Mini Project: TDL Application (with TDD)

Project Overview

This mini project is the culmination of the course, a chance for learners to apply everything they've practised over the two days to a small but complete feature. It's designed to resemble a realistic frontend application, with the kinds of responsibilities a developer might encounter: managing state, working with user input, dealing with asynchronous data, applying conditional logic, and handling navigation.

The task is to build a To-Do List app using test-driven development. Learners should begin with test cases and allow their code to emerge from the process of making those tests pass. They should work in pairs or small groups where possible, but individual development is also fine. The emphasis here is not on creating something large or beautiful, but on building something confidently and well-tested — with tests that act as a safety net as the code evolves.

Functional Requirements

The app should have two main views: a **Task List** and a **Completed Tasks** view. The user should be able to:

- View a list of current tasks
- Add a new task
- Mark a task as completed
- Navigate between "Active" and "Completed" views
- Fetch existing tasks from a mocked API (asynchronously)
- Handle loading and error states gracefully

Navigation should be handled via routing. State should be local (no need for Redux or external stores), but learners are welcome to lift shared logic into custom hooks if they choose.

TDD Expectations

Learners should start with a simple test that describes the first bit of behaviour. Typicall y, rendering an empty task list. From there, they should follow the **Red Green Refactor** c ycle for every step of the implementation. For example, they might:

- Write a test that checks whether the "Add Task" input appears

- Write another that simulates typing and submitting a task
- Then write a test that asserts the new task appears in the DOM

When introducing asynchronous behaviour, such as fetching initial tasks, learners should mock the service layer using jest.mock() and assert against loading indicators and the resulting list once the data has resolved.

Routing should be tested by rendering the component inside a MemoryRouter with different initial routes, and verifying that the correct content is shown (e.g. showing only completed tasks on the /completed route).

Mock functions (jest.fn()) should be used to verify calls when appropriate. For example, confirming that a “Save” or “Complete” button triggers the expected handler.

Testing Expectations

The final solution should include tests that cover:

- **Component behaviour:** adding, removing, and displaying tasks
- **State changes:** toggling a task between active and complete
- **Routing logic:** only completed tasks appear on the /completed route
- **Async behaviour:** tasks fetched from a mocked service on initial load
- **UI feedback:** “Loading...” appears while fetching, error shown on failure
- **Test doubles:** use of jest.mock() and jest.fn() to isolate behaviour and verify usage

What Success Looks Like

A successful submission isn’t one with the most features. It’s one where the code is led by tests, and where the tests clearly describe and verify the behaviour of the app. Learners should be able to run their test suite with confidence and point to each feature knowing it has been driven by a clear test case.

At the end, allocate time for informal demos or peer walkthroughs. Let learners explain how they structured their tests, how they approached problems, and what surprised them. This helps solidify the thinking behind TDD. Not just the technical syntax, but the mindset shift it encourages.