Comprehensive Test Plan: Secure Todo Application

1. Introduction

This document outlines the comprehensive test plan and strategy for the Secure Todo Application, encompassing both the backend API and frontend End-to-End (E2E) automation testing. The primary objective is to ensure the overall quality, functionality, and reliability of the application by validating its components at different layers.

2. Testing Scope

The testing strategy covers two primary layers:

2.1. Backend API Testing

Focuses on validating the functionality, reliability, and performance of the Node.js Express backend API endpoints. This layer ensures that the server-side logic and data operations work as expected, independently of the frontend.

Key areas tested:

- **User Authentication:**
  - Successful user login with valid credentials.
  - Rejection of login attempts with invalid username or password.
  - Handling of missing login credentials.
- **Item Management (CRUD Operations):**
  - Retrieval of all existing items.
  - Creation of new items with proper data validation (e.g., missing name).
  - Updating existing items (e.g., name, completion status).
  - Deletion of items, including handling of non-existent items.
- **Authorization:**
  - Ensuring protected endpoints require a valid authentication token.
  - Verification of proper handling for missing or invalid tokens.
- **Utility Endpoint:**
  - Functionality of the `/reset-todos` endpoint for test environment setup.

2.2. Frontend E2E Testing

Simulates real user interactions within the React-based frontend application to validate critical user flows and ensure the entire system (frontend + backend) works cohesively from a user's perspective.

Key user journeys and functionalities tested:

- **User Authentication Flow:**
  - Successful login and redirection to the todo list.
  - Unsuccessful login attempts and corresponding error messages.
  - Successful logout and return to the login screen.
- **Todo Management Flow:**
  - Adding new todo items via the UI.
  - Viewing newly added and existing todo items.
  - Editing (updating) an existing todo item through the UI.
  - Deleting a todo item via the UI.
  - Verification of UI updates after CRUD operations.
- **Data Synchronization:**
  - Confirmation that UI actions accurately reflect changes in the backend and vice-versa.
- **Basic UI Responsiveness:**
  - Verification that key interactive elements are present and functional.

3. Test Coverage Areas

The comprehensive test coverage aims to validate critical paths and ensure robustness across both backend and frontend layers:

- **Functional Coverage:** All core user stories and API functionalities are covered with both positive and negative test cases.
- **Integration Coverage:** E2E tests specifically validate the seamless interaction between the frontend UI, the API, and the underlying data store.
- **Error Handling Coverage:** Tests include scenarios for invalid inputs, unauthorized access, and non-existent resources to ensure appropriate error responses and UI feedback.
- **Data Persistence & State:** Verification that data changes are correctly persisted in the backend and accurately reflected in the frontend's state.
- **Isolation & Reliability:** Automated setup/teardown procedures (e.g., database resets) ensure test independence and repeatability.

4. Tools Used and Why

A robust set of Node.js-based tools has been selected for comprehensive test automation:

- **Jest (for API Tests):**
  - **Why:** A delightful JavaScript Testing Framework with a focus on simplicity and performance. Its integrated assertion library and rich ecosystem make it ideal for unit and integration testing of backend APIs. It offers excellent developer experience and generates detailed reports.
- **Supertest (for API Tests):**

    - **Why:** A high-level abstraction for testing HTTP requests, built on top of Superagent. It allows for fluent API testing by sending requests directly to the Express application (without needing to run a separate HTTP server), making API tests fast, reliable, and easy to write.
- **Mocha (for E2E Tests):**
    - **Why:** A flexible and mature JavaScript test framework that provides a clear structure for writing and organizing E2E test suites (`describe`) and individual test cases (`it`). Its extensibility allows integration with various assertion libraries and reporters.
- **Selenium WebDriver (for E2E Tests):**
    - **Why:** The industry-standard tool for automating web browsers. It enables the simulation of real user interactions (clicks, typing, navigation, waiting for elements) directly within a web browser, providing genuine end-to-end validation of user flows.
- **Chai (for E2E Tests):**
    - **Why:** A versatile BDD/TDD assertion library that provides expressive and readable syntax (`expect(...).to.be.true`, `expect(...).to.include(...)`) for validating test outcomes in E2E scenarios, enhancing test readability.
- **Axios (for E2E Test Setup/Teardown):**
    - **Why:** A promise-based HTTP client used within E2E test hooks (`beforeEach`, `after`) to interact directly with the backend API. This ensures efficient and reliable test data setup (e.g., resetting todos) and cleanup, crucial for maintaining test independence and a clean test environment.
- **Jest-HTML-Reporter (for API Test Reporting):**
    - **Why:** Generates comprehensive, interactive HTML reports for Jest test runs. This provides a clear, visual overview of API test results (pass/fail, duration, details), making it easy for non-technical stakeholders to review and understand test outcomes.
- **Mochawesome (for E2E Test Reporting):**
    - **Why:** An elegant and interactive HTML reporter specifically for Mocha. It produces highly detailed, self-contained HTML reports for E2E test runs, including visual summaries, test durations, and categorized results, significantly enhancing the professionalism and clarity of E2E test reporting.

