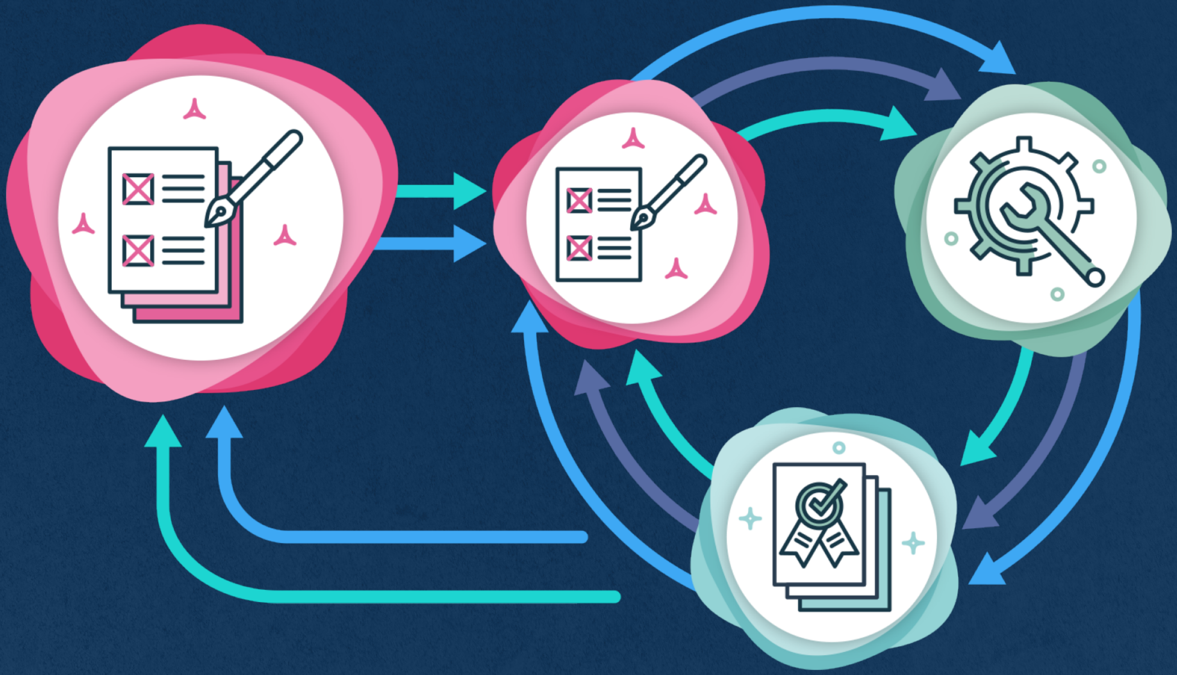
OUTSIDE-IN
React Development

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 2022-10-13

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 can come up around asynchronous behavior in component tests. We'll look at the good reason React might give 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 formatted 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.
- Zac Brooks for designing the cover of the book.
- 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

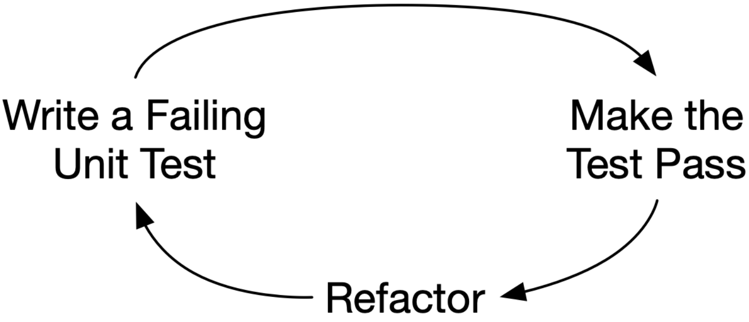
Now that we've gotten our testing terms straight, let's look at the way our agile development approach recommends going about testing: test-driven development. In the next chapter we'll describe what TDD means in detail, and we'll explain how the unintuitive practice of writing your tests first leads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

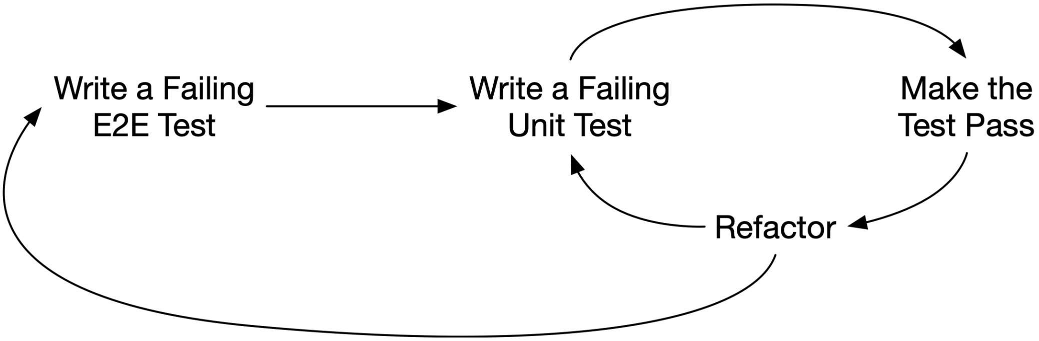
To see if we can overcome these downsides to traditional TDD, let's consider an alternate way to approach test-driven development. Referred to as *outside-in TDD,* it provides a structure for using end-to-end and unit tests together in a complementary way, solving the above problems and providing additional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

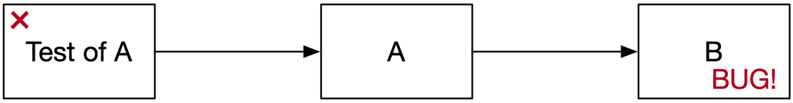
Write the Code You Wish You Had

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.

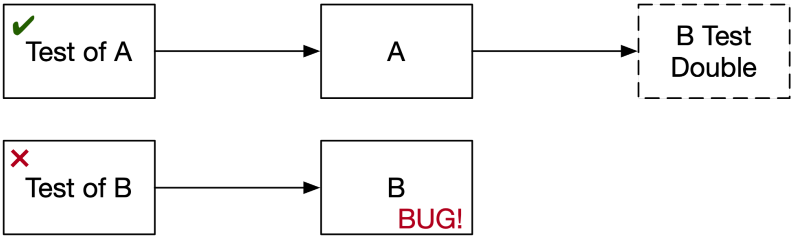
Using test doubles to isolate your units from one another has the benefit of providing good defect localization: when a bug is introduced, your tests will pinpoint which unit has the bug and what exactly is going wrong.

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.



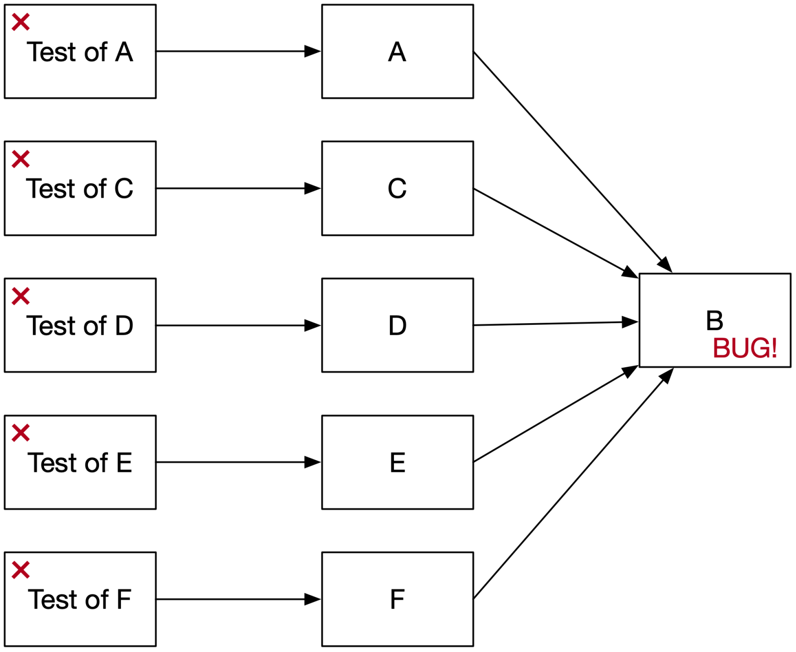
A test of module A integrated with module B. B has a bug so the test of A fails.

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.



