Outside-In React Development

A TDD Primer

Josh Justice

Outside-In React Development

A TDD Primer

Josh Justice

This book is for sale at http://leanpub.com/outside-in-react-development

This version was published on 2022-06-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Go Beyond the Book

Thank you for reading this book! This is Josh Justice, the author, and I hope it will be a big help to you. But the help doesn't stop here.

I'd love to invite you to connect with me and other readers online to:

- Let me know if the book is helpful to you
- Ask questions and provide feedback about the book
- Share your own approach to React testing and TDD and get input
- Get help with React testing challenges on your own projects
- Hear about updates about this book and other resources from me, including pre-releases and discounts

Visit https://outsidein.dev/connect to see the latest ways to connect, including email, social media, and online discussion. We hope to see you soon!

Introduction

Testing on the Front-end

The World Wide Web started out as a platform for displaying only static documents, but now it hosts fully-featured interactive applications. The JavaScript language and browser APIs allow building user interfaces that are as rich as conventional desktop and mobile applications in many respects. Front-end JavaScript frameworks like React abstract away many of the low-level details of managing rich user interfaces, allowing developers to focus on delivering business functionality.

Back when JavaScript was only used to provide a little enhancement on top of server-rendered web pages, testing JavaScript code was both difficult and, in many cases, unnecessary. But with many web applications now implementing their entire user interface in JavaScript, testing that JavaScript code has become essential. Responding to this need, in the past few years the open-source community has heavily invested in JavaScript testing tools—test runners, framework-specific testing libraries, and browser automation tools.

But good testing tools aren't the only thing needed for developers to achieve a positive testing experience. Many developers loathe writing tests, and JavaScript developers are no exception. Maybe the tests take much longer to write than the corresponding production code. Maybe one change to production code breaks many tests, necessitating lots of effort to fix them. Maybe the tests fail sporadically for no obvious reason. Maybe the tests don't actually seem to be confirming anything that matters. Whatever the reason, many JavaScript developers feel like testing isn't delivering on its promise to make their apps better.

There's a good reason that testing is so challenging. After many years of practicing, studying, and having conversations about software testing, I've become convinced that testing is an irreducibly complex topic. There might be only one obvious way to implement a given feature, but there are usually several ways to test it, each with pros and cons that make it difficult to know which to choose. Judgment is necessary. And you can't learn judgment from a quick read of a testing tool's documentation: judgment only comes after time and experience.

But there is one hack we can use to develop our judgment more quickly. We can listen to *others'* experiences and learn from them—not only the experience of front-end JavaScript

developers, but also developers on other platforms who have struggled with the same testing challenges. This book examines testing principles that emerged in other programming environments and applies them to front-end development. These principles center around the practice of test-driven development—specifically, a variety known as *outside-in* test-driven development. This practice helps you write tests that thoroughly cover your app but are loosely-coupled so they don't hinder you from making changes.

Who Is This Book For

You're likely to benefit from reading this book if you would describe yourself as:

- **A front-end developer new to testing**. If you haven't written unit or end-to-end tests before, this book will teach you how to write both. You'll get experience with excellent testing tools and with techniques that will help you maximize the value of these tests and minimize their cost.
- **A front-end developer new to test-driven development**. If you write your front-end tests after you write your production code, this book will show you why you might consider writing your tests first. You'll learn how test-driven development makes it easier to fully cover the functionality of your app with tests, and you'll see how it prevents test fragility by keeping your tests focused on the interface rather than the implementation.
- **An experienced TDDer new to the front-end**. This was me when I moved into front-end development. I didn't want to leave behind the TDD practices that had helped me so much in server-side apps, but there weren't many resources on how to do TDD effectively in modern front-end frameworks. This book will help you apply the TDD techniques you love to React, and it will show how the different constraints of the front-end environment might lead to small adjustments to your TDD approach.
- **A front-end TDDer who only writes component tests, or writes them before end-to-end tests**. This is sometimes referred to as "classic TDD" or "middle-out TDD" because you start with the inside of your app—in the case of front-end apps, your components. You'll see how end-to-end tests complement your unit tests by adding a different kind of coverage, and how they can help your code become even more focused by steering you away from TDDing unneeded functionality. You'll also see how Cypress overcomes some of the pain points you may have experienced with end-to-end testing tools in the past.

Chapters

This book consists of three parts.

Part One, **Concepts**, lays out big-picture ideas related to outside-in TDD:

- *Why Agile?* describes some problems that commonly occur in software development, and it shows how addressing those problems is the goal of agile development practices. These practices include small stories, evolutionary design, test-driven development, refactoring, and others.
- *Testing Concepts* introduces some important terms related to software testing and clarifies how they will be used in this book.
- *Why TDD?* goes into detail about the agile development practice that this book focuses on: test-driven development. It will explain the surprising benefits of writing tests before you write production code, including regression safety, test robustness, and speed of development.
- *Outside-In TDD* describes the variant of TDD this book will follow, known as outside-in TDD. It explains how, in this approach, end-to-end tests and unit tests work together to confirm both the external and internal quality of your software.

Part Two, **Exercise**, walks you through putting outside-in TDD into practice by building the first few features of a real application in React.This part consists of the following chapters:

- *About This Exercise* describes the exercise in general and introduces the tech stack we'll be using.
- *Project Setup* sets up both the codebase and process we'll use throughout the exercise. We'll list out the stories we'll work on, create the project, and configure it. Before we even write the first feature we'll get tests running on a CI service and get the code automatically deploying to a hosting service.
- *Vertical Slice* puts outside-in TDD into practice with our first feature: reading data from an API server. We'll stay focused on a minimal feature slice that touches all layers of the app so that each will begin to be built out.
- *Refactoring Styles* shows how thorough test coverage allows us to implement functionality and styling in two separate steps. We'll take our plain-looking app and apply a nice look-and-feel to it, relying on the tests to confirm we haven't broken anything.
- *Edge Cases* adds polish to our first feature: visual feedback for loading and error states. Testing these edge cases at the unit level will keep our end-to-end tests simple and fast, and TDD will ensure that all the edge cases are covered by tests.

- *Writing Data* brings everything we've learned together as we build out a second feature: writing data to the API server. We see how to test HTML forms and verify data posted to the server. First we'll build out the core functionality in an outside-in TDD loop with end-to-end and unit tests, then we'll test-drive edge cases with additional unit tests.
- *Exercise Wrap-Up* reflects back on how the outside-in TDD process went over the course of the exercise and summarizes the benefits we gained by following that process.

Part Three, **Going Further**, provides supplemental material that builds on the first two parts:

- *Integration Testing the API Client* walks you through how you can directly test your API layer by testing your wrapper in integration with the third party code. We didn't do this in the main exercise because it doesn't provide a lot of value in this case, but it's a good tool to have in your tool belt if you find it would increase your confidence.
- *Asynchrony in React Testing Library* dives deeper into some challenges that came up during the exercise around asynchronous behavior in component tests. We'll look at the good reason React is giving us warnings, try alternate ways to address the root cause, and examine the outcome of those changes on our tests.
- *Next Steps* wraps up the book by pointing to additional resources you can use to learn more about test patterns, testing tools, TDD, refactoring, and other agile practices.

Prerequisites

You'll find this book easiest to follow if you already have the following:

Familiarity with React and Redux

The second part of this book is an exercise building a front-end application in React and Redux. It's helpful if you already have some familiarity with the stack you choose. We won't be using any features that are too advanced, and we'll explain what's happening as we go. But we won't explain *everything* about how these libraries work or why. Because of this, if the stack you choose isn't already familiar to you, it's recommended that you go through an introductory tutorial about that stack first. The React and Redux web sites include excellent documentation and are a great place to start:

- React web site[1]
- Redux web site[2]

[1] https://reactjs.org
[2] https://redux.js.org

Familiarity with Jest or Mocha

Jest and Mocha are two popular JavaScript testing libraries that are fairly similar. Jest is the main unit testing library we'll use in this book. But if you only know Mocha, don't worry: we won't be using any features of Jest that are too advanced, so you should be able to follow along easily enough. But if you haven't used either Jest or Morcha before, it's recommended that you look through the introductory parts of the Jest docs to get familiar.

- Jest web site[3]

Our end-to-end test framework, Cypress, uses Mocha under the hood instead of Jest. But in our Cypress tests we won't use many Mocha APIs; we'll mostly be using Cypress-specific ones. So don't worry about getting familiar with Mocha if you haven't used it before.

