University of San Carlos
Department of Computer and Information Sciences and Mathematics
Talamban Campus, Cebu City, Philippines

IT 3202 - Software Quality Assurance

Final Project
CRUD Web Application Utilizing JSON Web Token
Test Sheets

Submitted by:
Nichole Vine Alburo

Submitted to:
Mr. Keenan Paul Mendiola

May 2024

UNIT TESTING: USER

Test Title: Adding User Functionality
Test Steps:
`

1. **Setup:**
   - Import required modules and mock dependencies.
   - Initialize the database connection pool.
2. **Test Case 1: Adding a new user with unique email**
   - Test Step 1: Mock the database query to return an empty result set (indicating no duplicate emails).
   - Test Step 2: Define a Jest mock callback function.
   - Test Step 3: Call addUser function with a unique email and password, passing the callback.
   - Test Step 4: Verify that the database was queried to check for existing emails.
   - Test Step 5: Verify that the database was queried to insert the new user.
   - Test Step 6: Verify that the callback was called without any error, and with an object argument.
3. **Test Case 2: Adding a user with a duplicate email**
   - Test Step 1: Mock the database query to return a result set containing a user with the same email.
   - Test Step 2: Define a Jest mock callback function.
   - Test Step 3: Call addUser function with an email that already exists and a password, passing the callback.
   - Test Step 4: Verify that the database was queried to check for existing emails.
   - Test Step 5: Verify that the callback was called with an error.
   - Test Step 6: Verify that the error object has a status code of 409 (Conflict).

Expected Outcomes:

- Test Case 1 should pass, indicating successful addition of a new user.
- Test Case 2 should pass, indicating proper handling of duplicate email errors.

Notes:

- Ensure that the mock implementation of the database query behaves as expected in each test case.
- Verify that the callback functions are invoked correctly with the expected arguments and error handling

Test Title: Fetching Users' Details Functionality

Test Steps:

1. **Setup:**
   - Import required modules and mock dependencies.
   - Initialize the database connection pool.
2. **Test Case 1: Fetching users successfully**
   - Test Step 1: Mock the database query to return a list of mock users.
   - Test Step 2: Define a Jest mock callback function.
   - Test Step 3: Call getUsers function, passing the callback.
   - Test Step 4: Verify that the database was queried to fetch users.
   - Test Step 5: Verify that the callback was called without any error, and with the expected list of users.
3. **Test Case 2: Handling errors when fetching users**
   - Test Step 1: Mock the database query to simulate a database connection error.
   - Test Step 2: Define a Jest mock callback function.
   - Test Step 3: Call getUsers function, passing the callback.
   - Test Step 4: Verify that the database was queried to fetch users.
   - Test Step 5: Verify that the callback was called with an error object indicating the database connection error.

Expected Outcomes:

- Test Case 1 should pass, indicating successful retrieval of users.
- Test Case 2 should pass, indicating proper error handling when there's an issue with the database connection.

Notes:

- Ensure that the mock implementation of the database query behaves as expected in each test case.
- Verify that the callback functions are invoked correctly with the expected arguments and error handling.

UNIT TESTING: STUDENTS

Test Title: Adding Student Functionality

Test Steps:

1. **Setup:**
   - Import required modules and mock dependencies.
   - Initialize Express app and mount the router.
   - Mock the database query and repository functions.
2. **Test Case 1: Adding a new student successfully**
   - Test Step 1: Mock the repository function to simulate successful addition of a student.
   - Test Step 2: Send a POST request to the endpoint with valid student data.
   - Test Step 3: Verify that the response status is 201 (Created).
   - Test Step 4: Verify that the response body contains a success message.
3. **Test Case 2: Handling missing name or email**
   - Test Step 1: Send a POST request to the endpoint with missing name or email.
   - Test Step 2: Verify that the response status is 400 (Bad Request).
   - Test Step 3: Verify that the response body contains an error message indicating missing name and email.
4. **Test Case 3: Handling duplicate email error**
   - Test Step 1: Mock the repository function to simulate a duplicate email error.
   - Test Step 2: Send a POST request to the endpoint with a duplicate email.
   - Test Step 3: Verify that the response status is 409 (Conflict).
   - Test Step 4: Verify that the response body contains an error message indicating the duplicate email error.
5. **Test Case 4: Handling database error**
   - Test Step 1: Mock the repository function to simulate a database error.
   - Test Step 2: Send a POST request to the endpoint.
   - Test Step 3: Verify that the response status is 500 (Internal Server Error).
   - Test Step 4: Verify that the response body contains an error message indicating the database error.

