DEPLOYED WEB APPLICATION: https://nichole.alburo.ph/

GITHUB REPOSITORY: https://github.com/nicvhine/teacher-web

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

Submitted by:
Nichole Vine Alburo

Submitted to:
Mr. Keenan Paul Mendiola

May 2024

TABLE OF CONTENTS

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.

Test 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.

Test 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.

AUTOMATION TESTING: REGISTER PAGE

Test 1: ***'displays form inputs and registers a user'***

- This test checks if the form inputs are displayed properly and if a user is successfully registered.
- Test Steps:
    - Type 'test@example.com' into the input field with type "email".
    - Type 'password123' into the input field with type "password".
    - Click on the submit button.
    - Check if a success message with the text 'User registered successfully!' is displayed.

Test 2: ***'displays error message when form is submitted with empty fields'***

- This test verifies if an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Click on the submit button without filling any fields.
    - Check if an error message with the text 'Email and password cannot be empty.' is displayed.

Test 3: ***'displays error message when registration fails'***

- This test simulates a scenario where registration fails (e.g., due to an email already being taken).
- Test Steps:
    - Intercept the registration request and mock a response with a 409 status code (conflict) and an error message indicating that the email is already taken.
    - Type 'existinguser@example.com' into the input field with type "email".
    - Type 'password123' into the input field with type "password".
    - Click on the submit button.
    - Wait for the registration request to complete.
    - Check if an error message with the text 'This email is already registered. Please use a different email.' is displayed.

Test 4: ***'navigates to login page when "Login Here" link is clicked'***

- This test checks if clicking on the "Login Here" link navigates to the login page.
- Test Steps:
    - Click on the link containing the text 'Already registered?'.
    - Check if the URL includes '/' indicating navigation to the login page.

AUTOMATION TESTING: LOGIN

Test 1: '*displays form inputs and logs in a user'*

- This test verifies that the form inputs are displayed correctly and a user can successfully log in.
- Test Steps:
    - Visit the login page.
    - Type 'test@example.com' into the email input field.
    - Type 'password123' into the password input field.
    - Click on the submit button.
    - Check if the URL includes '/class', indicating successful login and navigation to the class page.

Test 2: *'displays error message when form is submitted with empty fields'*

- This test ensures that an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Visit the login page.
    - Click on the submit button without filling any fields.
    - Check if an error message with the text 'Email and password cannot be empty.' is displayed.

Test 3: *'displays error message when login fails'*

- This test simulates a scenario where login fails due to incorrect credentials.
- Test Steps:
    - Intercept the login request and mock a response with a 401 status code (unauthorized) and an error message indicating invalid credentials.
    - Visit the login page.
    - Type 'invalid@example.com' into the email input field.
    - Type 'incorrectpassword' into the password input field.
    - Click on the submit button.
    - Wait for the login request to complete.
    - Check if an error message with the text 'Invalid credentials' is displayed.

Test 4: *'navigates to registration page when "Register Here" link is clicked'*

- This test confirms that clicking on the "Register Here" link navigates to the registration page.
- Test Steps:

    - Visit the login page.
    - Click on the link containing the text 'Haven't registered yet?'.
    - Check if the URL includes '/register', indicating navigation to the registration page.

AUTOMATION TESTING: CLASS PAGE

Test 1: 'displays form inputs and adds a class'

- This test checks if the form inputs are displayed properly and if a class is successfully added.
- Test Steps:
    - Type 'Test Class' into the input field with placeholder "Class Name".
    - Type 'This is a test class' into the textarea with placeholder "Description".
    - Type 'Test Group' into the textarea with placeholder "Group Number".
    - Type '2024' into the input field with placeholder "Start Year".
    - Type '2025' into the input field with placeholder "End Year".
    - Click on the submit button.
    - Check if a success message container with the text 'Class added successfully' is displayed.

Test 2: ***'displays error message when form is submitted with empty fields'***

- This test verifies if an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Click on the submit button without filling any fields.
    - Check if an error message container with the text 'All fields are required' is displayed.

Test 3: ***'navigates to class details page when class item is clicked'***

- This test ensures that clicking on a class item navigates to the class details page.
- Test Steps:
    - Click on the first class item.
    - Check if the URL contains '/dashboard/' indicating navigation to the class details page.

AUTOMATION TESTING: DASHBOARD PAGE

Test 1: ***'displays class navigation links'***

- This test verifies that the class navigation links are displayed correctly.
- Test Steps:
    - Visit the dashboard page for class ID 1.
    - Check if the 'Dashboard', 'Student List', 'Task Management', 'Class Management', 'Class List', and 'Logout' links exist.

Test 2: ***'fetches class information when classId is provided'***

- This test ensures that class information is fetched and displayed when the class ID is provided.
- Test Steps:
    - Intercept the GET request to fetch class information and mock a response with class information from the server.
    - Visit the dashboard page for class ID 1.
    - Wait for the request to complete.
    - Verify if class information such as 'Class Name' and 'Teacher' are displayed in the dashboard container.

Test 3: ***'navigates to class list when "Class List" link is clicked'***

- This test checks if clicking on the "Class List" link navigates to the class list page.
- Test Steps:
    - Visit the dashboard page for class ID 1.
    - Click on the 'Class List' link.
    - Verify if the URL includes '/class', indicating navigation to the class list page.

AUTOMATION TESTING: STUDENT PAGE

Test 1: ***'displays form inputs and adds a new student'***

- This test verifies that the form inputs are displayed correctly and a new student can be added.
- Test Steps:
    - Visit the student list page.
    - Type 'John Doe' into the name input field.
    - Type 'john@example.com' into the email input field.
    - Submit the form.

    - Check if a success message with the text 'Student added successfully!' is displayed.

Test 2: ***'displays error message when form is submitted with empty fields'***

- This test ensures that an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Visit the student list page.
    - Submit the form without filling out any inputs.
    - Check if an error message with the text 'Name and email cannot be empty.' is displayed.

Test 3: ***'filters students based on search query'***

- This test checks if students are filtered based on the search query.
- Test Steps:
    - Visit the student list page.
    - Type 'john' into the search input.
    - Verify if only matching students are displayed in the table body.

Test 4: ***'toggles student status when clicked'***

- This test verifies that the student status toggles when clicked.
- Test Steps:
    - Visit the student list page.
    - Assuming there's at least one student rendered with active status, find the status span of the first student.
    - Click on the status span.
    - Verify if the status toggles successfully by checking the color of the status span.

AUTOMATION TESTING: TASK PAGE

Test 1: ***'displays form inputs and adds a new task'***

- This test verifies that the form inputs are displayed correctly and a new task can be added.
- Test Steps:
    - Visit the task management page.
    - Type 'New Task' into the title input field.
    - Type 'This is a new task' into the description input field.

    - Type '2024-06-01' into the due date input field.
    - Click on the 'Add Task' button.
    - Check if a success message with the text 'Task added successfully' is displayed.

Test 2: ***'displays error message when form is submitted with empty fields'***

- This test ensures that an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Visit the task management page.
    - Click on the 'Add Task' button without filling out any inputs.
    - Check if an error message with the text 'Fields cannot be empty.' is displayed.

Test 3: ***'filters tasks based on search query'***

- This test checks if tasks are filtered based on the search query.
- Test Steps:
    - Visit the task management page.
    - Type 'new' into the search input.
    - Verify if only matching tasks are displayed in the table body.