Code Formatting

When we are displaying whole new blocks of code, they'll be syntax highlighted like this:

```
export default function App() {
  return <div>Hello, world.</div>;
}
```

Because we'll be using test-driven development, we'll be spending less time writing large chunks of code and more time making tiny changes to existing code. When that happens, we'll strike through the lines to remove and bold the lines to add:

```
import RestaurantScreen from
  './components/RestaurantScreen';

export default function App() {
~~  return <div>Hello, world.</div>;~~
  return (
    <div>
      <RestaurantScreen />
    </div>
  );
}
```

[3] https://jestjs.io

About the Author

Josh Justice

I'm Josh Justice, and I've worked as a professional software developer since 2004. For the first 10 years I worked in server-rendered web applications (the only kind of web applications most of us had back then). I wasn't writing any automated tests; every change I made required manually retesting in the browser. As you might imagine, that resulted in a lot of builds sent back from QA, a lot of delayed releases, and a lot of bugs that made it to production anyway. I was fortunate enough not to have to work too many nights or weekends, but there was always the very real threat of an evening phone call about a production issue I needed to fix urgently. I tried different programming languages and frameworks to see if they would help make my apps more reliable, but none made much of a difference.

Eventually I was introduced to unit testing and browser automation testing, and I saw a glimmer of hope. Unfortunately, the language ecosystem and teams I was working on at the time didn't have much experience with automated testing; we tried to learn it but didn't have much success. That all changed when I started working in Ruby on Rails. In Ruby, the testing paths are well-trodden. I was able to learn from experienced testers and see the way they approached writing tests. They shipped code to production as soon as the pull request was merged, confident it would work because the tests passed—and then they made *me* do the same! I saw security patches applied in a matter of minutes instead of weeks; as soon as the tests were green we knew it was safe to release.

Testing wasn't the only thing I learned from the Ruby community. With the safety that test coverage gave us, we had the confidence to make tiny improvements to the code constantly. We renamed variables, methods, and classes to more clearly describe what they were intended to do. We split long, complex methods into smaller ones so that each was a short, easily-understood series of steps at a single level of abstraction. We rearranged code so that new features fit in cleanly instead of being hacked in with increasingly-complex

conditionals. When improvements like these are consistently applied to a codebase, I found that I could jump into that codebase for the first time and understand it quickly. This not only made my development more productive; it also made it a lot more fun. I spent a lot less time worried and stressed, and a lot more time interested and excited—and *that* made me more productive, too.

A few years after my professional focus shifted to Ruby, it shifted again, this time to front-end development. Because of all the benefits I'd seen from testing, one of the first things I looked for was how to test front-end web applications. The answer at the time was effectively "we're working on it." The community was still working through fundamental questions about how to build front-end apps that were consistent, performant, and simple—and with those fundamentals in flux, testing approaches were necessarily in flux as well. I tracked with community conversations about testing practices as they evolved over the years, and I became increasingly convinced that the testing principles I'd learned on the backend applied just as much to front-end apps. Those practices weren't widely applied on the front-end, but it wasn't usually because of inherent differences between the platforms: usually, it was due to a lack of information flow from one programming ecosystem to another.

That lack of testing information flow into the front-end community is the problem I've tried to address over the years by creating a variety of front-end testing resources. This book is the culmination of that process so far. It encompasses the practices I'm most convinced lead to applications that are reliable, maintainable, and evolvable. These are the practices I reach for on the front-end, and that I would reach for on any platform, language, or framework. This book contains the advice I most commonly give in code reviews and pairing sessions. These practices were passed along to me having stood the test of time, and I believe they'll continue to do so for you.

Thanks

It's no exaggeration to say that this book has no original content and is only an arrangement of the good ideas of others. I'd like to thank the following people for playing a role in the ideas or the process of writing it.

- Kent Beck for creating and writing about TDD, Extreme Programming, and other practices that help geeks feel safe in the world.
- Nat Pryce and Steve Freeman for evolving TDD by creating and writing about mock objects and outside-in TDD.
- Jeffrey Way for introducing me to TDD and object-oriented design.

- Toran Billups for helping me see the possibilities of outside-in TDD on the front end.
- The Big Nerd Ranch web team, past and present, for teaching and exemplifying agile development.
- Myron Marston and Erin Dees for making outside-in TDD so practical in Ruby.
- Kent C. Dodds, Edd Yerburgh, and the Cypress team for creating great front-end test tooling.
- Atlanta tech meetup organizers, connect.tech, and Chain React for opportunities to speak and teach about testing.
- Jack Franklin and Justin Searls for informing and challenging my thoughts on front-end testing.
- James Shore for consistently championing the relevance of TDD to JavaScript.
- Matthew Strickland and Jonathan Martin for encouraging and inspiring me to write and create other content.
- The creators of Docusaurus and VuePress for creating great platforms for publishing online content.
- Graham Lee, Brian Marick, Ron Jeffries, Noel Rappin, and Elisabeth Hendrickson for being generous enough to field my out-of-the-blue questions.
- My wife Jennifer and my three children, Emily, Katherine, and James, for supporting me in my passion for programming, and for making me look forward to finishing work for the day.

What's Next

With that, we're ready to get started learning about the concepts behind outside-in front-end development. We begin by looking at the problems that agile development practices are intended to solve.

Part One: Concepts

1. Why Agile?

The Problem

Software projects rarely go as smoothly as we would like. We make project plans, but how often does the reality end up matching those plans? Nobody says it better than Sandi Metz:

> Unfortunately, something will change. It always does. The customers didn't know what they wanted, they didn't say what they meant. You didn't understand their needs, you've learned how to do something better. Even applications that are perfect in every way are not stable. The application was a huge success, now everyone wants more. Change is unavoidable. It is ubiquitous, omnipresent, and inevitable.
>
> — Sandi Metz, *Practical Object-Oriented Design*

When you first start building a new software system, things go smoothly and it's easy for you to add new functionality. You feel so productive! But as time goes on, that sense of productivity wanes. When you change a bit of code, something seemingly-unrelated breaks. To add just one new feature, you need to make dozens of changes throughout the codebase.

To make sure your app continues to work as you change it, you need some kind of testing. Maybe you don't have any automated tests, so you have to rely entirely on manual testing. But as your feature set grows, the effort required to manually test it grows at the same rate. With each release you either need to allow more time for manual testing, hire more testers, or retest less of your codebase and increase the risk of breakage. Or maybe you *do* have automated tests, but changing one feature causes lots of tests to break, so the majority of your development time goes toward maintaining tests. Worse, maybe those tests don't actually catch the kinds of bugs your app tends to have, so you have to do just as much manual testing *in addition to* maintaining the test suite!

Over the lifetime of a software system, more and more effort is needed to get a smaller and smaller result. What causes these diminishing returns? Ron Jeffries explains in *The Nature of Software Development*:

> The time needed to build a feature comes from two main components: its inherent difficulty, and the accidental difficulty of putting it into whatever code already exists. Teams are good at estimating the inherent difficulty. What makes us erratic, what makes us slow down, is the accidental difficulty. We call this difficulty "bad code."
>
> If we allow code quality to decline, some features go in easily, sailing right through. Others that seem similar get entangled in twisty little passages of bad code. Similar work starts taking radically different amounts of time.

How can you prevent the accumulation of "bad," messy code as the project changes? Often the first thing that comes to mind is to *minimize* change by putting more effort into up-front design. If changing the system leads to messy code, then making and sticking to a plan for how to structure the code should help, right? The problem is that, despite our best efforts, up-front designs rarely fit the requirements perfectly—and if the requirements *change*, all bets are off. Any code that doesn't fit with the design will be messy, causing development slowdown.

A common response to *this* problem is to design for flexibility. For example, we don't know for sure what data store we'll use or what communication mechanisms the user will want, so we make those parts of our system configurable and pluggable. We think of everything in the system that could vary and we isolate each of those pieces so they can be replaced. But just because the system *could* change in a certain way, that doesn't mean it *will*. And every bit of flexibility and pluggability we build into the system has a cost: it's indirection that makes the code harder to understand and work with. When the system doesn't end up changing along the lines of a given configuration point, that configuration point adds cost without providing any benefit. And if a change is needed that *wasn't* one of the ones we built a configuration point for, we'll have to write messy code to add it in after all.