Expected Outcomes:

- Test Case 1 should pass, indicating successful addition of a new student.
- Test Case 2 should pass, indicating proper handling of missing name or email.
- Test Case 3 should pass, indicating proper handling of duplicate email error.
- Test Case 4 should pass, indicating proper handling of database errors.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.

Test Title: Fetching Students' Details Functionality

Test Steps:

1. **Setup:**
   - Import required modules and mock dependencies.
   - Initialize Express app and mount the router.
   - Mock the database query and repository functions.
2. **Test Case 1: Adding a new student successfully**
   - Test Step 1: Mock the repository function to simulate successful addition of a student.
   - Test Step 2: Send a POST request to the endpoint with valid student data.
   - Test Step 3: Verify that the response status is 201 (Created).
   - Test Step 4: Verify that the response body contains a success message.
3. **Test Case 2: Handling missing name or email**
   - Test Step 1: Send a POST request to the endpoint with a missing name or email.
   - Test Step 2: Verify that the response status is 400 (Bad Request).
   - Test Step 3: Verify that the response body contains an error message indicating missing name and email.
4. **Test Case 3: Handling duplicate email error**
   - Test Step 1: Mock the repository function to simulate a duplicate email error.
   - Test Step 2: Send a POST request to the endpoint with a duplicate email.
   - Test Step 3: Verify that the response status is 409 (Conflict).
   - Test Step 4: Verify that the response body contains an error message indicating the duplicate email error.
5. **Test Case 4: Handling database error**
   - Test Step 1: Mock the repository function to simulate a database error.
   - Test Step 2: Send a POST request to the endpoint.
   - Test Step 3: Verify that the response status is 500 (Internal Server Error).
   - Test Step 4: Verify that the response body contains an error message indicating the database error.

Expected Outcomes:

- Test Case 1 should pass, indicating successful addition of a new student.
- Test Case 2 should pass, indicating proper handling of missing name or email.
- Test Case 3 should pass, indicating proper handling of duplicate email error.
- Test Case 4 should pass, indicating proper handling of database errors.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.

Test Title: Updating Student Details Functionality

Test Steps:

1. **Setup**:
    - Import required modules and mock dependencies.
    - Initialize Express app and mount the router.
    - Mock the database query and repository functions.
2. **Test Case 1: Updating student status successfully**
    - Test Step 1: Clear all mocks to ensure a clean state.
    - Test Step 2: Mock the repository function to simulate successful update of student status.
    - Test Step 3: Send a PUT request to the endpoint with valid student status data.
    - Test Step 4: Verify that the response status is 200 (OK).
    - Test Step 5: Verify that the response body contains a success message.
3. **Test Case 2: Handling database error while updating student status**
    - Test Step 1: Clear all mocks to ensure a clean state.
    - Test Step 2: Mock the repository function to simulate a database error while updating student status.
    - Test Step 3: Send a PUT request to the endpoint.
    - Test Step 4: Verify that the response status is 500 (Internal Server Error).
    - Test Step 5: Verify that the response body contains an error message indicating the failure to update student status.
4. **Test Case 3: Handling missing status**

    - Test Step 1: Send a PUT request to the endpoint with missing status.
    - Test Step 2: Verify that the response status is 400 (Bad Request).
    - Test Step 3: Verify that the response body contains an error message indicating missing status.

Expected Outcomes:

- Test Case 1 should pass, indicating successful update of student status.
- Test Case 2 should pass, indicating proper handling of database errors while updating student status.
- Test Case 3 should pass, indicating proper handling of missing status.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.

