# 5 Testing

## 5.1 Unit Testing

**Frontend:** We will be unit testing our react components, pages, and UI.

To test the components and their functionality, we will use Jest test cases in order to check that the components are working as they should. We will use a combination of Jest and Enzyme to test the pages and UI to make sure that the frontend renders correctly as it should.

**Backend:** The units we are going to test are the database, API's and server.

To test the databases we are going to be using MySQL workbench because it is a very popular open source database management tool for MySql databases which we are using. With this we can write queries, execute them and then get the results to test the database.

For API's we are going to be using Postman to create our tests and then set it up in the CI pipeline. This will ensure that there is end to end functionality of the web application. Later on we will also need to test the security of API's as we will need to secure specific information going over HTTP.

We will also need to be testing the performance and load testing of the database and servers, the most popular load testing tool we found was HammerDB which supports the most popular databases including MySQL databases.

## 5.2 Interface Testing

**Frontend:** In order to accomplish live messaging and updating during live lectures, the frontend and backend will be communicating through WebSockets. This is a very critical interface, as it is the fundamental interface that enables the primary feature of our application. This interface will be tested through the creation of a mock frontend data consumer and a mock backend data producer on the same machine, which will then use our WebSocket infrastructure to communicate and verify successful messages. The mocking features of Jest can be used to accomplish this, as our frontend and backend are both written in JavaScript.

We will also be creating a test that will both post a message/discussion to an archived lecture and then test that we can retrieve that message and it is categorized with the correct lecture and date. This will be tested using a local server as to not actually store the test messages in the database. We will be setting up these tests as we establish connections between the frontend and the backend.

**Backend:** For our backend we have essentially 2 interfaces. The first interface is our express application, and our second interface is a mysql database. Our express server communicates with our database for many of our endpoints so testing is a high priority. To test this interface we have created a local script that creates a local database and starts our express server locally. The script automatically adds data to our database and we test the express server to database connection by calling our various endpoints. In the future we plan to add a stage to our deployment pipeline that will check if our health endpoint works. The health endpoint will simply check if the express server and database can communicate properly. If this test fails the pipeline will fail.

**Tools:**

- Docker
- Bash

## 5.3 Integration Testing

**Frontend:** The most critical integration path relevant to the frontend is the loading and routing between various pages. Since we are using React with a single root element, this isn't quite as straightforward, and special care will need to be taken to ensure it works well. Automated testing of this integration would come down to simply loading an example nested page (i.e. not the root page) and verify that it loaded correctly. In order to load a page like this, the routing logic would have to run successfully in sync with the page loading/rendering, so this will work as an integration test.

Besides page loading/routing, proper display of fetched backend data is the other important integration. Testing the exact rendering of the data in the DOM would likely be excessive, but there are other strategies we will leverage to test this integration. For example, when testing the display for the list of courses, we will test to ensure that the proper number of courses is being displayed.

**Backend:** The most important components that will need to have integration tests are the ability to call the APIs, then for the APIs to be able to fetch data from the database and return it to the calling user. A great example of this would be to stand up a "Health" endpoint that ensures it is able to connect to the database, grab some data, perform some sort of simple calculation on it, and then return a success or failure result. In order to test this, and other similar but possibly more simple tests (i.e. only testing the integration of the Express app and the database), our main tool will be Jest.

**Tools:**

- Jest

## 5.4 System Testing

The root system of our application is the operation of live lectures, and the most essential system-level interaction that needs to function correctly is the start-to-finish sending and receiving of messages from users. If that works correctly, then we can have much higher confidence in our application, as all communication during live lectures takes the form of a "message," whether that is a conventional text message or a multiple-choice poll. For unit testing, testing the frontend message producers, backend message processors, and frontend message consumers would suffice. Also, for the backend, testing our api controllers will suffice since these controllers handle all of the logic for our apis. For integration testing, testing the successful operation of WebSockets will ensure correctness for communication. Additionally, the backend will test its integration with the front-end -> server -> database by having a health check on deployments. This check will call a health endpoint that will return success if the api call succeeds in making a database connection.

**Tools:**

- Jest
- Docker
- Bash

## 5.5 Regression Testing

**Frontend:** Along with the use of CI/CD and standard agile development methodologies, the frontend can assure that visually the app is rendering as it should through the use of snapshots. Enzyme and Jest give us the ability to generate snapshot tests that can determine if the build is rendered correctly. This alongside the use of CI/CD covers that our features being pushed go through without breaking.

**Backend:** We are implementing CI/CD for our web application, it allows us to continuously integrate new software from our developers which helps with delivery and deployment. Regression testing on the CI/CD pipeline can be used to make the newer releases of our application more efficient and operate better because it can reduce execution time, testing and software quality in general. Our testing will be driven by many requirements and one of them is making sure that new features or changes pass all prior unit tests on a development branch before merging to the master branch, which will be our main application. This can be achieved using the CI/CD pipeline we have implemented. With this method we can ensure that the master branch will always be working and won't need to spend lots of time debugging the program if an error occurs before it works again. The main features that should not break as they are critical to our application is going to be security of data and end to end functionality,

## 5.6 Acceptance Testing

**Frontend:** We will show our client the functionality of each of the pages, as well as the ascetic to ensure it matches with what he had in mind. We will also have potential student(s) or testers that will try to navigate our application to make sure it is both intuitive and something that they would enjoy using.

**Backend:** We will show our client the APIs and validate with him that they are providing the expected functionality in an acceptable manner.

## 5.7 Security Testing

**Frontend:** As the frontend will be communicating through HTTPS and will not be responsible for storing sensitive data (minus the session information, which will be stored in browser storage where it's isolated from other webpages), security testing is not applicable for this side of the application.

**Backend:** The backend involves authentication on requests to our server for security. When the user logs in they are given a session and this session is required to make any future api calls to the server. The way we test this is we have added middleware to our server. All requests go through this middleware and the middleware checks if a user has a valid session before the user is able to make a request. We have a local startup script that uses docker to create a local database and then our server is also started locally. We use this local startup to test our apis and make sure that a user can only make a request if they have a valid session.

## 5.8 Results

Currently we have just exited the design and planning phase and are currently in the process of developing the main critical features of our application. So far, we have not performed/written any tests but as we continue to implement our application, we will write tests to ensure functionality according to the requirements and we will have more information to report in the final draft of the design document.