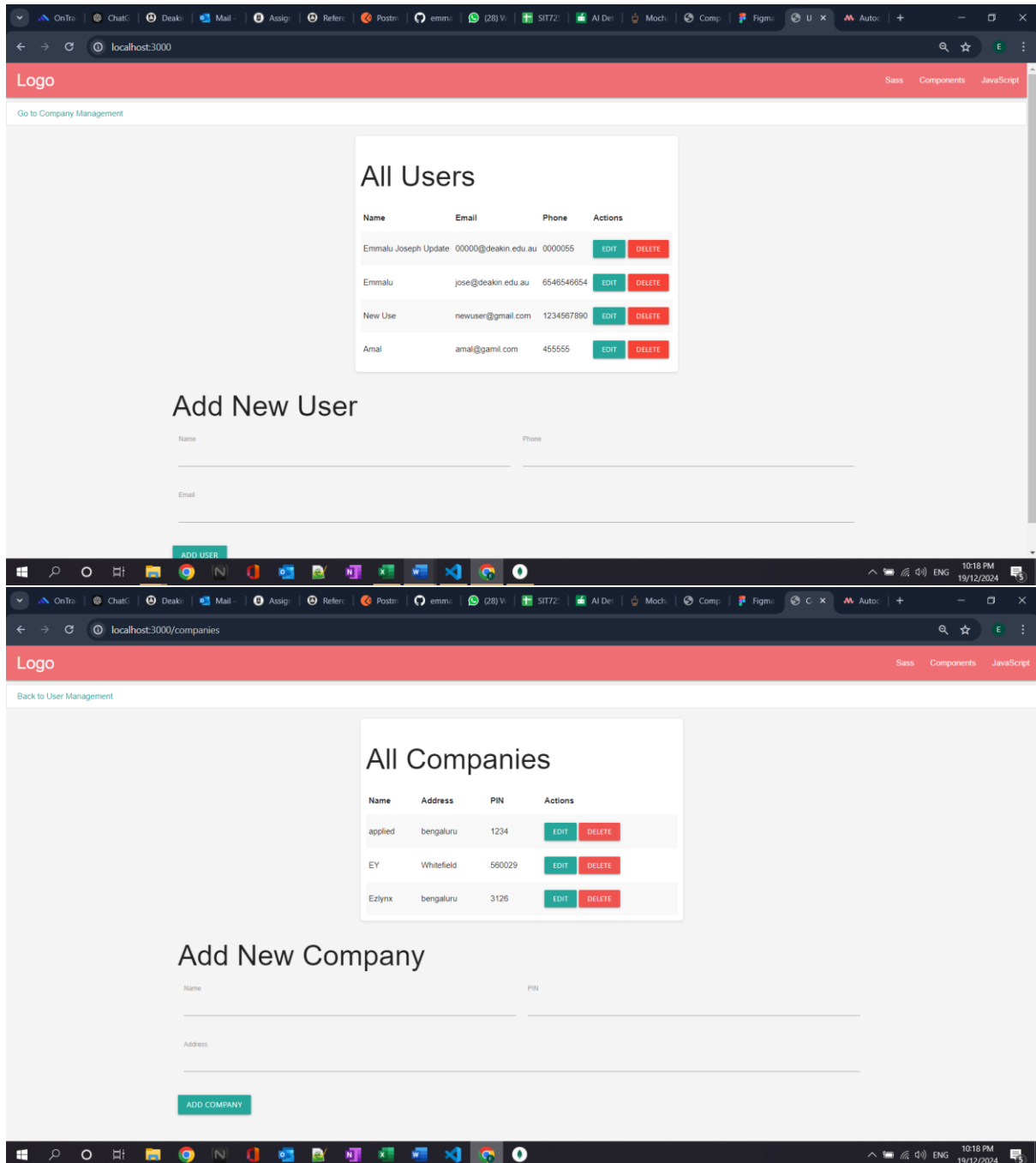
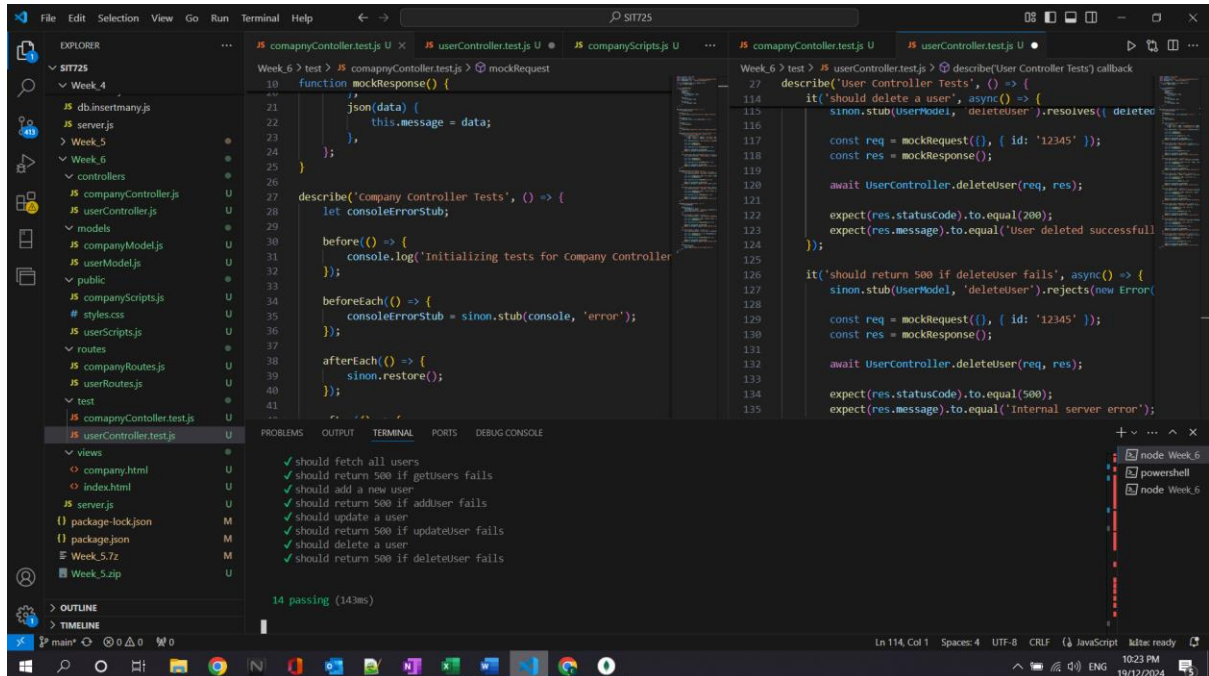


