DESCRIPTION PAGE

Frontend Unit Testing

📁 Project Name: doctoral progress tracking (Frontend)

🧪 Tools & Libraries Used:

- **React** (frontend library)
- **TypeScript** (for type safety)
- **Jest** (test runner and assertion library)
- **React Testing Library** (for DOM testing)
- **Vitest** (configured as testing framework, from `vitest.config.ts`)

🧪 Test Coverage Overview:

1. **Component Rendering Tests**
   - **Objective:** Ensure that individual components render correctly without crashing.
   - **Test Strategy:**
     - Render components like `Navbar`, `Login`, `Dashboard`, etc.
     - Use `screen.getByText()` or `getByRole()` to assert important elements are present.
     - Check component loading behavior and key DOM structure.
2. **Form Interaction Tests**
   - **Objective:** Validate that user inputs, forms, and buttons function correctly.
   - **Test Strategy:**
     - Simulate form filling and submission (e.g., login form, publication entry form).
     - Use `fireEvent.change()` and `fireEvent.submit()` or `userEvent`.
     - Assert on validation messages, success messages, or navigation changes.
3. **Navigation & Routing Tests**
   - **Objective:** Ensure React Router navigates to the correct pages.
   - **Test Strategy:**
     - Simulate clicks on navigation elements (`<Link>` or buttons).
     - Check if the route changes and the correct component is displayed.
4. **Role-Based Rendering Tests**
   - **Objective:** Ensure correct content appears based on logged-in user roles (Student, Supervisor, Coordinator).
   - **Test Strategy:**
     - Mock user roles.
     - Assert that components and features visible are restricted/allowed as per role.
5. **API Mock Tests**
   - **Objective:** Mock API calls and ensure UI behaves properly.
   - **Test Strategy:**
     - Use `msw` (Mock Service Worker) or Jest mocks to intercept fetch/axios.
     - Simulate success and failure responses.

DESCRIPTION PAGE

- Check loading spinners, error alerts, and success updates.

6. **Edge Cases**
   - **Objective:** Handle scenarios like empty responses, invalid inputs, and permission-denied responses.
   - **Test Strategy:**
     - Test against blank datasets, slow loading, or 401/403 responses.
     - Ensure fallback UI is shown and no crashes occur.

Step-by-Step Commands for Setting Up and Running Tests

1. Install vitest and React Testing Library with necessary dependencies

. npm install -D vitest @vitejs/plugin-react jsdom

2. Install React Testing Library & utilities

. npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event

3. (Optional but useful) Vitest UI for visual test running

. npm install -D @vitest/ui

4. To run all tests in CLI

. npx vitest run

5. To run all tests in watch mode (recommended while developing)

. npx vitest

6. To open the interactive Vitest UI

. npx vitest –ui

Unit Test Case Description – SE Project Backend

Project Name:

Doctoral Progress Tracker (Backend)

Purpose of Unit Testing:

DESCRIPTION PAGE

The purpose of the unit tests is to ensure that each individual component (repository and controller classes) of the SE Project performs as expected. The focus was on testing core functionalities such as CRUD operations and response behaviors for different endpoints and repository methods.

Tools Used:

- **JUnit 5** – For writing and running test cases
- **Mockito** – For mocking dependencies such as repositories and service layers
- **Spring Boot Test** – For integration with Spring Boot context
-

Sample Tests:

- Verifying correct return value for repository queries
- Checking `@GetMapping` and `@PostMapping` responses in controller
- Testing fallback and edge cases (empty results, null values)

3. Commands to Run Tests

Maven has been used:

1.To compile tests only

. mvn test-compile

2.To run all test cases

. mvn test

3. To run tests with detailed about

. mvn test –X