Whether from a lack of flexibility or from unnecessary indirection, it seems inevitable that development slows down as systems grow. How can we escape this dilemma?

How Agile Helps

Keeping the pace of development fast as systems grow is one of the main goals of agile software development. Rather than trying to resist or anticipate change, an agile team embraces change and adopts practices that help them effectively respond to that change. Here are some of the most central agile development practices:

Small Stories

We break the work up into minimum units of user-visible functionality. For example, an agile team doesn't build out an application's entire data layer at once. Instead, we build one user-facing feature, including just enough of the data later, business logic, and user interface to get it working. When that work is finished, we start the next story and add another slice of data layer, business logic, and user interface.

Evolutionary Design

As we're adding this functionality story-by-story, we don't try to predict everything our application will need and design an architecture that will satisfy all of it…because we know we'll be wrong. Instead, we strive to make the system's design the best fit for its functionality today, for the story we're currently working on. When we start the next story, then we adjust the design of the system to match *that* story's functionality. With a design that is constantly being custom-fit to the present reality, we should never have a system that is either under-designed with hacked-in changes or over-designed with unused flexibility.

Test-Driven Development

As we build our stories, we write the test first, and we only write production code in response to a failing test. This ensures that every bit of our logic is covered by test, so that as we rearrange the design of our system we know that we haven't broken anything. This level of test coverage significantly reduces the need for manual testing, which means our application can grow without the manual testing time increasing indefinitely. TDD also helps us identify and fix design issues in our code that could cause future development slowdown.

Refactoring

At several different moments during the agile process, we refactor: that is, we improve the arrangement of the code without changing its functionality. Refactoring is the third step of the TDD cycle, where after the test is passing we refactor to *better* code that keeps the test passing. Another time we refactor is while we're preparing to add new functionality: we consider if we can rearrange the code so that the new functionality fits in more naturally.

Code Review

We ensure that the person who wrote a bit of code isn't the only one who is familiar with it. We want another set of eyes on the code to find bugs and improvements, and we want to make the code easy to understand by any team member who will work on it in the future. Pull requests are a common way to do code review, and they can work well as long as reviewers are focusing on thoroughly understanding the code and not just giving a cursory glance.

Continuous Integration (CI)

When the term "continuous integration" was coined it referred to integrating team members' work together at least daily instead of using long-running branches. One key aspect of CI is having an integration machine that will automatically build and test the app to ensure that integrated code is always working. Today we have cloud services referred to as "continuous integration" services that handle that building and testing on both main branches and pull request branches. But just using one of those services doesn't mean you're practicing CI: you also need to merge in your branches frequently.

Continuous Delivery (CD)

Agile teams have the ability to release their system at any moment. To accomplish this, they ensure the main source control branch runs successfully and doesn't include incomplete work. In the rare case that one of these problems does happen, fixing it is the team's highest priority. CD also involves automating the steps to release the system. This doesn't mean that the team necessarily *does* release to production every time a new feature is completed, but they have the ability to do so.

Abstractions

Agile teams order their work to deliver the most important user-facing functionality first. One strategy they use is to reach for shared solutions and libraries rather than writing all their functionality from scratch. Shared solutions include community standard build systems, UI libraries, back-end frameworks, and hosting solutions like Netlify and Heroku. Every day you spend custom-building an implementation detail of your tech stack is a day you aren't delivering features to the user. When the team discovers that an abstraction is slowing down their delivery of business functionality, then and only then do they write lower-level code themselves.

Agile Team Practices

Most of the practices above are *technical* practices involving how individuals work with their code. There are also agile practices that are less technical and more focused on how individuals within a team work together. These practices aren't addressed in this book, but they are equally important. An effective agile team will be intentional about their approach to:

- Deciding what roles should be included on the team and how they should collaborate
- Eliciting needs and feedback from business users
- Writing and organizing stories
- Deciding whether or not estimation would provide value, and deciding how to do that estimation
- Coordinating work for a bit of user-facing functionality across multiple disciplines such as design, front-end, back-end, infrastructure
- Measuring their progress in terms of velocity or other metrics

To learn more about this broader scope of agile practices, check out Agile Methodology Resources.

What's Next

In this book we'll examine and try out most of the agile technical practices described above, with a particular emphasis on test-driven development. But before we can get to test-driven development, the next thing we need to do is lay a foundation of core testing concepts and define the testing terms we'll use in this book.

2. Testing Concepts

Gaining experience in any area of programming requires learning a variety of technical terms—and the area of testing is no different. Testing terms present a particular challenge: they have a tendency to be given many different definitions, often contradictory ones. To begin talking about testing more in depth, then, we need to lay out the terms we'll be using. We'll look at the variety of ways the terms are used in the industry, and we'll define how they will be used in this book.

Assertions and Expectations

One of the most foundational concepts of automated testing is an assertion: a check that something that *should* be the case really *is* the case. Many test frameworks have one or more `assert…()` functions that do just that:

```
assert.equal(sum, 42);
```

The test runner we'll be using for unit tests, Jest, has a slightly different terminology. Jest uses an `expect()` function, which allows you to chain function calls together to check a condition, resulting in test code that (arguably, to some people) reads more like natural language:

```
expect(sum).toEqual(42);
```

Checks that use an `expect()` function are often referred to as "expectations". In this book you'll see the terms "assertion" and "expectation" used interchangeably. There's no practical difference, other than that it reads a bit more naturally in a sentence to say that "In this test we *assert* that X is true".

The end-to-end testing tool we'll be using, Cypress, offers the ability to make a variety of assertions with the `.should()` method. But our Cypress tests are so simple that we won't end up needing any explicit `.should()` calls. Instead, we will call a method to look for an element on the page that `.contains()` a certain string. If that string is found on the page, the test will proceed, but if it isn't found, the test will fail. This is effectively an assertion even though the method isn't named "assert, "expect," or "should."

Unit Tests

The term "unit test" refers to an automated test of a portion, or unit, of your code. The differences in how people use the term "unit test" involve what they consider a "unit" to be.

The narrowest definition of a "unit" is a single function, object, or class. If a given unit depends on any other functions or objects, they will be replaced by a test double in the test. Of course, all code needs to depend on built-in language primitives, functions, and classes, so those are allowed in even the narrowest unit tests. And an exception is sometimes made for low-level utility libraries that effectively extend the language's standard library, such as Lodash for working with arrays and objects or `date-fns` for working with dates. But no other real dependencies are used in the test.

Another definition of a "unit" allows the code under test to interact with the framework it's built with but not other classes or functions in your application's code. When you have a React component function that uses React functionality for state and life cycle events, testing that component requires running it through React so that necessary functionality works. But that test can still isolate that component from data store dependencies and even child components.

Finally, a "unit" can be scoped to the function or object under test along with *all* of its real dependencies within the application. This is the approach generally taken by the inventors of the modern unit testing and test-driven development. With front-end application components, using a component's real dependencies includes rendering all of its child components. Some would refer to this kind of test as an "integration" test because of the potentially large amount of first-party and third-party code you are "integrating with" in the test. Whichever term is used, the most important thing is that everyone on your team is using a consistent approach and terminology for their tests.

In this book, we'll refer to the tests for our components as "unit tests." Because they will involve rendering the components, they will integrate with React. The tests will render all of the component's children, including third-party UI library components. However, our components under test will be isolated from our data store: instead of connecting to the real store, we'll make assertions on the messages our components attempt to send to the store.

Our unit tests of data layer code will run that code in integration with our data layer library, Redux. Functions in our data layer code that cooperate together will be tested together as well. For example, our data layer's architecture separates out functions that make asynchronous calls from functions that persist the returned data. Rather than testing these two types of function in separate tests, we'll test them in integration with one another in a single test. But

we'll isolate our data layer from our API client, so that our data layer tests aren't dependent on the API client.

End-to-End Tests

Whereas unit tests run against portions of your application's code, another type of test runs against the entire application and simulates a user interacting with it. There are a variety of terms for this kind of test, with meanings that are similar but not quite identical.

The terms "UI automation test" and "browser automation test" focus on the mechanics of the test: it automates interactions with the user interface. The term "browser automation test" is limited to tests of web applications, but "UI automation test" can apply to tests of either web applications or native mobile and desktop applications. These terms can be used to refer either to tests written by developers for features they built themselves (as we'll be doing in this book) or to tests written by test automation engineers for features built by others.