Test 4: ***'toggles task status when clicked'***

- This test verifies that the task status toggles when clicked.
- Test Steps:
    - Visit the task management page.
    - Assuming there's at least one task rendered with pending status, find the status span of the first task.
    - Click on the status span.
    - Verify if the status toggles successfully by checking the color of the status span.

AUTOMATION TESTING: CLASS MANAGEMENT PAGE

Test 1: ***'displays class details and updates them'***

- This test verifies that the class details are displayed correctly and can be updated.
- Test Steps:
    - Visit the settings page.
    - Clear and type 'New Class Name' into the name input field.
    - Clear and type 'New Class Description' into the description input field.
    - Clear and type '2024' into the start year input field.

    - Clear and type '2025' into the end year input field.
    - Click on the 'Update Class' button.
    - Check if a success message with the text 'Class updated successfully' is displayed.

Test 2: ***'displays error message when form is submitted with empty fields'***

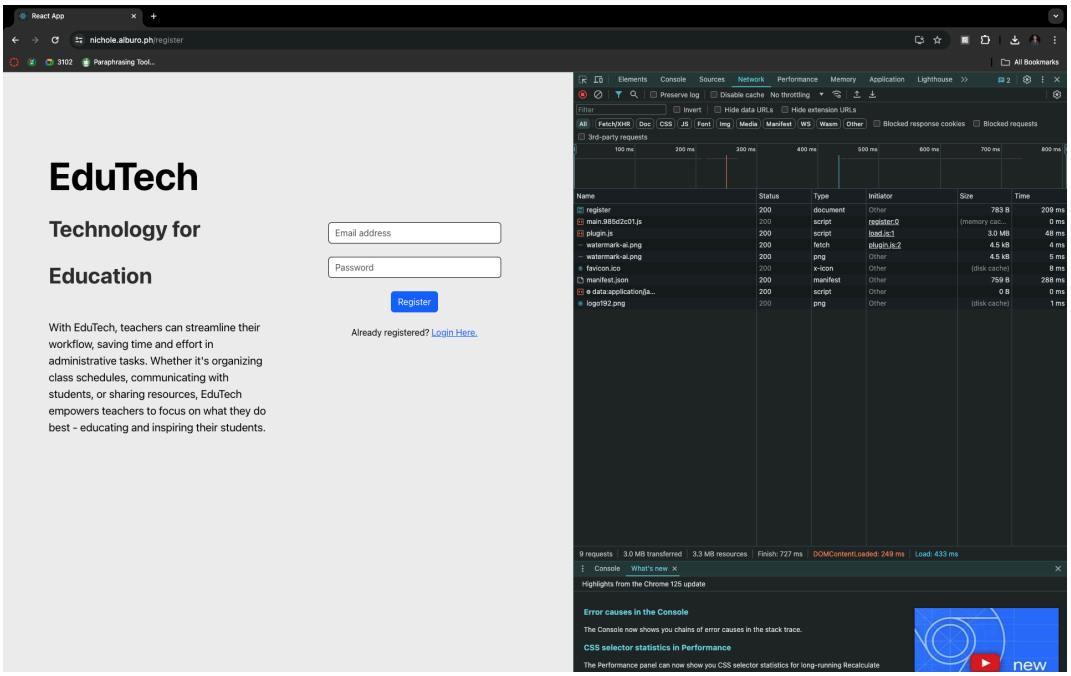
- This test ensures that an error message is displayed when the form is submitted with empty fields.
- Test Steps:
    - Visit the settings page.
    - Click on the 'Update Class' button without filling out any inputs.
    - Check if an error message with the text 'Fields cannot be empty.' is displayed.

Test 3: ***'deletes class when "Delete Class" button is clicked'***

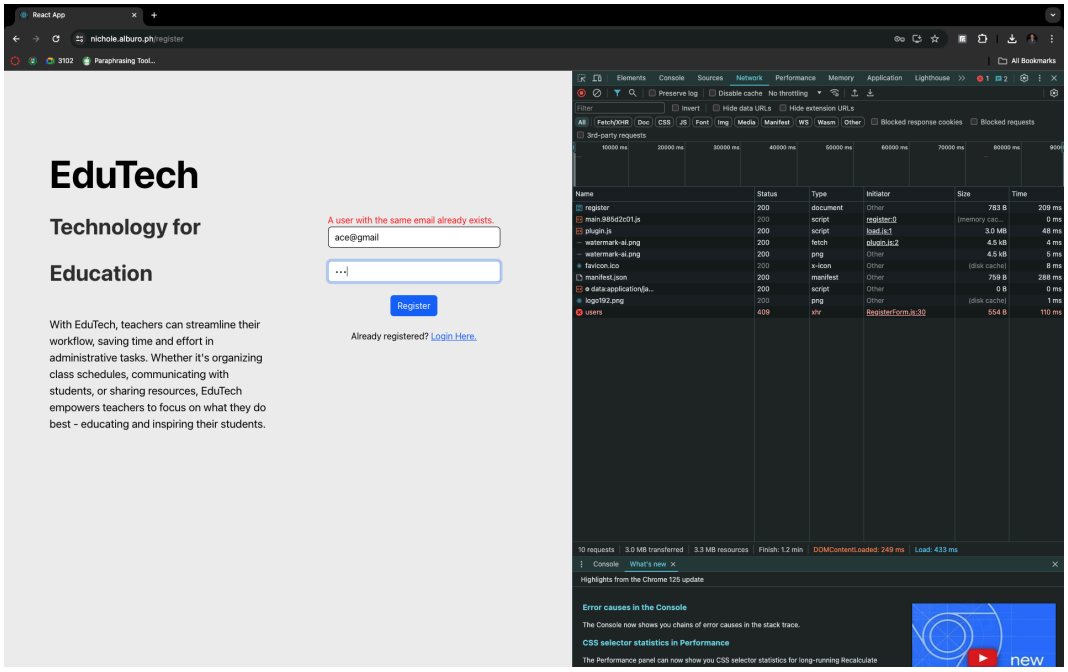
- This test checks if the class can be deleted when the "Delete Class" button is clicked.
- Test Steps:
    - Visit the settings page.
    - Click on the "Delete Class" button.
    - Assuming you have a confirmation dialog, handle it accordingly.
    - Verify if the class is deleted by checking if the URL changes to '/class'.

