Introduction

TDD in JavaScript

QA

Learn. To Change.

Overview

Course Goals

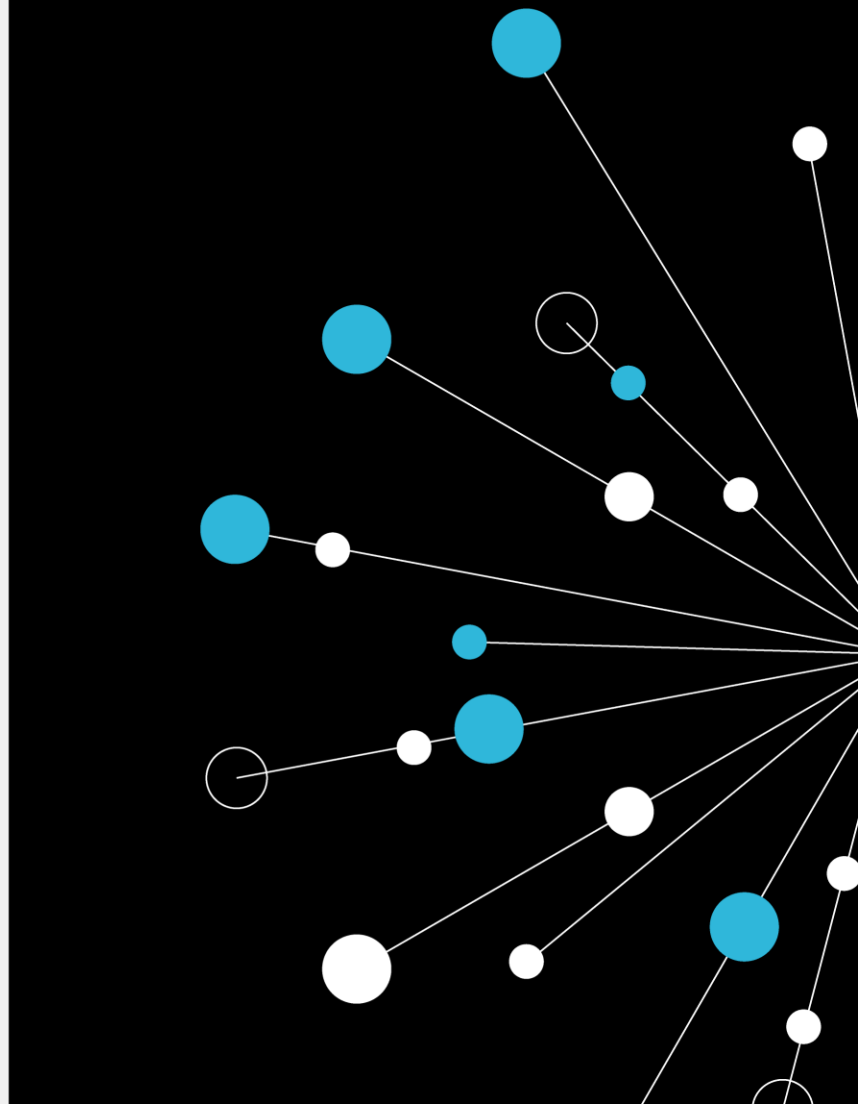
- Develop effective TDD habits using pure JavaScript
- Write cleaner, more maintainable, and testable code
- Gain practical experience with Jest testing techniques

What You'll Learn

- TDD workflow: Red → Green → Refactor cycle
- Writing unit and integration tests with Jest
- Using mocks and test doubles effectively
- Safe refactoring practices guided by tests

Hands-On Approach:

- Practical QuickLabs
- Mini-project with incremental complexity
- Immediate feedback and iteration on tests



QA

Course Delivery

Focused Explanations:

- Clear, concise theory just enough to support practice.

Live Demonstrations

- Observe the TDD process step-by-step in action using Jest.

Hands-On Exercises

- Practical coding tasks after each major topic.

Continuous Feedback & Iteration

- Review tests and outcomes together.
- Discuss real-world relevance and practical trade-offs.

> Hear and forget, see and remember, do and understand.

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

QA

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

Test core JavaScript behaviours

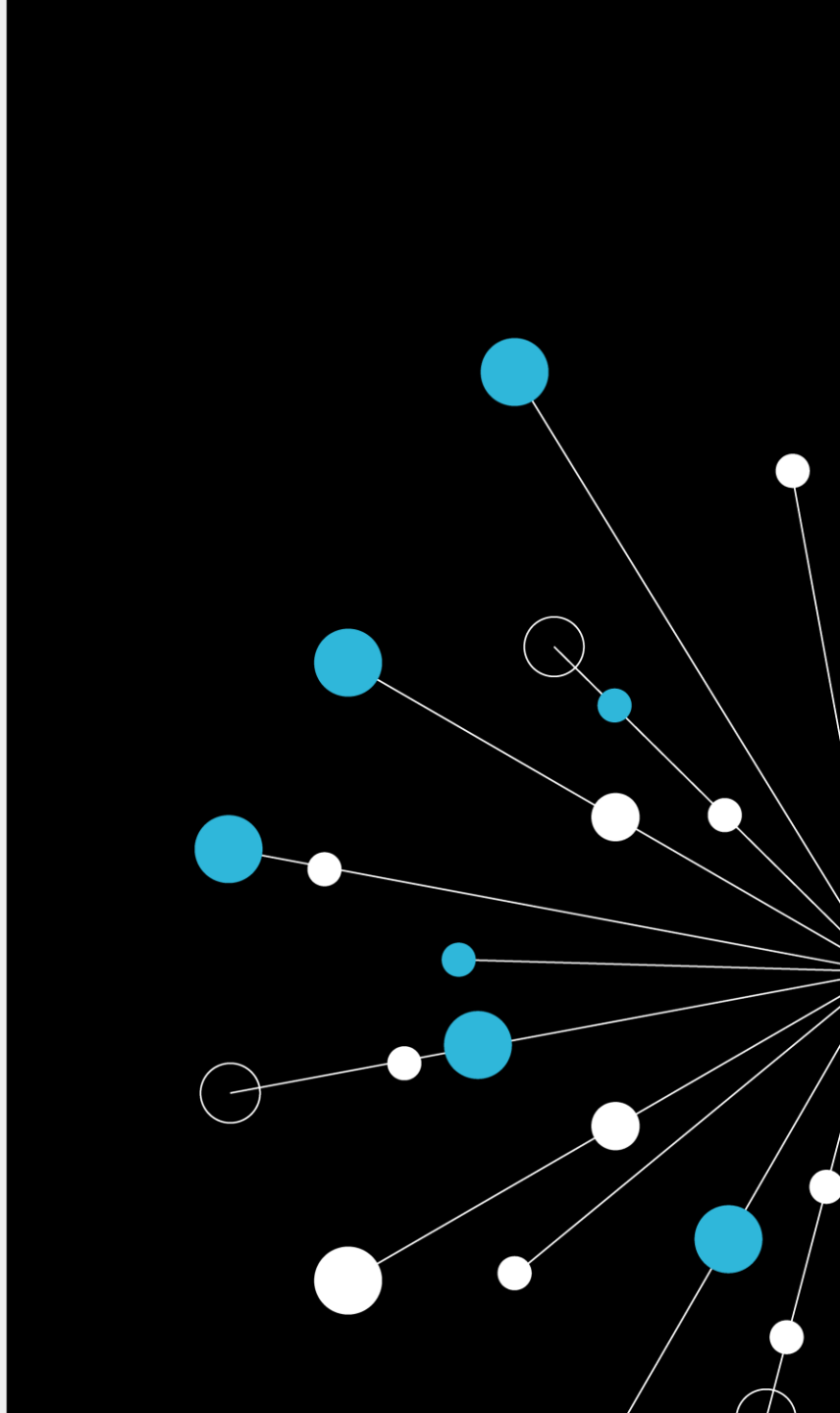
- Functions, Logic and state transitions
- DOM updates and user interactions
- Async flows and edge cases

Use test doubles where they count

- Mocks, stubs and spies.

Understand structure:

- Refactor safely with high test coverage.