The terms "acceptance test", "feature test", and "functional test" focus on the scope of a test in this category: it covers a single feature, a user-facing bit of functionality. When this is your mental model for a test, you will tend to write the test to make it closely match the way a user thinks about the interaction, so that once it passes the user can feel confident accepting the feature as working correctly. Interestingly, although these types of tests are usually written to simulate user interactions, there is an alternative: a feature can also be tested by making a sequence of requests directly to your business logic code and verifying the resulting data. But in this book, as is typical in front-end development, we will run our feature tests through the UI of our app.

The terms "end-to-end test" and "system test" refer to testing an entire system together, not just part of it. But if you were hoping that programmers would at least be able to agree on what "the whole system" means, your hopes will be dashed: even these terms are used in different ways. You can test a front-end application and allow it to access the real back-end system, or you can isolate the front-end from the back-end and only test the front-end "system" "end-to-end." In the latter case, instead of referring to the test as "end-to-end" you might choose to refer to it as an "integration" test because you're testing all of the front-end code integrated together but not testing "end-to-end" from the standpoint of the running production system. But remember that some developers would use the term "integration test" to refer to the type of *unit* test we'll be writing. Whatever terms you choose, you probably shouldn't use the same term for two things at the same time, or things will get confusing: "the two kinds of test we have are integration tests and integration tests!"

In this book we'll write tests that drive the UI of our application, organized by feature. We will isolate our front-end from the back-end API to focus on testing our front-end application on its own. We'll use the term "end-to-end test" for these tests because it's commonly used in the front-end development world.

Stubbing and Mocking

The terms "stubbing" and "mocking" refer to replacing real production dependencies with simpler dependencies that makes testing easier. These terms can be used at both the unit and end-to-end testing level.

In unit tests, the general term for this type of testing dependency is a "test double." The term is analogous to a "stunt double" in a movie: a dependency that stands in for another dependency. In programming languages that are purely object-oriented, test double libraries are designed to create doubles for an entire object, but in JavaScript test double libraries are usually designed to create doubles for one function at a time.

Two of the more commonly used types of test double are "stubs" and "mocks." The two terms are often used interchangeably, but their original definitions had a difference of meaning.

A *stub* function is one that returns hard-coded data needed for the current test. For example, consider a component that calls a function that retrieves data from a web service. In one test you might stub that function to return a promise that resolves with web service response data and confirm that the component displays that data correctly. In another test you might stub the same function to return a promise that rejects with an error message, and test that the component correctly handles that error. In both cases, stubbing made it straightforward for you to configure the scenario you wanted to test.

A *mock* function is one that allows you to make assertions about whether and how it was called. A mock function can also optionally be configured to return data if the calling code requires it, just like a stub function. But mocks are particularly useful for cases where no return value is used by the calling code. For example, consider a form component that calls a function when it is submitted, passing it the form data. When testing that component, you could mock the function, fill out and submit the form, then check that the function was called with the data you filled out. The form might or might not use a return value from that function, but either way you want to make sure the function was called with the right arguments.

End-to-end tests don't often involve using test doubles to replace portions of your front-end application code, because you're usually testing your whole front-end application together.

Instead, when you replace a dependency in an end-to-end test it's generally something external to your front-end application. Cypress, the end-to-end test framework we'll be using, allows controlling a number of browser APIs, but in this book the only dependency we'll be controlling is the back-end web service. "Stubbing" the back-end involves setting up hard-coded HTTP responses to give the front-end app the data it needs for different scenarios. "Mocking" the back-end involves making assertions about what HTTP requests were sent to the back-end, which is especially useful when writing data. For example, if you create a new record, it's not enough to confirm that that record is shown in the UI: you also need to ensure it's properly sent to the back-end. Mocked requests allow you to check that the request was made with the correct data.

In the exercise we work through in this book, we'll take advantage of stubbing and mocking in both our unit and end-to-end tests.

What's Next

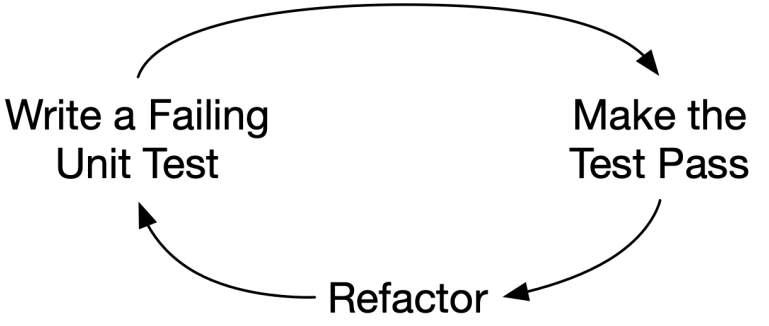
Now that we've gotten our testing terms straight, let's look at the way our agile development approach recommends going about testing: test-driven development. In the next chapter we'll describe what TDD means in detail, and we'll explain how the unintuitive practice of writing your tests first leads to a number of benefits.

3. Why Test-Driven Development?

This book will walk you through a number of agile development practices, but it has a particular focus on one such practice: test-driven development. Test-driven development is the practice of writing a test for application functionality before you write the functionality itself. It follows a three-step process, "Red-Green-Refactor":

1. **Red**: write a test for a small bit functionality that does not yet exist, and watch it fail.
2. **Green**: write only enough production code to pass the test.
3. **Refactor**: rearrange the test and production code to improve it without changing its functionality.

Then the cycle repeats: you write a test for the *next* bit of functionality and watch it fail, etc. This repeating sequence of three steps can be visualized as a loop:



The TDD Loop

Why would you want to follow test-driven development? There are a number of common problems in programming that test-driven development helps to solve. Let's take a look at some of them.

Regression Safety

As you add new features and make changes to existing features, you need a way to make sure you don't introduce any regressions—*unintended* changes to the application's

functionality. Full manual retesting isn't a scalable solution because it becomes impractical as the application grows larger, so an automated test suite is needed. Most developers who write tests do so after the corresponding production code is complete, an approach called "test-after development." But there are several things that make it difficult to achieve regression safety with test-after development.

First, with test-after development, you may find that some of the production code you wrote is difficult to write a test for. This can happen if your production code has a complex interface or many dependencies on the rest of your application. When a bit of code is designed in a way that is hard to test, developers will often adopt complex testing approaches that attempt to work around the problem. These approaches can be a lot of work and tend to make tests fragile—problems that can lead to giving up on testing the code altogether. Instead, when you find that a bit of code is hard to test, it's better to rearrange the code to make it more testable. (Some would say you shouldn't "design your code for the sake of the tests," but this is really designing the code to make it usable in new contexts, and a test is just one such context.) Rearranging the code can make it more testable, but it can be demotivating to do so because it risks breaking the code before tests can be put in place.

There's a second obstacle to achieving regression safety using test-after development: it makes it difficult to fully specify the functionality of your app. Fully specified means your tests cover every important behavior of your app: if the tests are passing, you can be sure the app is working. But how can you tell if you have enough tests to fully specify the app's functionality? You could try test coverage metrics, which indicate for each function, statement, and branch whether or not it is executed during a test. But metrics can't tell you if you are making assertions about every important result; in fact, you can max out the test coverage metrics without making any assertions at all! Also, when you have complex boolean expressions, metrics don't generally check if you are testing every possible combination of boolean values. Testing *every* boolean combination is often too many possibilities. Not every boolean combination is important for your application, but some are, and a tool can't make the call which is which: you need to make it yourself. Because of all this, code coverage tools can't guarantee that your application's functionality is fully specified, which can allow bugs to make it to production.

In contrast to these problems with test-after development, **test-driven development results in a test suite that provides thorough regression safety for your application**. You won't end up with code that can't be tested, because the test is what resulted in that code being written. And by definition TDD results in tests that fully specify your functionality: every bit of logic you've written is the result of a failing test that drove you to write it. Because of this, you can have high confidence that your test suite will catch unintentional changes.

Robust Tests

Even if you see the theoretical value of testing, in practice it may feel like your unit tests have a cost that outweighs their benefit. Sometimes it can seem like whenever you make the smallest change to production code, you need to change the tests as well, and the test changes take more time than the production code changes. Tests are supposed to ensure our changes don't break anything, but if the test fails when we make a change, how much assurance are we really getting?

