Mini Project: TDL Application (with TDD)

# Project Overview

This final lab is a chance for learners to consolidate everything they’ve learnt throughout the course. The challenge is to build a small but meaningful Angular application using test-driven development driving the design of components, services, and routing through carefully structured tests.

The project is a To-Do List application, with support for adding, listing, completing, and filtering tasks. Crucially, learners should not jump straight into coding the UI, they should begin each piece of functionality with a failing test and allow the implementation to be guided by the expectations captured in that test.

This is a realistic, structured exercise that mirrors real-life Angular feature development. It includes user interaction, async data, routing, state management, and isolated logic. All testable with the tools and techniques used over the past two days.

# Functional Requirements

Learners will build an Angular app that allows users to:

* View a list of current tasks
* Add new tasks using a reactive form
* Mark tasks as complete
* Navigate between active and completed task views
* Fetch existing tasks from a (mocked) service asynchronously
* Display loading indicators and errors as appropriate

Routing should be set up to switch between /tasks (active tasks) and /completed (completed tasks), and users should be able to navigate easily using a simple nav or toolbar.

# TDD Expectations

The project must be developed using a test-first approach throughout. Encourage learners to treat each test as a design prompt, what should this component do under specific conditions?

For example:

* A test should assert that the “Add Task” input appears and the submit button is disabled if the input is empty.
* A second test might simulate a valid entry and confirm that the form submission triggers a save function or updates the UI.
* Learners should mock the TaskService using a Jest spy and provide it via useValue in the TestBed configuration.
* When testing async fetches, use fixture.whenStable() to wait for the promise or observable to resolve, then make assertions.

For navigation, learners should configure RouterTestingModule and simulate route changes. They should assert that the component renders only the tasks relevant to the current route (e.g. completed tasks only on /completed).

# Testing Expectations

The final test suite should cover:

* **Form and UI interactions**: input fields, validation, submission behaviour
* **Async behaviour**: service call is stubbed and response is tested
* **Component logic**: list updates after adding or completing a task
* **Routing behaviour**: tasks are filtered correctly based on route
* **Error and loading states**: relevant UI feedback is rendered as expected
* **Use of test doubles**: mock TaskService and confirm it’s called with expected values

Tests should follow the Arrange–Act–Assert pattern and use the Angular testing utilities introduced during the course.

# What Success Looks Like

A good submission will include clearly structured test files, minimal but functional components, and evidence that each new piece of logic was developed through testing first. Code should be clean, testable, and expressive. Functionality should work, but the priority is on test quality, not feature quantity.

At the end of the session, allow time for learners to share what they’ve built. Encourage them to reflect on how TDD influenced their design choices and where they felt most (or least) confident. This is also a good time to reinforce how TDD can become a part of everyday Angular development. Not just a lab activity, but a practical mindset.