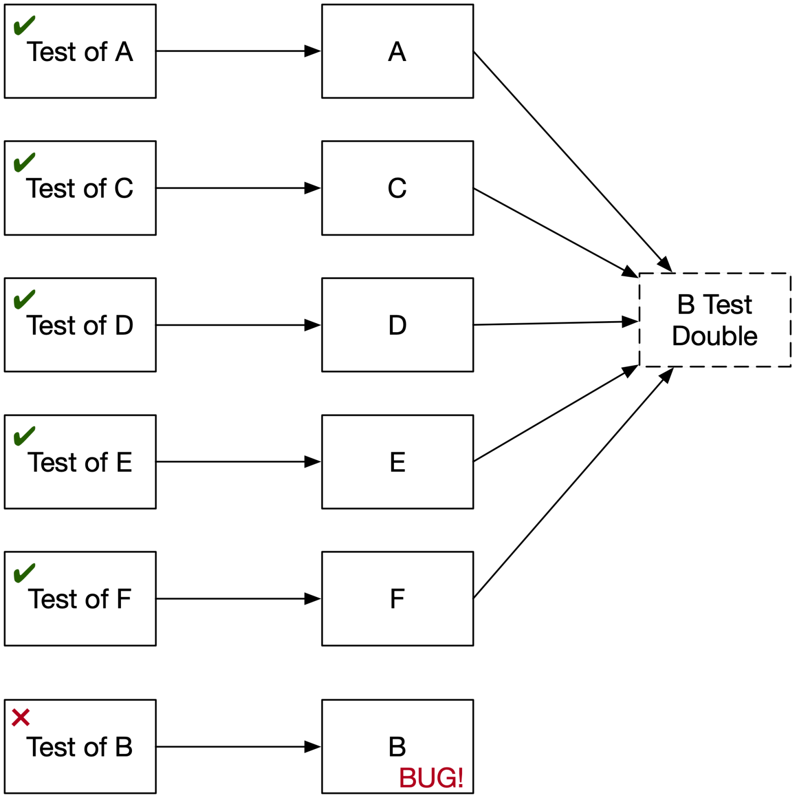
A test of module A integrated with a test double, and a test of buggy module B separately. Only the test of B fails.

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.

6. Project Setup

In this chapter, we'll set up our project and our development process. We won't write any app-specific functionality yet. Instead, what we'll do is:

- Make a list of our stories to work on
- Get our development machine dependencies installed
- Get React installed and running
- Get linting and auto-formatting working
- Get E2E and component tests running
- Get our tests running on CI
- Get our application automatically deploying to a hosting service

That's a lot, but we want to do it all before we write our first line of production code so that we have support in place for our agile process. And it won't take too long because we'll use powerful tools to help us get there. Once that's all in place, we'll be ready to start implementing our application's first feature in the next chapter.

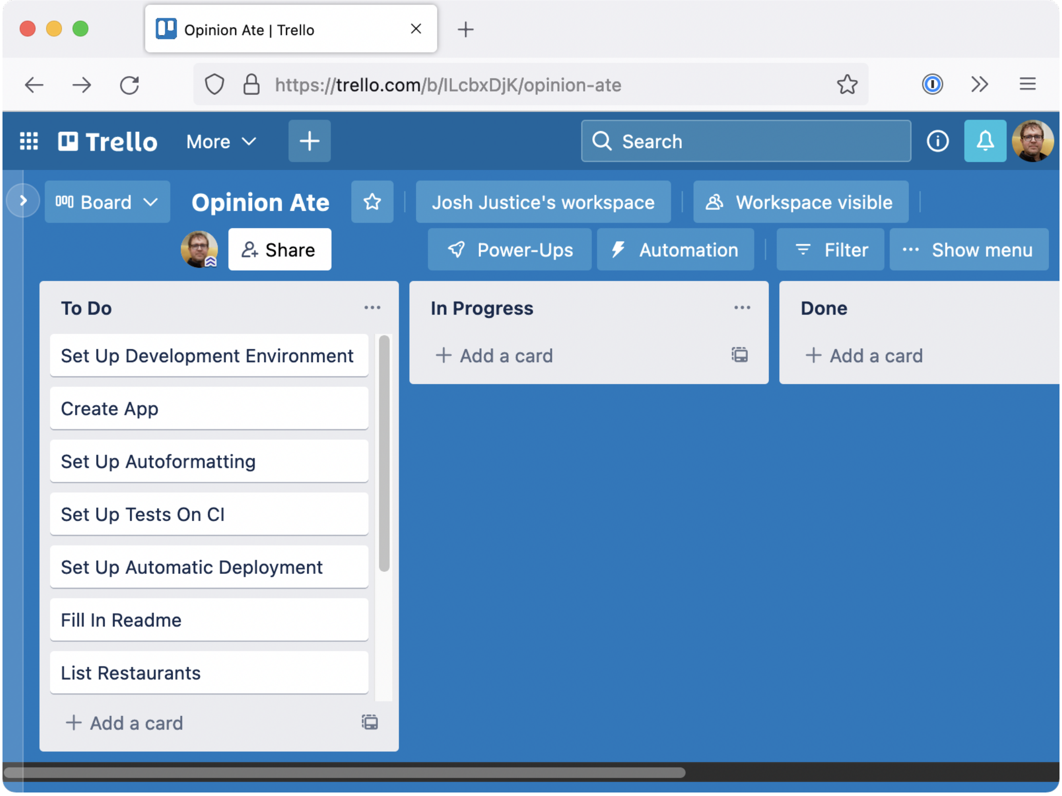
Making a List of Stories

Agile developers often use the term "stories" to refer to units of work. This term emphasizes the fact that agile developers stay focused on delivering units of work that are useful to the user. Rather than "building out the data model" (which would not make for a very compelling tale!), a story would be more likely to be something like "users can see a list of restaurants they've entered."

Before we start our work, we need a way to track it. In this exercise it'll just be you working by yourself, so you could just keep a to-do list on paper or a to-do app if you like. But when doing agile development in a team environment, it's a good idea to have a shared tracker. Trello is a great flexible tool that is useful for tracking work.

If you want to try out Trello as part of this exercise, go to https://trello.com and sign up for a free account. Create a new board and name it "Opinion Ate". Create three lists: "To Do", "In Progress", and "Done". In the "To Do" column, add a card for each of the following stories:

- Set Up Development Environment
- Create App
- Set Up Auto-Formatting
- Set Up Tests on CI
- Set Up Automatic Deployment
- Fill In Readme
- List Restaurants
- Style App with Material Design
- Show Loading and Error States
- Add Restaurants



To Do

- Set Up Development Environment
- Create App
- Set Up Autoformatting
- Set Up Tests On CI
- Set Up Automatic Deployment
- Fill In Readme
- List Restaurants

In Progress

Done

Trello board with stories

If you'd rather not use Trello, make a list of the above stories in your project management tool or todo list app of your choice.

Setting Up Development Environment

Our first task is "Set Up Development Environment". In Trello, drag that card to the "In Progress" column.

Here are the tools we'll need to install:

- Git
- Node
- Yarn
- An editor

Git

Version control is essential for all developers, and if you're an agile developer it gives you even more benefits. You'll be taking a lot of small steps, and you need to be able to track them to make sure they aren't lost. Although we won't get into it in this book, focused and well-explained commit messages are essential for communicating to your teammates as well. Git[15] is probably the most popular version control tool right now. We'll use GitHub for source control as well as for pull requests and CI functionality.

As of the time of writing, the latest version of git is 2.37.3. You can run `git --version` to check what version you have installed. But since we'll only be using the most basic git features, using an older version is unlikely to cause problems.

Node

We'll be using Create React App as our build tool. Like most front-end build tools, it runs on top of Node.js[16], so you'll need Node installed locally.

Node's "LTS" (long-term support) version is recommended for most users. As of the time of writing, the LTS version is 16.17.1. You can run `node -v` to check what version you have installed.

[15] https://git-scm.com
[16] https://nodejs.org

Yarn

Yarn 1.x[17] is a client for `npm`, the Node package manager service. `npm` has its own client, but I find Yarn to be faster, more predictable, and more reliable. I would recommend Yarn for any JavaScript project.

The latest major version of Yarn is 2.x. It is a significant architectural change, and many Node ecosystem tools don't yet support it. This exercise uses the previous major version instead, Yarn 1.x. If you install Yarn via the instructions linked above, Yarn 1.x is what you'll have available. The latest version as of time of writing is 1.22.19. You can run `yarn -v` to check what version you have installed.

If you'd prefer to use `npm`, you can still follow this exercise, you'll just need to replace any `yarn` commands with the equivalent `npm` commands.

An Editor

There is a wide variety of editors that are good for React development. One of the most popular free ones is Visual Studio Code[18].

Once you have git, Node, Yarn, and an editor set up, you're done setting up your development environment. Drag the "Set Up Development Environment" story to the "Done" column in Trello.

Creating the App

Our next story is "Create App"; drag it to "In Progress".

Create a new React app:

```
$ yarn create react-app --scripts-version 5.0.1 opinion-ate
```

The `--scripts-version` option should ensure you get the same version of the `react-scripts` package as this book, to minimize the chance of incompatibilities.

Create React App will start the installation process. When it completes, your application will be created and ready to use.

