

Test-Driven Development

React & Angular

Welcome, everyone. Over the next two days, we’re going to dive deep into Test-Driven Development (TDD)—specifically how to apply it effectively in modern frontend development using React or Angular.

You already know how to build features. This course is about improving how you **design, structure, and verify** those features using tests.

Even if you’ve written some tests before, TDD takes it a step further—writing tests first and letting them shape your implementation. That may sound backward at first, but it’s one of the best ways to catch bugs early, build with confidence, and write clean, testable code.

We’ll keep things very practical, hands-on, and grounded in real-world patterns—nothing too academic or theoretical. You’ll come away with tools you can use in your day-to-day work.

1



Overview

**Course Objectives**

* Build strong TDD habits for frontend development
* Write cleaner, more testable React or Angular code
* Get hands-on with practical Jest techniques

What We’ll Cover

* TDD mindset and workflow
* Writing unit and integration tests with Jest
* Testing components, services, and hooks
* Mocking and test doubles in frontend contexts
* Refactoring with safety nets

**Hands-On Focus**

* Paired and solo exercises
* TDD walkthroughs in both frameworks
* Daily mini-projects with evolving requirements

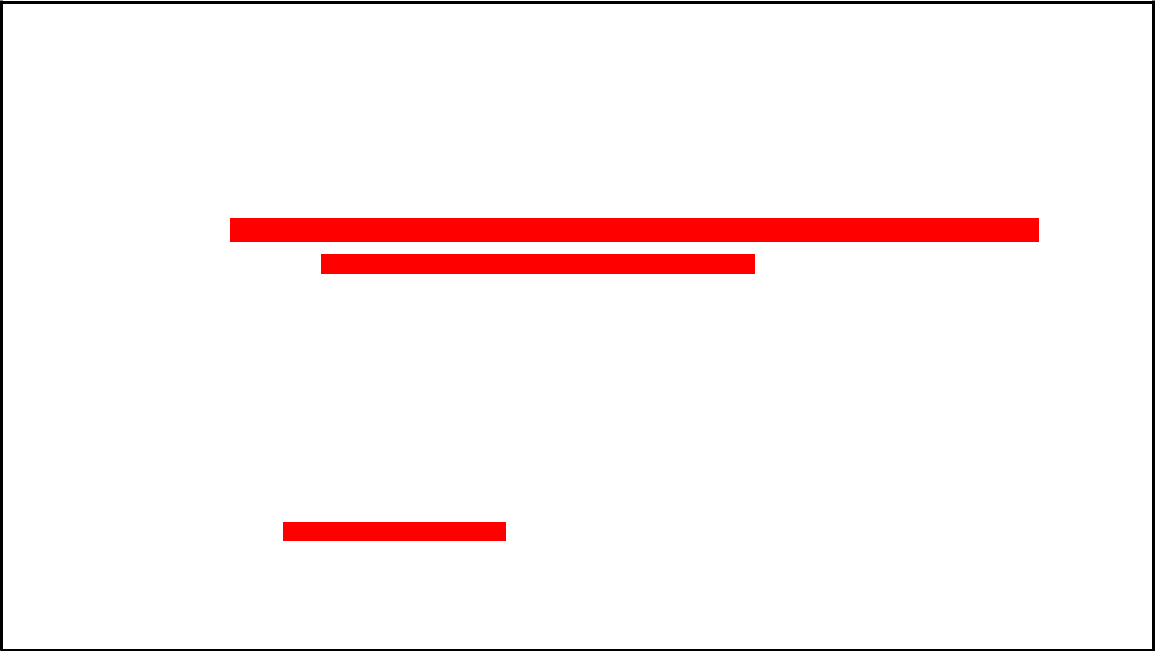
This is where we set the tone. The goal isn’t just to learn how to use Jest—we’re here to build your TDD muscle so you can think like a test-first developer.

We’ll look at both React and Angular examples, but you don’t need to know both. You’ll work in the framework you're most comfortable with.

This course is intensely practical—you’ll be coding, refactoring, and testing constantly. Every section includes guided exercises and real-time feedback loops.

We’re not just looking for ‘green tests’—we’re aiming to build software that’s easier to change without fear. That’s the real value of TDD.

2



Administration

**Access**

* Virtual lab : [PROVIDED FROM US PROBABLY – need to double check]
* Project starter code: **cloned from repo or from provided docs**
* Tools: Node.js, VS Code, browser devtools, test runner (Jest)

**Schedule**

* Morning: 9:30 – 12:30
* Afternoon: 1:30 – 5:00
* Breaks: 10:45 & 3:15 (15 mins)
* Lunch: 12:30 – 1:30

**Support**

* Technical issues: [email/contact phone]
* Help from peers is encouraged too — TDD thrives on discussion

Let’s go over a few practical details before we dive into the code.

You should have access to the lab environment or Git repo for the starter project. We’ll make sure you’re set up to run tests locally using Jest.

We’ll follow a typical training rhythm—morning and afternoon sessions with breaks around midpoints, and an hour lunch.

If you hit any snags—technical or conceptual—speak up! You can message me directly or post in the group chat. Don’t struggle in silence.

Also, lean on each other—**pairing and peer feedback** are part of the TDD process.

3



Course Delivery “Hear and forget, see and remember, do and understand.

**Short Explanations**

* Just enough theory to frame the practice

**Live Demos**

* Watch TDD in action in both React and Angular

**Hands-On Coding**

* Exercises after every major concept
* Paired and solo

**Feedback & Iteration**

* Discuss test outcomes, trade-offs, and real-world relevance

This is not a lecture-heavy course. You won’t be passively absorbing content—you’ll be writing a lot of code, starting today.

I’ll introduce each concept with a quick explanation, show you how it looks in practice with a live demo, and then hand it over to you to do it yourself.

That’s why this quote is front and centre:

**‘Hear and forget, see and remember, do and understand.’**

You’ll retain this much more effectively by getting your hands dirty. And remember: TDD isn’t something you understand immediately—it’s something you develop a feel for through repetition and reflection.”

4



Training Experience

**Two-Way Interaction**

* Ask questions anytime — there are no “dumb” ones
* Share your thought process out loud during coding

**Group Collaboration**

* Pair programming and group review
* Learn from different testing styles and approaches

**Individual Growth**

* Build confidence in writing your own tests
* Develop a test-first mindset that sticks beyond the course

This course isn’t just about teaching TDD—it’s about creating an environment where it makes sense to learn it.

You’ll hear from me, yes—but you’ll also hear from each other.

Ask questions, challenge ideas, share your coding decisions.

That’s where the real learning happens.

We’ll work in pairs, review group outcomes, and reflect on different approaches to the same problem.

TDD is rarely about ‘one right way’—it’s about deliberate decisions and fast feedback.

And through all of that, your goal is personal growth: you’ll leave this course with better instincts for how to build testable code and work with confidence.”

5



Workshop Outcomes

By the end of this workshop, you’ll be able to:

**Apply the TDD lifecycle confidently**

* Red → Green → Refactor — and repeat with purpose

**Write effective tests using Jest**

* For React / Angular projects

Test core frontend behaviours

* Component rendering, state, and events
* Routing and async data flows

**Use test doubles where they count**

* Mock functions, components, and modules cleanly

**Leverage snapshot testing**

* When it's useful — and when to avoid it

**Understand structure:**

* Testable components, custom hooks/services, and contracts between layers

We’re covering a lot in two days, and you’ll be building skills across multiple dimensions of frontend testing.

You’ll internalize the TDD cycle — Red, Green, Refactor — and not just the mechanics, but the thinking behind it.

You’ll get very comfortable writing Jest tests for real React or Angular components — including testing events, state, props/inputs, and async flows.

Mocking can feel like dark magic at first, but we’ll break it down so you’ll know when and how to mock functions, child components, or modules for control and isolation.

You’ll also see how to work with snapshot testing, use it responsibly, and write tests that assert the contract between components and their consumers.

Whether you’re a React or Angular developer, the labs are mirrored so you’ll see how each framework handles testing differently, and where the core

6

principles stay the same.