When a test needs to change *every* time its production code changes, this is a sign of an over-specified test. Usually what is happening is that the test is specifying details of the production code's implementation. Instead, what we want is a test of the *interface* or *contract* of the production code: given a certain set of inputs, what are the outputs and effects visible to the rest of the application? Our test shouldn't care about what's happening *inside* the module as long as what's happening *outside* of it stays consistent. When you're testing the interface in this way, you can rearrange a module's implementation code to make it easier to add in a new requirement while the existing tests for that module continue to pass, confirming no existing requirements are broken.

Test-driven development guides you toward testing the interface rather than the implementation, because there *is* no implementation yet at the time you're writing the test. At that time, it's easy to visualize the inputs and outputs of the production code you want to write, and it's harder to visualize implementation details such as internal state and helper function calls. Because you can't visualize the implementation, you can't write a test that's coupled to it; instead, your test specifies only the interface of the code. This helps you build up a test suite that is less fragile, that doesn't need to change every time production code changes. As a result, the value of your test suite for regression safety goes up and the cost of maintaining it goes down.

Speed of Development

As we discussed in Why Agile?, as applications grow over time, the speed of development tends to get slower and slower. There is more code, so when you need to make a change, more existing code is affected. There is an increasing (sometimes *exponentially*-increasing) amount of effort needed to add functionality. It's even possible to reach a point where it takes all the developers' effort just to keep the system working, and adding new functionality is impossible.

Why does this slowdown happen? Because when you wrote the code in the first place you couldn't foresee all future requirements. Some new features you add will fit easily into the existing code, but many will not. To get those new features in you have to resort to workarounds that are complex and inelegant but get the job done. As these workarounds multiply, you end up with code that is very difficult to understand and change. Some of your functions end up as massive sequences of unrelated branching logic, and the amount of effort to follow what one of them is doing can be overwhelming.

One way test-driven development speeds up development is by guiding you toward the simplest implementation. We programmers can tend to jump to conclusions by coming up with sophisticated ways to implement a module, but often the needs of an application are simpler than that. TDD leads you to start with a simple implementation and only refactor to a complex one when there are enough tests to force you to do so. If you don't need those tests yet, you don't need that costly complexity that isn't adding value.

Test-driven development also guides you to write the simplest interface. Before you write the implementation of a module, you write the interface presented to the rest of your application for using it. When you write the function call first, you're a lot less likely to end up with a function that takes eight positional arguments and a lot more likely to think of a simpler interface for that function. Interface thinking helps ensure your code presents a clean abstraction to the rest of the application. This reduces the effort required for future developers to understand the calling code, lowering the cost of maintenance.

When you need to add additional functionality, you want to avoid workarounds that will slow you down over time. Instead, you need to adjust the code as you go so that it's easy to add in the new requirement. An ideal regression test suite would give you the confidence to make these changes. But if you have even a *little bit* of doubt in your test suite, you'll hesitate because you don't want to risk breaking something. Using a workaround will be safer than reorganizing the code. But after an accumulation of many such workarounds you can end up with a codebase that is a mess of giant functions with deeply-nested conditional logic that continues to get slower and more fragile to work with.

With test-driven development, you have a regression test suite you know you can trust, so you can clean up the code any time with very little friction. You can make the code just a bit clearer or simpler, and if the tests are green you will have a high degree of confidence that you haven't broken anything. Over time these tiny improvements add up to a codebase that looks like it was designed from the start knowing what you know now. The simple, clear code helps your development speed stay fast.

Another cause of development slowdown is dependencies. If each part of your code talks to many others then changes are likely to have a ripple effect throughout your codebase: each

small change will cascade into many more necessary changes. Code that is easy to change is loosely-coupled, with few dependencies on other bits of code. Why does code end up with many dependencies? One reason is that it's difficult to visualize your code's dependencies in production use. Your application is arranged just so, and all the bits are ready and available for your code to use them—which is not so great when things need to change.

Unit testing reveals the dependencies in your application because they are an instance of reusing your code in a second context. If the code has few dependencies, it will be easy to use on its own and therefore relatively easy to write a test for. But if the code depends on the rest of your application being available, it will be difficult to write a test for it. This difficulty can help you identify a dependency problem, but it won't help you *solve* that problem. Breaking those dependencies will require changing your code, and you don't yet have the code under test to be able to change it safely.

Test-driven development helps you avoid writing code with too many dependencies in the first place. Because you're writing the test first, you'll quickly see if too many dependencies are required to set up the test, and you can change your strategy before you even write the production code. As a result, you'll end up with code with minimal dependencies. This means that changes you make in one bit of code will be less likely to require changes in many other places in your app, allowing you to deliver features more quickly and smoothly.

When Not to TDD?

Test-driven development provides many benefits, and the argument of this book is that far more projects would benefit from it than are using it today. That said, this doesn't mean TDD is a fit for every software project. Let's look at the cases where TDD might *not* be such a good fit, but also warnings for each about why you shouldn't make that decision lightly.

Throwaway Code

If your code will be used only a few times and then discarded, there is little need for robustness or evolving the code. This is true if you know *for sure* the code won't be used on an ongoing basis.

However, many programmers have worked on a project that everyone agreed was "only a proof-of-concept" but nonetheless ends up shipped to production. The risk you take by not TDDing in this case is that if it *does* end up shipped to production, you will already be started down the path of having code that isn't well-specified by tests.

Rapidly-Changing Organizations

If the business is undergoing frequent fundamental changes, code is more likely to be discarded than evolved, so it isn't valuable to prepare to change that code. An example would be a startup that is pivoting frequently. In systems built for an organization like this, it can be better to limit your test suite to end-to-end tests of the most business-critical flows in the application.

But what about when the business settles down and needs to start evolving on a stable codebase? At this point the code will already be written and it will be difficult to add thorough test coverage you can have confidence in.

Systems That Won't Change

For domains where the needs are well-understood, the system may not evolve much over time. It's a system with a known end state, and once that state is reached further feature changes will be minimal. So if you put effort into getting the initial design right and don't make many mistakes, you won't need to make a lot of changes. I've worked in domains like this.

It's easy to *think* that things are certain and will never change, because we humans find comfort in certainty. Nonetheless, many programmers' experience is that software often requires more changes than you think it will. And if nothing else, the environment your code runs in will need to change, as operating systems and web browsers are upgraded. And you will at least need to update your underlying libraries for security patches. After any of these changes your application needs to be regression tested, and you will have backed yourself into a corner where you don't have the test coverage ready.

Spikes

Say you have a general idea for a feature, but you don't know exactly how you want it to work. You want to play around with different alternatives to see how they feel before you commit to one. In this approach, you don't *know* what to specify in a test, and if you did specify something it would likely be thrown out 15 minutes later. So instead, you just write the feature code and see how it works out. This approach is called a "spike,"" and it's looked upon favorably in TDD circles. The question is, once you settle on a final approach, do you keep your untested code as-is, or do you try to retrofit tests around it? TDD advocates would recommend a third option: treat the spike as a learning process, and take those lessons with you as you start over to TDD the code. This takes some extra effort, but when you're familiar

with a technology most applications won't require too many spikes: most features are more boring than that!

Human Limitations

One objection to test-driven development is that you can *theoretically* get all the same benefits just by knowing the above software design principles and being disciplined to apply them. Think carefully about the dependencies of every piece of code. Don't give in to the temptation to code workarounds. In each test you write be sure that every edge case is covered. That seems like it should give you a codebase and test suite that are as good as the ones TDD would give you.

But is this approach practical? I would ask, have you ever worked with a developer who isn't that careful all the time? If not, I'd like to know what League of Extraordinary Programmers you work for! Most of us would agree that most developers aren't that careful *all* the time. Do you want to write your code in such a way that only the most consistently careful developers are qualified to work on your codebase? That approach leads to an industry that is so demanding that junior developers can't get a job and senior developers are stressed by unrealistic expectations.

Let me ask a more personal question: are *you* always that careful? *Always?* Even when management is demanding three number-one priorities before the end of the day? Even when you're sick? Even in the middle of stressful life events or world events?

Programming allows us to create incredibly powerful software with relative ease, and as a result, programmers can be tempted to subconsciously think that they have unlimited abilities. But programmers are still human, and we have limited energy, attention, and patience (especially patience). We can't perform at our peak capacity 100% of the time. But if we accept and embrace our limited capacities, we will look for and rely on techniques that support those limitations.