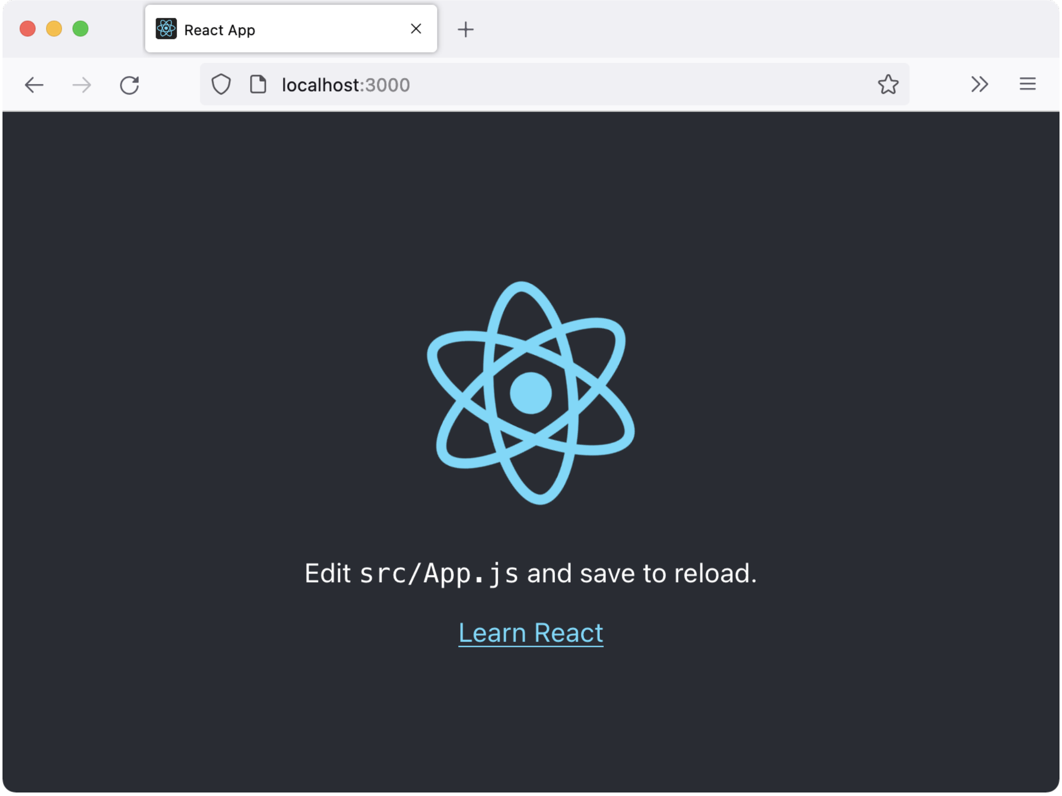
Open `package.json` and note the scripts we have available:

[17] https://classic.yarnpkg.com

[18] https://code.visualstudio.com

- `start` to run the app locally
- `build` to create a release build of the app
- `test` to run unit tests, including React component tests
- `eject` for if we ever want to move away from Create React App

Let's try it out. Run `yarn start`. Your app should automatically open in your default browser, with a spinning React logo.



Edit `src/App.js` and save to reload.

Learn React

React app intro screen

In the console you'll see output like the following:

```
Compiled successfully!

You can now view opinion-ate in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.126:3000

Note that the development build is not optimized.
To create a production build, use yarn build.

webpack compiled successfully
```

With this, we can move our “Create App” task in Trello to “Done”.

Setting Up Auto-Formatting

Our next story is “Set Up Auto-Formatting”; drag it to “In Progress”.

Create React App includes a built-in ESLint config to check your code for issues while it runs. We can extend this config by adding Prettier auto-formatting and support for the Cypress test files we’ll be adding.

Add this rather lengthy list of packages:

```
$ yarn add --dev \
     eslint-config-prettier@8.5.0 \
     eslint-plugin-cypress@2.12.1 \
     eslint-plugin-prettier@4.2.1 \
     prettier@2.7.1
```

Then open `package.json` and update the `"eslintConfig"` key to match the following:

```
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest",
    "prettier"
  ],
  "plugins": [
    "prettier",
    "cypress"
  ],
  "env": {
    "cypress/globals": true
  },
  "rules": {
    "prettier/prettier": "warn"
  }
},
```

Next, let’s tweak the rules Prettier will use for auto-formatting our code.

Add another entry to `package.json` called `"prettier"` and add the following:

```
"prettier": {
  "arrowParens": "avoid",
  "bracketSpacing": false,
  "singleQuote": true,
  "trailingComma": "all"
},
```

These options configure Prettier with some helpful options to match common JavaScript practice. It also makes git diffs a bit simpler to read, which is helpful not only for this book but also for code reviews in your own projects.

Now, set up ESLint integration with your editor. For example, Visual Studio Code has the ESLint extension[19].

After enabling this integration, you may need to restart your editor to make sure it picks up the latest config changes. Then open `src/App.test.js`. Notice the warning on line 1 inside the curly brackets: Prettier is suggesting removing the spaces inside the curly braces. If you’ve

[19] https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

enabled auto-formatting on save, which I recommend, when you save the file those spaces will be removed automatically.

To make sure our app builds correctly, we need to make sure there are no lint warnings in any of the default JavaScript files created in our app. Create React App will let the app run locally with warnings, which is helpful to not slow down your work. But when you try a production build it will fail if there are any lint warnings, which is good to make sure we've handed any linting issues.

To help find and correct any lint issues in our app, let's set up an npm script to run the lint command. Add the following to `package.json`:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "lint": "eslint src",
  "eject": "react-scripts eject"
}
```

This command will run ESLint on all JavaScript files in our `src/` directory.

Let's try it. Run `yarn lint`. You should see warnings something like the following:

```
$ yarn lint
yarn run v1.22.19
$ eslint src

/Users/josh/apps/opinion-ate/src/App.test.js
  1:9  warning  Replace `·render,·screen·` with `render,·screen`  prettier/pret\
tier

/Users/josh/apps/opinion-ate/src/index.js
  11:22  warning  Insert `,`  prettier/prettier

/Users/josh/apps/opinion-ate/src/reportWebVitals.js
  3:33  warning  Replace `·getCLS,·getFID,·getFCP,·getLCP,·getTTFB·` with
  `getCLS,·getFID,·getFCP,·getLCP,·getTTFB`  prettier/prettier

□ 3 problems (0 errors, 3 warnings)
```

```
  0 errors and 3 warnings potentially fixable with the `--fix` option.

□  Done in 1.55s.
```

Notice the message that says "3 warnings potentially fixable with the `--fix` option". ESLint's auto-fix functionality ran earlier when we saved our file in the editor. The `--fix` flag is how you can use this auto-fix functionality from the command line.

Run `yarn lint --fix`. If all the warnings you got were auto-fixable, the command should complete without any warnings. Run `git status` and you should see all the files that you got warnings about have changes applied.

Let's go ahead and commit these configuration changes to linting and formatting. Small, focused commits make it easier for other developers to review, and keep us accountable to really understanding what is changing in our code. Create React App initializes our app with a git repo, so we can just add the changes:

```
$ git add .
$ git commit -m "Set up linting and auto-formatting"
```

With this, we can drag "Set Up Auto-Formatting" to "Done".

Running Tests on CI

Next is "Set Up Tests on CI"; drag it to "In Progress".

Create React App automatically sets up an example component test for us. Before we run that test on our CI server, let's confirm it works for us locally. Open `src/App.test.js`. Note that it's testing the `App` component. Now run `yarn test`. You may get a note "No tests found related to files changed since last commit." If so, press the "A" key to run all tests.

You should see the following:

```
PASS  src/App.test.js
 ✓ renders learn react link (32ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.816s
Ran all test suites.

Watch Usage: Press w to show more.
```

By default, Jest continues running in the terminal so that tests will automatically rerun as we make changes to our code. For now, press ctrl-C to leave the test runner.

One of the testing libraries Create React App installs by default is `@testing-library/user-event`. As of this writing, there is a version of that library out newer than the one Create React App installs, and that version includes some improvements that will make our tests simpler. Let’s upgrade to that version:

```
$ yarn add --dev "@testing-library/user-event@14.4.3"
```

Commit this change:

```
$ git add .
$ git commit -m "Update User Event library"
```

Now we need to install Cypress for end-to-end tests. Run:

```
$ yarn add --dev cypress@10.9.0
```

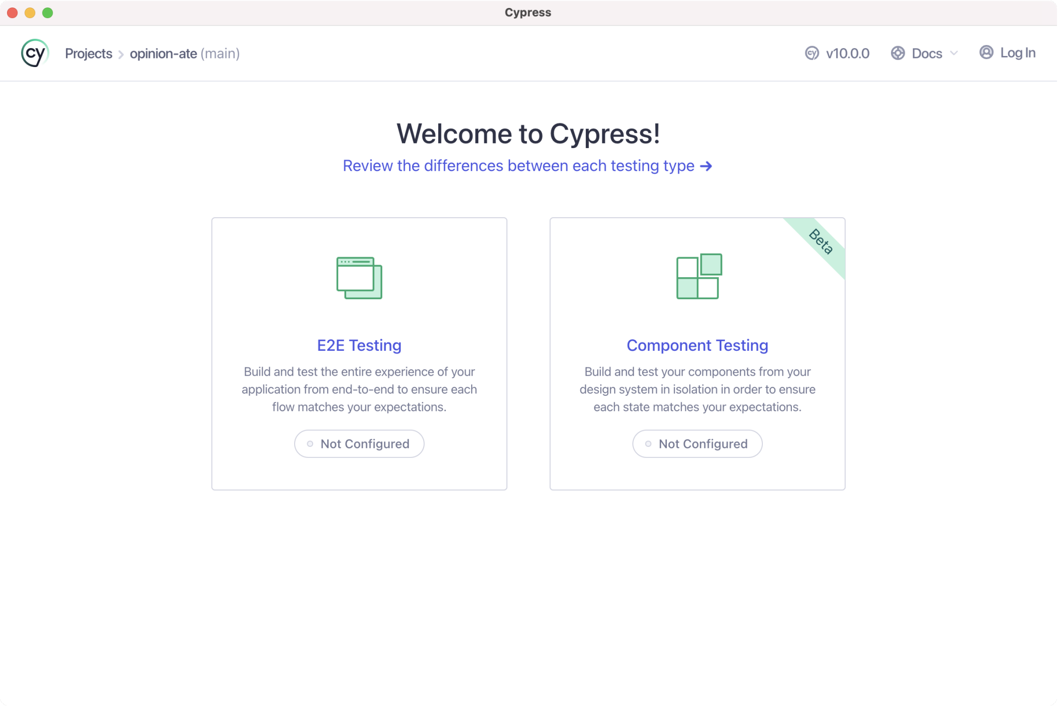
When it completes, in your `package.json`, add a new script:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "cypress": "cypress open",
  "lint": "eslint src",
  "eject": "react-scripts eject"
}
```

Now run that command:

```
$ yarn cypress
```

A Cypress window will open. You'll see a "Welcome to Cypress!" message with an option to configure "E2E Testing" and "Component Testing".



Welcome to Cypress!

Review the differences between each testing type →

E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

Welcome to Cypress window

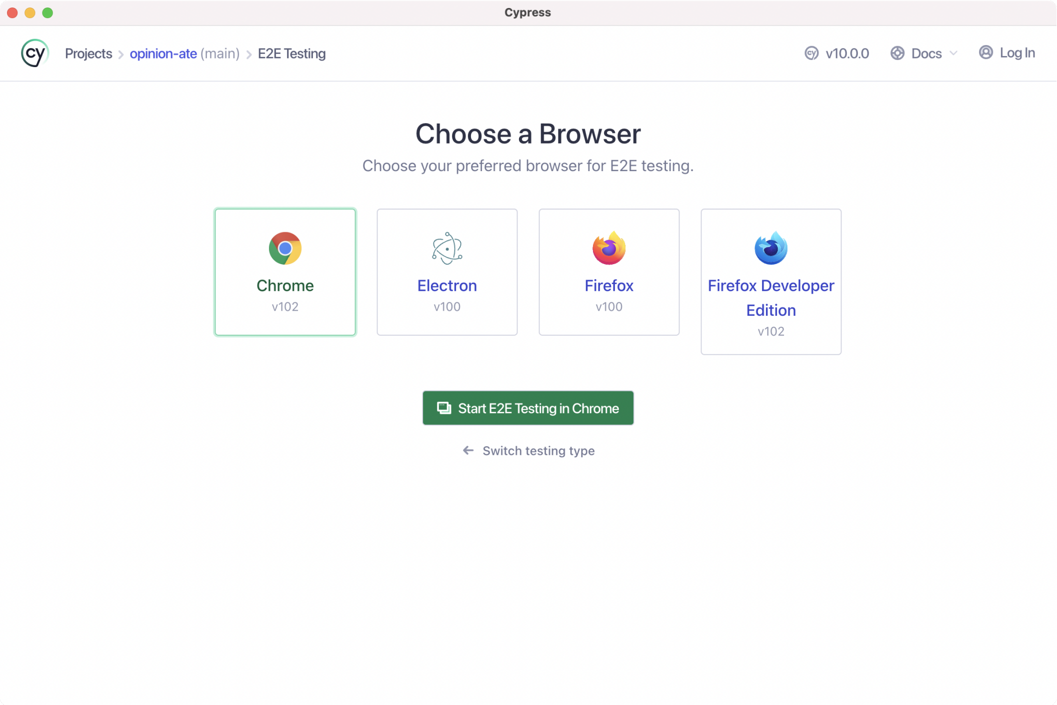
We will only be using Cypress for E2E testing, so click “E2E Testing”. You should see a message “We added the following files to your project” with a few files shown. Quit Cypress.

Now let’s tweak some of the files Cypress created. In the root of your project, there should now be a `cypress.config.js` file. Open it and replace the contents with:

```
const {defineConfig} = require('cypress');

module.exports = defineConfig({
  e2e: {
    baseUrl: 'http://localhost:3000',
  },
});
```

This configures Cypress to interpret all relative URLs relative to the root of our local React app.

Now, run Cypress again with `yarn cypress`. This time, “E2E Testing” will show as “Configured”. Click it. Next you’ll be prompted to “Choose a Browser”.


Cypress
Projects > opinion-ate (main) > E2E Testing
v10.0.0
Docs
Log In

Choose a Browser

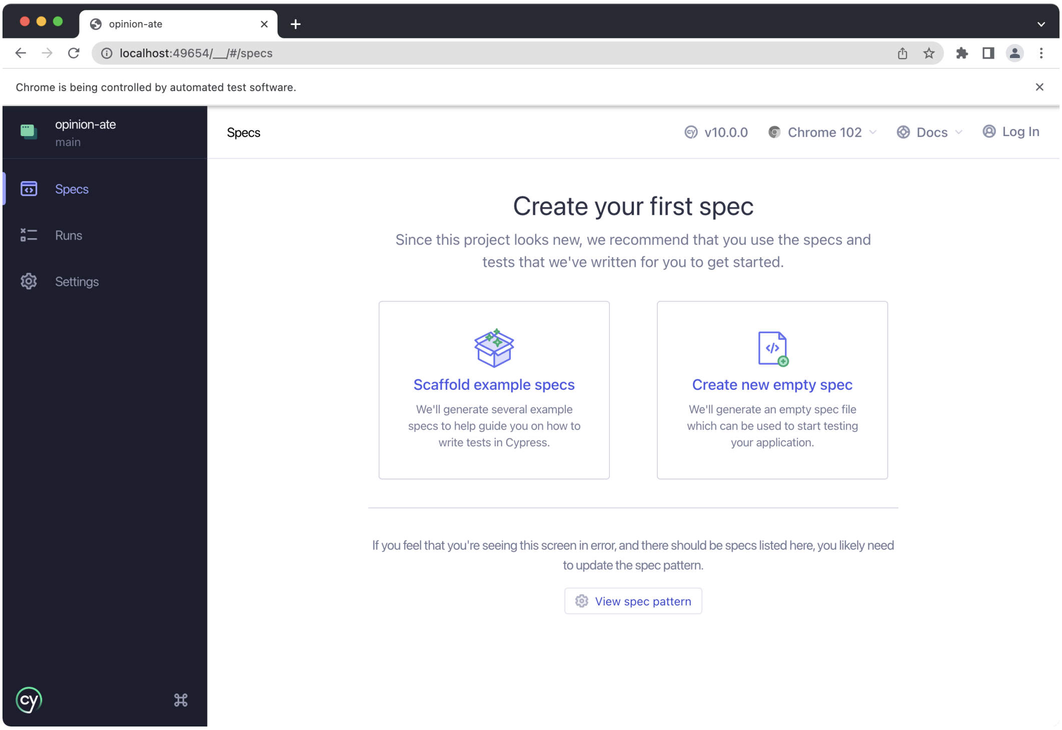
Choose your preferred browser for E2E testing.

Chrome
v102
Electron
v100
Firefox
v100
Firefox Developer Edition
v102
Start E2E Testing in Chrome
Switch testing type

Cypress "Choose a browser" window

For the sake of this exercise let's stick with the default of Chrome. Click the "Start E2E Testing in Chrome" button. The browser you selected will open, and Cypress will be opened in it.

Next you're prompted to "Create your first spec".


opinion-ate
localhost:49654/__/#/specs
Chrome is being controlled by automated test software.
opinion-ate
main
Specs
Runs
Settings
Specs
v10.0.0
Chrome 102
Docs
Log In

Create your first spec

Since this project looks new, we recommend that you use the specs and tests that we've written for you to get started.

Scaffold example specs

We'll generate several example specs to help guide you on how to write tests in Cypress.