QA

Assumptions

We’re assuming you already have:

JavaScript experience

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

Comfort with:

- The command line
- Cloning repos
- Running scripts with **npm** or **node**
- Using Git for version control

Introductions

When it’s your turn, share:

Your name and current role

Your experience with:

- JavaScript/TypeScript
- Testing (any kind — unit, E2E, manual)

One thing you’re hoping to get from this course

QA

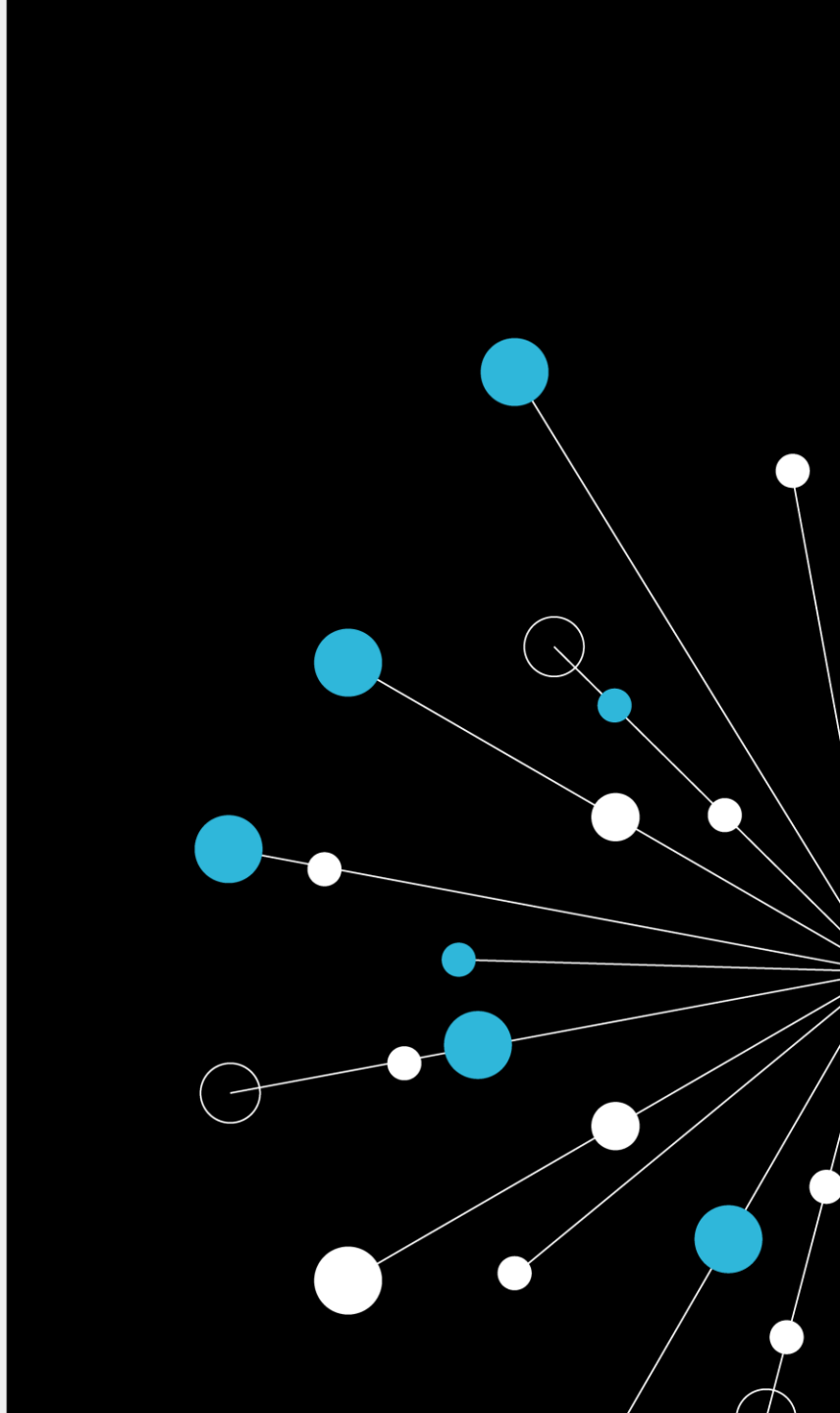
Any Questions?

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.



QA

JavaScript

Test-Driven
Development

QA
Learn. To Change.

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

QA

TDD - A Mindset and a Method

Tests as Living Documentation

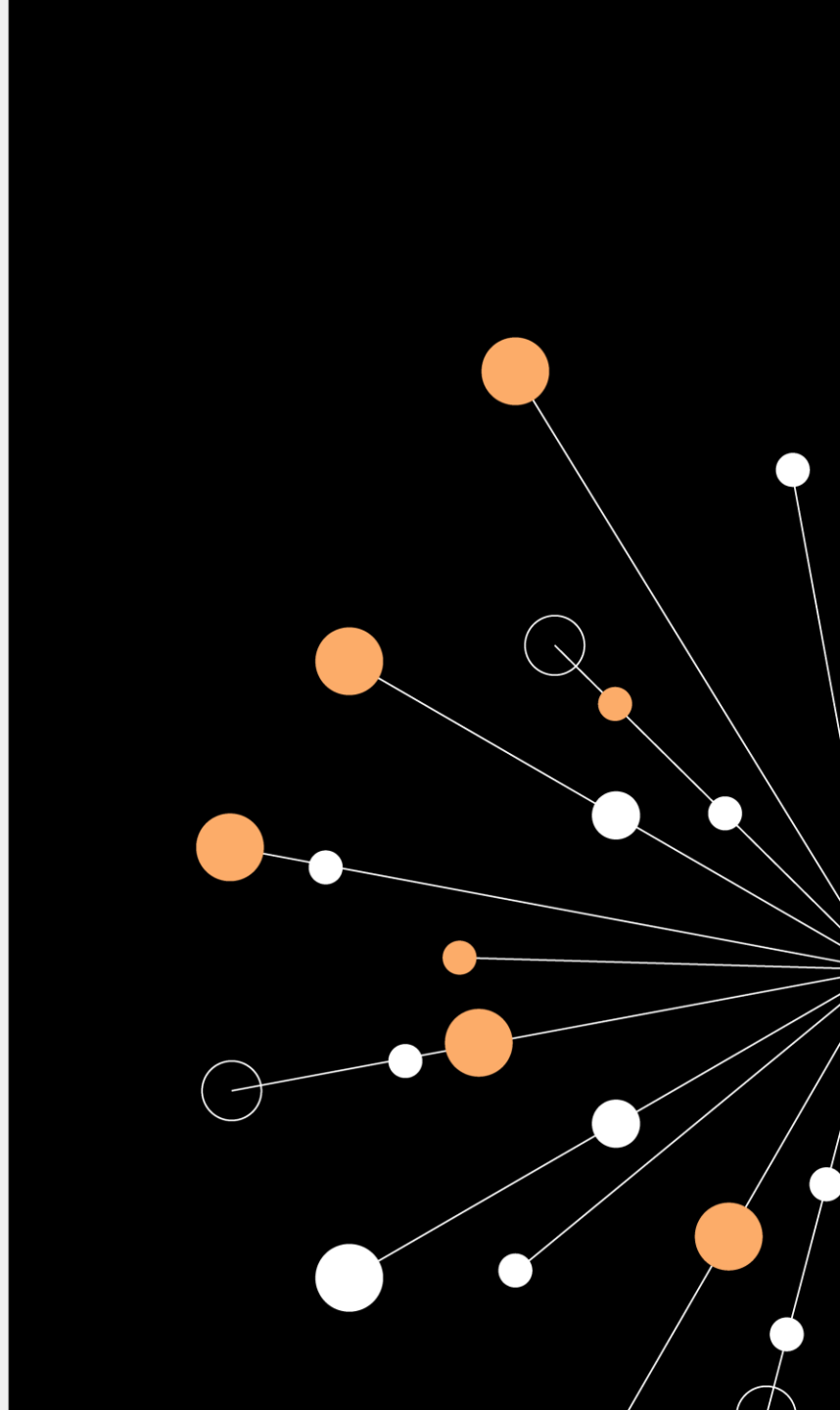
- Tests show *what the code is supposed to do*
- Clear, readable, and close to the source