Assumptions

We’re assuming you already have:

**React or Angular experience**

* You’re comfortable building apps in your chosen framework

**JavaScript or TypeScript fluency**

* You can read, write, and reason about modern JS/TS

**Basic CLI and Git usage**

* You’ve cloned a repo, run commands, and committed code

This isn’t an intro to React, Angular, or JavaScript—we’re assuming you already build apps in one of these frameworks and can move comfortably in a modern codebase.

You’ll be writing code in either JavaScript or TypeScript (depending on your setup), and working from the terminal and a code editor.

You won’t need advanced Git chops, but you should be able to clone repos and commit changes.

If anything’s unfamiliar, speak up early.

The goal isn’t to catch you out—it’s to build your testing muscle on top of the skills you already have.

7



Introductions

When it’s your turn, share:

**Your name and current role**

**Your experience with:**

* React or Angular
* JavaScript/TypeScript
* Testing (any kind — unit, E2E, manual)

One thing you’re hoping to get from this course

Before we dive in, let’s take a minute to get to know each other.

You’ll be working together a lot over the next two days, so it helps to know who’s in the room.

When it’s your turn, tell us your name, what you do, and your experience level with React or Angular, JS/TS, and testing — whether that’s unit testing, manual QA, Cypress, anything.

Then, most importantly, tell us what you’re hoping to get out of this course.

It could be:

* ‘I want to stop writing tests after the fact.’
* ‘I want to get better at mocking in React.’
* ‘I’ve never done TDD and I want to try it.’

This helps me shape how I deliver the content to fit your needs.

-> “If you were honest, how do you feel about writing tests

today?

Love it? Avoid it? Curious? Let us know.”

8



Any Questions?

**Golden Rule:**

“There is no such thing as a stupid

question.”

**Amendment:**

“… even when asked by the instructor.”

**Corollary:**

“A question never resides in a single mind.”

**Ask anytime. Speak up. Stay curious.**

This is your standing invitation: **ask questions—whenever they come up**.

There are no stupid questions. And if you’re wondering something, chances are someone else is too.

I’m a big fan of this version of the Golden Rule:

* There’s no such thing as a stupid question
* Even when *I* ask it
* And chances are, **your question is helping someone else silently wondering the same thing**

So whether it’s ‘What does this Jest error mean?’ or ‘Why would I even write a test here?’—bring it up. That’s how we learn together.

9



TDD

Test

Driven

Development

Now let’s move on to the fundamentals of TDD or **Test-Driven Development**

10



What’s the Problem?

**Tests written *after* implementation**

* Often rushed or skipped entirely
* Don’t shape the design — just “test what we wrote”

**Poor or inconsistent coverage**

* Key logic untested
* Edge cases missed
* Bugs sneak through or reappear

**Low confidence during changes**

* Fear of breaking things
* Refactoring becomes risky
* Developers avoid touching older code

Before we talk about TDD, let’s look at why we’re even here.

In traditional setups, testing is often treated as an **afterthought** — something done when the feature is ‘done’ or even skipped when time runs short.

What we end up with is **incomplete test coverage**, test suites that lag behind the real code, and **low confidence** in whether changes will break things.

This kills velocity over time. Refactoring becomes scary. And we stop trusting our tests to catch regressions.

TDD is a response to these problems. It doesn’t just help you write tests — it helps you **write better, more testable code**, with higher confidence and less risk.

11



TDD – An Approach – A Philosophy

**Tests as Living Documentation**

* Tests show *what the code is supposed to do*
* Clear, readable, and close to the source

**Confidence to Change Code**

* You’ll know when something breaks — and where
* Safer refactoring, easier collaboration

**Better, More Targeted Coverage**

* Test the “why” and “what,” not just the “how”
* Edge cases don’t get forgotten

TDD isn’t just a technique — it’s a way of thinking.

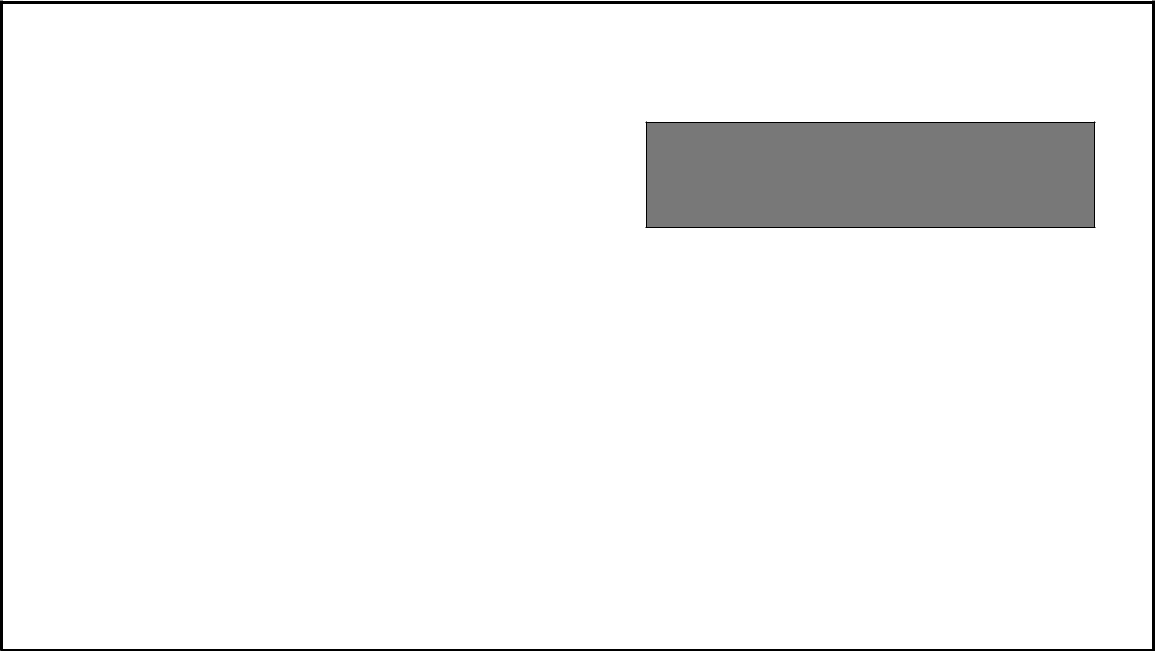
First, your tests become **living documentation**. Anyone can read them and understand the intent behind your code — they’re not buried in a spec document.

Second, you build **confidence**. When your tests fail, they tell you *exactly what broke* and *where*. That makes changing or refactoring code far less scary.

And third, you end up with **stronger, more intentional test coverage** — not because someone said ‘aim for 80%’, but because you’re designing tests around **what matters** from the start.

TDD helps you write code that’s not only functional, but also **more maintainable and explainable**.

12



|  |  |  |
| --- | --- | --- |
| Tests → Development | “ **You wouldn’t build a product without** |  |
| **TDD flips the usual sequence:** |  |
| **quality checks in place first.** |  |

* Traditional: Code → Then test
* TDD: Test → Then code

**Why?**

* Forces you to define *what success looks like*
* Surfaces design decisions early
* Builds with confidence from the start

Let’s ground this with a real-world analogy.

Imagine a car manufacturer building vehicles without having **quality checks** in place until the very end. That’s risky, wasteful, and expensive.

TDD flips our typical dev mindset.

Instead of **writing code and hoping it works**, we start by defining **what working even means** — in the form of a test.

That test becomes a contract. It guides the implementation and forces us to clarify the behaviour we actually want.

It might feel strange at first — writing tests for something that doesn’t exist yet

— but think of it like setting up **inspection points on an assembly line**. The better they are, the smoother the process becomes.

13



Test Coverage – Test After Approach

**Risks of Writing Tests After the Code:**

* Missed logic paths (especially edge cases)
* Testing only what’s *already* working
* False sense of safety — “we have tests” ≠ “we have good tests”
* Fragile code that's hard to refactor safely

**Result:**

* Incomplete coverage
* Bugs sneak in later
* Fear around changing code

This is the trap we fall into with test-after development.

We write the code first — maybe under time pressure — and then come back to write tests afterward.

But by then, the code has already taken shape, and we’re just writing tests that **confirm what we built**, not challenge it.