UNIT TESTING: TASKS

Test Title: Adding Task Functionality

Test Steps:

1. **Setup:**
    - Import required modules and mock dependencies.
    - Initialize Express app and mount the router.
    - Mock the database query and repository functions.
2. **Test Case 1: Adding a task successfully**
    - Test Step 1: Clear all mocks to ensure a clean state.
    - Test Step 2: Mock the repository function to simulate successful addition of a task.
    - Test Step 3: Send a POST request to the endpoint with valid task data.
    - Test Step 4: Verify that the response status is 201 (Created).
    - Test Step 5: Verify that the response body contains a success message.
3. **Test Case 2: Handling database error while adding a task**
    - Test Step 1: Clear all mocks to ensure a clean state.
    - Test Step 2: Mock the repository function to simulate a database error while adding a task.
    - Test Step 3: Send a POST request to the endpoint.

    - Test Step 4: Verify that the response status is 500 (Internal Server Error).
    - Test Step 5: Verify that the response body contains an error message indicating the failure to add a task.
4. **Test Case 3: Handling missing required fields**
    - Test Step 1: Send a POST request to the endpoint with missing required fields.
    - Test Step 2: Verify that the response status is 400 (Bad Request).
    - Test Step 3: Verify that the response body contains an error message indicating missing required fields.

Expected Outcomes:

- Test Case 1 should pass, indicating successful addition of a task.
- Test Case 2 should pass, indicating proper handling of database errors while adding a task.
- Test Case 3 should pass, indicating proper handling of missing required fields.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.

Task Title: Fetching Tasks' Details Functionality

Test Steps:

1. **Setup:**
    - Import required modules and mock dependencies.
    - Initialize Express app and mount the router.
    - Mock the database query and repository functions.
2. **Test Case 1: Fetching tasks for the given class successfully**
    - Test Step 1: Mock the repository function to simulate successful fetching of tasks for the given class.
    - Test Step 2: Send a GET request to the endpoint with a valid class ID.
    - Test Step 3: Verify that the response status is 200 (OK).
    - Test Step 4: Verify that the response body contains the expected tasks.

3. **Test Case 2: Handling database error while fetching tasks**

  - Test Step 1: Mock the repository function to simulate a database error while fetching tasks.
  - Test Step 2: Send a GET request to the endpoint.
  - Test Step 3: Verify that the response status is 500 (Internal Server Error).
  - Test Step 4: Verify that the response body contains an error message indicating the failure to fetch tasks.

Expected Outcomes:

- Test Case 1 should pass, indicating successful fetching of tasks for the given class.
- Test Case 2 should pass, indicating proper handling of database errors while fetching tasks.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.

Task Title: Updating Task Details Functionality

Test Steps:

1. **Setup:**
   - Import required modules and mock dependencies.
   - Initialize Express app and mount the router.
   - Mock the database query and repository functions.
2. **Test Case 1: Updating task status successfully**
   - Test Step 1: Mock the repository function to simulate successful update of task status.
   - Test Step 2: Send a PUT request to the endpoint with valid task status data.
   - Test Step 3: Verify that the response status is 200 (OK).
   - Test Step 4: Verify that the response body contains a success message.
3. **Test Case 2: Handling database error while updating task status**
   - Test Step 1: Mock the repository function to simulate a database error while updating task status.
   - Test Step 2: Send a PUT request to the endpoint.
   - Test Step 3: Verify that the response status is 500 (Internal Server Error).

    - Test Step 4: Verify that the response body contains an error message indicating the failure to update task status.
4. **Test Case 3: Handling missing status**
    - Test Step 1: Send a PUT request to the endpoint with missing status.
    - Test Step 2: Verify that the response status is 400 (Bad Request).
    - Test Step 3: Verify that the response body contains an error message indicating missing status.

Expected Outcomes:

- Test Case 1 should pass, indicating successful update of task status.
- Test Case 2 should pass, indicating proper handling of database errors while updating task status.
- Test Case 3 should pass, indicating proper handling of missing status.

Notes:

- Ensure that the mock implementations of repository functions behave as expected in each test case.
- Verify that the responses are correctly structured with the appropriate status codes and error messages.