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



QA

Tests → Development

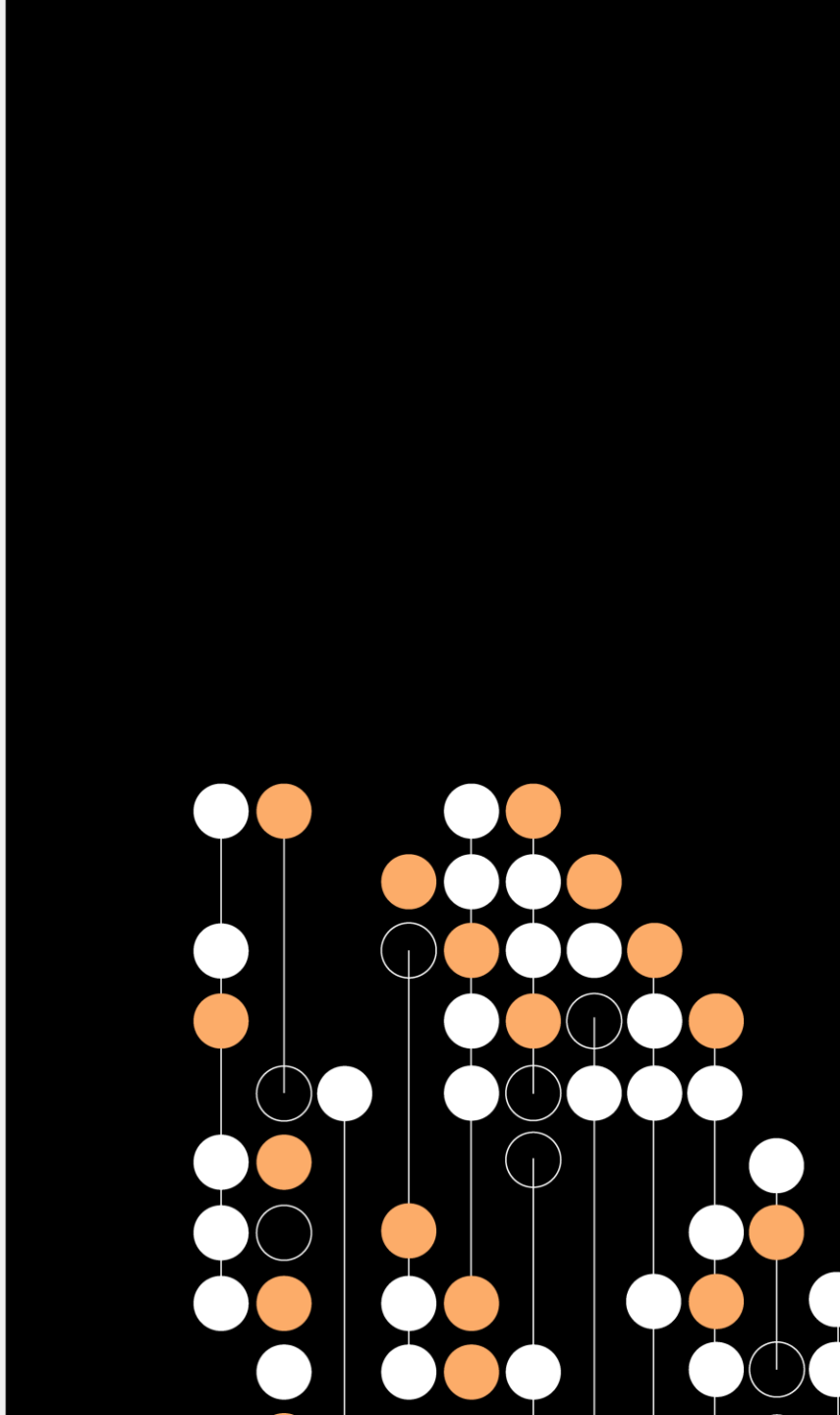
TDD flips the usual sequence:

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

Why flip it?

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

> You wouldn’t build a product without quality checks in place first.



QA

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
- Fear of breaking things limits change

Result:

- Incomplete coverage
- Bugs sneak in later
- Confidence Drops

QA

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

QA

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:

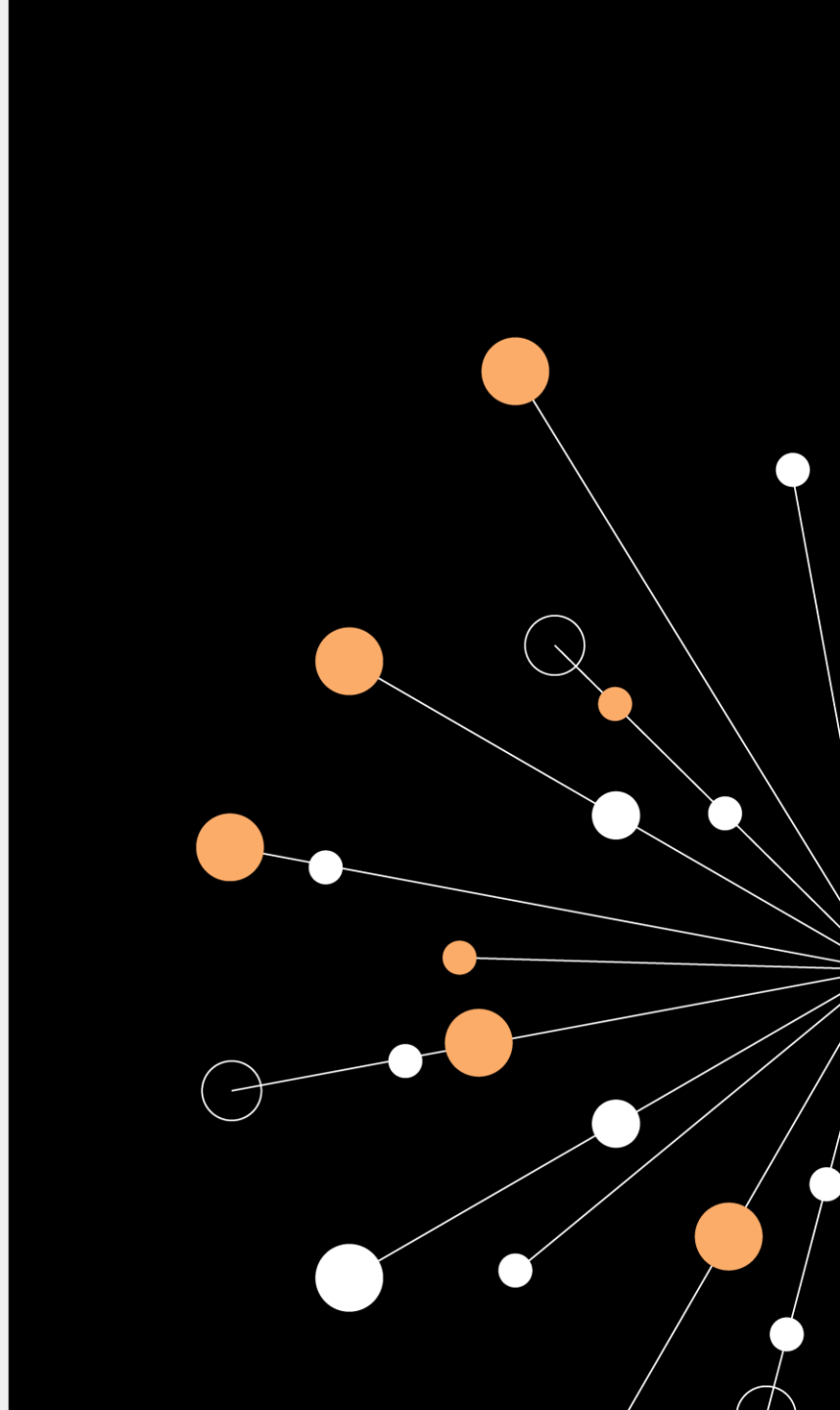
Write just enough code to **make the test pass**

- "No polish, no extras, just pass the test."

Refactor:

Improve the code without changing behaviour

- "Now that it works, clean it up with confidence."



QA

Where do Tests Come From?

