Day 27 — Testing in React

Week 4 — Quality Assurance & Reliability

Topic: Jest + React Testing Library (RTL) — simple, modern, user-focused testing (2025 best practices)

🎯Learning Goals

- Understand *why* we test React apps.
- Learn the minimal setup for **Jest** + **React Testing Library**.
- Write **simple, reliable tests** that verify component behavior (not implementation details).
- Practice Manual testing steps + Automated tests (unit + integration) with tiny examples.

Why Test (short)

- Prevent regressions when refactoring
- Catch bugs before users do
- Make changes with confidence
- Document expected behavior

Manual testing is great for learning and quick checks. Automated tests give repeatable safety.

Testing Tools (2025 standard)

- **Jest** — test runner + assertions (fast, widely used)
- **React Testing Library (RTL)** — query the DOM like a user (preferred over implementation-based testing)
- **@testing-library/jest-dom** — helpful matchers (toBeInTheDocument, etc.)
- **user-event** — simulate real user interactions (typing, clicking)

Setup (Vite / Create React App / Plain React)

Install minimal dev dependencies:

```
npm install --save-dev jest @testing-library/react @testing-library/jest-dom
@testing-library/user-event
```

Add a test script to `package.json` (if needed):

```
"scripts": {
  "test": "jest --watchAll"
}
```

If using Create React App, testing is preconfigured (runs `npm test`).

⛳Testing Philosophy (Keep it simple)

- Test behavior, not implementation. Use queries that a user would use (`getByText`, `getByRole`, `getByLabelText`).
- Keep tests small and deterministic. Avoid network calls in unit tests (mock them).
- Use manual testing to validate UX, then write automated tests for critical flows.

Example 1 — Counter Component

Component: `Counter.jsx`

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h3>Count: <span data-testid="count">{count}</span></h3>
      <button onClick={() => setCount(c => c + 1)}>Increment</button>
      <button onClick={() => setCount(c => c - 1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

Test: `Counter.test.js`

```
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';

test('renders count text', () => {
  render(<Counter />);
  expect(screen.getByText(/Count:/i)).toBeInTheDocument();
});

test('increments counter on click', () => {
  render(<Counter />);
  const inc = screen.getByText(/Increment/i);
  const count = screen.getByTestId('count');
  fireEvent.click(inc);
  expect(count.textContent).toBe('1');
});
```

Notes: - `render()` mounts the component in a test DOM. - `screen` queries the DOM. - `fireEvent` simulates clicks. For richer user-like events prefer `user-event`.

Example 2 — TodoList (Integration Test)

Component: `TodoList.jsx`

```
import { useState } from 'react';

export default function TodoList() {
  const [todos, setTodos] = useState([]);
  const addTodo = () => setTodos(t => [...t, `Task ${t.length + 1}`]);
  return (
    <div>
      <button onClick={addTodo}>Add Task</button>
      <ul>
        {todos.map((t, i) => <li key={i}>{t}</li>)}
      </ul>
    </div>
  );
}
```

Test: `TodoList.test.js`

```
import { render, screen, fireEvent } from '@testing-library/react';
import TodoList from './TodoList';

test('adds tasks when button clicked', () => {
  render(<TodoList />);
  const btn = screen.getByText(/Add Task/i);

  fireEvent.click(btn);
  fireEvent.click(btn);

  const items = screen.getAllByRole('listitem');
  expect(items.length).toBe(2);
});
```

Useful RTL Queries (recommended)

- `getByRole('button', { name: /submit/i })`
- `getByLabelText('Email')` for form inputs
- `getByText(/Welcome/i)` for static text
- `findBy...` for async elements (returns a promise)
- Prefer `getByRole` and `getByLabelText` — they are accessible and user-centric.

🪁Mocking Network Calls (simple)

- For components that call APIs, mock fetch/axios with `jest.mock()` or use msw (Mock Service Worker) for more realistic tests.

Simple mock example (axios):

```
jest.mock('axios');
import axios from 'axios';
axios.get.mockResolvedValue({ data: [{ id:1, title:'a'}] });
```

Debugging Tests

- `screen.debug()` prints the test DOM
- Run `jest --watch` for quicker iterations
- Use `.only` on tests during development: `test.only(...)`

Minimal Manual Testing Checklist

- Open app in browser
- Click primary buttons
- Submit simple forms
- Inspect console for errors
- Repeat important flows after changes

Manual testing + a few automated tests gives fast confidence.

🎯Exercise — Day 27 (15–25 minutes)

- Create a small `Counter` and `TodoList` components (if you haven't yet).
- Write automated tests:
- Counter renders and increments/decrements.
- TodoList adds items on button click and shows correct count.
- Run tests with `npm test` and fix any failures.

Bonus: - Use `user-event` to type into an input and submit a form. - Mock a simple API call for a component using `jest.mock`.

Final Notes

- Keep tests focused and simple. Tests that mirror user interactions are the most stable.
- Start with manual testing habits, then add automated tests for critical paths.
- When ready, expand to CI integration and coverage thresholds.

Next: Day 28 — Advanced Testing Patterns & CI/CD (optional)