SCREEN CAPTURES FROM THE DEPLOYED APP

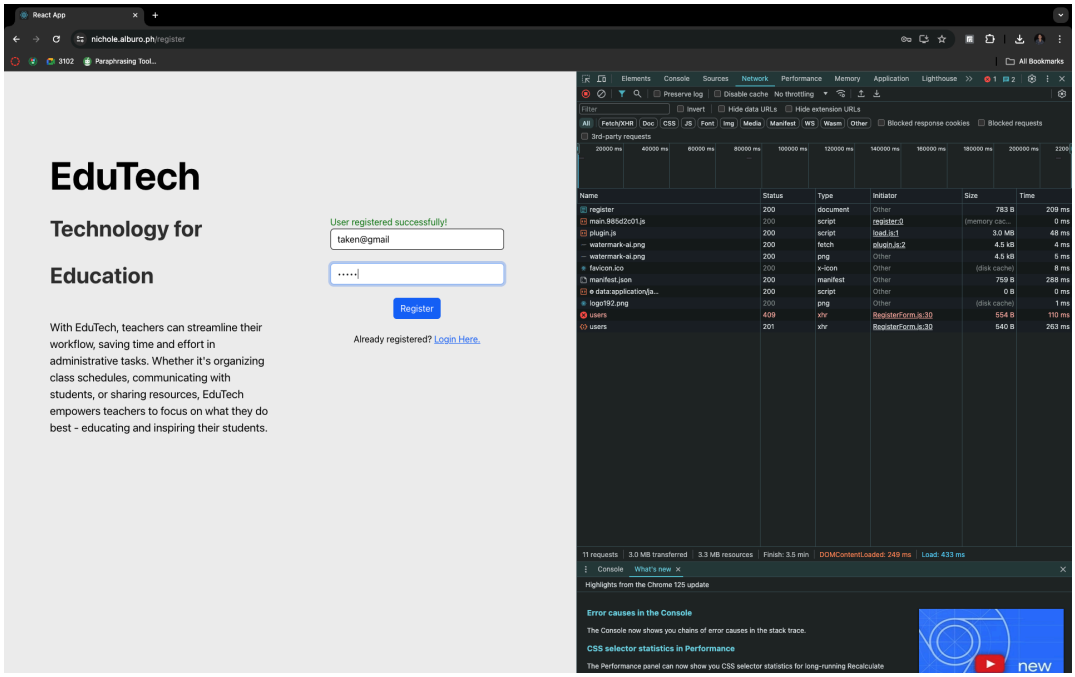
REGISTER PAGE



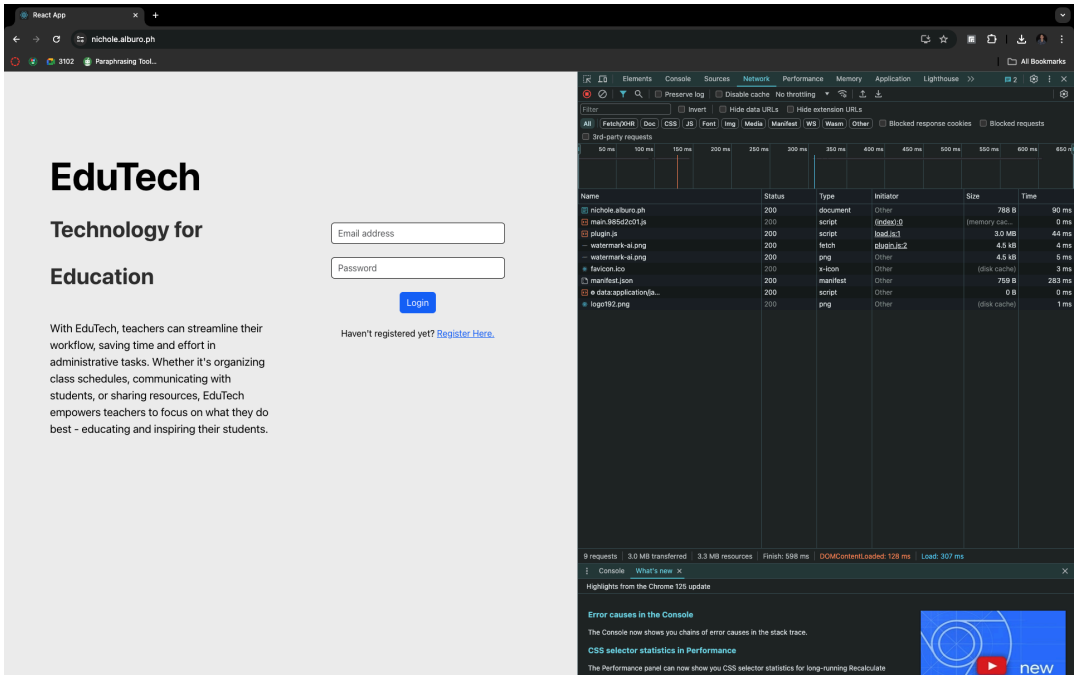
Placed email is already registered



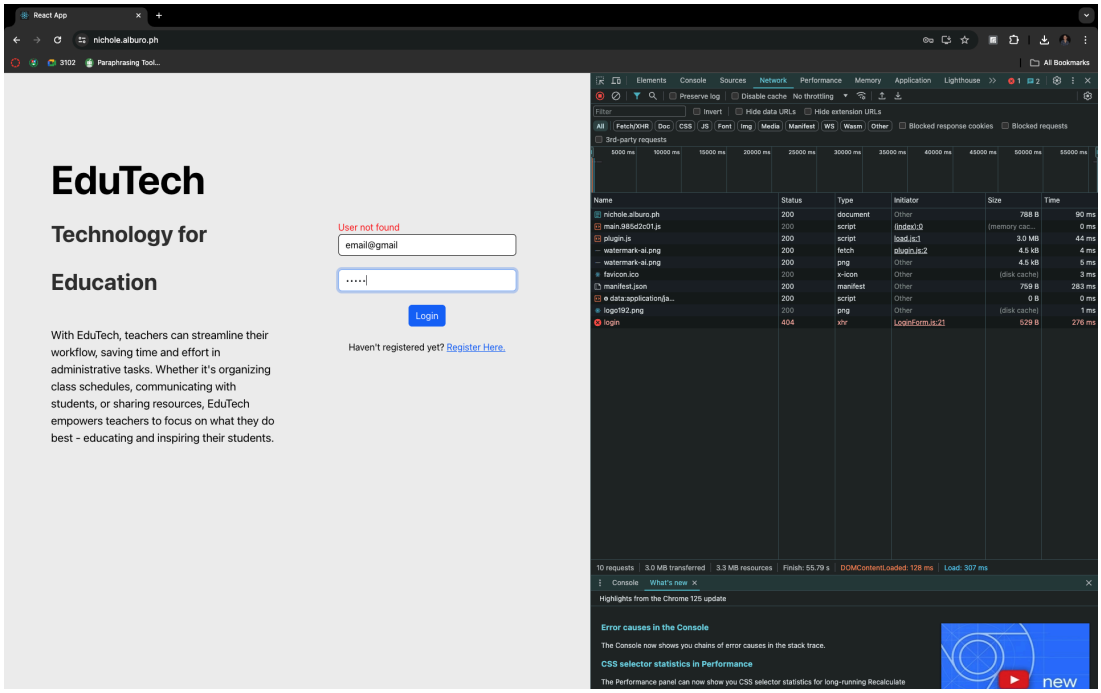