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

QA

Turning Requirements into Tests

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** after data is fetched
- Content does **not** render while loading

Red phase test:

```
it('shows a loading message while fetching data', () => {
  document.body.innerHTML = '<div id="status"></div>';
  showLoading(); // function that sets loading message
  const status = document.getElementById('status');
  expect(status.textContent).toBe('Loading...');
});
```

QA

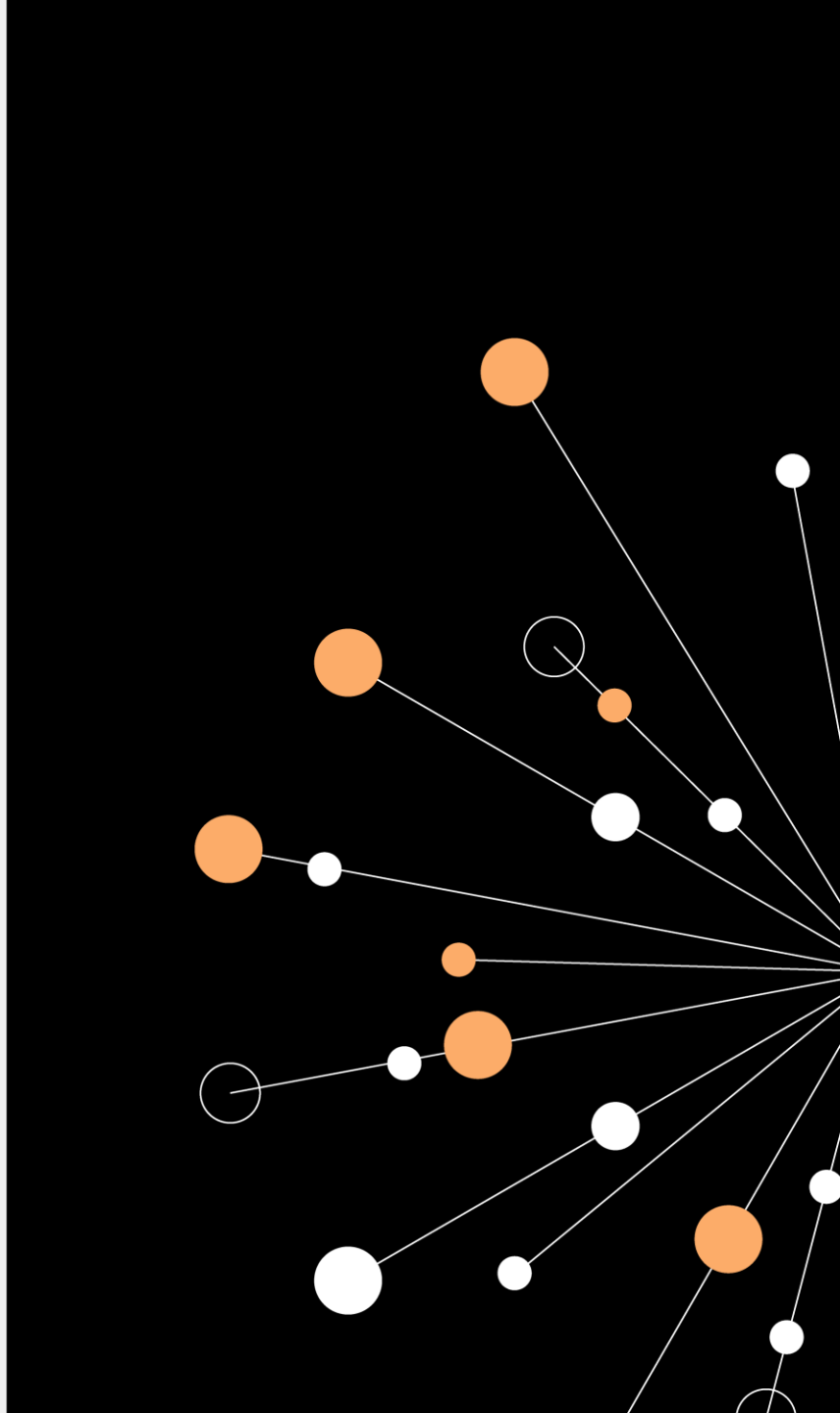
Requirements Hierarchy

Requirements Flow Like This:

1. **Product Owner** defines goals
   e.g. “User needs to see feedback while waiting”
2. **Dev Team** breaks it into functional behaviours
   “Show loading indicator until fetch completes”
3. **Frontend Developer** builds UI components
   Using `textContent`, to show and hide messages

Tests connect each layer:

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



QA

So where is the Unit Test?

A unit test checks the smallest testable bits of code

- A single function or behaviour
- Input → Output
  (Given this input, do we get the correct result?)
- DOM changes or events triggered by logic
- Isolated from:
  - Network requests
  - File access
  - External dependencies

Examples:

- Does a function return the correct value?
- Does clicking a button update the right element?
- Does validation logic handle edge cases?

QA

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

Think like a tester:

- What should work?
- What should never happen?
- What might break?

QA

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:

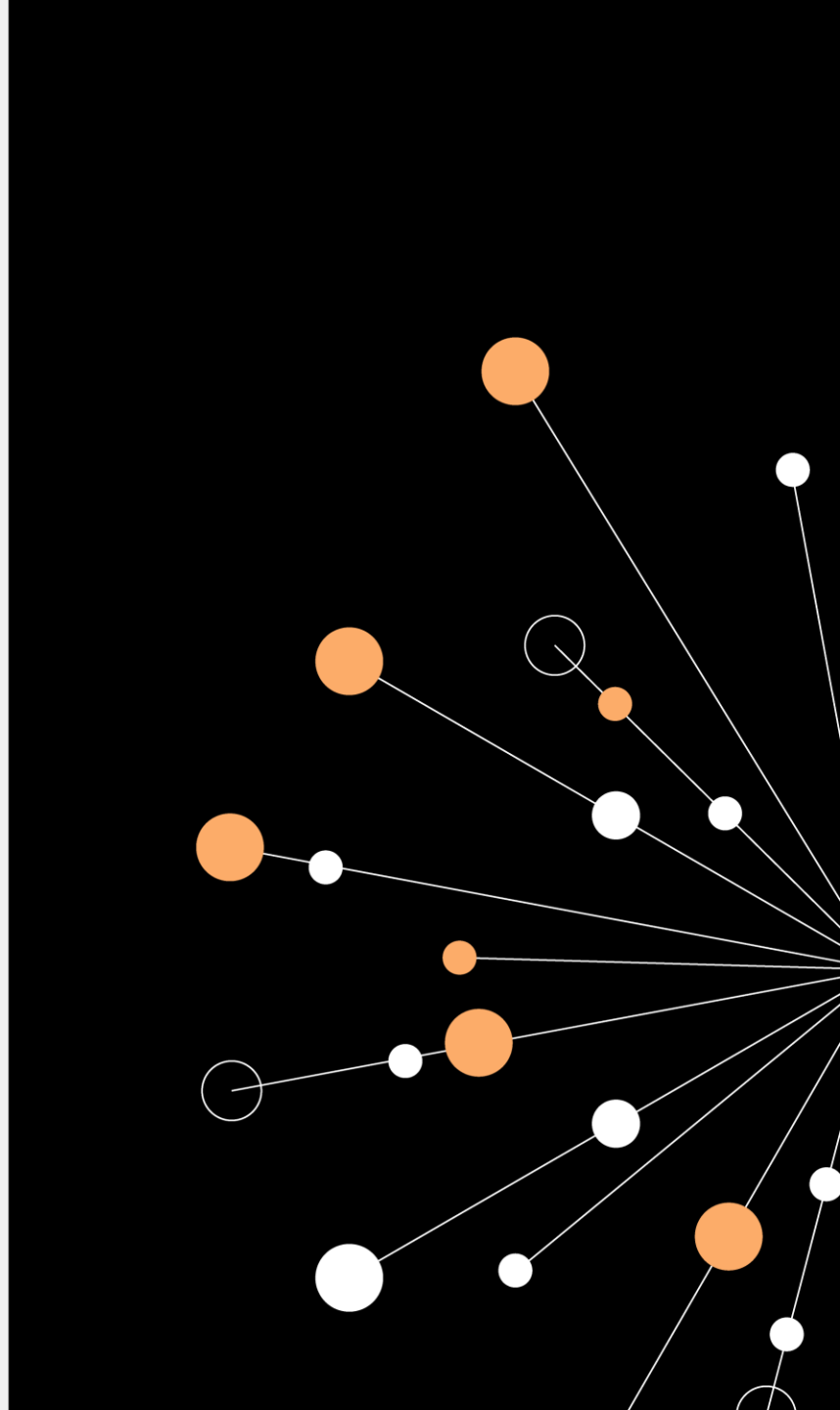
1. Write a failing test
2. Write just enough code to make it pass
3. Refactor (if needed), tests must still pass

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



QA

Unit Tests

QA

Learn. To Change.

