

Testing

Test pyramid, unit tests, mocking, React Testing Library, E2E

1. What to test

3. Testing React components

2. Unit tests (Arrange-Act-Assert)

4. API and end-to-end tests

Draw on any slide, then Present (arrow keys) or export the board as a PDF.

1. What to test

Definition

Tests give confidence to change code. Write many fast unit tests, some integration tests, and a few end-to-end tests of critical user flows.

Level	Tests	Tools
Unit	one function/module in isolation	Vitest, Jest
Integration	pieces together: API + DB, component + hooks	Supertest, Testing Library, test DB
End-to-end	the real app in a browser	Playwright, Cypress

Tip

Test behaviour, not implementation details — tests should survive a refactor that keeps behaviour the same.

2. Unit tests (Arrange-Act-Assert)

A tiny test runner to show the idea

```
function slugify(s) {
  return s.toLowerCase().trim().replace(/^[a-z0-9]+/g, '-').replace(/^-|$/g, '')
}

const tests = []
const test = (name, fn) => tests.push({ name, fn })
const expectEqual = (actual, expected) => {
  if (actual !== expected) throw new Error(`expected "${expected}", got "${actual}"`)
}

test('lower-cases and joins words', () => expectEqual(slugify('Hello World'), 'hello-world'))
test('drops punctuation', () => expectEqual(slugify('React 19: new!'), 'react-19-new'))
test('trims dashes', () => expectEqual(slugify(' --Hi-- '), 'hi'))

for (const t of tests) {
  // ... 2 more lines in the notes
}
```

The same with Vitest / Jest

```
import { describe, it, expect, vi } from 'vitest'

describe('slugify', () => {
  it('lower-cases and joins words', () => {
    expect(slugify('Hello World')).toBe('hello-world')
  })
})

it('calls the API once and returns the user', async () => {
  const fetchUser = vi.fn().mockResolvedValue({ id: 1, name: 'Asha' }) // mock
  const service = makeUserService({ fetchUser }) // inject the dependency
  await expect(service.getName(1)).resolves.toBe('Asha')
  expect(fetchUser).toHaveBeenCalledTimes(1)
})
```

- Mock: replaces a dependency and records calls. Stub: returns canned data. Spy: wraps a real function to observe it.
- Mock the boundaries (network, time, randomness), not your own logic.
- Fake timers (vi.useFakeTimers()) for debounce/timeout code.

3. Testing React components

React Testing Library: test like a user

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'

it('shows an error for an invalid email', async () => {
  render(<SignUpForm onSubmit={vi.fn()} />)
  await userEvent.type(screen.getByLabelText(/email/i), 'not-an-email')
  await userEvent.click(screen.getByRole('button', { name: /sign up/i }))
  expect(await screen.findByText(/valid email/i)).toBeInTheDocument()
})
```

- Query priority: `getByRole` → `getByLabelText` → `getByText` → `getByTestId` (last resort).
- `getBy` throws if missing, `queryBy` returns null (use it to assert absence), `findBy` waits (async).
- Mock the network with MSW so components use their real data-fetching code.

4. API and end-to-end tests

API test with Supertest

```
import request from 'supertest'
import { app } from '../src/app.js'

it('rejects a user without an email', async () => {
  const res = await request(app).post('/api/users').send({ name: 'Asha' })
  expect(res.status).toBe(400)
  expect(res.body.error).toMatch(/invalid/i)
})
```

E2E with Playwright

```
import { test, expect } from '@playwright/test'

test('user can log in', async ({ page }) => {
  await page.goto('/login')
  await page.getByLabel('Email').fill('asha@example.com')
  await page.getByLabel('Password').fill('correct horse')
  await page.getByRole('button', { name: 'Log in' }).click()
  await expect(page.getByRole('heading', { name: 'Dashboard' })).toBeVisible()
})
```

- Use a separate test database; reset or seed it per test file.
- E2E: test the critical paths (sign-up, checkout, payment), not every edge case.
- Coverage is a guide, not a goal — 100% coverage with weak assertions proves little.
- TDD: write a failing test (red), make it pass (green), clean up (refactor).

Revise

Many unit, some integration, few E2E. Arrange-Act-Assert, mock only boundaries, test behaviour. RTL queries by role/label like a user; MSW for the network; Supertest for APIs; Playwright for critical flows; run all of it in CI.

Rapid-fire Q&A (1/2)

What is the testing pyramid?

Lots of fast unit tests at the base, fewer integration tests, and a small number of slow end-to-end tests at the top.

Unit vs integration vs E2E tests?

Unit tests check one piece in isolation; integration tests check pieces working together; E2E tests drive the real app like a user.

Mock vs stub vs spy?

A stub returns canned data, a mock replaces a dependency and records calls for assertions, a spy wraps a real function to observe it.

What should you mock?

Boundaries you don't control or that are slow or random: network, time, randomness, third-party services — not your own logic.

getBy vs queryBy vs findBy in Testing Library?

getBy throws if not found, queryBy returns null (for asserting absence), findBy waits asynchronously.

What is TDD?

Write a failing test first, write just enough code to pass it, then refactor — red, green, refactor.

Rapid-fire Q&A (2/2)

Is 100% code coverage the goal?

No — coverage shows what ran, not whether behaviour is checked. Focus on meaningful assertions for important paths.

How do you test code that uses `setTimeout`?

With fake timers (`vi.useFakeTimers` / `jest.useFakeTimers`) and advancing time manually.