That means **key logic paths often go untested** — especially edge cases and failure scenarios.

You get that false comfort of having ‘some tests’, but the test suite might not catch regressions, and you won't trust it when you really need it.

Refactoring becomes risky, because you’re not sure what areas the tests are protecting. And the longer this continues, the more fragile the system gets.

14



Test Coverage – Test Before Approach

**Benefits of Writing Tests *Before* (TDD):**

* Forces you to define **expected behaviour** early
* Surfaces edge cases while they still matter
* Guides simpler, more testable logic
* Drives **focused, high-value coverage**

**Result:**

* More complete and intentional coverage
* Confidence to refactor and evolve code
* Fewer bugs, faster feedback

Here’s the flip side of what we just saw.

When you write tests *first*, you’re designing with clarity.

You're thinking: *What do I expect this code to do? What cases should it handle?*

That thought process alone results in **stronger coverage** — not because you chased a percentage, but because you had to be precise about the behaviour from the start.

You’ll catch edge cases sooner. You’ll write smaller, more modular functions because your test needs them to be that way.

And your tests will reflect *why* the code exists, not just what it does.

TDD creates a kind of safety net: when your code starts to drift, your tests will catch it — fast.

15



TDD Lifecycle

**Red:**

* Write a **failing test** that defines the next bit of functionality
  + "It fails because the feature doesn’t exist yet — that's good!"

**Green:**

* Write just enough code to **make the test pass**
  + "No polish, no extras — just pass the test."

**Refactor:**

* **Improve the code** without changing behaviour
  + "Now that it works, clean it up with confidence."

This is the **heartbeat of TDD** — the Red, Green, Refactor loop .

* You start with **Red**: write a test that fails because the functionality isn’t there yet.
* Then **Green**: write just enough code to make it pass. Don't overbuild.
* Then **Refactor**: clean up the code, improve the design — because the test gives you safety.

You’ll repeat this loop dozens of times in even a short session. It feels mechanical at first, but it becomes second nature — like test-driven breathing.

The key is discipline. Don’t skip ahead. Don’t write tests *after* the fact. Let this loop **drive your development decisions.**

16



Source of Test Data

**Your best source: Feature specs & user stories**

* “As a user, I want…” becomes:

*What does success look like?*

*What should never happen?*

**Translate specs into test cases:**

* **Happy paths** → primary test
* **Edge cases** → additional tests
* **Error handling** → defensive tests

**Use tests to:**

* Define **behaviour early**
* Clarify requirements
* Catch spec misunderstandings fast

TDD doesn't start in code — it starts with **the feature you're building.**

User stories, acceptance criteria, product specs — these are all rich sources of test ideas. They're how we know what the system should do, and more importantly, what it should *not* do.

When a story says, ‘As a user, I want to reset my password via email,’ we can turn that into tests:

* What happens when a valid email is entered?
* What about an invalid email?
* What if the user isn’t registered?

TDD helps you **nail down these behaviours early**, reducing rework and catching gaps in understanding before you ship.

17



Upgrade Scenario

**Feature Request (User Story):**

"As a user, I want to see a loading spinner while data is being fetched, so I know something is happening.“

**What should we test?**

* Spinner **shows** while data is loading
* Spinner **hides** once data is fetched
* Content does not render while loading

**Red phase test:**

it('shows a spinner while loading', () => {

render(<MyComponent isLoading={true} />);

expect(screen.getByRole('status')).toHaveTextContent('Loading...');

});

This is where TDD starts to feel real.

Let’s say your product owner requests a new feature: a spinner should appear while the app is fetching data.

Before you build anything, ask:

What does this actually mean from the user's perspective?

*What should always happen?*

What should never happen?

From this one story, we’ve already got three meaningful test cases — and that’s before touching the implementation.

The test you see here is the Red phase — it’ll fail at first, because we haven’t built the logic or the spinner yet.

And that’s perfect. That failure will guide us as we move forward.

18



Requirements Hierarchy

**Requirements Flow Like This:**

1. **Product Owner** defines goals

e.g. “User needs to see feedback while waiting”

1. Dev Team breaks it into functional behaviours

“Show loading indicator until fetch completes”

1. **Frontend Developer** builds UI components

<LoadingSpinner /> , conditional rendering logic

**Tests connect each layer:**

* Are we delivering the intended user behaviour?
* Does the component reflect the spec accurately?

Requirements don’t magically become components — they go through a chain of understanding.

It starts with the Product Owner, who defines the user need or business requirement. That’s often expressed in a user story.

The development team then breaks that down into specific behaviours the app must support.

And finally, the frontend developer turns that into a real UI — using components, props, state, hooks, and so on.

TDD helps you tie it all together. Your test becomes the glue that ensures what you're building at the UI layer actually reflects what was asked for.

If something gets lost in translation, a failing test will catch it early — before the user does.

19



So where is the Unit Test?

**Unit = Smallest testable unit**

* For frontend devs, this usually means:
  + **React** → Component, Hook, or Function
  + **Angular** → Component, Service, or Pipe

**What are we testing?**

* Inputs (props / @Inputs)
* Outputs (events / @Outputs / UI changes)
* Internal logic (state, conditional rendering)

**Your responsibility:**

* Test the component in isolation
* Avoid testing framework internals or browser APIs

When we say ‘unit testing’ in frontend, we’re usually talking about components.

In React, that might be a functional component or a hook. In Angular, it might be a component, service, or pipe.

Our job is to write tests that focus on the behaviour of that unit in isolation

— based on its inputs and outputs.

We don’t need to test React or Angular itself — they’re already tested.

We test:

* What happens when props or inputs change?
* How does the UI respond to user interaction?
* What gets rendered based on state?

TDD keeps these units **small, clear, and verifiable**, which leads to better architecture and less coupling over time.

20



Structure of Tests

**Tests come in many forms:**

* **Acceptance criteria** (from user stories)
* **Spec-by-example** (scenarios with expected outcomes)
* **Truth tables** (inputs vs. outputs mapped clearly)
* All of these define expected state or behaviour

Your job:

Turn these into meaningful **test values and assertions** for your code

Tests don’t always start as code — they start as information.

That might be:

* An acceptance criterion: *‘Button should be disabled until input is valid’*
* A spec-by-example: *‘If value is X, output should be Y’*
* A truth table: listing combinations of inputs and expected results

They all point to stateful expectations — what your component should be doing in a given scenario.

Your job as an engineer is to **extract the logic**, **capture the cases**, and **inject the right data** into your tests — whether that’s props, mocked services, orsimulated user events.

And here’s the key:

**There is no magic bullet.**

It takes critical thinking and experience to structure good tests — just like writing good code.”

21



QuickLab 1: Stepper Component

* Use the TDD cycle (Red → Green → Refactor) to build a counter component that adjusts by a configurable step value.
* What to Test:
  + Initial count = 0
  + + increases, – decreases
  + Step defaults to 1, is configurable via props/@Input

Emphasise small testable steps

Encourage learners to only write code to pass the current test

Support pairing or solo work, TDD is the main skill being developed

**React Tips**

* Start with a failing test: initial value = 0
* Add **+** and **–** buttons; test increment/decrement
* Add a step prop (step={5}) and test its effect
* Use: useState, fireEvent.click, getByText, getByRole

**Angular Tips**

* Write test using TestBed for initial value = 0
* Add **+** and **–** buttons to template and test updates
* Add @Input() step = 1 and test custom step (e.g. step="5")
* Use: ComponentFixture, detectChanges, jest.fn(), click()

22



TDD Takeaways

**TDD is built from incremental unit tests**

* Each test drives one small change
* You write **only** the code needed to pass the current test

**Follow the Red → Green → Refactor rhythm:**

1. Write a failing test
2. Write just enough code to make it pass
3. Refactor (if needed) — tests must still pass

**Work with source control:**

* **Commit after green**
* Push after refactor
* Track small changes — know what broke and when

**Add tests incrementally**

* If all tests fail at once, you won’t know what caused it
* Small test steps = clear failure signals

If there’s one rhythm to take away from TDD, it’s this:

**Red → Green → Refactor → Commit → Push → Repeat.**

You’re writing one test at a time. You’re writing the minimum code to satisfy it.

