The Pragmatic Programmers

Test-Driven React

Second Edition

Find Problems Early,
Fix Them Quickly,
Code with Confidence

Trevor Burnham
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Introduction

I vividly remember the first time I wrote code. I was ten years old and utterly obsessed with robots. The local public library must have lent me every book they had on the subject. One of those books had an appendix, "Write Your Own Robot in BASIC." I ran to my parents' computer, fired up `qbasic` (bundled with the cutting-edge MS-DOS operating system), and fed in the instructions for my robot companion.

The program was unimpressive by today's standards. It was a primitive version of what we would now call a "chatbot." It would give you a prompt like this one:

```
>>> Greetings, human. How are you feeling today?
```

Then it would wait for you to enter a recognizable string like `tired` and give an appropriate response, something like this:

```
>>> I am sorry to hear that. How about a nice cup of coffee?
```

And it would continue. The only keywords in the entire program were `IF`, `THEN`, and `GOTO`.

Even though my chatbot wouldn't stand a snowball's chance in a Turing test, the exercise was a revelation to me: I could actually *create* something just by *typing*. Now it seemed that robots were old news. Computers were where it's at!

As computers have grown more capable, software has grown more complex, and that thrilling feeling has become more elusive. New layers of abstraction have empowered me to do more with less code but at the cost of constant uncertainty: *will my code do what I intended?*

Test-driven development (TDD) is the art of minimizing that uncertainty, allowing you to feel confident about your code from the moment you write it. How? By making a few assertions about that code beforehand. This groundwork sets up a short, satisfying feedback loop: as soon as you write your code, the tests light up green. Afterward the tests remain in place, standing guard against regressions.

I don't always use TDD, but when I do, I feel a little bit closer to the magic of that first coding experience. All of the rigamarole of modern software development fades away. I can focus all my energies on reaching toward the green light.

What's in This Book

This is a book about React. But it's not like any other book about React. This is a book about writing React code in a joyful way. You might learn a few new things about React, but that's not my goal. My goal is to help you write better code and to have more fun doing it.

In Chapter 1, Test-Driven Development with Jest, on page ?, you'll get a taste of test-driven development, a programming methodology that uses tests to create a feedback loop as you work. You'll meet Jest, a lightning-fast test framework which is the perfect companion for TDD.

Chapter 2, Integrated Tooling with VS Code, on page ?, will introduce you to some of my favorite tools: VS Code, an amazingly powerful editor; TypeScript, a dramatic enhancement to the JavaScript language; and ESLint and Prettier, the ultimate code beautification duo. You'll experience the wonder of instantaneous feedback as you code.

Then in Chapter 3, Testing React with Testing Library, on page ?, you'll start writing React components and testing them with the aptly named React Testing Library. You'll build a complex component the TDD way.

Chapter 4, Styling in JavaScript with Styled-Components, on page ?, is all about style. You'll use the styled-components library to add pizzazz to your React components without the need for separate CSS files. You'll also learn about testing styles and using one of Jest's most powerful features: snapshots.

In Chapter 5, Refactoring with Hooks, on page ?, you'll learn some important techniques for refactoring React components. You'll extract pieces of functionality into hooks, encouraging code reuse and allowing core components to stay small and easy to test. And you'll look at your components with X-ray vision through the power of the React Devtools.

Finally, in Chapter 6, Continuous Integration and Collaboration, on page ?, you'll meet all the tools you'll need to share what you've built with the world. You'll run your tests in the cloud with Travis CI, enforce your project's rules with Husky, and create beautiful, interactive documentation with Storybook.

What's Not in This Book

This is not an introduction to JavaScript. If you're new to the language, or if you just want a refresher, I highly recommend Kyle Simpson's excellent *You Don't Know JS Yet*[1] series. Most of the code in this book will employ features added to the language as part of the ECMAScript 6 (also known as ES6 or ES2015) standard. Here's a quick test:

```
const stringifyAll = (...args) => args.map(String);
```

If any of that syntax is confounding, you'll find clarity in Simpson's series.

We'll also be using TypeScript. TypeScript is a superset of JavaScript that adds type annotations. The syntax may look strange at first, but the examples in this book should be fairly easy to understand. If you're interested in diving deeper, I recommend *Learning TypeScript [Gol22]* by Josh Goldberg.

Some familiarity with React is helpful, but not required. I'll give a brief explanation for each React concept we encounter. If you feel lost, a good resource is *React Quickly [BM23]* by Morten Barklund and Azat Mardan.

All tests in this book are unit tests, meaning the JavaScript code is tested in isolation. In production applications, I highly recommend adding functional tests using a tool like Playwright[2] in addition to unit tests.

Unit Tests vs. Functional Tests

Unit tests, by definition, test a single unit of code (such as a React component) in isolation. Nothing outside of that unit should affect whether the test passes or fails. But that's not how code works in the real world! In the context of a real application, whether your component works or not is likely to depend on the behavior of other components, on data returned by APIs, and on countless other factors.