Test-driven development is one such technique. Instead of thinking about the abstract question "does my code have too many dependencies?", we can just see if it feels difficult to write the test. Instead of asking the abstract "am I testing all the edge cases of the code?", we can focus on the more concrete "what is the next bit of functionality I need to test-drive?" And when we're low on energy and tempted to take shortcuts, our conscience won't remind us about all the big-picture design principles, but it might remind us "right now I would usually be writing the test first."

So can you get all the same benefits of TDD by being disciplined about software design

principles? Maybe on your best day. But TDD helps you consistently get those benefits, even on the not-so-good days.

Personal Wiring

In addition to *project* reasons you might not want to use TDD, there is also a *personal* reason. The minute-by-minute process of test-driven development is more inherently enjoyable for some people than others. Some developers will enjoy it even on projects where it doesn't provide a lot of benefit, and to other developers it feels like a slog even when they agree it's important for their project. If you fall into the latter group, you might choose to use TDD only on some portions of your codebase where the value is higher and the cost is lower.

As you can probably guess, I fall into the "enjoys TDD" category, so it would be easy for me to say "use TDD for everything." But if you don't enjoy it, I understand choosing to reach for it less frequently. But you're still responsible for the maintainability of your system. You need to be able to make changes without breaking key business functionality, and you probably want to be able to do so without having to do emergency fixes during nights and weekends. You need to be able to keep up the pace of development, and you probably want code that's easy to understand and a joy to work in rather than a tedious chore.

If you aren't using TDD to accomplish those goals, you need to find another way to accomplish them. As we saw in the previous section, "just try harder" isn't an effective strategy because it requires every team member to be operating at peak levels of discipline all the time. And unfortunately I don't have good suggestions for other ways to accomplish those goals. I'm not saying there aren't any; I just don't know them.

It's not fair that TDD and its benefits come more easily to some than others, but it's true. If you're in the latter group, that doesn't make you a worse developer: every developer has different strengths they bring to their team, and TDD isn't the only skill that matters. My encouragement would be this: if you've tried TDD and you feel like it isn't very enjoyable for you, don't assume it will always be that way. Give it a try in the exercise in this book. Practice it on your own. Maybe at some point a switch will flip and you'll find it more motivating. Maybe you'll find a few more parts of your codebase that TDD seems like a fit for.

What's Next

We've just seen how test-driven development works and the benefits it provides. This book will follow a particular kind of TDD approach called *outside-in* TDD. In the next

chapter, we'll see how outside-in TDD builds on the practices we've just examined, providing additional benefits and giving you additional confidence.

4. Overview of Outside-In TDD

Beyond Traditional TDD

Traditional test-driven development is a process that is specifically about unit tests: you create objects and call functions and methods. It's sometimes referred to as "middle-out TDD", because you start in the middle of your application building domain logic. This exclusive focus on unit tests comes with a few trade-offs.

First, because middle-out TDD works at the level of objects and functions, it doesn't address testing your UI. When TDD was created, UI testing technology was immature, unreliable, and difficult to use, so it wasn't incorporated into the process. Today we have better technologies for UI testing, especially on the web—ones that are more reliable, stable, and feasible for developers to write. But because these kinds of tests didn't exist at the inception of traditional TDD, it doesn't provide any guidance on how to incorporate end-to-end tests into your TDD workflow.

Another downside of middle-out TDD is the risk of building functionality that is unused or difficult to use. Say you put a lot of effort TDDing a module for handling data, and then you prepare to integrate it with the rest of your application. Maybe it turns out your data needs to be stored elsewhere, so you don't actually need the module you put so much effort into. Or maybe you discover that in order for your application to use the module it needs to have a different interface than the one you built it with, and you have to rework it.

Finally, a trade-off of the middle-out approach is that it usually (but not necessarily) involves testing code in integration with its dependencies—the other code it works with in production. The upside of this approach is that it can catch bugs in how modules integrate with one another. But it also means that a bug in one lower-level module can cause failures in the tests of many higher-level modules. There can also be a lack of defect localization: the tests aren't able to pinpoint where the problem originates in a lower-level module because they only see the result that comes out of the higher-level module.

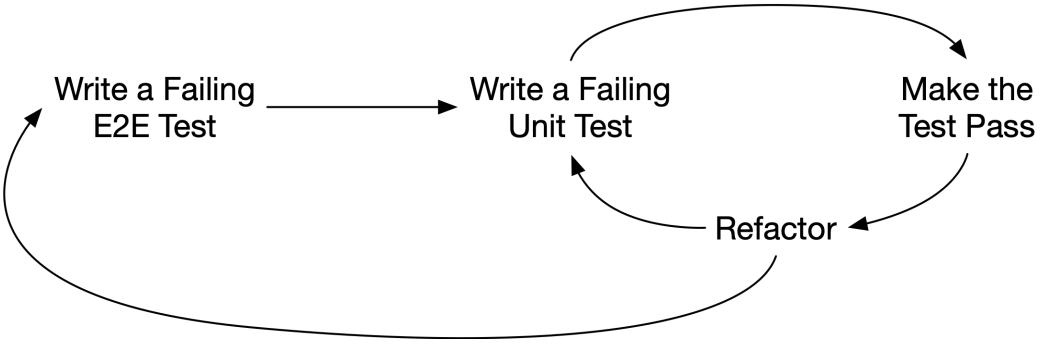
To see if we can overcome these downsides to traditional TDD, let's consider an alternate way to approach test-driven development. Referred to as *outside-in TDD,* it provides a structure for using end-to-end and unit tests together in a complementary way, solving the above problems and providing additional benefits. Let's see how.

The Two-Level TDD Loop

Remember the concept of the TDD loop: red, green, refactor? The first thing outside-in TDD adds is a *second* TDD loop outside the first one:

1. Write an E2E test and watch it fail.
2. Step down to a unit test and use the Red-Green-Refactor loop to implement just enough functionality to get past the current E2E test failure.
3. Step back up and rerun the E2E test to see if it passes. As long as it still fails, repeat the unit-level loop to address each E2E failure in turn.

These steps can be visualized as a two-level loop:



The outside-in TDD loop

This style of TDD is called "outside-in" because you start from the outside of your application: the user interface, as tested by the E2E test. Then you step inward to implement the low-level functionality needed to implement the desired outwardly-visible behavior.

Now that we've seen what outside-in TDD entails at a high level, let's look at its component parts to see how they work to address the TDD problems we saw above and provide additional benefits.

The Role of End-to-End Tests

In outside-in TDD, end-to-end tests work together with unit tests, providing test coverage of the same application functionality in a complementary way. Each type of test provides a different value. First let's look at the role of end-to-end tests.

End-to-end tests confirm that your application does what the user wants it to do. At the most basic level, they ensure that the logic you built is actually reachable through the user interface. They also ensure that all the code works together correctly in the context of the running app—the maximum level of test realism. This is sometimes referred to as "external quality:" from the outside, the app works.

End-to-end tests provide a safe way for you make major changes to your app without breaking anything. They're able to do this because as long as the test can still find UI elements that match what it's looking for, everything about the implementation of the app can change. You can replace entire function or object hierarchies, for example if you want to change the technology used for your data layer. You can even reuse the same Cypress tests if you rewrite your application in another framework. Our React exercise demonstrates this: it has almost identical Cypress tests to an older version of the exercise written in Vue.js! For large changes like these, unit tests don't provide a lot of safety because they will fail when units and the ways they interact are replaced.

Another benefit of the end-to-end tests produced by outside-in TDD is that they help you build only what you need. In outside-in TDD, each end-to-end test focuses on one user-facing feature. When you start working on the feature, you write an end-to-end test for it, then you build out the minimum code necessary to get the end-to-end test passing. When it passes, you're done with that feature. This ensures that you only build what is immediately useful to provide functionality to a user. It also prevents code from being written with an interface the app can't use, because you write the code that calls into the module before you write the module itself.

The Role of Unit Tests

With all these benefits of end-to-end tests, is there any need to write unit tests too? Why not stop with the end-to-end tests?

Whereas end-to-end tests confirm the external quality of your app, unit tests expose its "internal quality" by showing how your units are used. The attributes of your code we discussed in "Speed of Development" are all aspects of internal quality. Do your units have clear and simple interfaces? Are they easy to instantiate for tests, or are there a lot of required dependencies that are going to make them harder to change? As your application grows, these factors affect how easy it is to make changes to it—but these factors are invisible to end-to-end tests. You can have an app that works reliably from the outside, but is a mess of spaghetti code on the inside, and that means you'll have trouble handling future change. Unit tests help steer you towards good design attributes that pay off in the long run.