And you’re using source control — not as an afterthought, but as part of the TDD cycle.

When a test passes, **commit it**. If you refactor, **make sure your tests still pass**, then push.

This helps you isolate changes and catch regressions fast.

And here’s the trap: if you try to do too much at once — multiple tests, multiple features — and something breaks, you won’t know where or why.

**Small steps make debugging easier, changes safer, and progress visible.**

23



Anatomy of a Unit Test

**Arrange**

* Set up the component, inputs, dependencies, mocks

Render component, provide props or inject services

**Act**

* Trigger the behaviour you're testing

Simulate a click, input, API call, or change detection

**Assert**

* Verify the outcome

Check UI, emitted event, updated state, or side effect

Every good test follows the **AAA pattern** — Arrange, Act, Assert.

* Arrange means setting up the test scenario: render the component, mock any services or props, and get the test environment ready.
* Act is where you perform the action that drives the behaviour — like a click, an input, or a service call.
* Assert checks that the result is what you expected — is something rendered? Did a value change? Did a method get called?

This pattern works the same whether you’re using React Testing Library or Angular’s TestBed — the structure is universal.

Once you internalize AAA, your tests become **clearer, more predictable, and easier to debug**.

24



Code Examples

**React** **Angular**

it('displays a greeting after clicking the button', () => { it('displays a greeting after clicking the button', () => {

// Arrange // Arrange

render(<Greeting />); fixture = TestBed.createComponent(GreetingComponent);

component = fixture.componentInstance;

// Act fixture.detectChanges();

fireEvent.click(screen.getByRole('button', { name: 'Say hi' }));

// Act

// Assert const button = fixture.nativeElement.querySelector('button');

expect(screen.getByText('Hello, user!')).toBeInTheDocument(); button.click();

}); fixture.detectChanges();

// Assert

expect(fixture.nativeElement.textContent).toContain('Hello, user!');

});

25



Qualities of a good Unit Test

**A good unit test:**

* Focuses on the CUT (Component Under Test)

The single function, hook, service, or component you're validating

* Is Isolated

No I/O, no real network or DB access — use test doubles (mocks, stubs, spies)

* Is Fast

Runs in **milliseconds** — essential for quick feedback in TDD

* Is Independent

Doesn’t rely on global state or test order — works on its own

* Is Repeatable

Same result every time — avoids flaky or non-deterministic behaviour

* Is Readable

Clear and intentional — tells future devs what matters and why

When we talk about a good unit test, we’re really asking:

is this test giving me value?

Is it helping me change code with confidence?

Here’s what that looks like:

* First, it’s focused on the CUT — the Component Under Test. That could be a component, a service, a function, or a custom hook.
* It’s isolated. That means no database calls, no HTTP requests, no file system operations. We use test doubles — like mocks and stubs — to simulate those interactions.
* It’s fast. Unit tests should run in milliseconds. That speed is what makes TDD possible.
* It’s independent and repeatable — it doesn’t break if another test runs first, and it doesn’t pass only sometimes.
* And finally, it’s readable — future you (or your teammates) should understand what’s being tested and why without having to decode it.

27

If your test follows these principles, it's more likely to survive over time and help you build safer, more maintainable code.



Implementation Example

Make the Test Pass — Then Stop

function Greeting() {

const [showMessage, setShowMessage] = useState(false);

return (

<div>

<button onClick={() => setShowMessage(true)}>Say hi</button> {showMessage && <p>Hello, user!</p>}

</div>

);

}

Let’s look at a perfect example of the Green phase in TDD — where we write just enough code to make the failing test pass.

The test expects a greeting to appear when the button is clicked.

In response, we write the smallest possible component that satisfies that behaviour.

Nothing more.

No greeting before the click.

No extra conditions.

Just enough logic to make the test go green.

This might feel a bit too minimal at first — but that’s the point. TDD is about focus.

One behaviour at a time. The rest will come later, through new tests.

Resist the urge to overbuild.

You’re not writing the whole feature — you’re writing just enough to prove that this slice works.

29



What is Snapshot Testing?

* Captures a component’s rendered output (DOM structure)
* Compares against a stored reference snapshot
* Fails if anything changes — helpful for UI regressions
* When it’s useful:
  + Presentational components (e.g. cards, modals, menus)
  + Static content, layout consistency
  + Only on stable UI — not rapidly changing logic
* Watch out for:
  + Snapshot fatigue: hard to review large diffs
  + Overuse = brittle tests
  + Prefer semantic assertions for behaviour

Snapshot testing sounds great at first.

You render a component, save its DOM output, and Jest will tell you if it ever changes.

But like many good tools, it’s best used in moderation.

Snapshots are great for **layout-heavy** or **pure presentational** components

— things like cards, menus, headers —

where you want to catch unexpected changes in structure or style.

But they become **noise** when used on:

* Dynamic components
* Logic-heavy flows
* Components with conditional rendering

Also, reviewing snapshot diffs in PRs is **hard**.

You often get diffs like <div> changed to <section> — but no context why.

30

So: use it for static UI, and use it sparingly.



Example React Snapshot

it('matches snapshot', () => {

const { asFragment } = render(<ProfileCard name=“Hugh" />); expect(asFragment()).toMatchSnapshot();

});

First run saves the snapshot; future runs compare to it.

If the DOM output changes, the test fails.

29



Interaction Testing

* What to test:
  + Clicks, typing, keyboard use — simulate real interactions
  + Changes in the DOM — visible text, elements appearing/disappearing
  + State reflected in UI, not internal variables
* Use:
  + fireEvent or userEvent in React
  + Direct DOM interaction in Angular (click(), dispatchEvent())
* Follow the Arrange → Act → Assert pattern:
  + Arrange: Render/setup the component
  + Act: Simulate user input
  + Assert: Confirm visible change in the DOM

Interaction testing focuses on **how users interact with your app,** not how your code behaves internally.

You're testing visible outcomes like:

* A message showing up
* A button being disabled
* A modal appearing

This is where your tests start to feel **real.**

You're checking whether the app responds to users as expected.

Whether in **React** or **Angular**, follow the classic **Arrange → Act → Assert** pattern:

1. Arrange your component
2. Act by simulating a user event (like clicking a button)
3. Assert that the DOM updated accordingly

In React, you use fireEvent or userEvent. In Angular, you trigger DOM

33

interactions directly and follow with updates.

fixture.detectChanges() to reflect



Interaction Test Examples

React

it('shows a message after button click', () => {

render(<Greeting />);

fireEvent.click(screen.getByText('Say hi'));

expect(screen.getByText('Hello, user!')).toBeInTheDocument();

});

Angular

it('displays message on click', () => {

const fixture = TestBed.createComponent(GreetingComponent);

fixture.detectChanges();

fixture.nativeElement.querySelector('button').click();

fixture.detectChanges();

expect(fixture.nativeElement.textContent).toContain('Hello, user!’);

});

Both examples show **how to simulate interaction** and verify what changed.

In React, we use fireEvent to simulate a click and assert the new content.

In Angular, we manually trigger the button’s click event, then run detectChanges() to update the DOM, and assert what’s visible.

Both follow the AAA pattern making your tests clear, predictable, and behaviour-driven.

31



CUT and its Dependents

CUT = Component Under Test

* Your focus — the “black box” you’re testing
* React component, Angular service, custom hook, etc.

**Unit Test**

* Talks to the CUT via **inputs and outputs** only
* Doesn’t peek inside or care how it works

**Dependencies (mocked/faked)**

* Anything the CUT relies on:
  + Props / Inputs
  + Services, APIs, global state, child components
* Handled with **test doubles**

When you write a unit test, your mindset should be this:

I don’t care how the thing works — I care what it does.

That’s black-box testing.

The CUT — Component Under Test — could be a React component, an Angular service, or a simple function.

Whatever it is, you test it by interacting with its inputs (like props or method calls), and observing its outputs (like DOM changes, return values, or emitted events).

Everything it relies on — like API clients, child components, or shared services — should be **mocked**.

You're isolating the unit so the test remains focused, fast, and reliable.