So why write unit tests at all? Two reasons. First, unit tests are much faster. For TDD, it's critical to get feedback in seconds, not minutes. Second, unit tests are easier to write. That's why I recommend starting with unit tests, then adding functional tests after you have a working application (but before shipping it to production!).

Even as your application's functional requirements change, many of its components will remain the same, so unit tests will continue to provide value. Functional tests, on the other hand, will need to be rewritten as the application evolves.

1. https://github.com/getify/You-Dont-Know-JS
2. https://playwright.dev

What's New in the Second Edition

React development has seen an incredible evolution from 2019 to 2023, with dramatic improvements for both the developer experience and the results we can deliver to users. This edition reflects many of those improvements.

The most obvious change is the shift from JavaScript to TypeScript, Microsoft's typechecked JavaScript superset. TypeScript was a bleeding-edge technology back in 2019; today it's the industry standard for writing reliable code. Once you start using TypeScript, you'll wonder how you ever did without it! All of the React code in this book is now in TypeScript, and TypeScript concepts are explained for newcomers.

React code looks very different now than it did a few years ago, thanks to the addition of hooks in React 16.8. Hooks are a powerful feature that allows stateful components to be defined as a single function instead of a class. Additionally, hooks are a neat way of encapsulating reusable functionality across components. The chapter originally entitled "Refactoring with Higher-Order Components" has been replaced with one called Chapter 5, Refactoring with Hooks, on page ?, and all React code has been updated to reflect best practices as of React 18.

On the testing side of things, the Enzyme framework has fallen in popularity in favor of the simply named React Testing Library. It offers a conceptually different approach to testing React components: instead of making assertions about a component's state or the React tree it generates, you fully render the component and look at the resulting DOM tree. This approach to React testing has proven more intuitive (and less finicky) and is embraced in this edition.

Lastly, we've seen a revolution in web application build tools and frameworks in the last few years. The first edition of this book walked readers through the process of setting up a build chain with Webpack and Babel, step by step. Today there are a number of "batteries included" frameworks that require little to no configuration to compile modern React code. Some, like Next.js[3] and Remix,[4] are full-stack frameworks with a built-in Node.js instance for server-side rendering (SSR). This edition uses Vite,[5] a lightweight alternative to Webpack with a highly performant integrated testing library, Vitest.

3. https://nextjs.org/
4. https://remix.run/
5. https://vitejs.dev/

How to Read the Code Examples

This book takes a hands-on, project-driven approach, which means that source files often change over the course of a chapter. When a code example is a work in progress, its file name (relative to the project root) is shown as a comment at the top of the snippet:

```
// src/MyComponent.test.tsx
import { render, screen } from "@testing-library/react";
import MyComponent from "./MyComponent";

describe("MyComponent", () => {
  it("renders a <div>", () => {
    render(<MyComponent />);
    expect(screen.getByRole("div")).toBeInTheDocument();
  });
});
```

As a source file changes over the course of a chapter, familiar sections are omitted with ... and new/edited lines are highlighted:

```
// src/MyComponent.test.tsx
...
describe("MyComponent", () => {
  ...
  it("accepts a `className` prop", () => {
    render(<MyComponent className="test-class" />);
    expect(screen.getByRole("div")).toHaveClass("test-class");
  });
});
```

The final version of a source file within a chapter has a download link at the top instead of a comment:

intro/src/MyComponent.test.tsx

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import MyComponent from "./MyComponent";

describe("MyComponent", () => {
  it("renders a <div>", () => {
    render(<MyComponent />);
    expect(screen.getByRole("div")).toBeInTheDocument();
  });

  it("accepts a `className` prop", () => {
    render(<MyComponent className="test-class" />);
    expect(screen.getByRole("div")).toHaveClass("test-class");
  });

  it("triggers `onClick` when clicked", async () => {
    const onClick = vi.fn();
```

```
    render(<MyComponent onClick={onClick} />);
    const nextButton = screen.getByRole("button");
    const user = userEvent.setup();
    await user.click(nextButton);
    expect(onClick).toHaveBeenCalled();
  });
});
```

Online Resources

You can find the source code for the projects in this book on the PragProg website.[6] You can also use the site to report errata. Help make this book better for other readers!

Mantra: Code with Joy

At its best, coding is an exercise in imagination and exploration, an exciting journey into the unknown. At its worst, it feels like stumbling in the dark. Which kind of experience you'll have is largely determined by feedback. The next time you're feeling frustrated, take a step back and ask yourself what kind of feedback would help you move forward. What question can you ask about your code that would bring clarity? Can you turn that question into a test?

I hope this book will help you bring more joy to your work by instilling a habit of seeking feedback early and often. Let's begin!

Trevor Burnham
trevorburnham@gmail.com
Cambridge, MA, August 2024

6. https://pragprog.com/titles/tbreact2