5. How to Run the Tests

To execute the full suite of automated tests, both the frontend and backend applications need to be running. Browser drivers (like ChromeDriver) must also be properly configured.

Project Structure Overview:

Your project structure is assumed to be as follows, with a main root directory containing three sub-directories:

```
your-main-project-folder/
├── secure-todo-app-frontend/   (Contains the React frontend application)
```

```
├── todo-api-backend/           (Contains the Node.js Express backend API and
its API tests)
└── selenium-e2e-tests/         (Contains the Selenium E2E tests)
```

Prerequisites (Run in separate terminal sessions):

1. **Start Frontend Application:**
    - Open a terminal and navigate to the frontend project directory:
    - `cd your-main-project-folder/secure-todo-app-frontend`
    - Start the development server:
    - `npm start`
    - (The app typically runs on `http://localhost:3000`)
2. **Start Backend API:**
    - Open a **new** terminal and navigate to the backend project directory:
    - `cd your-main-project-folder/todo-api-backend`
    - Start the backend server:
    - `node server.js`
    - (The API typically runs on `http://localhost:5000`)
3. **Install Test Dependencies (One-time setup per directory):**
    - If you haven't already, ensure all `npm install` commands have been run in each of the three project directories (`secure-todo-app-frontend`, `todo-api-backend`, `selenium-e2e-tests`) to install their respective dependencies.
4. **ChromeDriver Setup:**
    - Ensure `ChromeDriver` (or the appropriate WebDriver for your target browser) is installed on your system and its executable path is added to your system's PATH environment variable. This allows Selenium to control the browser.

Execution Steps (Run in new terminal sessions):

1. **Run Backend API Tests:**
    - Open a **new** terminal and navigate to the backend project directory:
    - `cd your-main-project-folder/todo-api-backend`
    - Execute the API tests (Jest):
    - `npm test`
    - **Viewing Reports:**
        - Terminal will show a summary.

        - A detailed HTML report (`test-report.html`) will be generated in the `todo-api-backend` directory.
        - A code coverage report (`coverage/lcov-report/index.html`) will be generated in `todo-api-backend/coverage`.
2. **Run Frontend E2E Tests:**
    - Open a **new** terminal and navigate to the E2E test project directory:
    - `cd your-main-project-folder/selenium-e2e-tests`
    - Execute the E2E tests (Mocha/Selenium):
    - `npm run test:e2e`
    - **Viewing Reports:**
        - Terminal will show a summary.
        - A detailed HTML report (`mochawesome-report/mochawesome.html`) will be generated in `selenium-e2e-tests/mochawesome-report`.

6. Assumptions and Limitations

- **Assumptions:**
    - Node.js and npm (or Yarn) are installed on the execution environment.
    - The backend API is designed to be accessible on `http://localhost:5000`.
    - The frontend React application is designed to be accessible on `http://localhost:3000`.
    - The backend includes a `/reset-todos` endpoint, crucial for test data isolation and cleanup.
    - Operating system environment variables (e.g., PATH for ChromeDriver) are correctly configured.
    - Tests are executed in a stable network environment.
- **Limitations:**
    - **Cross-Browser Compatibility (E2E):** The current E2E setup is primarily configured for Chrome. Extensive cross-browser testing would require additional WebDriver configurations.
    - **Performance Load Testing:** These tests focus on functional correctness, not concurrent user load or server stress testing.
    - **Accessibility Testing:** Dedicated automated accessibility audits are not integrated into the current E2E test suite.
    - **Visual Regression Testing:** The tests do not include tools for visual comparison to detect unintended UI layout or styling changes.
    - **Backend Database State:** The backend uses an in-memory data store for simplicity, meaning data resets on server restarts. For more complex scenarios, a persistent database with dedicated test data management would be considered.
    - **Unit/Integration Test Coverage (Frontend):** This document focuses on E2E and API testing; it does not detail unit or more granular integration tests that might exist within the frontend application itself (e.g., React component tests).