This keeps your tests **deterministic**, **focused**, and **easy to reason about**.

32



QuickLab 2: Contact Form

* Use the TDD cycle to build a contact form that:
  + Captures user input
  + Validates required fields
  + Submits data via a callback or event
* What to Test:
  + Inputs render with correct labels
  + Form validation (required fields)
  + Invalid submission shows error
  + Valid submission emits correct data

Encourage learners to test validation logic, user flow, and submission behaviour

Focus on input → state → output

TDD forces clear thinking about UX and form handling

**React Tips**

* Render form; assert inputs appear with getByLabelText
* Simulate input with fireEvent.change
* Submit with empty fields → expect error message
* Submit with valid data → expect onSubmit() call
* Use: useState, fireEvent.submit, getByText, queryByText

**Angular Tips**

* Use TestBed to assert form renders with name + email inputs
* Check initial form state: invalid, untouched
* Simulate valid input → form becomes valid
* Submit with empty fields → expect error message
* Submit valid form → assert EventEmitter or handler is triggered

37

• Use: Reactive Forms, fixture.detectChanges(), FormBuilder



QuickLab 3: Modal

* Use TDD and interaction testing to validate a modal component that:
  + Opens/closes based on props or @Input()
  + Emits a close event
  + Passes accessibility expectations
* What to Test:
  + Modal is hidden by default
  + “Open” button displays modal
  + “Close” button hides modal / emits event
  + Snapshot or role attributes reflect expected DOM structure

Reinforce **UI state + event output testing**

Snapshot ≠ logic testing → use it for **structural checks** only

Recommend role-based assertions (getByRole('dialog')) to support accessibility best practices

**React Instructions**

* Write snapshot tests: closed vs open modal (asFragment() or toMatchSnapshot())
* Simulate “Open Modal” → expect modal content
* Simulate “Close” button → expect content to disappear
* Use isOpen (prop) and onClose (callback)
* Test for role="dialog" or aria-modal

**Angular Instructions**

* Pass @Input() isOpen = true → expect modal content
* @Input() isOpen = false → expect no modal content
* Click “Close” → spy on @Output() onClose.emit()
* Use fixture.detectChanges() after interactions

39

• Optionally test accessibility roles/attributes



Dependents Takeaway

**Tight Coupling Causes Fragile Tests**

* Test breaks when a child component changes, even if behaviour is correct
* Too much reliance on **internal details** of dependencies
* Mocking becomes complex or brittle

**Symptoms:**

* You’re asserting on child implementation, not the CUT
* Tests fail after a refactor, not a behaviour change
* Test setup feels overly complex or bloated

**Solutions:**

* Use **shallow testing** or **test doubles** for child components
* Mock external services (HTTP, DB, global state) cleanly
* Only assert what the **CUT itself controls**

One of the biggest causes of flaky or overcomplicated tests is **tight coupling to dependents** — things like child components, services, or APIs.

If your test breaks every time a child component changes — even when the parent component’s behaviour is correct — that's a red flag.

It usually means you’re reaching inside other components or asserting on things outside the responsibility of the CUT.

The solution is to **mock or stub** those dependents.

Only test what the current component should control:

* Is it rendering the child?
* Is it calling the service correctly?
* Is it emitting the right output?

Keep your test boundaries clean. Let each unit own its own tests.

35



Test Doubles

Now let’s move on to Test Doubles

36



Test doubles types

* **Stub** – *Replace a function and control its output*
  + Use when you want to fake a return value *“Always return X when called.”*
* **Mock** – *Fake a whole dependency and track interactions*
  + Use when you want to verify calls and control behaviour “Was it called? With what arguments?”
* Spy – Wrap a real function to observe behaviour
  + Use when you want to track usage **without replacing logic** *“Did the real function get called?”*

These terms get used interchangeably a lot, but they actually serve different purposes.

* A **stub** is like a controlled shortcut — it replaces a function or method and just gives back a value you decide.
* A mock does that too but also keeps track of how it's used: how many times, with which arguments, and so on. It’s like a stub with memory.
* A **spy** wraps a real function instead of replacing it. You let the original logic run, but you also get a record of how it was used.

In Jest, we typically use jest.fn() for all three

— and then we adjust how we use it depending on whether we’re stubbing, mocking, or spying.

The trick is to pick the right tool for the job

— don’t overmock, and don’t spy unless you really need to.

37



The Importance of Test Isolation

**Isolated tests:**

* Focus only on the **Component Under Test (CUT)**
* Do **not** perform real:
  + Network calls
  + Database access
  + File or I/O operations

**Test Doubles enable isolation:**

* Stubs, mocks, and fakes simulate external dependencies
* A **design technique** that encourages **looser coupling** and **more testable code**

**Isolation benefits:**

* **Confidently test code before all dependencies are ready**
* **Run tests in CI pipelines** without fragile external calls
* **Move faster** without waiting on other teams or services

Isolation isn’t just a testing preference — it’s a fundamental design strategy.

When your tests are isolated:

* They’re fast
* They’re reliable
* And they can run anywhere — including inside your **build pipeline**

We use test doubles (like mocks and stubs) to simulate things the component depends on — maybe a service, a fetch call, a database, or a child component.

This lets us test the unit **without needing the rest of the system to be finished.**

Imagine you’re waiting on another developer to finish an API.

With test doubles, you don’t need to wait — you can simulate the behaviour and continue writing your component.

Also, remember: pipelines are unforgiving.

44

A test that hits a real service or DB is **guaranteed to break at the worst time** — often for reasons that have nothing to do with your code.

Isolation gives you freedom to move quickly, and confidence to refactor without fear.



Stubs – The problem

Component Example:

<FileLoader fileId=“99” />

Or

<app-file-loader [fileId]=“'99'"></app-file-loader>

**The Problem:**

* FileLoader fetches file contents from an external API
* Without stubbing, the test:
  + Hits real network or fails without one
  + Depends on live data, timing, or external setup
  + Becomes slow and flaky

**The Fix: Stub the data fetch**

* Replace real API call with a **controlled return value**
* Make the component testable in isolation

Let’s look at a realistic example that breaks test isolation: a component like FileLoader.

This component might make an API call like GET /files/:id to fetch and render file content.

That’s fine in production — but in a unit test, it causes problems:

* If the API’s not available, the test fails
* If the API returns different data tomorrow, the test fails again
* The test is now slow and fragile

This is where **stubs** come in.

Instead of calling the real API, we stub the fetch method or service to return a known response like { content: 'Sample text’ }.

That gives us a consistent test environment and a fast feedback loop.

The test now controls the data, and we can verify that the component behaves correctly without waiting on external systems.

46



Stubs – The solution

React Angular

// Stub the API call // Stubbed service

jest.mock('./fileService', () => ({ const fileServiceStub = {

fetchFile: jest.fn().mockResolvedValue({ content: 'Hello world!' }), fetchFile: jest.fn().mockResolvedValue({ content: 'Hello world!' }),

})); };

it('shows file content after load', async () => { beforeEach(() => {

render(<FileLoader fileId=“99" />); TestBed.configureTestingModule({

expect(await screen.findByText('Hello world!')).toBeInTheDocument(); declarations: [FileLoaderComponent],

}); providers: [{ provide: FileService, useValue: fileServiceStub }]

}).compileComponents();

});

it('renders file content from stubbed API call', async () => {

const fixture = TestBed.createComponent(FileLoaderComponent);

const component = fixture.componentInstance;

component.fileId = '123';

fixture.detectChanges();

await fixture.whenStable();

expect(fixture.nativeElement.textContent).toContain('Hello world!');

});

Here’s how we solve the external dependency problem using stubs in both React and Angular.

In React, we use jest.mock() to replace the real fetch function with one that returns a canned response.

In Angular, we provide a mocked service using the testing module’s providers array, and again use jest.fn() to return a controlled response.

In both cases, the test:

* Runs quickly
* Doesn’t depend on the real API
* Only fails if the component doesn’t render the response correctly

This lets us **test in isolation** while still simulating real user flows.