Successfully registered



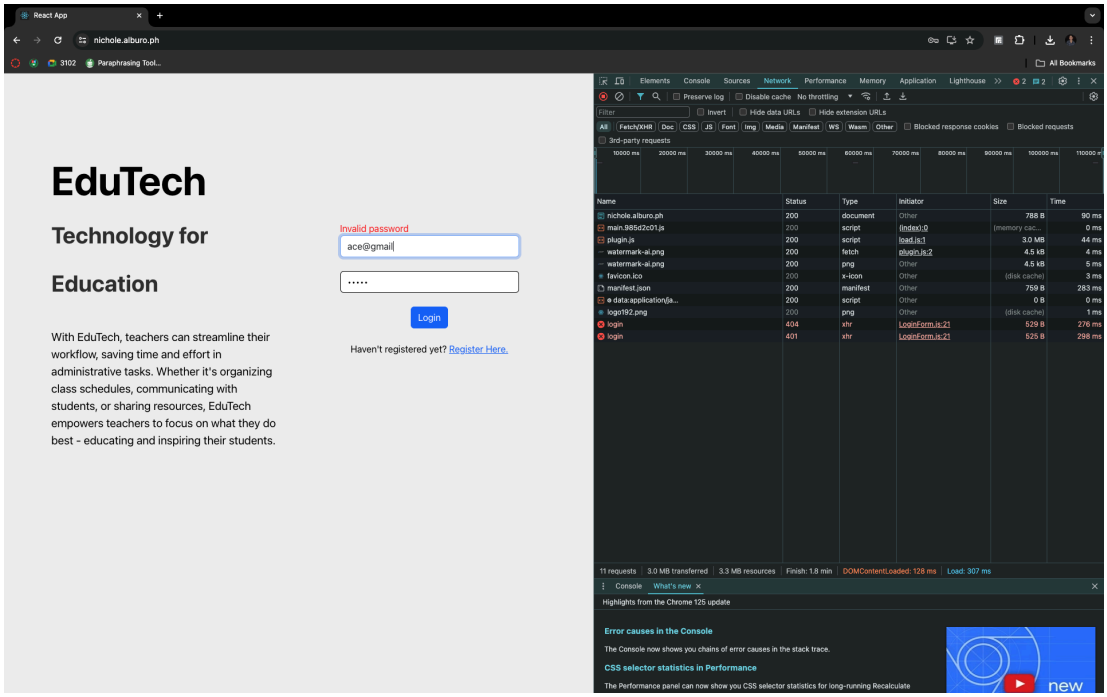
LOGIN PAGE



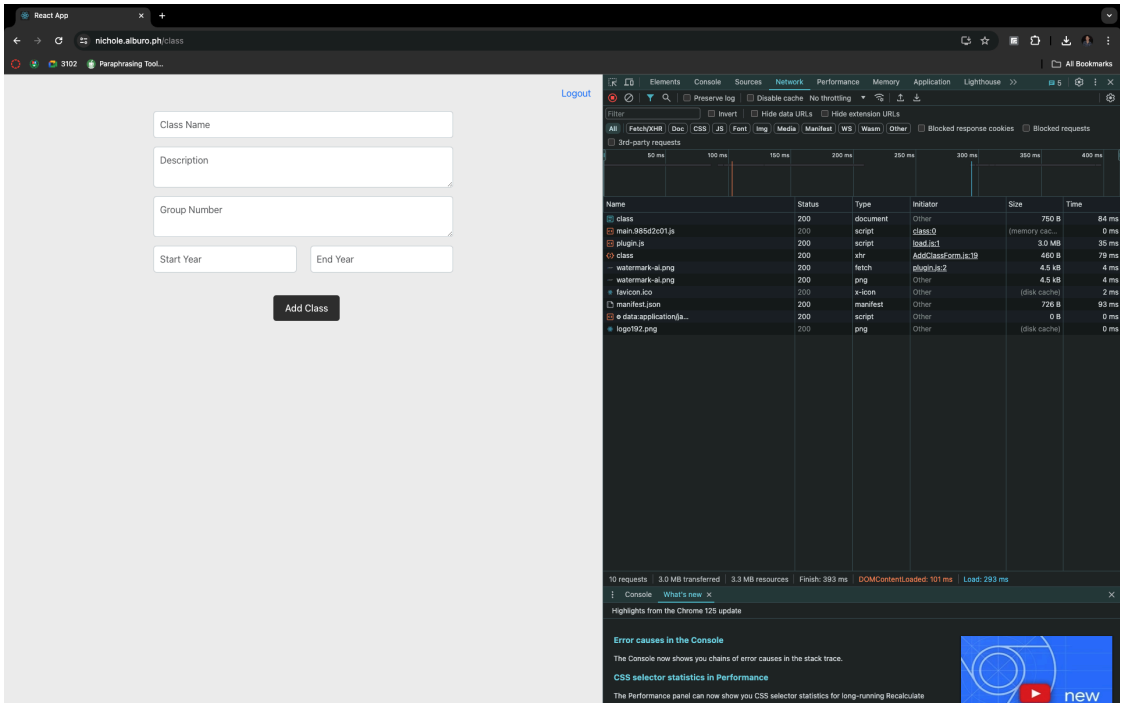
User is not found/registered



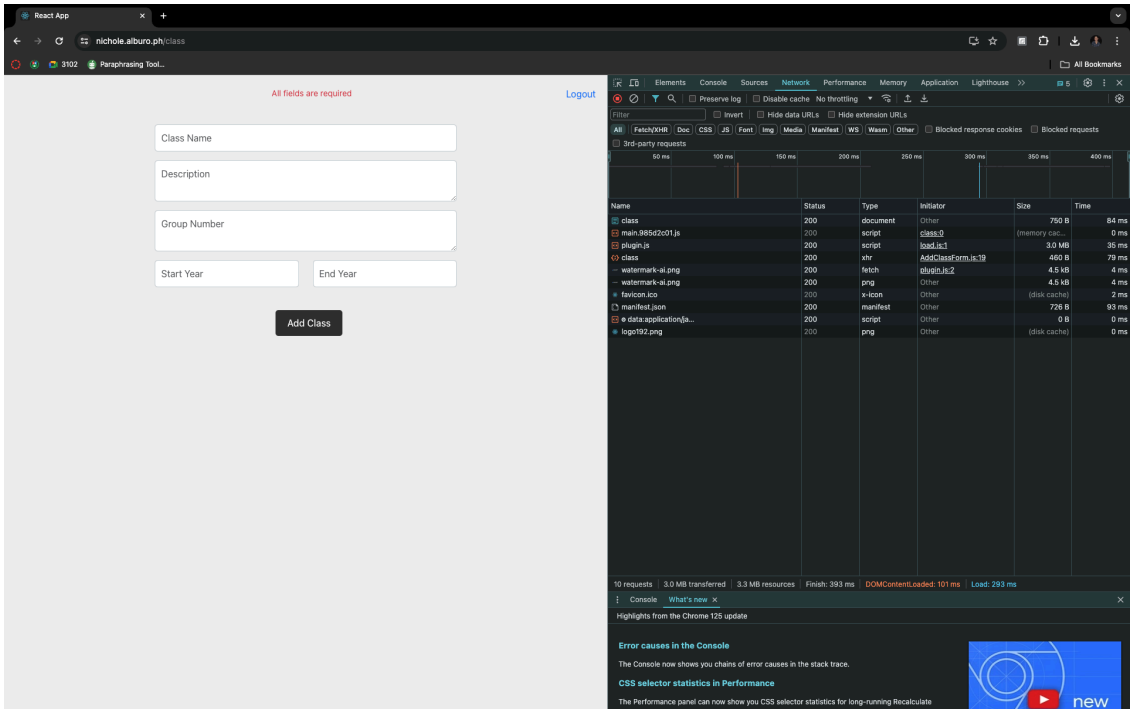