Anatomy of a Unit Test

Arrange

Set up your environment, variables, and DOM elements

Prepare test data or mock behaviour

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 DOM, or side effect

QA

Code Example

Unit Test

```
it('shows a greeting when the button is clicked', () => {
          // Arrange
          document.body.innerHTML = `
                    <button id="greet">Say hi</button>
                    <p id="message" style="display:none">Hello, user!</p>
          `;

          const button = document.getElementById('greet’);
          const message = document.getElementById('message’);

          // Act
          button.addEventListener('click', () => {
                    message.style.display = 'block’;
          });
          button.click();

          // Assert
          expect(message.style.display).toBe('block’);
});
```

QA

Qualities of a good Unit Test

- **Focused**
  Tests one clear behaviour
- **Isolated**
  Doesn’t depend on external systems (network, database, etc.)
- **Fast**
  Runs in milliseconds, supports fast feedback
- **Independent**
  Doesn’t rely on other tests or shared state
- **Repeatable**
  Always produces the same result
- **Readable**
  Easy to understand and maintain

QA

Implementation Example

Make the Test Pass — Then Stop

```
function showGreeting() {
	const message = document.getElementById('message');
	message.style.display = 'block';
}
```

```
<button id="greet">Say hi</button>
<p id="message" style="display:none">Hello, user!</p>

<script>
	document.getElementById('greet').addEventListener('click', showGreeting);
</script>
```

QA

Interaction Testing

What to test:

- User events: clicks, typing, keyboard input
- Changes in the DOM: text updates, element visibility
- Behaviour that reflects state, not internal variables

Use:

- `addEventListener`, `.click()`, `.value`, `.dispatchEvent()`
- Assertions on DOM elements using `textContent`, `style`, `classList`, etc.

Follow the Arrange → Act → Assert pattern:

- **Arrange**: Set up HTML and any required state
- **Act**: Simulate the interaction
- **Assert**: Confirm visible change in the DOM

QA

Interaction Test Example

Clicking a button reveals a message

```
it('reveals a message when the button is clicked', () => {

        // Arrange
        document.body.innerHTML = `
             <button id="show">Show message</button>
             <p id="message" hidden>Hello!</p>
        `;

        const button = document.getElementById('show');
        const message = document.getElementById('message');

        button.addEventListener('click', () => {
          message.hidden = false;
        });

        // Act
        button.click();

        // Assert
        expect(message.hidden).toBe(false);
});
```

QA

Dependents Takeaway

Tight Coupling Causes Fragile Tests

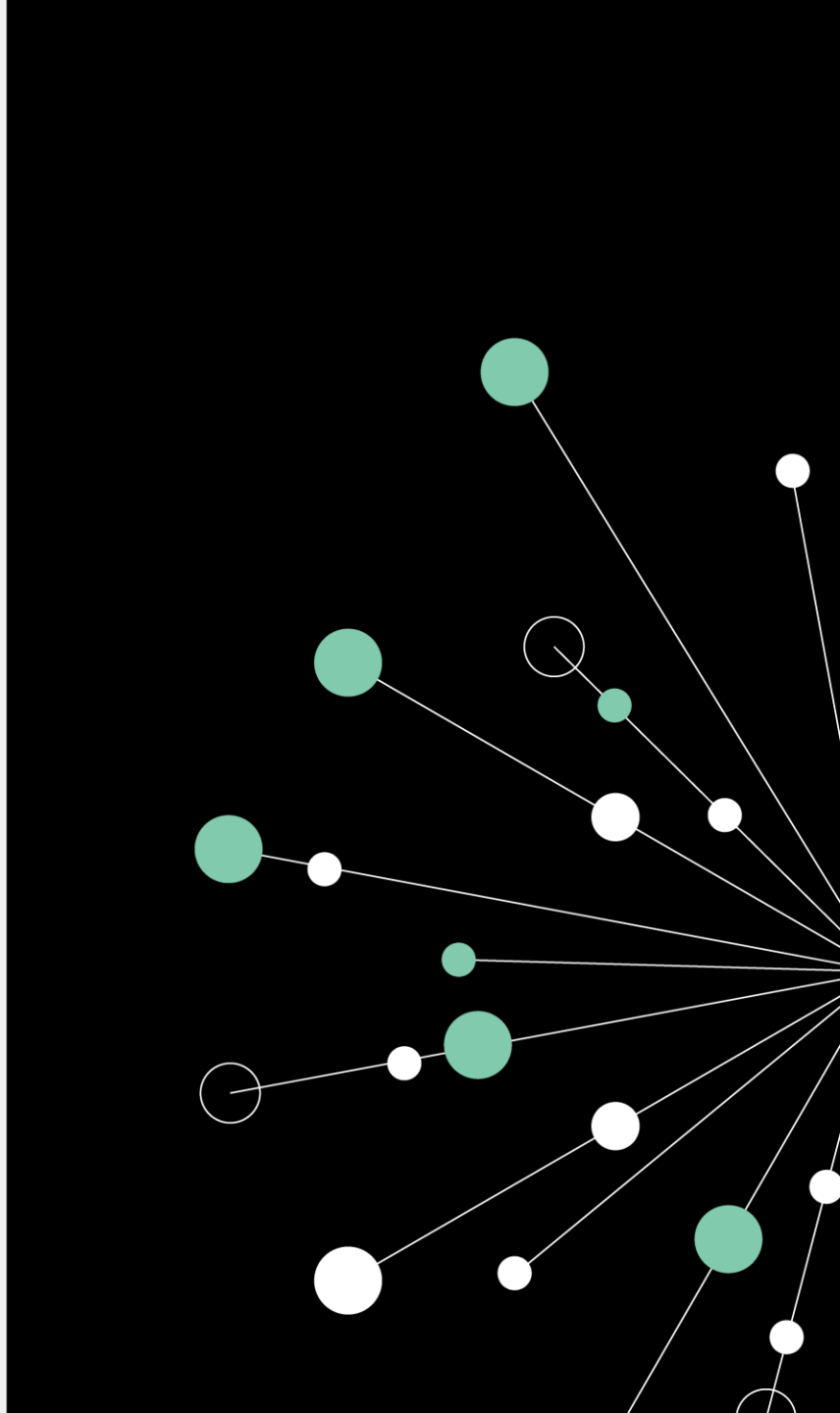
- Tests break when inner details change, even if behaviour stays the same
- You end up testing everything except what you actually wrote

Symptoms

- Tests fail after a harmless refactor
- You're asserting on things deep inside other modules
- Setup becomes complex or brittle

Solutions

- Only test what your code controls
- Mock out dependencies clearly
- Keep functions and DOM logic separate



Test Doubles

QA

Learn. To Change.

Test doubles types

Stub - *Replace a function and control its output*

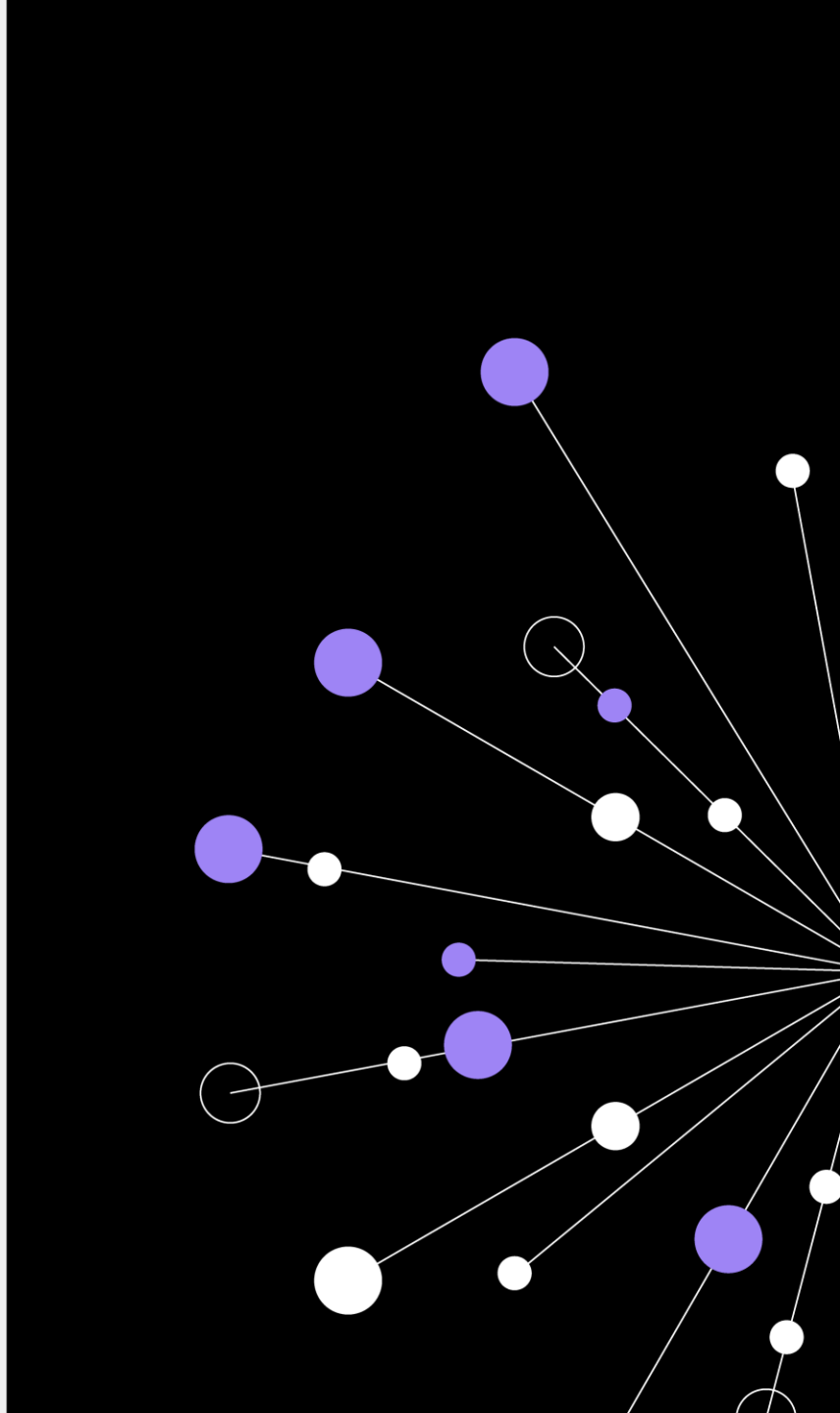
- Use when you want to fake a return value
  “Always return X when called.”

