Documentation for Setting Up the Testing Environment and Running Tests Locally

Prerequisites

Before setting up the testing environment, ensure you have the following software installed:

- **Node.js** (version 14.x or later)
- **npm** (comes with Node.js)
- **Git** (for version control)

Setting Up the Testing Environment

Clone the Repository:

```
git clone https://github.com/your-username/your-repo.git
cd your-repo
```

1. **Install Dependencies:**

   ```
   npm install
   ```
2. **Configure Environment Variables**: Create a `.env` file in the root directory and configure the necessary environment variables:
   env

   ```
   DATABASE_URL=your_database_url

    API_KEY=your_api_key
   ```
3. **Set Up the Database**: Depending on your database setup, you may need to run migrations and seed data:

   ```
   npm run db:migrate

   npm run db:seed
   ```
4. **Run the Development Server**: Start the development server to ensure everything is set up correctly:

```
npm start
```

5. **Running Tests Locally**

Run Unit Tests:

```
npm test
```

1. This will run all unit tests using your configured testing framework (e.g., Jest, Mocha).

Run Integration Tests:

```
npm run test:integration
```

2. This command is specific to integration tests that might require a running instance of the application or database.

Run End-to-End (E2E) Tests:

```
npm run test:e2e
```

E2E tests simulate user interactions and require the application to be running. Ensure the server is running in a separate terminal:

```
npm start
```

3. View Test Coverage:

   ```
   npm run test:coverage
   ```

4. This will generate a coverage report, usually in the `coverage` directory. Open the `index.html` file in your browser to view detailed coverage information.

Test Plan

Click on this link for more information: **Testing Strategy**.

Our testing strategy is divided into three main categories:

1. **Unit Tests**: Focus on individual functions and components. These tests are fast and should cover a wide range of edge cases.
2. **Integration Tests**: Ensure that different parts of the system work together as expected. These tests typically involve multiple units or components.
3. **End-to-End (E2E) Tests**: Simulate real user scenarios to validate the entire application flow. These tests are the most comprehensive but also the slowest.

Test Cases

1. **Unit Tests:**
    - **Functionality**: Test individual functions for correct outputs given a set of inputs.
        - Example: Testing a `calculateSum` function with different sets of numbers.
    - **Component Rendering**: Ensure UI components render correctly with various props.
        - Example: Testing a `Button` component to ensure it renders correctly with different labels and states.
2. **Integration Tests:**
    - **API Endpoints:** Verify that API endpoints return expected results.
        - Example: Testing a `GET /api/users` endpoint to ensure it returns a list of users.
    - **Database Interactions**: Ensure that database operations are performed correctly.
        - Example: Testing a function that retrieves user data from the database.
3. **End-to-End Tests:**
    - **User Authentication:** Validate the entire login flow from entering credentials to accessing the dashboard.
        - Example: Testing the login form, successful authentication, and redirect to the dashboard.
    - **Form Submissions:** Test the complete process of submitting forms and handling responses.
        - Example: Testing a contact form to ensure data is submitted correctly and the user receives a confirmation message.

Coverage Goals

Our goal is to achieve the following test coverage metrics:

- Unit Tests: 90% of functions and components should be covered.
- Integration Tests: 80% of API endpoints and database interactions should be covered.
- End-to-End Tests: 70% of critical user flows should be covered.