40



QuickLab 4: File Loader

* Use TDD and test doubles to verify a component that:
  + Loads async data
  + Shows loading and fallback UI
  + Stubs the real service/API for isolation
* What to Test:
  + "Loading..." appears initially
  + Stubbed data loads correctly
  + Fallback UI shows on failure or empty response
  + No real HTTP requests are made

Reinforce test isolation and behaviour-driven testing

Emphasize waiting for async behavior (findByText, whenStable)

Each framework handles loading state and data injection differently.

**React Tips**

* Create a fileService with fetchFile(id)
* Use jest.mock() to return { content: 'Sample text’ }
* Assert "Loading..." shows initially
* Assert loaded content appears via findByText
* Simulate empty/error response → assert fallback UI
* Use: useEffect, useState, queryByText

**Angular Instructions**

* Create a stubbed FileService using jest.fn().mockResolvedValue(...)
* Provide it in TestBed via useValue
* Use fixture.whenStable() to await async data
* Use: of(), throwError(), ComponentFixture, detectChanges()

41



Stubs - Takeaways

**What Stubs Do Well**

* Simulate responses from APIs, services, or dependencies
* Enable fast, isolated tests with controlled behavior
* Improve test **reliability** and reduce external coupling

**Limitations of Stubs**

* Can lead to **repetitive setup** across many tests
* Hard to maintain when:
  + Behaviour becomes **complex**
  + Tests require **variations** of the same stub
* Do not track usage — they only return values
* Can obscure actual interaction logic

Stubs are powerful — they let us test components in isolation by simulating external behaviour.

They’re especially useful when we’re waiting on APIs, mocking services, or testing fallback states.

But they’re not perfect.

As your app grows, you'll find that **stubbing the same thing over and over** across different test files becomes tedious.

You might start duplicating canned responses or manually recreating service stubs with different return values.

Also — stubs are **passive**.

They don’t tell you whether a method was called or *how* it was used.

They’re great for returning fixed values, but not great if you want to **verify interactions** — that’s where **mocks** and **spies** come in.

42

So: stubs are a great tool.

But they’re not the only one. As complexity grows, we’ll need to **track behaviour**, not just fake outputs.



Test Doubles and Verification Testing

Now let’s move on to **Verification Testing**

43



Spy Object

**What is a Spy?**

* A **spy** wraps a real function — it still runs as normal
* Allows you **observe usage**:
  + Was it called?
  + With what arguments?
  + How many times?
* Keeps the **actual behaviour intact**

**When to use a Spy:**

* You want to **verify behaviour**, but still run the real logic
* You're testing internal methods (Angular)
* You're watching utility function usage (React)

A **spy** is like a passive observer — it watches a real function without replacing its behaviour.

Unlike mocks or stubs, which stop the original logic from running, **spies let everything run normally**.

You just get to monitor what happened: whether it was called, how often, and with what arguments.

This is great when:

* You still want the actual logic to execute
* You want to track internal class methods in Angular
* Or you want to observe utility functions used inside a React component

Spies are great for **behaviour verification**, especially when you don’t want to lose the logic’s real effect — just want to confirm that it happened.

44

Spying with Jest

React

jest.spyOn(utils, 'formatName');

it('calls formatName with the correct value', () => {

render(<UserProfile name=“Erin" />);

expect(utils.formatName).toHaveBeenCalledWith(Erin');

});

Angular



it('calls the real formatName method', () => {

const component = new UserProfileComponent();

const spy = jest.spyOn(component, 'formatName');

component.displayName(‘Erin');

expect(spy).toHaveBeenCalledWith(‘Erin');

});

These two examples show spies in action in both React and Angular:

* In **React**, we spy on a utility function imported from a module and verify that it was called with the expected input when the component renders.
* In **Angular**, we create a component, spy on one of its internal methods, call a function that would trigger it, and assert that the method was actually used.

Both cases keep the original behaviour intact — we're not replacing anything, we're just **watching** what happens.

45



Mock Objects

**Mock Object = Test Double with Memory**

* Replaces a real dependency with a **fake** that tracks usage
* Created using tools like jest.fn()or jest.mock()

**What You Can Check:**

• Was it called?

• How many times?

|  |  |  |  |
| --- | --- | --- | --- |
| • With what arguments? | “ | **Mocks observe and report** |  |
| **— so you don’t have to guess what happened.** |  |
| • What did it return? |  |

Think of a **mock object** like a **mystery shopper** in a store.

They don’t just act like a normal customer — they **watch and record everything**:

* Did the cashier greet them?
* What was said during the interaction?
* How many times did they ask for help?

In testing, a mock object works the same way. It steps in for a real service or function

— like a save handler, API call, or injected Angular service — and keeps a full record of what happened:

* + Was it called?
  + With what?
  + How often?

This is powerful for **verification testing** — especially when the behavior is what matters more than the result.

In Jest, you can create these mocks easily with jest.fn() or jest.mock(), and

54

use built-in matchers like toHaveBeenCalledWith() to assert those interactions.



Mocks for Verification

|  |  |  |
| --- | --- | --- |
| React | Angular |  |
| const saveMock = jest.fn(); | const mockService = { |  |
|  | saveUser: jest.fn(), |  |
| render(<SaveButton onSave={saveMock} />); | }; |  |
| fireEvent.click(screen.getByText('Save')); | beforeEach(() => { |  |
|  |  |
| expect(saveMock).toHaveBeenCalledWith('user1'); | TestBed.configureTestingModule({ |  |
|  | declarations: [UserFormComponent], |  |
|  | providers: [{ provide: UserService, useValue: mockService }] |  |
|  | }).compileComponents(); |  |
|  | }); |  |
|  | it('calls saveUser when form is submitted', () => { |  |
|  | const fixture = TestBed.createComponent(UserFormComponent); |  |
|  | const component = fixture.componentInstance; |  |
|  | component.userId = 'user1'; |  |
|  | fixture.detectChanges(); |  |
|  | component.submitForm(); // simulate submission |  |
|  | expect(mockService.saveUser).toHaveBeenCalledWith('user1'); |  |
|  | }); |  |

Mocks are your go-to tool when you want to **verify interactions** — not just data or UI changes.

You’re not checking what the component *renders* — you’re checking what it *does*.

In React, that might be confirming a prop function was called, like onSave() when a button is clicked.

We pass a jest.fn() into the component, trigger the behavior, then use toHaveBeenCalledWith() to make sure the interaction was correct.

Now, let’s talk about Angular. It looks a little different because we use **dependency injection**. Here, we:

* Replace the real UserService with a mock version in the test module
* Use jest.fn() for the saveUser() method
* Manually call the submitForm() method in our test
* Then verify that our mock method was called with 'user1’

This is a very clean and testable pattern in Angular: mock the service, trigger the interaction, and assert that it happened.

56

No HTTP needed. No side effects. Just a simple verification of behaviour.

Mocks like this give you the confidence to know your component is triggering the **right actions at the right time** — even when it doesn’t produce visible output.



QuickLab 5: Save Button / User Form

React Angular

* Use jest.fn() to verify that a component correctly calls an onSave callback with user input, and prevents submission when input is invalid.
* What to Test:
  + Text input updates correctly using fireEvent.change
  + Clicking “Save” with valid input calls onSave(value)
  + Clicking “Save” with empty input does not call onSave
  + (Optional) Input resets or shows confirmation after submission
* Test that a form submission correctly triggers a method on a mocked UserService using jest.fn(), and prevents submission when the form is invalid.
* What to Test:
  + Form renders with required input fields
  + Filling in the form updates Reactive Form state
  + Submitting valid form calls UserService.saveUser(data)
  + Submitting invalid form does not call the service
  + (Optional) Success message appears or form is reset after submit

This lab is about **mocking external handlers** and testing **behaviour**, not just UI.

Encourage learners to write both **positive (happy path)** and **negative (guard condition)** tests.