Placed password is incorrect



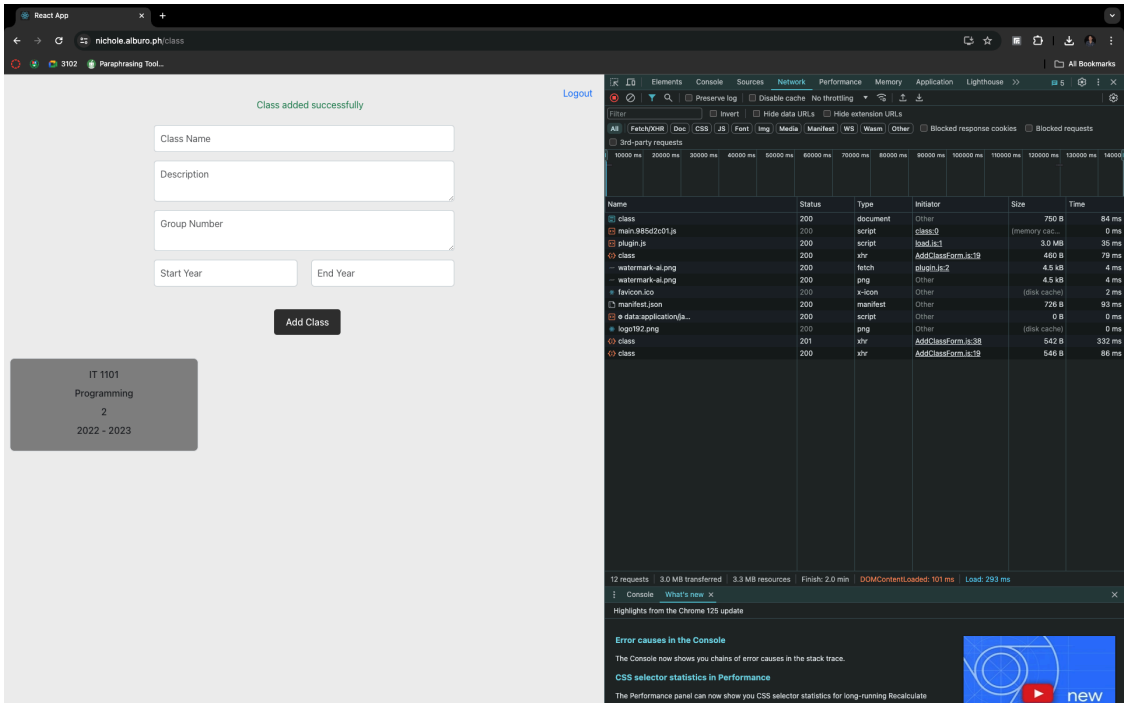
CLASS PAGE



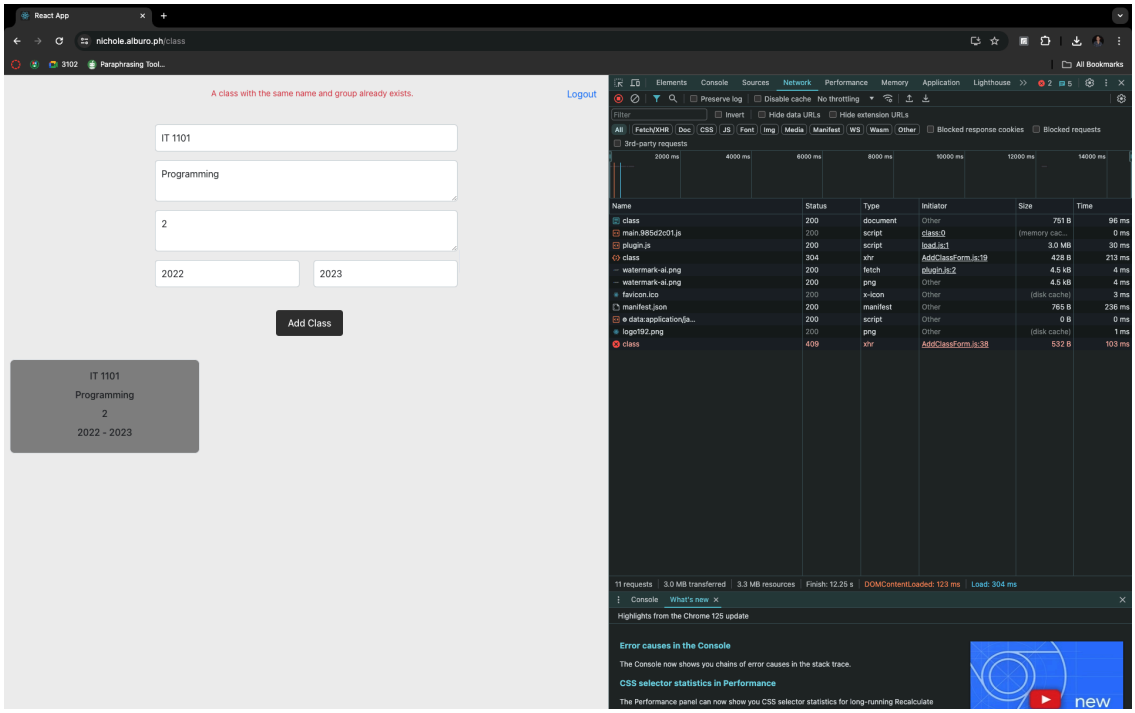
Field is empty



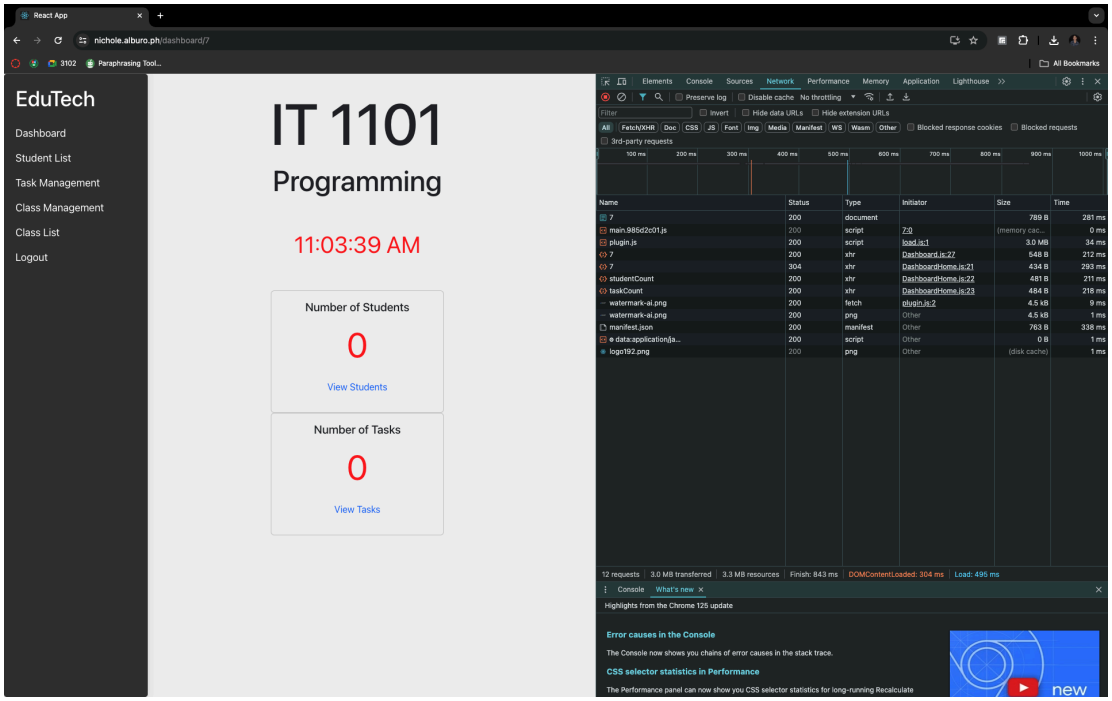
Class added successfully