GitHub Link: <https://github.com/emmalujosephwork/SIT725.git>

This is my week 5 web application I have added test cases to this (I have added a few materializecss to the week 5)

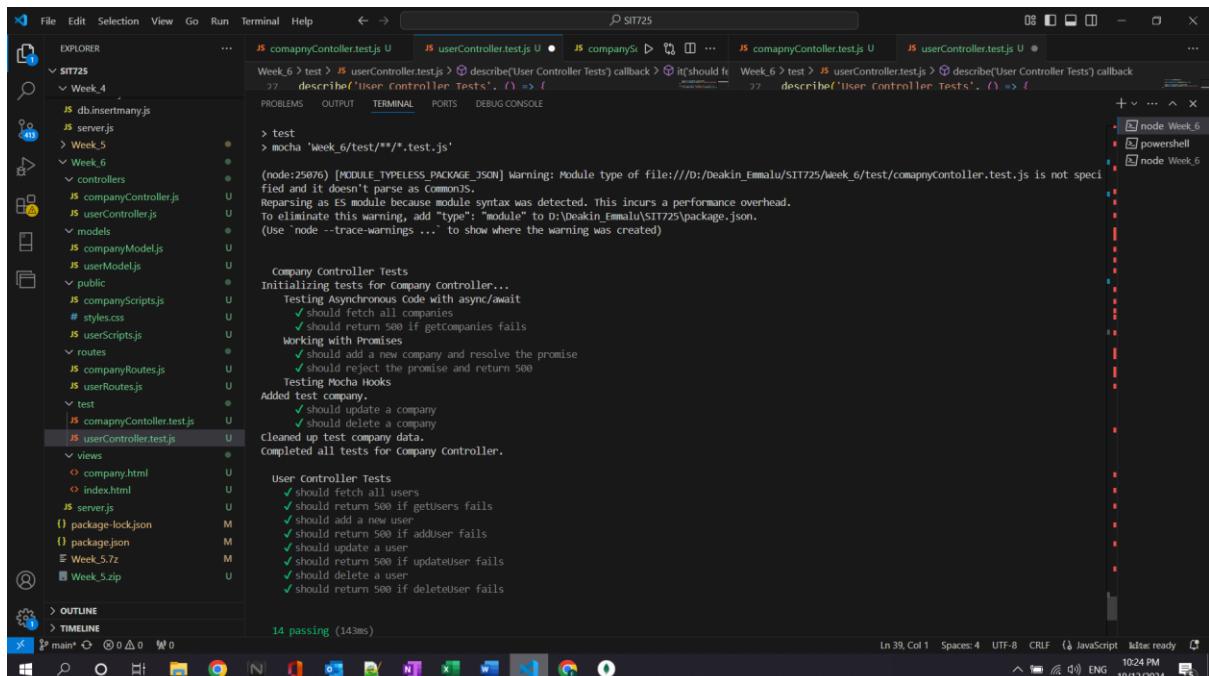


Below I am attaching the screenshot of test case files



The screenshot shows a VS Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure for 'SIT725' with folders for 'Week 4', 'Week 5', 'Week 6', 'controllers', 'models', 'public', 'routes', and 'test'. The code editor displays the contents of 'userController.test.js' and 'companyController.test.js'. The 'userController.test.js' file contains a 'describe' block for 'User Controller Tests' with several 'it' blocks for testing user creation, deletion, and updates. The 'companyController.test.js' file contains a 'describe' block for 'Company Controller Tests' with several 'it' blocks for testing company creation, deletion, and updates. The terminal at the bottom shows the output of the tests, indicating that all tests passed.