**React Tips**

* Create a SaveButton component with onSave(value: string) prop
* Use jest.fn() to mock the onSave callback
* Simulate typing into an input and clicking “Save”
* Assert:
  + onSave is called with the correct value
  + Empty submission **does not** call onSave
* Use: fireEvent.change, fireEvent.click, getByLabelText, getByText

**Angular Tips**

* Create a UserFormComponent using Reactive Forms
* Mock UserService.saveUser() with jest.fn()
* Fill in the form using querySelector
* Trigger form submission and verify:

58

* + Valid form → saveUser() called with correct data
  + Invalid form → saveUser() is **not** called
* Use: TestBed, ComponentFixture, spyOn, detectChanges(), whenStable()



Mocking Objects Takeaway

* Mocks vs Stubs – Key Difference

**Use stubs when:**

* You just need **fake data** to drive behaviour
* The interaction doesn’t matter — only the result does

**Use mocks when:**

* You want to **verify the relationship** between the CUT and a dependency
* You care if a method was called, how often, and with what

Let’s settle something that often causes confusion:

**Mocks are not stubs.**

They’re both types of test doubles, yes — but they serve different purposes.

* **Stubs** are all about control: *return this value when called.*
* **Mocks** are all about verification: *was this function called? how?*

People often say ‘mocking data,’ but that’s technically **stubbing** — you’re not verifying anything, just shaping the behaviour for the test.

A lot of testing tools (including jest.fn()) blur the line, because they let you create a function that does both.

That’s powerful, but it’s important to know **why** you’re using it.

So remember:

* If you just need predictable data to test the component’s flow → **stub it**
* If you want to make sure your component called a service correctly → **mock it**

60

Understanding this helps you write **more intentional, meaningful tests** — not just working ones.



Testing Async Behaviour in Components

**Key Practices:**

* Use built-in async utilities:
  + **waitFor**, **findBy\***, or **fixture.whenStable()** for async updates

Always check:

* **Loading state**
* **Success state**
* **Error state (if applicable)**

**Avoid:**

* Manual **setTimeout()** — it's slow and unreliable
* Making real HTTP calls — mock responses instead

When you test async behaviour in frontend components, you’re testing how the UI reacts over **time**, not just state.

These are often:

* Data-fetching components
* Delayed render effects (like spinners or transitions)
* Components using useEffect, ngOnInit, or Observables

The key is to **let the framework tell you when it’s ready** — don’t guess or hardcode delays.

Your job is to assert what users would see: a loading state first, followed by a successful (or error) state.

50



Async Testing Examples

React Angular

render(<UserList />); const fixture = TestBed.createComponent(UserListComponent);

expect(screen.getByText('Loading...')).toBeInTheDocument(); fixture.detectChanges();

await waitFor(() => await fixture.whenStable();

expect(screen.getByText(‘Boris')).toBeInTheDocument() expect(fixture.nativeElement.textContent).toContain(‘Boris');

);

In **React**, we render the component, assert that 'Loading...' appears first, then use waitFor to wait until 'Alice' appears in the DOM.

In **Angular**, we render with TestBed, trigger change detection, and then use fixture.whenStable() to wait for the async logic (e.g. service call or promise) to complete.

Both frameworks provide structured ways to handle asynchrony in tests

— no need to guess how long to wait.

51



QuickLabs 6: User List

* Test a component that loads user data asynchronously, and verify all UI states:
  + Loading
  + Success
  + Error
* What to Test:
  + Loading message appears first
  + User data renders correctly after fetch
  + Error message displays on failure
  + No real HTTP calls are made

Focus on **verifying all three async states**

Use mocks to control timing and simulate both success & failure

Reinforce use of findByText (React) and whenStable() (Angular) to await async UI updates

**React Tips**

* Create a userService.fetchUsers() that returns a Promise<User[]>
* Use jest.mock() to stub the service in tests
* Write tests for:
  + "Loading..." visible on initial render
  + Fetched user(s) rendered using findByText()
  + Rejected promise → error message or fallback UI shown
* Use: useEffect, useState, mockResolvedValue, mockRejectedValue

**Angular Tips**

* Create a UserService.getUsers() that returns Observable<User[]>
* Stub the service with jest.fn() and of() or throwError()
* Write tests to assert:

64

* + "Loading…” appears before data resolves
  + User list renders after success
  + Error message shows on failure
* Use: TestBed, detectChanges(), whenStable(), querySelector



Testing Route-Driven Components

**Don’t test the whole app — test the route context**

* Use **React MemoryRouter** or **Angular RouterTestingModule**
* Simulate navigation via mocks, not real routing
* Assert what the component **renders based on route state**

**Focus your tests on:**

* Rendered content for a given route
* **Active classes**, breadcrumbs, titles
* **Redirect behaviour** or conditional navigation

When you’re testing components that depend on the current route, the goal isn’t to test the full routing system

— that’s the framework’s job.

Your goal is to **simulate a routing context** and verify that the component behaves correctly.

In **React**, wrap your component with a <MemoryRouter> and set initialEntries to fake the current route. For example:

<MemoryRouter initialEntries={['/dashboard']}> <Dashboard />

</MemoryRouter>

In **Angular**, you’ll use RouterTestingModule to provide routing support, and **spy on navigateByUrl()** using jest.spyOn() if you want to verify navigationbehaviour without triggering it.

Common assertions:

• Does the component show the correct content for the current route?

66

* Are active nav items highlighted?
* Are users redirected based on logic (e.g. auth)?

This keeps your routing tests **isolated, fast, and focused**.



Route-Driven Test Example

|  |  |  |
| --- | --- | --- |
| React | Angular |  |
| // Dashboard.jsx | // user-page.component.ts |  |
| export function Dashboard() { | @Component({ ... }) |  |
| const location = useLocation(); | export class UserPageComponent { |  |
| return <h1>Current path: {location.pathname}</h1>; | constructor(private router: Router) {} |  |
| } | logout() { |  |
|  |  |
| // Dashboard.test.jsx | this.router.navigateByUrl('/login'); |  |
| import { MemoryRouter } from 'react-router-dom'; | } |  |
| it('displays current route path', () => { | } |  |
|  |  |
| render( | // user-page.component.spec.ts |  |
| <MemoryRouter initialEntries={['/dashboard']}> | it('redirects to login on logout', () => { |  |
| <Dashboard /> | const router = TestBed.inject(Router); |  |
| </MemoryRouter> | const spy = jest.spyOn(router, 'navigateByUrl'); |  |
| ); |  |  |
| expect(screen.getByText('Current path: /dashboard')).toBeInTheDocument(); | const fixture = TestBed.createComponent(UserPageComponent); |  |
| }); | fixture.componentInstance.logout(); |  |
|  | expect(spy).toHaveBeenCalledWith('/login'); |  |
|  | }); |  |

**React**

* **MemoryRouter** simulates navigation without needing a full app
* **initialEntries** sets the fake current path
* Use this for route-based content, redirects, or auth-protected routes

**Angular**

* **RouterTestingModule** injects a mock router
* You can spy on **navigateByUrl()** with Jest to verify redirects
* No need to trigger real navigation — just confirm the intent

In both frameworks, treat the router as a **context provider**, not the thing under test.

**Simulate**, don’t depend.

54



Testing Custom Logic

**Why test logic outside of components?**

* Keeps tests **fast, focused, and framework-light**
* Avoids unnecessary rendering or setup
* Encourages:
  + **Separation of concerns**
  + **Reusable**, **pure**, and **testable** logic

**Your targets:**

* **React:** Custom hooks (useCounter, userAuth, etc.)
* **Angular:** Injectable services (UserService, FormHelperService, etc.)

**Test directly:**

* Hook return values
* Service methods
* Side-effect-free logic

Some of the **cleanest, most effective tests** you can write live *outside* of the component tree.

When you extract logic into:

* **Hooks** (React) or
* **Services** (Angular),

…you make it easier to test without needing to worry about DOM, routing, or full render trees.

This keeps your logic:

* Small
* Focused
* **Testable in isolation**

You’re testing what matters without any extra setup.

55



Logic Testing Examples

React Angular

const { result } = renderHook(() => useCounter()); const service = TestBed.inject(MyService);

expect(result.current.count).toBe(0); jest.spyOn(apiClient, 'fetchData').mockReturnValue(of(mockData));

expect(service.getProcessedData()).toEqual(expected);

In **React**, renderHook() is like render() but for logic — it runs your hook, and gives you access to the returned values.

