as the backend is deployed with preloaded data in the database. Currently, load testing is done in a way such that a total of 10 users would try to execute their own ordered actions at the same time. This is done to ensure that up to 10 users are able to use the backend at the same time. Upon running the test, a log file is generated that contains the specific details of the test done, such as how long the backend took to service the request from a user, and whether the request failed.