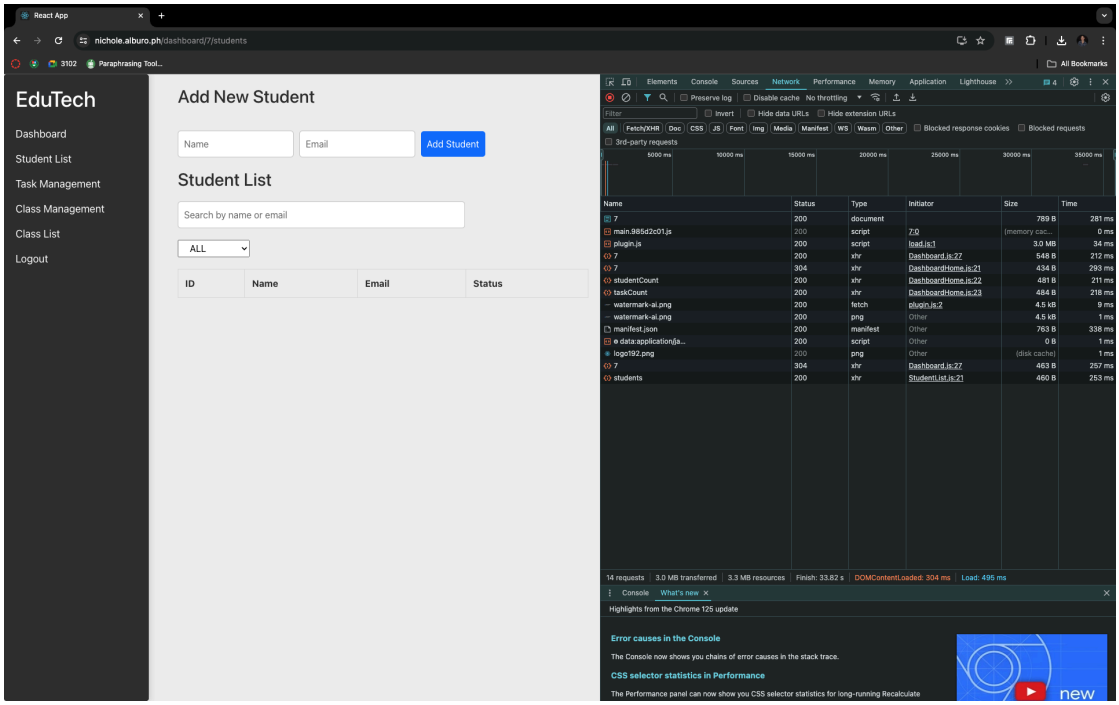
Duplicate details



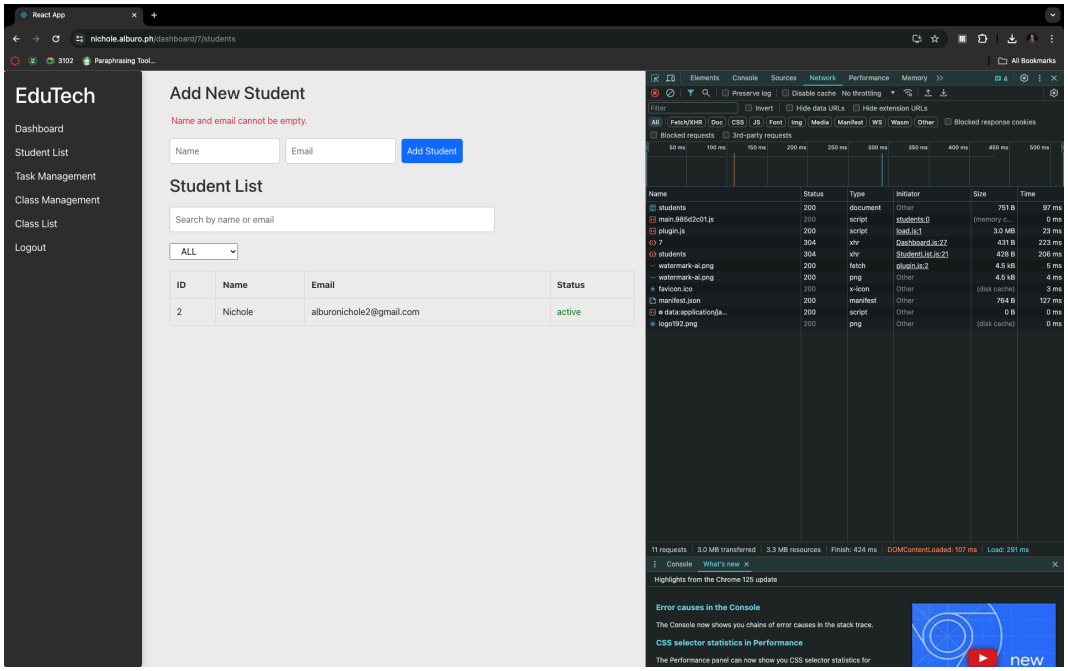
DASHBOARD PAGE



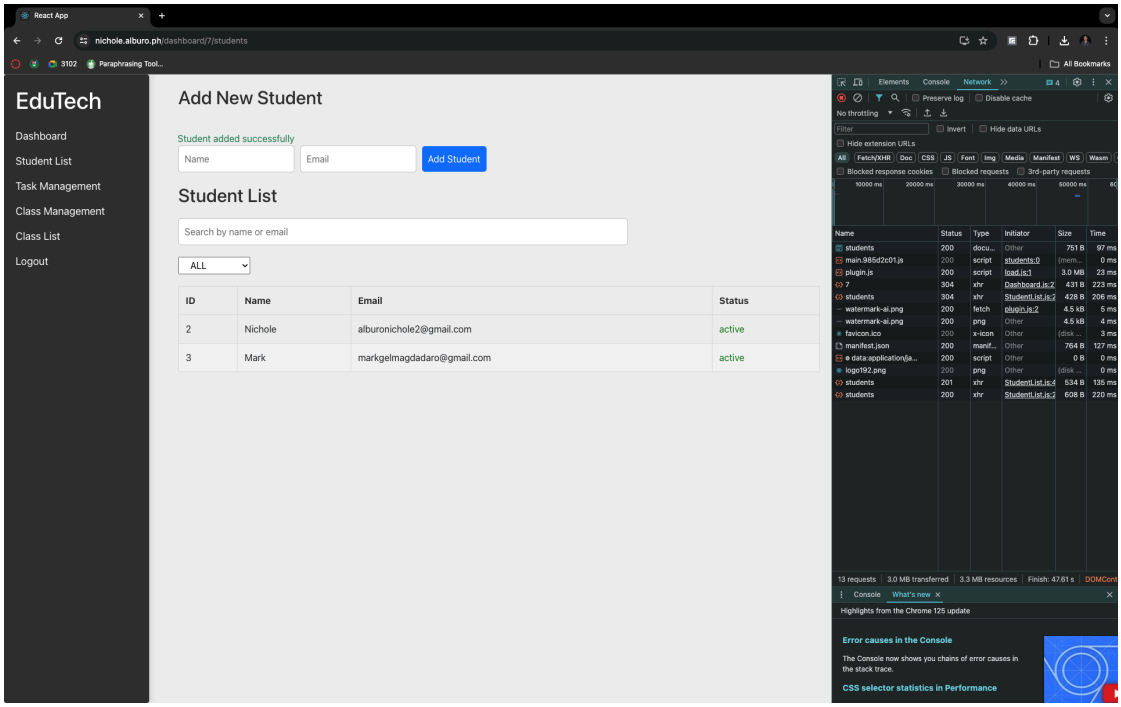
STUDENT PAGE



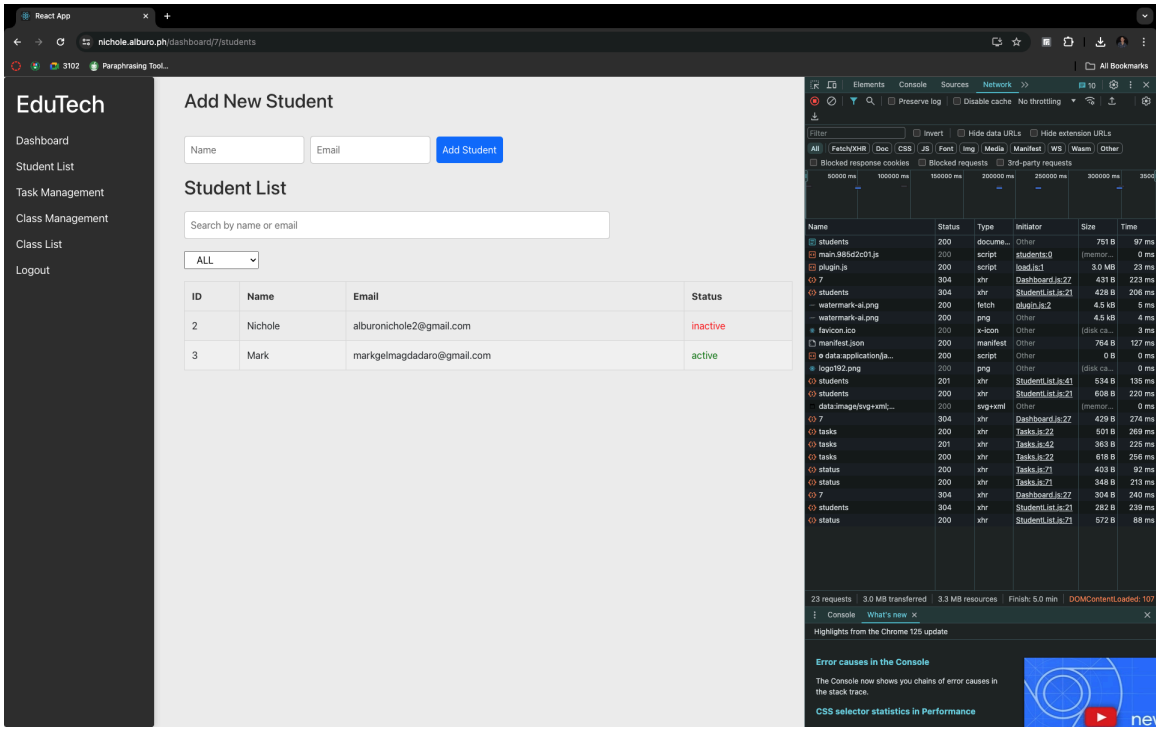