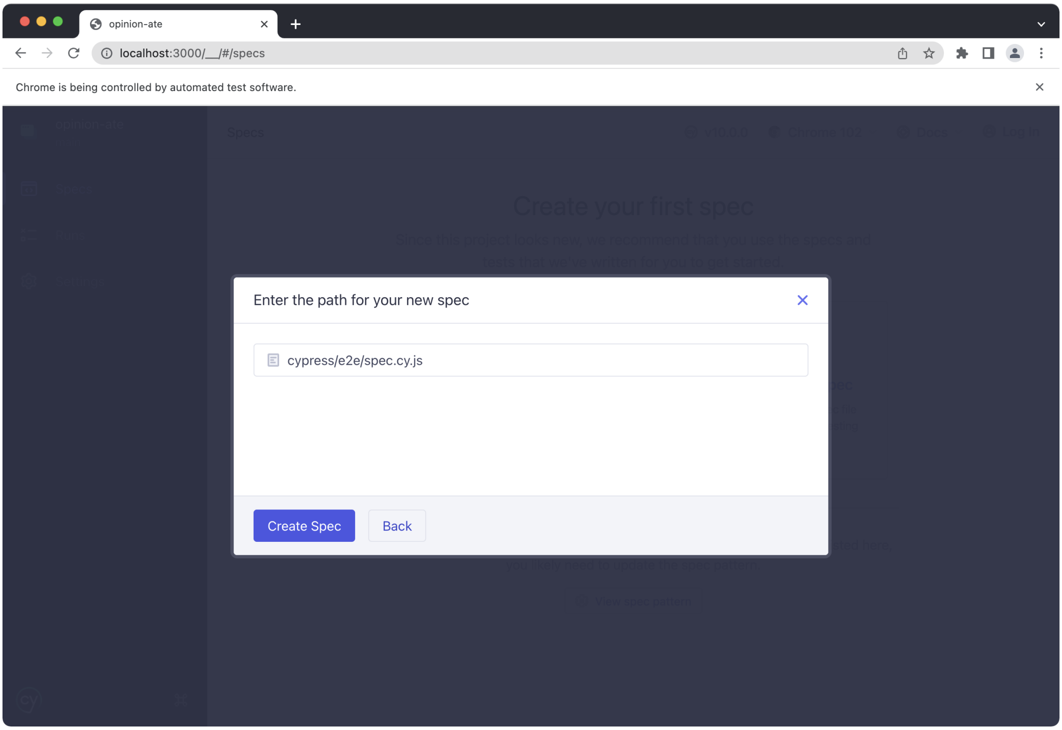
Create new empty spec

We'll generate an empty spec file which can be used to start testing your application.

If you feel that you're seeing this screen in error, and there should be specs listed here, you likely need to update the spec pattern.

Cypress "Choose a browser" window

Click "Create new empty spec". You'll be prompted to enter the file path, with a default of `cypress/e2e/spec.cy.js`.


opinion-ate
localhost:3000/__/#/specs
Chrome is being controlled by automated test software.
Create your first spec
Enter the path for your new spec
cypress/e2e/spec.cy.js
Create Spec
Back

Cypress "Name your first spec" window

Change it to `cypress/e2e/smoke.cy.js`, then click "Create spec". You'll be shown some default content of the spec. Click the X to close the window.

Open the `cypress/e2e/smoke.cy.js` folder Cypress created. Update the contents to the following:

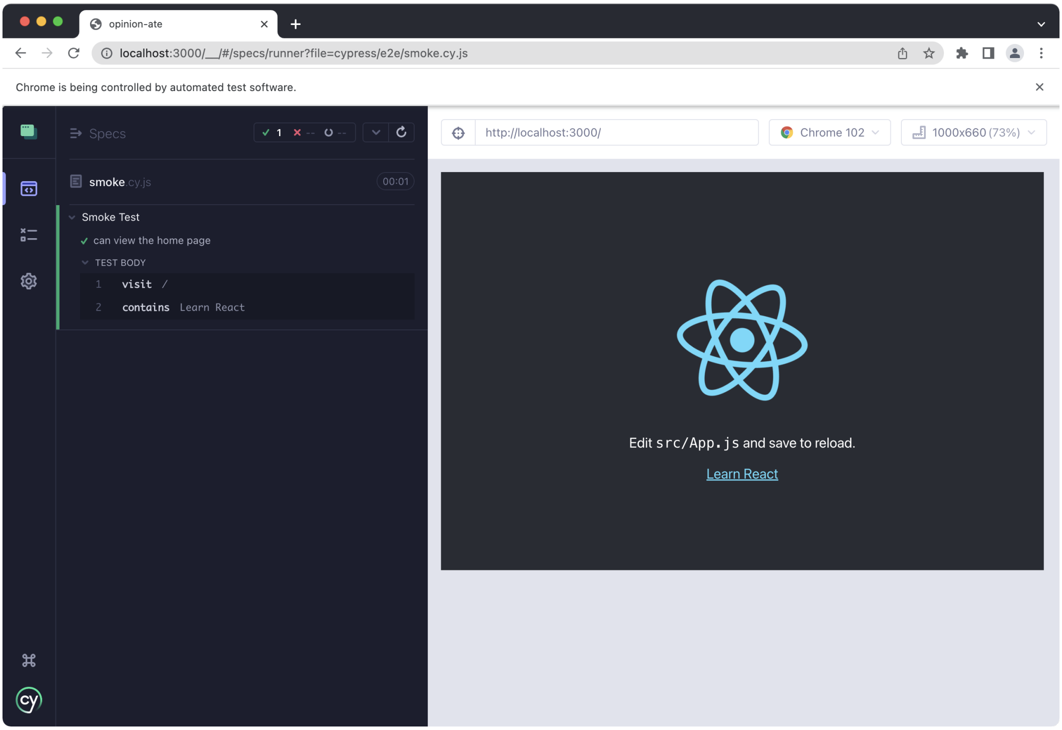
```
describe('Smoke Test', () => {
  it('can view the home page', () => {
    cy.visit('/');
    cy.contains('Learn React');
  });
});
```

This test will load up the root of our app and confirm it can see the text "Learn React" on it.

Save the changes.

Make sure your local React server is still running. If not, in another terminal, run `yarn start`.

Back in the Cypress window, click the smoke test. The test should run and pass.



Chrome is being controlled by automated test software.

Smoke Test
✓ can view the home page
TEST BODY

```
visit /
contains Learn React
```

Edit `src/App.js` and save to reload.

Learn React

Cypress smoke test passing

Add the new Cypress files to git:

```
$ git add .
$ git commit -m "Set up Cypress E2E tests"
```

When Create React App creates our project, it initializes a git repo and adds our code to it. Let's push it up to GitHub. On https://github.com, create a new GitHub repo under your account. Name it something like "opinion-ate".

Next, add it as the `origin` remote and push up the repo:

```
$ git remote add origin https://github.com/your-user-name/your-repo-name.git
$ git push --set-upstream origin main
```

In many development approaches, the next thing we would do would be to start building application functionality. After that, we might release to production. Later we might decide to try to add testing and continuous integration.

But we're going to go the opposite route in this book. We're going to set up CI and deployment from the very start, before we write a line of production code.

There are a number of great CI services with free starter plans, including CircleCI[20] and GitHub Actions[21]. For this exercise we'll go with GitHub Actions: it has a great feature set and is easy to reach for because every GitHub repo is set up for Actions automatically.

So far we've been making commits directly to our `main` branch, but once we start our feature work we'll use git branches instead. To get into the habit of that workflow, let's start now by creating a `ci` branch:

```
$ git checkout -b ci
```

In your project, create a `.github` folder, then a `workflows` folder under that. Then, inside `.github/workflows/`, create a file named `test.yml`. Add the following contents:

```
name: Test
on: [push]

jobs:
  test:
    name: Test
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3
      - name: Install Dependencies
        run: yarn install --frozen-lockfile
      - name: Unit Tests
        run: yarn test --watchAll=false
      - name: E2E Tests
        uses: cypress-io/github-action@v4
        with:
          start: yarn start
          wait-on: 'http://localhost:3000'
```

[20] https://circleci.com

[21] https://github.com/features/actions

Here's what's going on in this file:

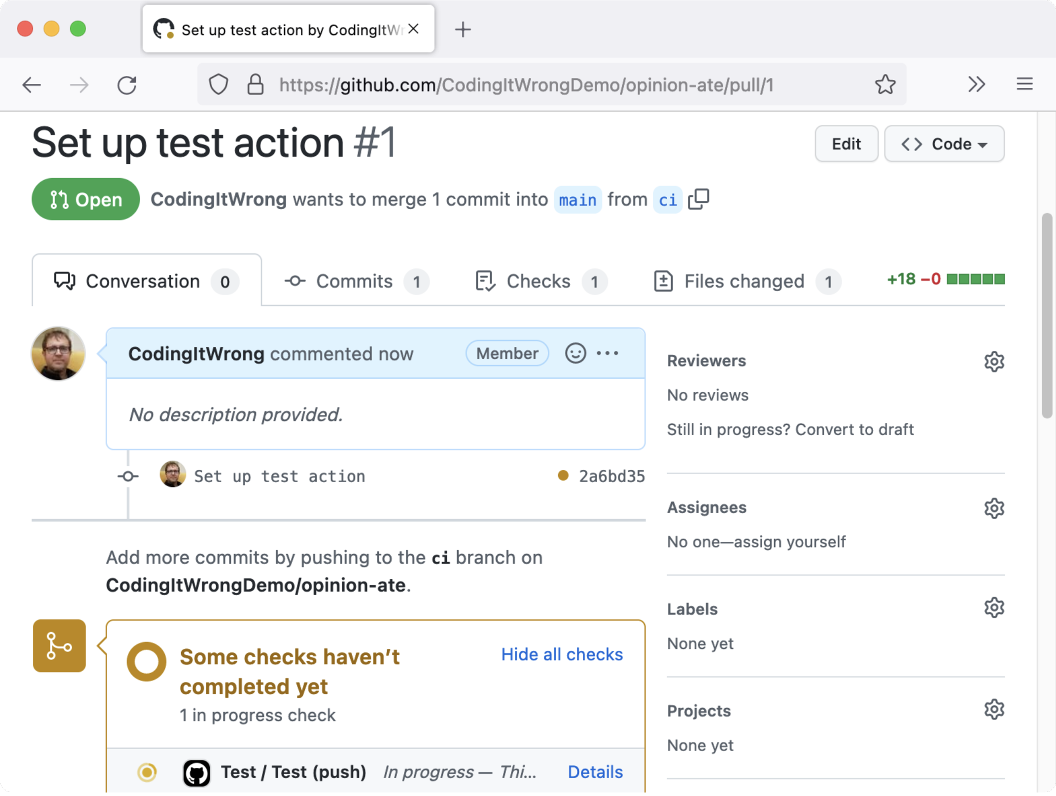
- We name the workflow "Test".
- We configure it to run any time code is pushed to the server. This means both PR branches and merges to the `main` branch will be tested.
- We configure a single job for the workflow, also named "Test".
- We configure it to run on a specific version of the Ubuntu distribution of Linux. You can also run on `ubuntu-latest`, but having a fixed version can prevent unpredictable and difficult-to-troubleshoot incompatibilities between the OS and testing tools.
- Now, we define the series of steps to run for the job. First, we use the GitHub Action `actions/checkout` to check out our code.
- We install our Yarn dependencies. The `--frozen-lockfile` flag is good to use on CI servers; it ensures Yarn will install exactly the versions that are in the lockfile instead of potentially updating them. This ensures that the versions running on CI are identical to what you're running locally.
- We run our unit tests. The `--watchAll=false` flag ensures they won't continue running and watching for file changes; they just run once.
- We run our E2E tests, using a GitHub Action helpfully provided by Cypress. We tell the action to start our React development server and to wait until it can see the running server before beginning the test suite.

Commit this `test.yml` file, then push up your changes to GitHub:

```
$ git add .
$ git commit -m "Set up test action"
$ git push -u origin ci
```

In the console output, GitHub will give you a URL to create a pull request. Open it in the browser, and click "Create pull request".

Underneath the PR description box and your commit, you should see a yellow "Some checks haven't completed yet" warning; this box shows the continuous integration checks that are running on your PR. Right now we only have one check, our GitHub Actions workflow, represented by the line that says "Test / Test (push)".



Set up test action by CodingItW

https://github.com/CodingItWrongDemo/opinion-ate/pull/1

Set up test action #1

Open CodingItWrong wants to merge 1 commit into main from ci

CodingItWrong commented now

No description provided.

Set up test action 2a6bd35

Add more commits by pushing to the **ci** branch on **CodingItWrongDemo/opinion-ate**.

Some checks haven't completed yet

1 in progress check

Test / Test (push) *In progress — Thi…*

Reviewers

No reviews

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

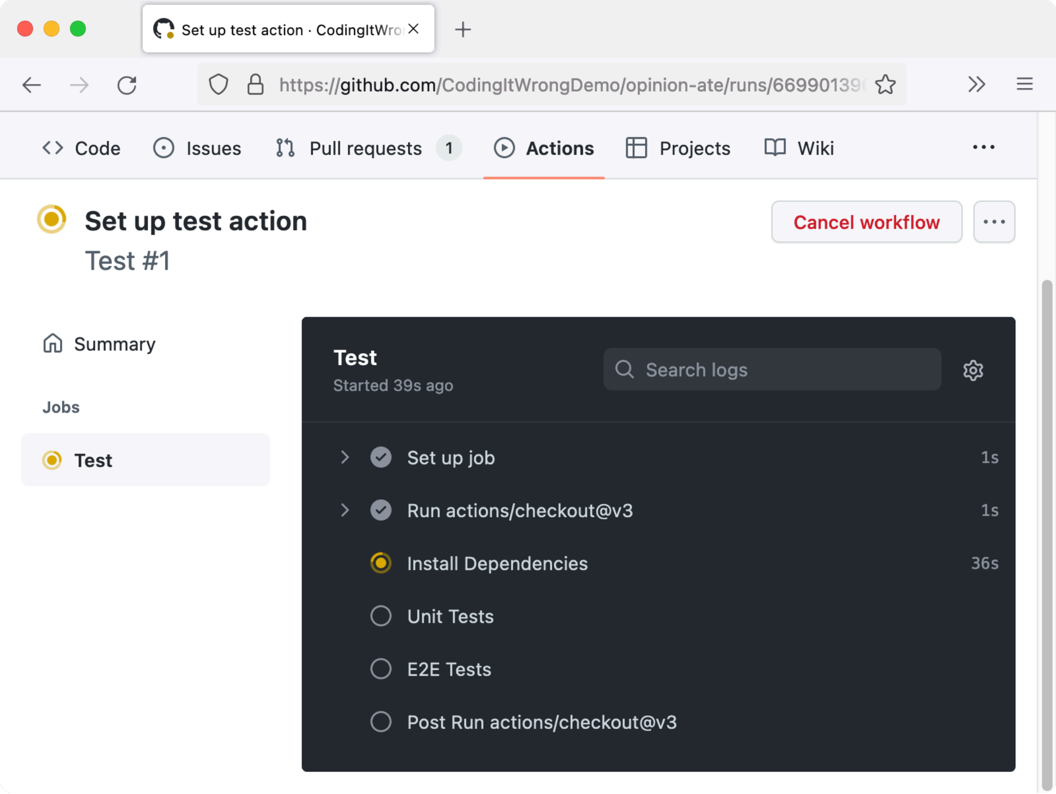
None yet

Projects

None yet

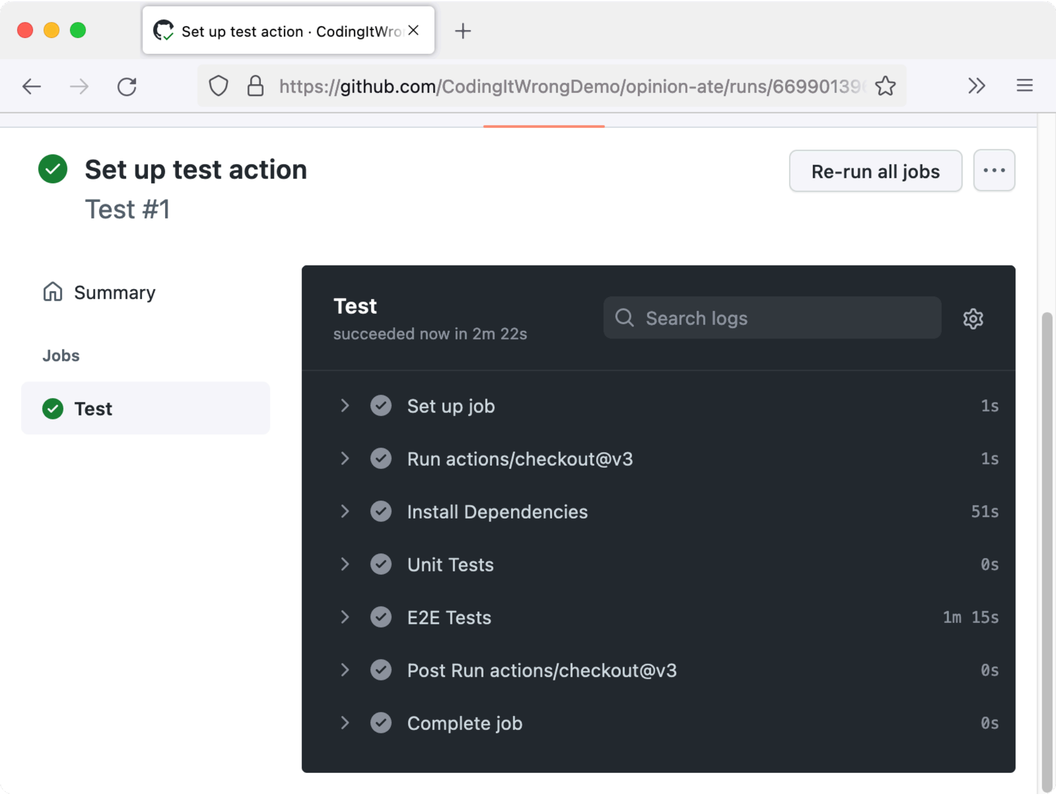
GitHub pull request with incomplete checks

Next to the "Test / Test (push)" line, click the "Details" link. You'll see an outline showing our GitHub Action workflow steps running.



GitHub Action workflow running

When the test run succeeds, the action will be marked as passed.