Tests the return value of your custom hook directly No need to render a full component

In **Angular**, you create the service using TestBed.inject(), mock any dependent services using jest.spyOn() or jest.fn(), and then test the output directly.

Inject mock dependencies and call service methods

Works great for transformation logic, data handling, or conditional flows

This keeps your tests **pure**, **fast**, and focused on **what the logic is supposed to do**, not how it’s rendered.

56



QuickLabs 7: Navigation Menu

* Test route-aware navigation components that:
  + Update based on current route
  + Trigger navigation on user interaction
  + Optionally style or render based on route state
* What to Test:
  + Initial route renders correct page
  + Clicking a nav item triggers navigation (React: view change; Angular: spy assertion)
  + Navigation UI reacts to route state
  + No real navigation occurs in test

Emphasize testing **routing context and outcomes**, not the router itself

Use MemoryRouter (React) or RouterTestingModule (Angular) to simulate navigation

In Angular, assert router interactions; in React, assert visible content changes

**React Tips**

* Wrap component in <MemoryRouter initialEntries={['/home']}>
* Use fireEvent.click to simulate link clicks
* Assert:
  + Correct content is rendered on initial route
  + Clicking a link updates the route and renders new content
  + (Optional) Active link styling or conditional rendering
* Use: react-router-dom, getByText, queryByText

**Angular Tips**

* Use RouterTestingModule to simulate routing
* Spy on router.navigateByUrl() with jest.spyOn()
* Trigger navigation via link or button click

71

* Assert:
  + navigateByUrl() is called with correct path
  + Component reflects changes with detectChanges()
  + (Optional) Conditional rendering or styling based on current route
* Use: TestBed, ComponentFixture, Router, @angular/router



Untestable Code



What is Untestable Code?

**Tightly Coupled Code**

* Business logic is locked inside UI rendering
* Direct calls to services, storage, or global objects
* No separation of concerns

**Hidden or Unreachable State**

* No clear way to observe changes or outcomes
* Behaviour depends on timeouts, DOM quirks, or external effects

**Difficult Dependencies**

* Hard-coded API calls or direct new instantiations
* No DI (Dependency Injection) or interface boundaries
* Dependencies can’t be replaced with stubs or mocks

Untestable code isn’t bad because it’s broken — it’s bad because it’s **hard to prove right** and **risky to change**.

There are a few common patterns that make code untestable:

1. **Tight coupling** — for example, when your data-fetching logic lives *inside* the render method or lifecycle hook.

There’s no way to isolate it or test it without bringing the whole component along for the ride.

1. **Hidden state** — like values buried in closures, set with timeouts, or only visible through side effects.

If your test can’t observe the outcome, it can’t verify the behaviour.

1. **Difficult dependencies** — hardcoded services, global functions, or direct instantiations (new SomeService()), which can’t easily be

replaced with test doubles.

These make isolation and control nearly impossible.

Our job is to **design for testability** — which often means designing for *clarity and separation*.

74

If it’s hard to test, it’s probably hard to change safely too.



What is a Contract?

* A **contract** is the expected interface between units
  + e.g. props, API responses, injected services
* Contract tests ask:
  + “Did I get what I expected?”
* Prevents silent mismatches between frontend and backend
* Catch errors **before runtime** — during development or build

Contract testing helps ensure that **one part of your app gets what it expects from another** — whether that’s a component receiving the right props, or aservice getting the correct shape of data from an API.

* In **React**, you can test the ‘contract’ by:
  + Mocking a fetch response and asserting the shape of the returned object
  + Or verifying how a component behaves based on expected props
* In **Angular**, it’s about **mocking the service** and ensuring the component or consumer **correctly expects and reacts** to that output structure.

Think of it like: *If the backend returns undefined instead of user.name, would my UI blow up or recover gracefully?*

There are tools for more formal contract testing:

* **Pact** (for consumer-driven contracts)
* **OpenAPI-based mocks** for verifying backend schemas

76

But at the unit level, it’s about **being intentional with what you expect from other layers**.



Designing Code That’s Easy to Test

**Design Principles:**

* **Separate logic from UI**
  + e.g. move business rules to hooks (React) or services (Angular)
* **Inject dependencies** instead of hard-coding them
  + Easier to replace in tests
* **Avoid global state** or side effects in components
  + Keep state and DOM interaction clean and predictable

This slide is the **antidote to untestable code**.

Testable code usually has three traits:

1. It’s **small** — focused on doing one thing well.
2. It’s **pure** — the logic doesn’t depend on timers, DOM, or random side effects.
3. It’s **injectable** — its dependencies can be replaced in a test without hacking around them.

In **React**, this might mean:

* Pulling logic out of components into **custom hooks**
* Passing functions via props instead of hardcoding API calls

In **Angular**, it means:

* Using **services** for logic and **constructor injection** for dependencies
* Avoiding side effects in components and using @Input/@Output clearly

The goal: build components that are **easy to understand in isolation** and easy to test without setting up the entire app around them.

78

This is testable architecture

— and it leads to cleaner code even when you’re not testing.



Mini Project: To-Do List using TDD

* Build a small but realistic To-Do List application, using TDD to drive every feature:
  + Start with a failing test
  + Let the test shape your component, service, or route
  + Deliver a clean, test-covered app.
* What to Test:
  + Initial render and empty state
  + Adding a task and marking it complete
  + Filtering by route
  + Async fetch with mock service
  + UI feedback: loading/error messages
  + Proper use of test doubles (jest.mock(), jest.fn())

This is the capstone so let learners take ownership.

Remind them to keep their test loop small and focused: **Red → Green → Refactor**

Encourage pair walkthroughs or demos to explain test-driven design decisions

**React Tips**

* Write tests before rendering or interaction logic
* Use jest.mock() for async data fetches
* Test:
  + Adding, displaying, and completing tasks
  + Routing (/tasks vs /completed) via MemoryRouter
  + Async UI states (Loading…, errors)
  + Interaction callbacks via jest.fn()

**Angular Tips**

* Drive components and services from tests
* Mock TaskService using jest.fn() + useValue
* Use RouterTestingModule for route assertions

80

• Test:

* Reactive form interaction and validation
* Task list rendering and completion
* Async loading and error handling
* Navigation between /tasks and /completed



|  |  |  |
| --- | --- | --- |
| Conclusion | “ **TDD isn't hard, skipping it just feels easier.** |  |
| **What We Learned** |  |

* TDD = **Write test → Write**
* Helps build **cleaner, more testable code**
* Catches bugs **before** they become problems
* Improves **confidence**, clarity, and collaboration

**Keep Practicing**

* TDD is a **discipline**, not a tool
* The more you do it, the **more natural it feels**
* Start small — one test at a time**code → Refactor**

Let’s wrap this up with the core message:

TDD is **not about writing lots of tests** — it’s about changing how we think when we write code.

It’s a **simple cycle**:

1. Write a test for the next behavior
2. Make it pass
3. Clean it up

Then repeat.

That simplicity is what gives it power — and what makes your code safer, easier to maintain, and more trustworthy.

But like any habit, TDD takes **practice**. It won’t feel natural overnight.

So start small. Write one test first. Build from there.

The best developers I know don’t write perfect code — they write code they’re confident they can change.

82

**That’s what TDD gives you.**

So take what you’ve learned and **try it out in your next feature, your next bugfix, or even your next personal project.**.



Feedback and Final Thoughts

**Thank You for Being Part of This**

* You showed up, coded hard, asked great questions
* TDD is a skill — and you're already on the path

**Your Feedback Matters**

* What worked well?
* What could be improved next time?
* Help us make this better for the next team

Before we finish, I just want to say a **genuine thank you** — for the time, energy, curiosity, and critical thinking you brought over the last two days.

TDD can feel awkward at first, but you pushed through the discomfort and took it seriously. That matters.

Now, we’d love your **feedback**:

* What landed well?
* What was confusing or rushed?
* What could we do better for the next group?

Please take a few minutes to fill out the feedback form — it helps us improve.

And finally — you’re not on your own now. If you want help applying TDD at work, or just want to bounce around a test idea, feel free to reach out.

Thanks you all again!

64