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?”



QA

The Importance of Test Isolation

Isolated tests:

- Focus only on the behaviour of the code under test
- Avoid side effects like:
  - Network calls
  - File system access
  - `LocalStorage` or global variables

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

QA

Stubs - The problem

Component Example:

```
function loadFile(id) {
       return fetch(`/files/${id}`)
         .then(res => res.text());
}
```

Why this is hard to test:

- Calls the real network
- Depends on external files
- Can fail or hang if offline

The Fix: Stub the data fetch

Replace `fetch` with a stub that returns fake data

QA

Stubs - The solution

Stub a fetch call

```
global.fetch = jest.fn(() =>
        Promise.resolve({
                text: () => Promise.resolve('Hello world!')
        })
);

it('shows file content after load', async () => {
        const content = await loadFile('99');
        expect(content).toBe('Hello world!’);
});
```

QA

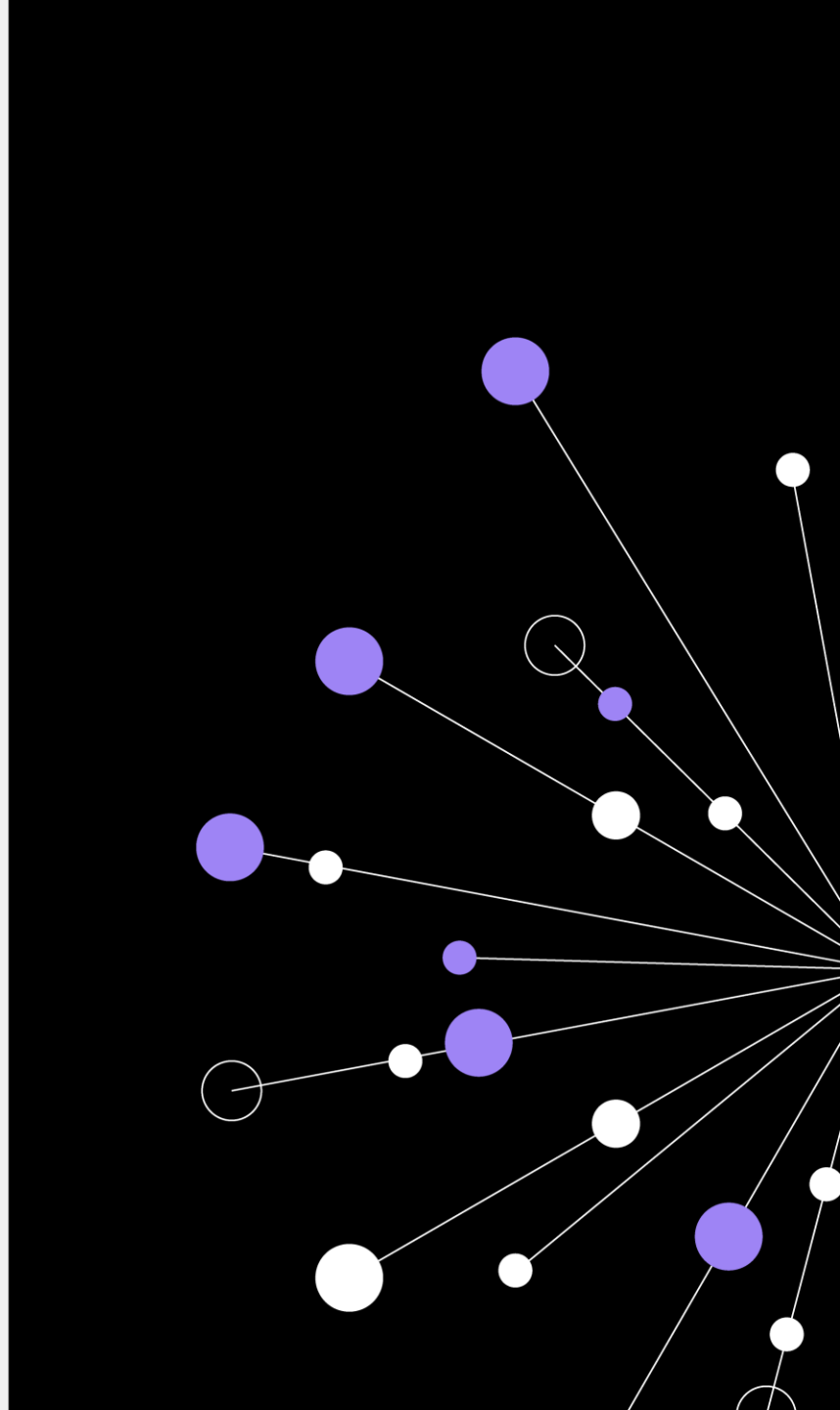
Stubs - Takeaways

What Stubs Do Well

- Simulate responses from APIs, services, or dependencies
- Enable fast, isolated tests with controlled behaviour
- 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



QA

Test Doubles and Verification Testing

QA

Learn. To Change.

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
- And you also want to track usage

QA

Spying with Jest

Track a helper function

```
const utils = {
        formatName: (name) => name.toUpperCase()
};

jest.spyOn(utils, 'formatName');

it('calls formatName with the correct value', () => {
        utils.formatName('erin');
        expect(utils.formatName).toHaveBeenCalledWith('erin’);
});
```

QA

Mock Objects

> Mocks observe and report – You don't have to guess what happened

What is a mock object?

A fake version of a function or service

Tracks

- If it was called
- How many times
- With what arguments

Made with

- `jest.fn()` - build a mock from scratch
- `jest.mock()` - replace an entire module

Why use mocks?