Set up test action · CodingItWro
https://github.com/CodingItWrongDemo/opinion-ate/runs/66990139
Re-run all jobs
Search logs

Set up test action
Test #1

Summary

Jobs

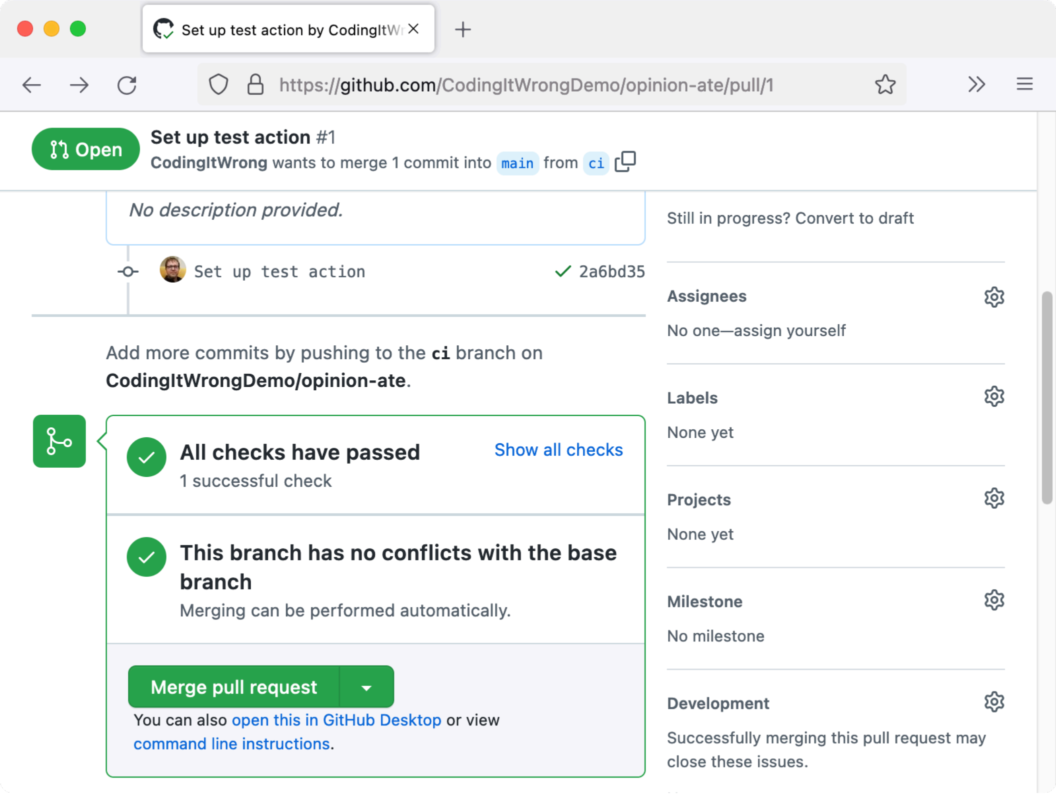
Test

Test
succeeded now in 2m 22s

```
Set up job                          1s
Run actions/checkout@v3             1s
Install Dependencies               51s
Unit Tests                          0s
E2E Tests                       1m 15s
Post Run actions/checkout@v3        0s
Complete job                        0s
```

GitHub Action workflow succeeded

If you go back to the Conversation tab of your PR, in the checks box you'll see a green "All checks have passed" message.



Set up test action by CodingItWrong

https://github.com/CodingItWrongDemo/opinion-ate/pull/1

Open **Set up test action #1**

CodingItWrong wants to merge 1 commit into `main` from `ci`

No description provided.

Set up test action ✓ 2a6bd35

Add more commits by pushing to the **ci** branch on **CodingItWrongDemo/opinion-ate**.

All checks have passed Show all checks

1 successful check

This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also open this in GitHub Desktop or view command line instructions.

Still in progress? Convert to draft

Assignees No one—assign yourself

Labels None yet

Projects None yet

Milestone No milestone

Development Successfully merging this pull request may close these issues.

GitHub pull request with checks passed

Go ahead and click "Merge pull request", then "Confirm merge", then "Delete branch".

In your local project, switch back to `main` and pull down the latest changes that we merged in from the branch:

```
$ git checkout main
$ git pull
```

With this, we can drag the "Set Up Tests on CI" task to "Done" in Trello.

Setting Up Automatic Deployment

Our next task is "Set Up Automatic Deployment"; drag it to "In Progress" in Trello.

Now we’re going to go ahead and deploy our application to production. Yes, even though it doesn’t do anything yet!

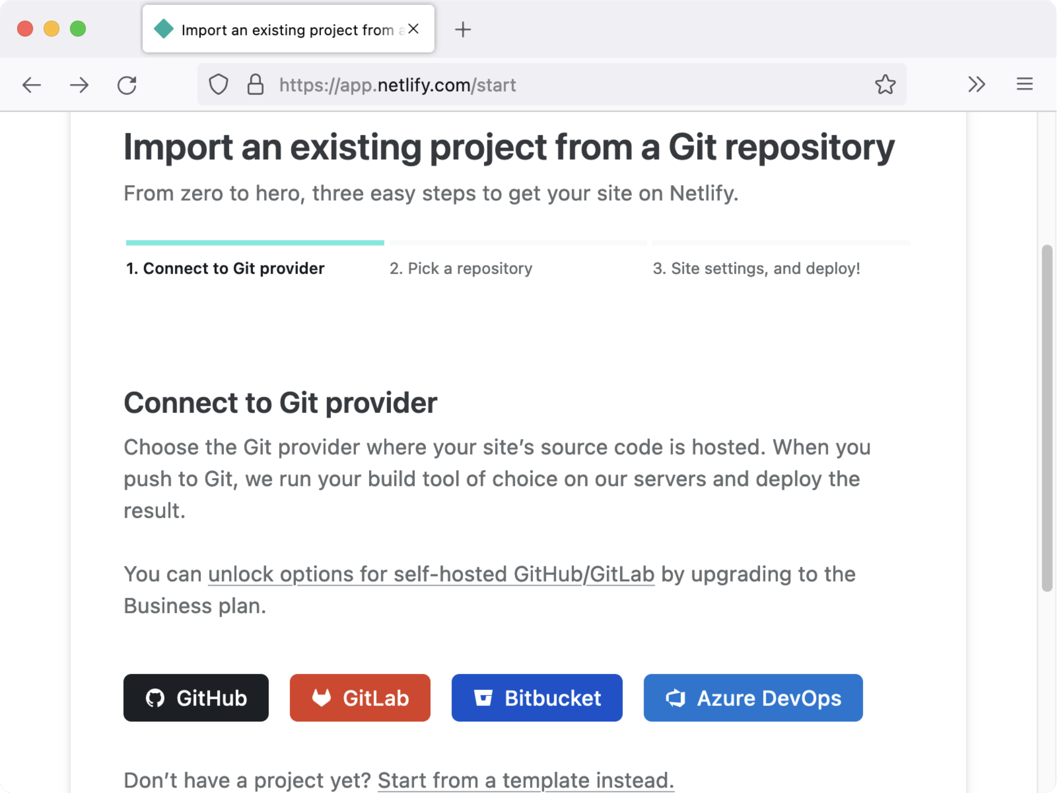
First let’s see how a production build works locally. Run `yarn build`. The files are written to a `build` folder in your project. Open it and see static assets including HTML, JS, and CSS files. Due to the way the file paths work, you can’t just open the HTML file in the browser, but they’ll work when deployed to a server.

There are many ways to deploy front-end apps. One easy way is a service like Netlify[22] that is set up to run your front-end build process for you. Netlify has a free Starter plan for individual users. You don’t need to provide a credit card, but just keep an eye out for emails about approaching your monthly limit of build minutes: if you go over the limit you’ll need to pay for more minutes or your sites will be shut down.

From Netlify’s home page, click the “Sign up” button. Netlify will prompt you with multiple ways you can sign up for an account. Since we will need to give it access to GitHub anyway, it might make sense to sign up with your GitHub account.

Once you’re signed in, click “Add new site”, then “Import an existing project”.