Unit tests also run much more quickly than end-to-end tests, which provides a number of benefits. This speed means you can keep the tests for the module you're working on running continually, so that when you introduce a bug you find out right away. This speeds also makes it feasible to cover every edge case with unit tests, something that isn't realistic with end-to-end tests for a system of any substantial size. This full coverage is what gives you the safety to refactor your code so you can make it better and better over time.

Because of the complementary value of end-to-end and unit tests, outside-in TDDers write both without thinking of it as duplicating effort. Instead, they see that each type of test covers the limitations of the other.

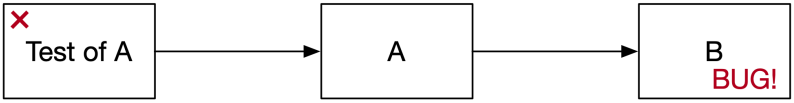
Write the Code You Wish You Had

Since the outside-in TDD process starts from the outside, how can you TDD code that depend on other code that hasn't been written yet? The solution is a practice called "writing the code you wish you had." When you are test-driving one unit of code, think about what functionality belongs in the unit itself and what functionality should be delegated to collaborators (other functions or objects). If that collaborator doesn't already exist, write the code you wish you had: pretend it does exist, and call it the way you'd like to be able to call it. In the test, use test doubles to take the place of that collaborator so you can verify how the unit you're testing interacts with this collaborator.

When you finish test-driving the current unit, your next step is to build any new collaborators that you scaffolded with test doubles. Test-drive the collaborators to match the interface you designed for them in the test of the first unit. Remember, only build the functionality necessary for that collaborator to satisfy the current feature—which might be less than *all* the functionality you could imagine it to have.

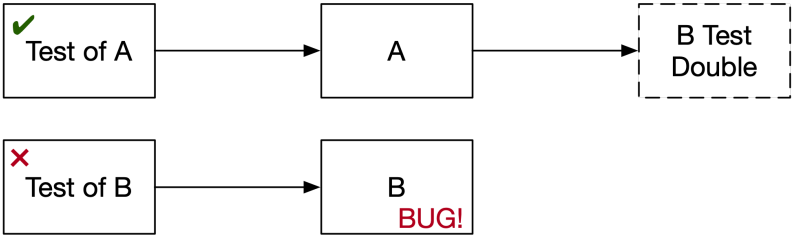
Using test doubles to isolate your units from one another has the benefit of providing good defect localization: when a bug is introduced, your tests will pinpoint which unit has the bug and what exactly is going wrong.

To see how this works, let's first consider the case where you *aren't* using test doubles. Say you have a module A that depends on module B, and in your test of module A you're allowing it to use the real module B. When there is a bug in module B, module A's test will fail even though the problem isn't in module A.



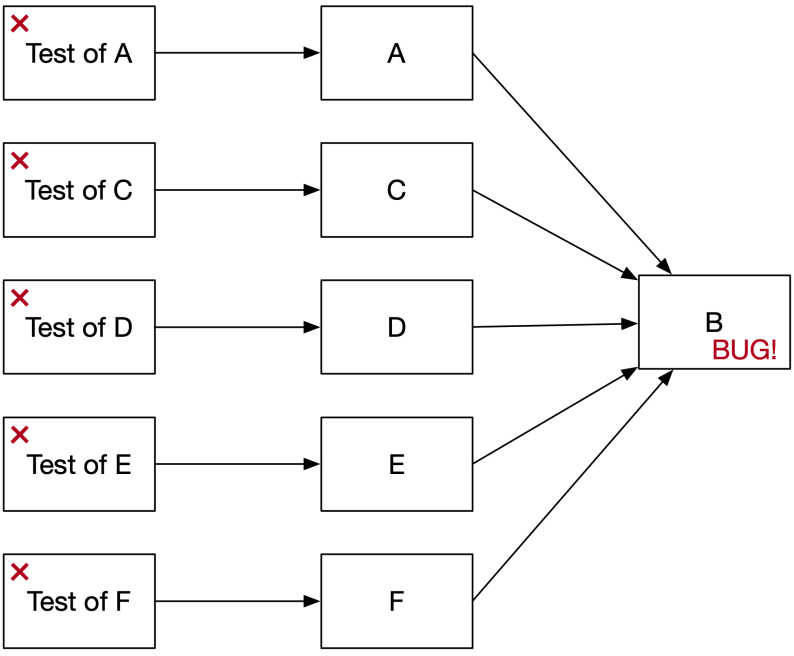
A test of module A integrated with module B. B has a bug so the test of A fails.

Now, what happens if in the test of module A we replace module B with a test double? This changes the meaning of module A's test to "*if* module B returns the correct result, module A behaves correctly," and because it passes you know that there is no bug in module A itself. Only the test of module B would fail, making it obvious that the problem is in module B.



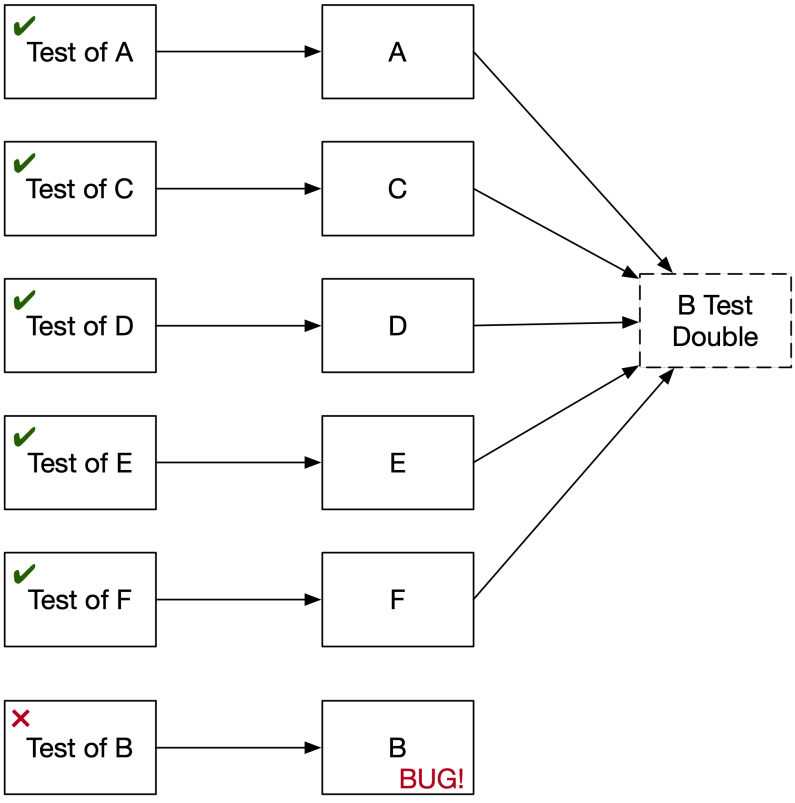
A test of module A integrated with a test double, and a test of buggy module B separately. Only the test of B fails.

This kind of test isolation provides even more benefit when a lower-level module is used by many higher-level ones. Say you have *five* modules that all depend on module B, and they are tested in integration with module B. If module B has a bug, the tests of *all five* higher-level modules would fail, making it hard to identify the underlying cause.



Five modules integrated with buggy module B. The tests of all five modules fail.

Using test doubles, the five tests for those modules would pass, and only the test for module B would fail—the ideal outcome to help you pinpoint bugs.



Five modules tested against a test double, and a test of buggy module B separately. The five modules' tests pass and only B's test fails.

It's common to hear criticisms of "mocking" in tests, and in fact those criticisms often apply equally to any kind of test double. Outside-in TDD provides a response to these criticisms by serving as an illustration of how mocks are intended to be used. (In fact, the creators of outside-in TDD are also the creators of mock objects!)

Criticisms of mocks that you might hear include:

- *"Mocks make your tests less realistic."* Considering the example above, does replacing module B with a test double make the test of module A unrealistic? No, it makes it more focused, testing module A in isolation from other code. True, it doesn't ensure all your units work together—but that's a job best suited to *end-to-end* tests, not unit tests. By relying on end-to-end tests for integration, you're free to test your units in isolation so you can get the benefits of isolated testing we've discussed.
- *"Mocks make your tests more complex because you end up creating mocks that return*

mocks that return mocks." If that happens, the problem is not with mocks but with the design of the code under test. It reveals that the code has deep coupling to other code. This is a sign that the production code should be changed to have simpler dependencies: specifically, to only call dependencies passed directly to it, so that only one level of mock is needed. Deep coupling is a problem that can be easy to miss when writing the code, but mocks help you see the problem so you can fix it. This is a point in mocks' favor.