Validate that your code interacts with dependencies correctly

QA

Mocks for Verification

Save button calls callback

```
const saveMock = jest.fn();

document.body.innerHTML = `<button id="save">Save</button>`;

document.getElementById('save').addEventListener('click', () => {
        saveMock('user1’);
});

it('calls onSave with user input', () => {
        document.getElementById('save').click();
        expect(saveMock).toHaveBeenCalledWith('user1’);
});
```

QA

Mocking Objects Takeaway

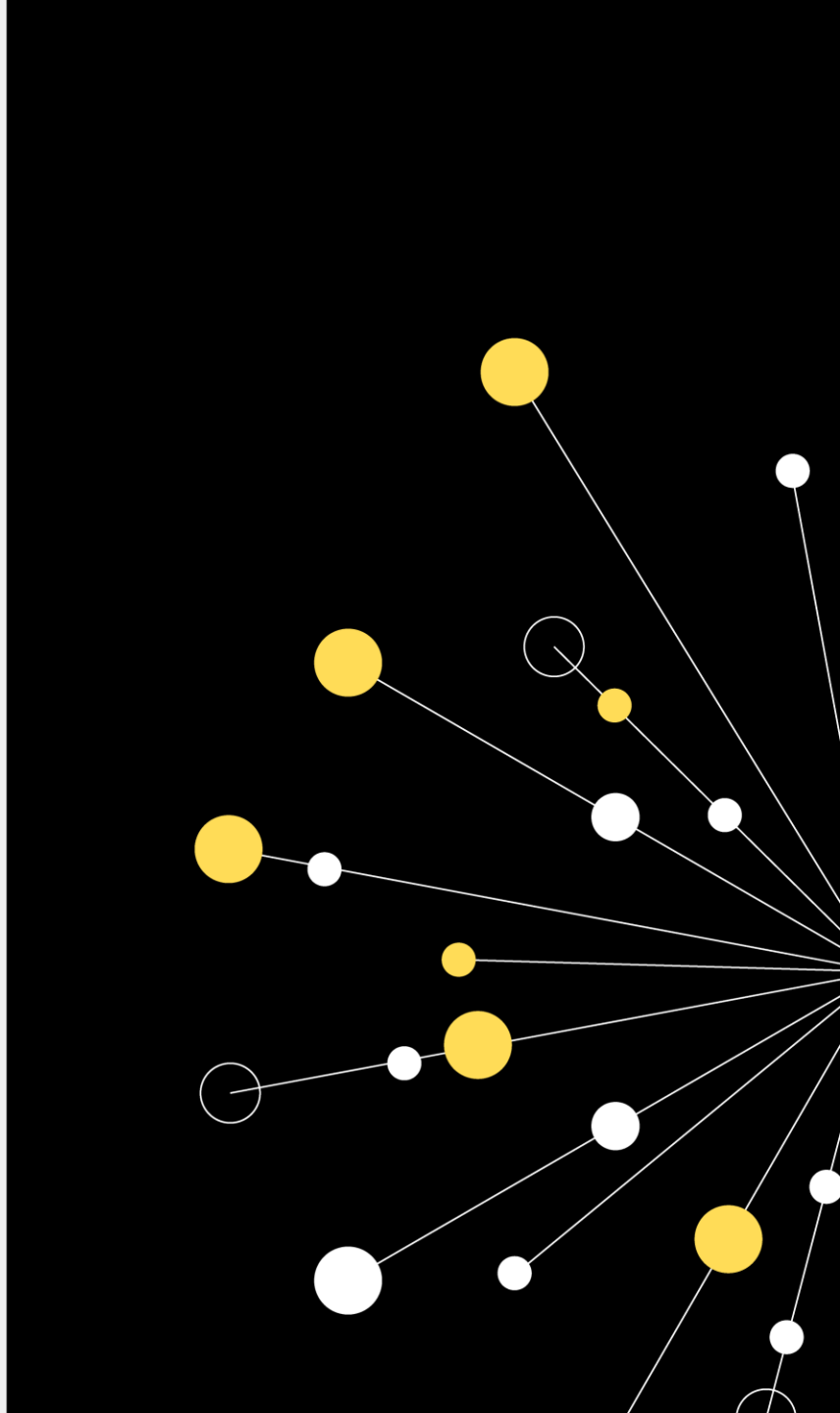
Mocks vs Stubs - Key Difference

Use stubs when:

- You just need **fake data** to drive behaviour
- The test only cares about what comes back
- You don’t need to check how the function was used

Use mocks when:

- You want to check if a function was called
- You care about how often, and with what arguments
- You want to verify interaction between modules



QA

Testing Async Behaviour

Key Practices:

Use built-in async utilities:

- **`async/await`**
- **`waitFor`, `findBy*`,** or custom delays

`Always test:`

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

Avoid:

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

QA

Async Testing Example

Load and display user data

```
function loadUser() {
          const el = document.getElementById('user');
          el.textContent = 'Loading...';
          return fetch('/user')
            .then(res => res.json())
            .then(data => {
                    el.textContent = data.name;
            });
}

global.fetch = jest.fn(() =>
  Promise.resolve({
          json: () => Promise.resolve({ name: 'Boris' })
  })
);

it('displays user name after fetch', async () => {
          document.body.innerHTML = `<div id="user"></div>`;
          await loadUser();
          const user = document.getElementById('user');
          expect(user.textContent).toBe('Boris');
});
```

QA

State-Based UI Testing

What do we mean by “state”?

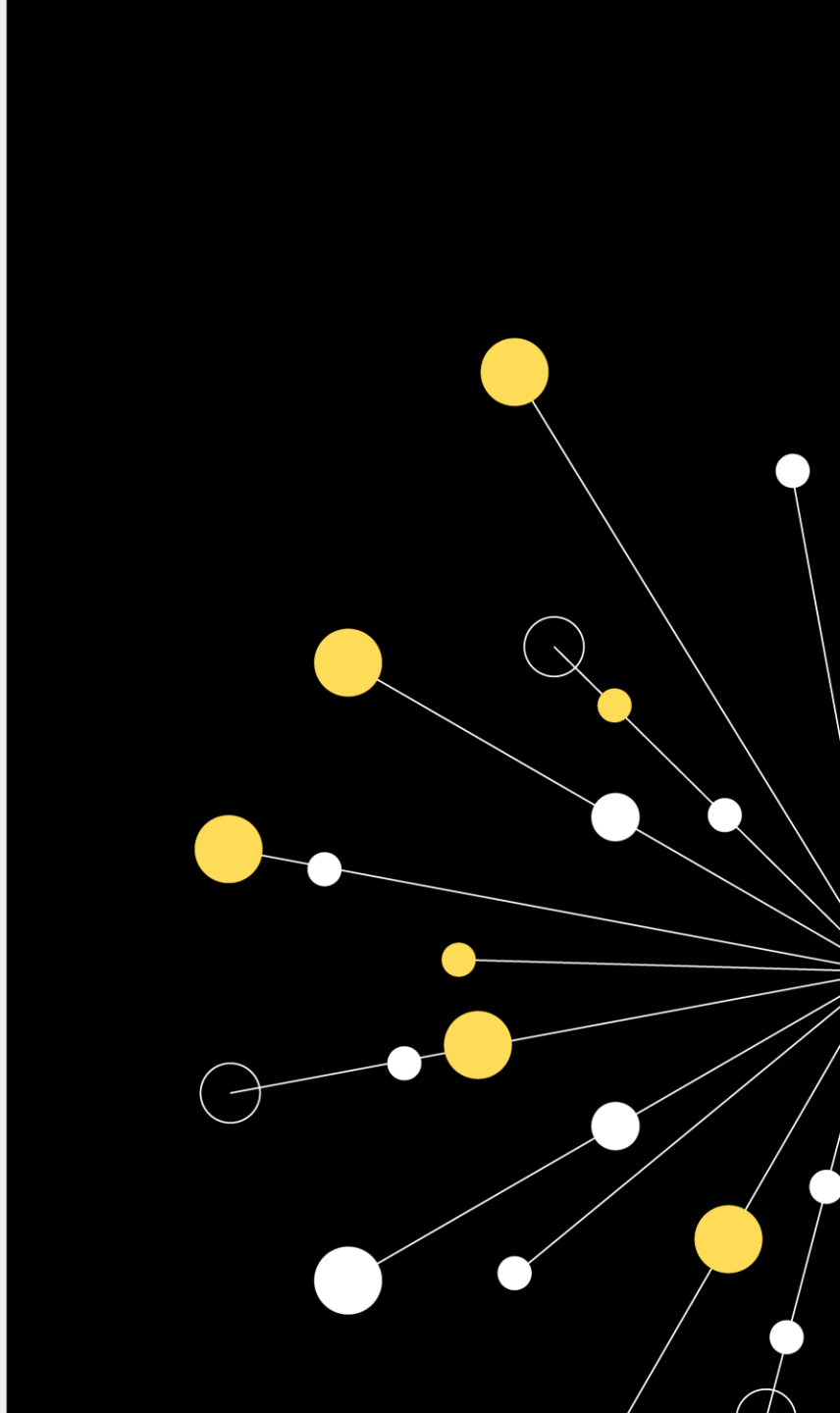
- A value that changes based on user interaction or data
- Tracked in variables, not just HTML

Examples of UI state:

- A menu that opens or closes
- A button that gets disabled after click
- A field that shows errors after input

Test it by:

1. Setting the initial state
2. Simulating interaction
3. Checking what changed



QA

State Testing Example

Toggle button UI

```
document.body.innerHTML = `
            <button id="toggle">Toggle</button>
            <p id="message" hidden>Hello!</p>
`;

document.getElementById('toggle’)
  .addEventListener('click', () => {
            const msg = document.getElementById('message');
            msg.hidden = !msg.hidden;
});
```

Toggle button Test

```
it('shows and hides the message on click', () => {
      const button = document.getElementById('toggle');
      const message = document.getElementById('message’);

      // Initial state: hidden
      expect(message.hidden).toBe(true);

      // First click: show
      button.click();
      expect(message.hidden).toBe(false);

      // Second click: hide
      button.click();
      expect(message.hidden).toBe(true);
});
```

QA

Untestable Code

QA

Learn. To Change.