Empty fields



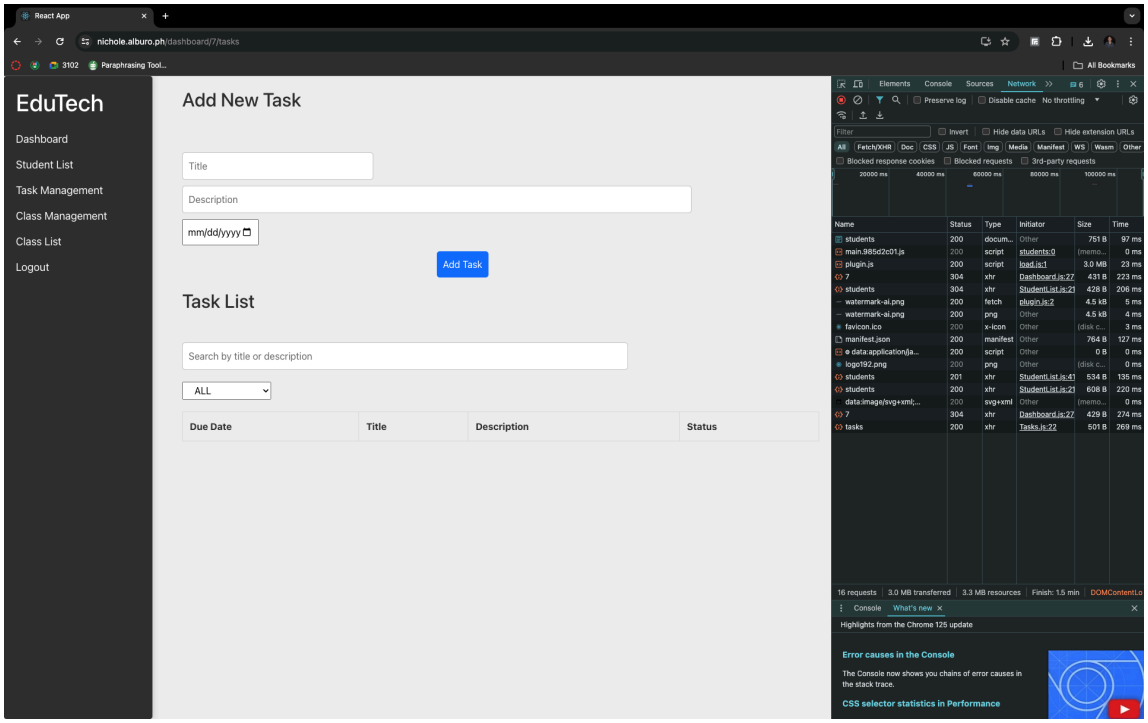
Student added successfully



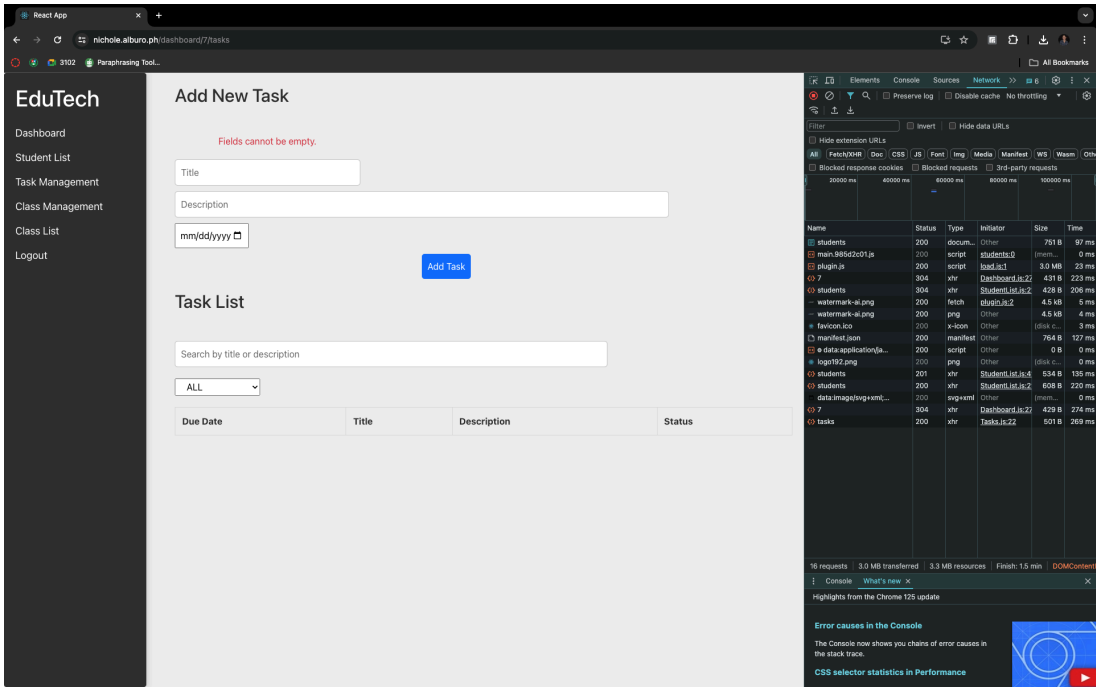
Switching of student's status



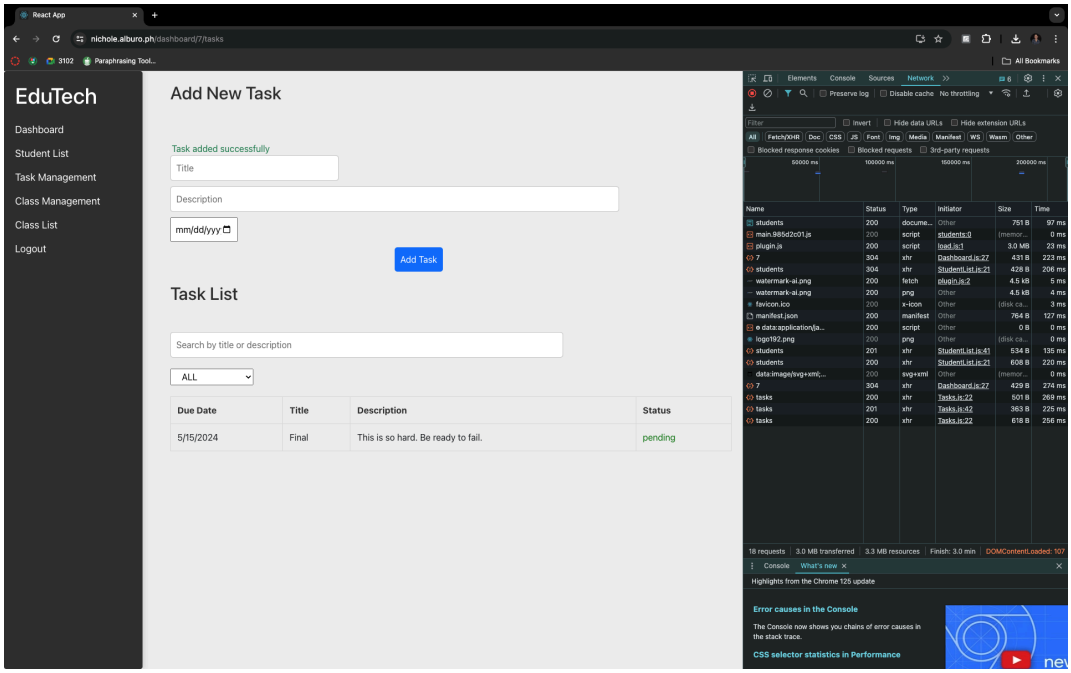