Below I am attaching test case file output



The screenshot shows a VS Code editor with a file explorer on the left and a code editor on the right. The file explorer shows the same project structure as the previous screenshot. The code editor displays the output of the tests, showing the 'describe' and 'it' blocks for 'User Controller Tests' and 'Company Controller Tests'. The terminal at the bottom shows the output of the tests, indicating that all tests passed. The output includes a warning about the module type of the file and a message about the performance overhead of the ES module syntax.

Types of Tests in My Test Files

1. Unit Tests

- **What It Does:** These tests focus on checking if individual methods in your controllers (like `getUsers` or `addCompany`) work as expected, all by themselves. They don't rely on the actual database or other systems.
- **Examples:**
 - Testing if `getUsers` correctly returns a list of users.
 - Testing if `addCompany` adds a new company and responds with success.

2. Testing Async Code

- **What It Does:** Since many of your methods use `async/await`, these tests ensure your code handles things like promises properly. They check both successful responses and failures.
- **Examples:**
 - Checking if `getCompanies` fetches the list of companies successfully.
 - Ensuring that if something goes wrong (like a database error), the method still responds with the appropriate error message and status code.

3. Error Handling Tests

- **What It Does:** These tests make sure that your controllers gracefully handle errors, such as when the database fails or invalid input is provided. The goal is to confirm that the right status codes (like 500) and error messages are sent back.
- **Examples:**
 - Testing `getUsers` when the database throws an error.
 - Making sure `updateCompany` sends a proper error response if something fails during the update.

4. Mocking and Stubbing

- **What It Does:** These tests use tools like `sinon` to "fake" the behavior of your models (like `UserModel` or `CompanyModel`). This means the tests don't rely on an actual database. Instead, you give fake responses to test just the controller logic.
- **Examples:**

- Pretending that getAllUsers returns a list of users without connecting to a database.
- Simulating an error in addCompany to see how the controller reacts.

5. Integration-Like Tests

- **What It Does:** These tests check if your controllers and models "talk" to each other the way they're supposed to. While it's not a full integration test (since the database is mocked), it still validates the interaction between the layers.
- **Examples:**
 - Testing if addCompany correctly calls the model and returns the expected result.
 - Verifying that deleteUser communicates properly with the UserModel to remove a user.

6. Hook-Based Tests

- **What It Does:** These tests use Mocha's before, after, beforeEach, and afterEach hooks to set up or clean up data before and after tests run. It's like preparing the stage for the tests and resetting everything afterward.
- **Examples:**
 - Using the before hook to create a test company before running update and delete tests.
 - Using after to clean up mock data after all the tests are done.
 - Using beforeEach and afterEach to reset stubs or mocks between individual tests.

7. Edge Case Testing

- **What It Does:** These tests check how your controllers handle unusual or extreme scenarios. This includes invalid input, missing data, or unexpected behavior.
- **Examples:**
 - Testing how addUser reacts when the required fields are missing.
 - Ensuring getCompanies doesn't crash if the database query fails.

Key Categories for Each Controller

User Controller

1. **Unit Tests:** Ensures methods like getUsers, addUser, updateUser, and deleteUser behave as intended.
2. **Error Handling:** Confirms proper responses when something goes wrong.

3. **Async Testing:** Validates methods that use `async/await`.
4. **Mocking:** Uses `sinon` to simulate database operations and isolate controller behavior.

Company Controller

1. **Unit Tests:** Checks methods like `getCompanies`, `addCompany`, `updateCompany`, and `deleteCompany`.
2. **Error Handling:** Tests how the controller handles failures like database errors.
3. **Integration-Like Testing:** Verifies the interaction between the controller and the mocked model.
4. **Hook-Based Testing:** Prepares or cleans up data before and after tests.
5. **Async Testing:** Ensures proper handling of asynchronous operations.
6. **Mocking:** Replaces actual database methods with mock responses to keep tests isolated.

These tests ensure several important aspects of your controllers' functionality:

1. **Validation of Individual Methods:** Unit tests are designed to confirm that each method works as expected when tested in isolation.
2. **Handling Failures Gracefully:** Error handling tests verify that the controllers respond appropriately and nothing breaks when unexpected issues, such as database errors, occur.
3. **Interaction Between Controllers and Models:** Integration-like tests ensure that the controllers communicate correctly with the models and perform the intended operations seamlessly.
4. **Proper Handling of Async Operations:** Async tests validate that asynchronous methods using `async/await` or promises are handled correctly, including both success and failure scenarios.
5. **Test Setup and Cleanup:** Mocha hooks ensure that the environment is prepared before each test and cleaned up afterward, keeping tests independent and reliable.
6. **Dealing with Unusual or Edge Cases:** Edge case tests examine how the controllers handle atypical scenarios or invalid inputs, ensuring robust and predictable behavior even in unexpected situations.