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

QA

What is a Contract?

A contract is an **agreement** between two parts of your code

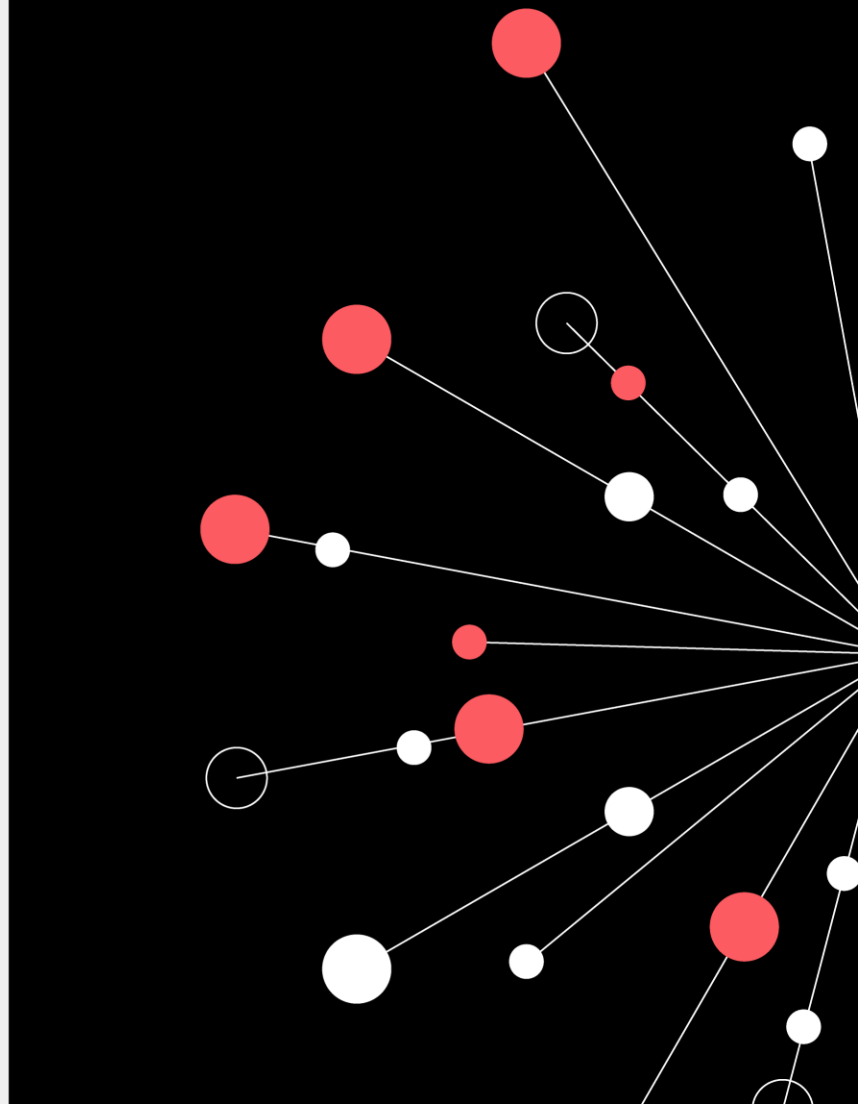
→ e.g. a function expects certain inputs and guarantees certain outputs

Examples

- A function returns a specific shape of data
- A callback will always be called with the right arguments
- An API response contains expected fields

Why it matters

- Contracts prevent silent failures
- Testing contracts catches broken assumptions early



QA

Designing Code That’s Easy to Test

Design Principles:

Separating logic from DOM updates

→ Move calculations or decisions into functions

Injecting dependencies

→ Pass in functions like `fetch` or `storage` instead of hard-coding them

Avoiding global state

→ Keep data local to functions, or pass it explicitly

Keeping UI updates predictable

→ Use `classList`, `hidden`, or `textContent` instead of manipulating raw HTML

QA

Conclusion

> TDD isn’t hard, skipping it just *feels* easier.

What We Learned

- TDD = **Write test → Write code → Refactor**
- 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

QA

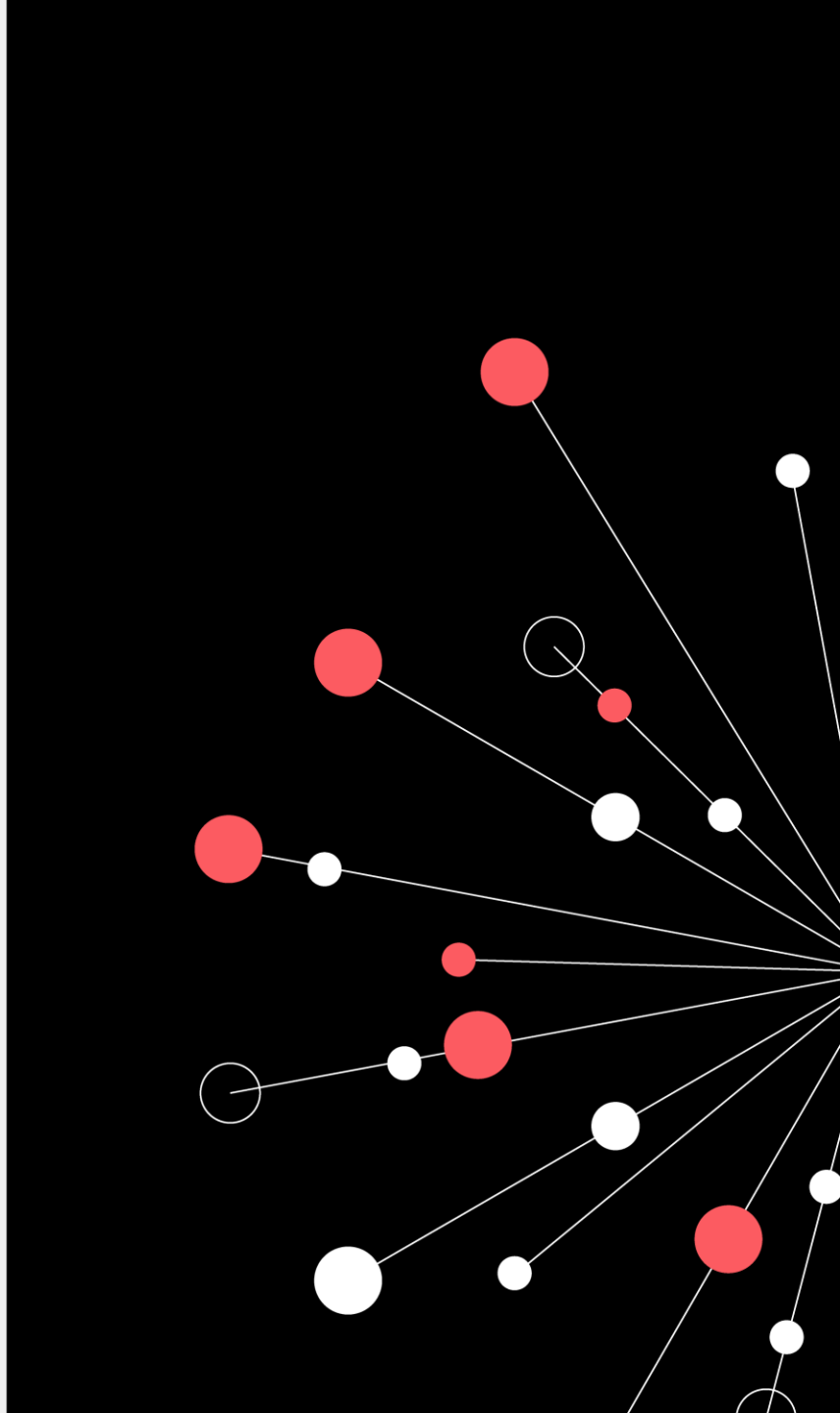
Feedback and Final Thoughts

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



Learn. To Change.