What's Next

In this chapter we saw that outside-in test-driven development involves a nested loop where you test-drive a feature with an end-to-end test and build out each necessary piece with a series of unit tests. We saw that end-to-end tests and unit tests work together, the former ensuring the external quality of your app and the latter ensuring the internal quality.

With this, we've completed our survey of agile development practices. The next part of this book is an exercise where we'll put these practices into use to build an app using React.

Part Two: Exercise

5. About This Exercise

To see outside-in test-driven development in action, let's walk through creating a few features in a simple front-end application. We'll build an app for rating dishes at restaurants, called Opinion Ate. We'll get to experience all parts of the outside-in TDD process, but note that we'll only get the app started and won't get anywhere near finishing it.

If you'd like to download the completed project, you can do so from the Opinion Ate React repo[4]. But I would highly encourage you to work through the exercise yourself. Even more so than programming in general, test-driven development requires practice to get into the habit and to really experience the benefits.

Connect and Get Help!
As you go through this exercise, if you get stuck or just want to talk about what you're learning, feel free to join the book's online chat at https://link.outsidein.dev/chat. It's a great way to connect with the author (me, Josh!) and other readers.

A Note on Learning

If you aren't used to test-driven development, the process can feel slow at first, and it can be tempting to give up. Stick with it through this guide! The value of TDD usually doesn't click until you've gotten a bit of practice. Anything new you learn is going to be slower while you're learning it. Once you've gotten some practice with TDD it'll be a tool in your tool belt, and then you'll be in a better position to decide whether and how often to use it.

As is the case with most TDD tutorials, the functionality we'll be writing here is so simple that it would probably be quicker to write it without tests. In real applications, the time TDD takes is offset by the time you save troubleshooting, tracking down production bugs, restructuring your code, manually testing, and struggling to write tests for code that wasn't written to be testable. It's difficult to demonstrate that kind of time savings in an exercise, but consider this: how much time do you spend on all those problems? How much more enjoyable would your development process be if you could significantly reduce them? Once you've learned TDD you can try it out on your real projects and see if you see improvements.

[4] https://link.outsidein.dev/repo

Tech Stack

Our application is going to follow a common architectural pattern for front-end apps involving three layers:

1. **User interface**: the components that make up the screens. Implemented in React.
2. **State management**: stores application data and provides operations to work with it. Implemented in Redux.
3. **API client**: provides access to a web service. Implemented using Axios.

If you use a different front-end library or any other libraries, don't worry: the testing principles and practices in this book apply to any front-end application. Go through the exercise, and afterward you'll be able to apply what you learn to your stack of choice.

Here's the full stack of libraries we'll use for our React application:

Build Tooling: Create React App

Create React App[5] allows running our application locally and building it for production. Depending on your production needs you might or might not want a more flexible build tool, like a custom webpack config or Parcel. This tutorial doesn't get into build configuration, though, so Create React App will work fine, and should be familiar to many readers.

State Management: Redux

Redux[6] is a state management library that's widely used in the React ecosystem. It used to be the go-to state management library for real-world React applications, but with the release of React's Context API and the `useReducer()` hook it isn't used quite as widely. Redux is now more likely to be used only in cases where you have fairly complex data structures that need to be widely shared throughout the application—which some would argue was the intended use in the first place.

The reason we're using Redux for this exercise is because it provides a strong boundary between the UI layer and the data layer. If you use React's built-in state APIs for your data layer it tends to be coupled to components and harder to test in isolation; doing so takes work, creativity, and discipline. By contrast, when you use Redux with React in the idiomatic way

[5] https://create-react-app.dev
[6] https://redux.js.org

you get an unmistakable separation between the UI layer and the data layer for free. Redux-Thunk actions aren't impacted by the React render cycle with its challenges to asynchronous testing; they are normal JavaScript async functions and can be easily tested as such.

Once you've seen the benefits of keeping your data layer separate from React, you'll have a goal to shoot for with any data layer approach you use.

State Management Asynchrony: Redux Thunk

Redux Thunk[7] is the recommended way to add asynchrony to a Redux data layer for most projects. It works directly with JavaScript's built-in promises, so this makes it a natural fit for most JavaScript developers.

HTTP Client: Axios

Axios[8] provides a nice simple interface for making web service requests. The browser's built-in `fetch()` function is close, but Axios removes some repetitive parts and improves on the API.

UI Components: MUI

Agile development is all about minimizing unnecessary work. For side projects, internally-facing systems, and MVPs, unless visual design is your passion you may be better off using an off-the-shelf component library. Plus, with a thorough test suite like the one we'll write, you can always refactor to a new visual design with confidence that you haven't broken any functionality. For this tutorial we'll go with MUI[9], a popular React implementation of Google's Material Design.

Test Runner: Jest

Jest[10] has one of the most popular JavaScript test runners for a number of years, especially in the React ecosystem. It includes everything you need out of the box for testing plain JavaScript code, including the ability to create test doubles.

[7] https://github.com/reduxjs/redux-thunk
[8] https://axios-http.com
[9] https://mui.com
[10] https://jestjs.io

Component Tests: React Testing Library

React Testing Library[11] (RTL) will help us write component tests. It's designed around testing the interface instead of the implementation, which aligns with the testing philosophy we'll take in this book: that way our tests are less likely to break as the application changes.

End-to-End Tests: Cypress

Cypress[12] is an end-to-end testing tool that was written with test-driven development in mind. Because it runs in the same browser context as your front-end app, it has insight into the event loop and network requests, reducing flake and allowing easy request stubbing. If you've had a bad experience with other browser automation tools in the past, Cypress will convince you that E2E tests *can* be valuable and enjoyable.

In addition to E2E tests, Cypress has been building component testing functionality, to meet some of the same needs as RTL. We haven't used it for this book because it's in beta as of the time of this writing and has less broad adoption than RTL. But its approach has some interesting benefits, and we'll be keeping an eye on it as it develops.

Continuous Integration: GitHub Actions

GitHub is extremely popular for source control, and it has a CI service built in as well: GitHub Actions[13]. There are other great CI options too, but the GitHub integration means that all we need to do is add an Actions config file and we're set to run our tests on every pull request.

Deployment: Netlify

For deploying front-end applications there's no service simpler than Netlify[14]. Just choose your repo and Netlify will automatically configure your build process, build your app, and deploy it. We'll only use the most basic Netlify features in this tutorial, but it also has features you'll need to take your app to production, such as adding a custom domain with an automatically-provisioned SSL certificate.

[11] https://testing-library.com/react
[12] https://www.cypress.io
[13] https://github.com/features/actions
[14] https://www.netlify.com

What's Next

Now that we've reviewed the tech stack we'll be using, it's time to get our app set up. In the next chapter we'll create our application and the environment it runs in, and we'll do some setup for our development process.

There's More!

You've reached the end of the free sample of *Outside-In React Development*.

To read the rest of this exercise, buy the full book here:

https://leanpub.com/outside-in-react-development

6. Project Setup

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Making a List of Stories

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Setting Up Development Environment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Git

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Node

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Yarn

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

An Editor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Creating the App

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Setting Up Auto-Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Running Tests on CI

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Setting Up Automatic Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Filling In the Readme

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

7. Vertical Slice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Setup

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Reviewing the Back-End

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

End-to-End Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Stepping Down to a Unit Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Stepping Back Up

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Unit Testing the Store

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Creating the API Client

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Pull Request Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

8. Refactoring Styles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

9. Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Loading Indicator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Component Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Data Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Error Flag

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Component Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Store Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

10. Writing Data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Main Functionality

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

End-to-End Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Unit Testing the Component

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Stepping Back Up

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Unit Testing the Store

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Creating the API Method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Clearing the Text Field

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Validation Error

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Server Error

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

11. Exercise Wrap-Up

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

What's Next

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Part Three: Going Further

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

12. Integration Testing the API Client

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

13. Asynchrony in React Testing Library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

`findBy*()` Methods For Simple Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

`waitFor()` For Complex Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Assessment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

14. Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

A Community of Practice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Testing Tool Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Books

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Outside-In TDD

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Test Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Agile Methodology

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.

Epilogue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/outside-in-react-development.