TASK PAGE



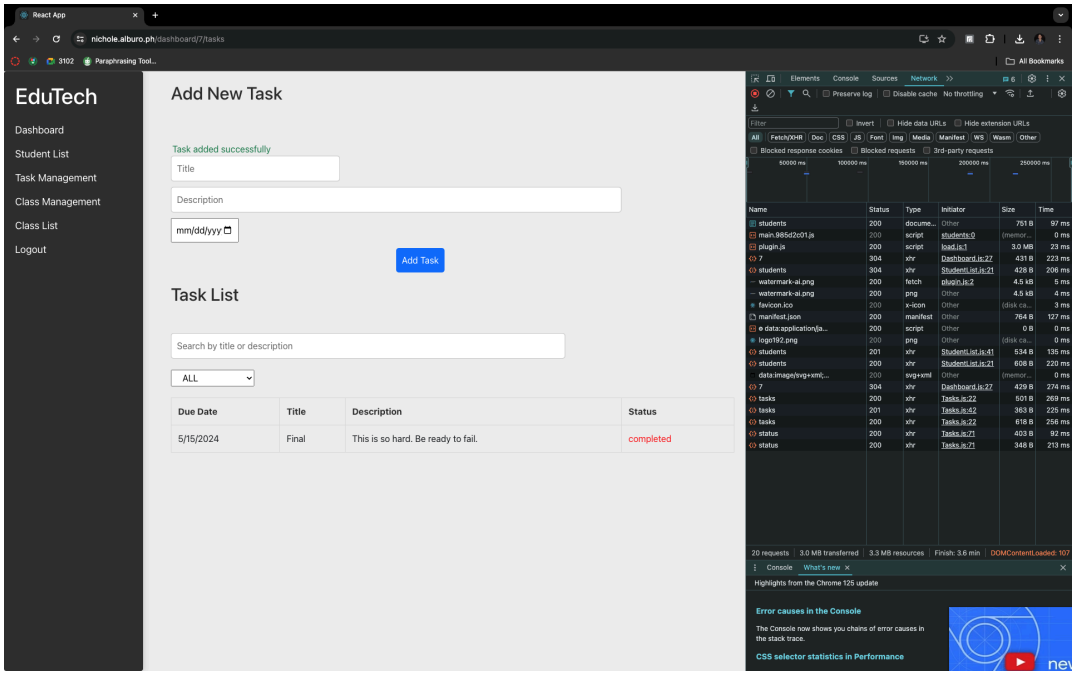
Empty Fields



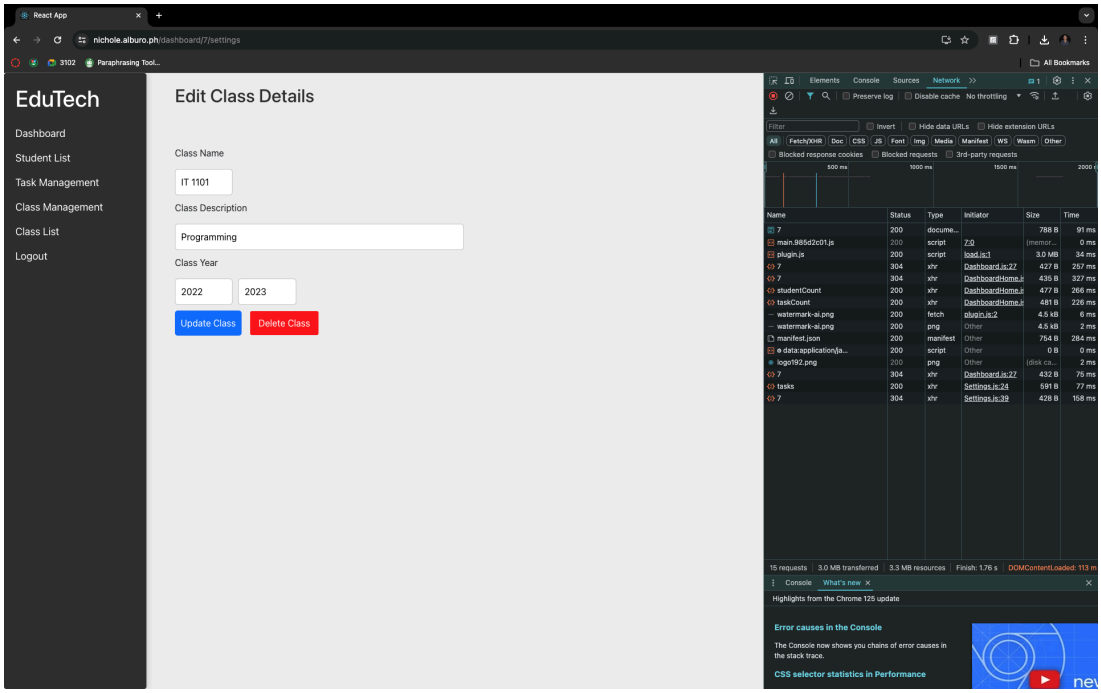
Task added successfully



Switching of task's status



CLASS DETAILS PAGE



Class details updated