[22] https://www.netlify.com



Import an existing project from a
https://app.netlify.com/start

Import an existing project from a Git repository

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider

2. Pick a repository

3. Site settings, and deploy!

Connect to Git provider

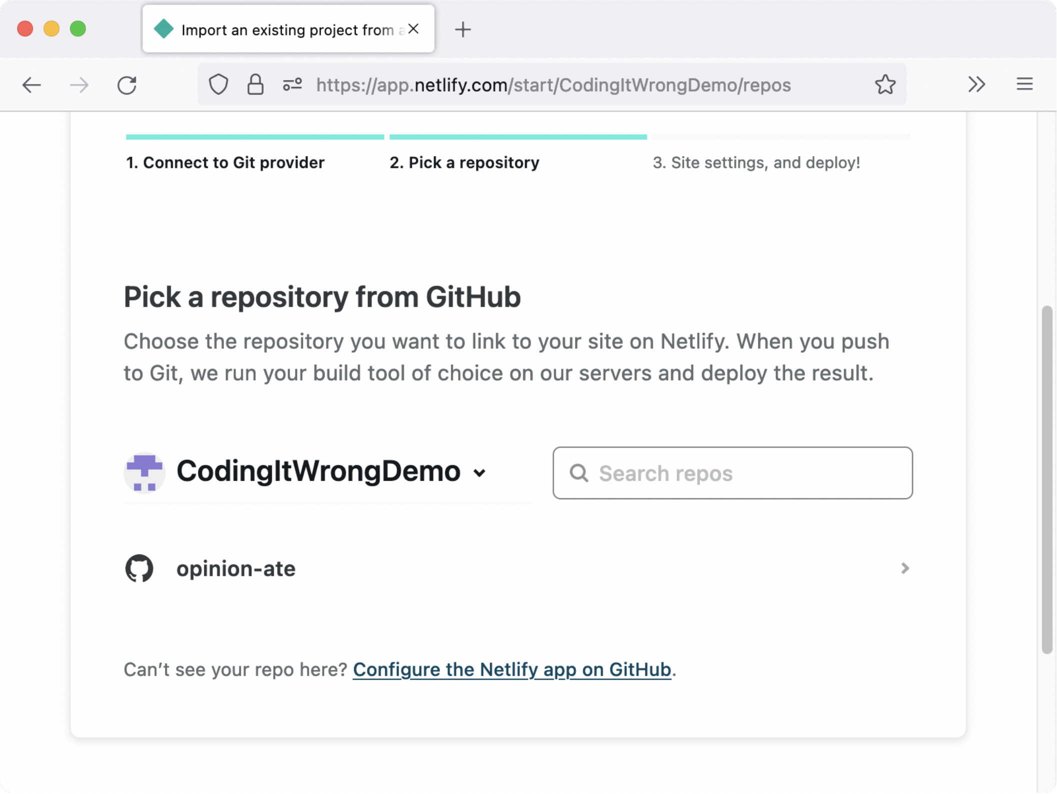
Choose the Git provider where your site's source code is hosted. When you push to Git, we run your build tool of choice on our servers and deploy the result.

You can unlock options for self-hosted GitHub/GitLab by upgrading to the Business plan.

Don't have a project yet? Start from a template instead.

Netlify import project page

Click the “GitHub” button. A list of all your repos will appear.


Import an existing project from a
https://app.netlify.com/start/CodingItWrongDemo/repos
1. Connect to Git provider
2. Pick a repository
3. Site settings, and deploy!

Pick a repository from GitHub

Choose the repository you want to link to your site on Netlify. When you push to Git, we run your build tool of choice on our servers and deploy the result.

CodingItWrongDemo
Search repos
opinion-ate

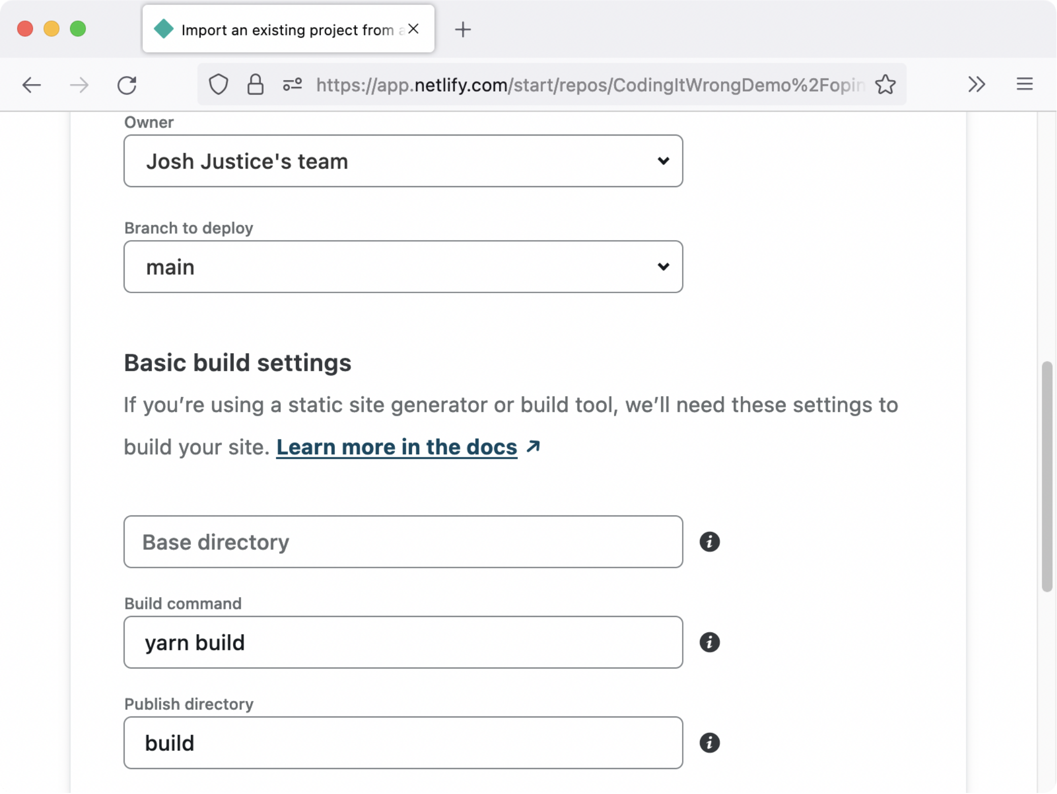
Can't see your repo here? Configure the Netlify app on GitHub.

Netlify pick repository page

Search for your repo and click it.

Next you'll be shown a "Site settings" screen allowing you to configure the build for the project. Use the following settings:

- Leave "Owner" and "Branch to deploy" with their default values.
- Under "Basic build settings", leave "Base directory" empty.
- You should see "Build command" pre-populated with `yarn build`, and `build` for the "Publish directory". Netlify has automatically detected that we're using Create React App and entered the settings for us. This means that Netlify will run that command, then take the files in that directory and deploy them.


Import an existing project from a
https://app.netlify.com/start/repos/CodingItWrongDemo%2Fopin

Owner: Josh Justice's team

Branch to deploy: main

Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site. Learn more in the docs ↗

Base directory:

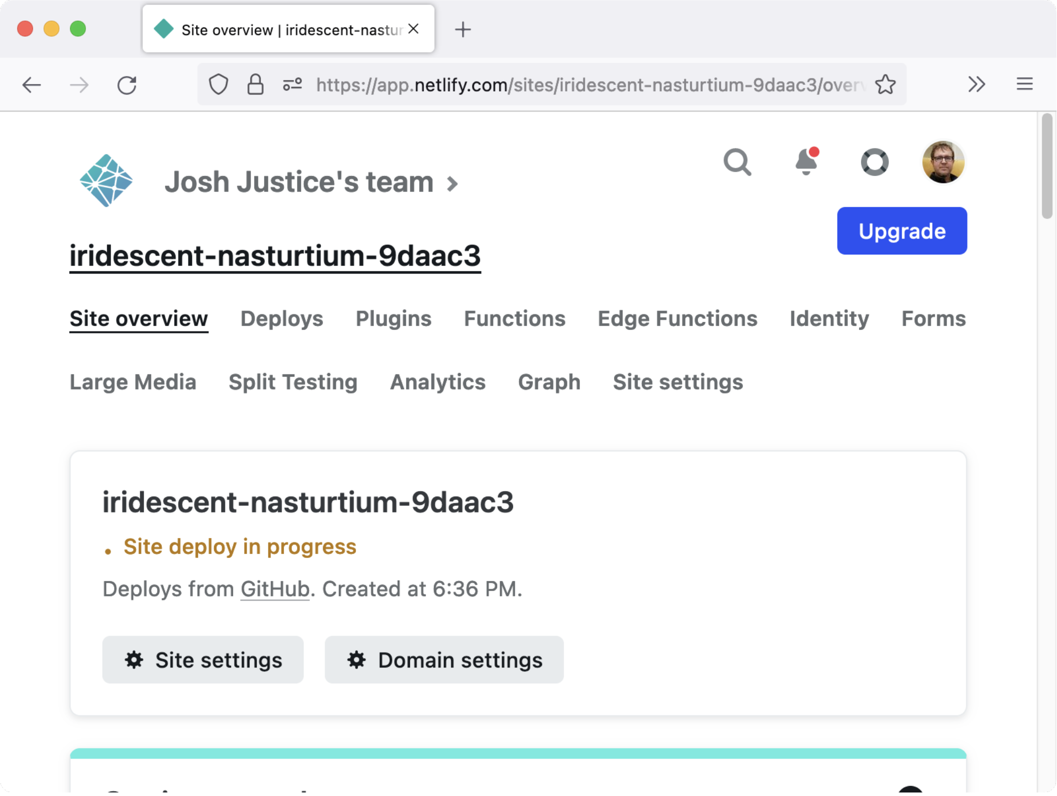
Build command: yarn build

Publish directory: build

Netlify site settings page

Once those values are configured, click "Deploy site".

You will be sent to the Site Overview page for your new site. In orange you'll see "Site deploy in progress".



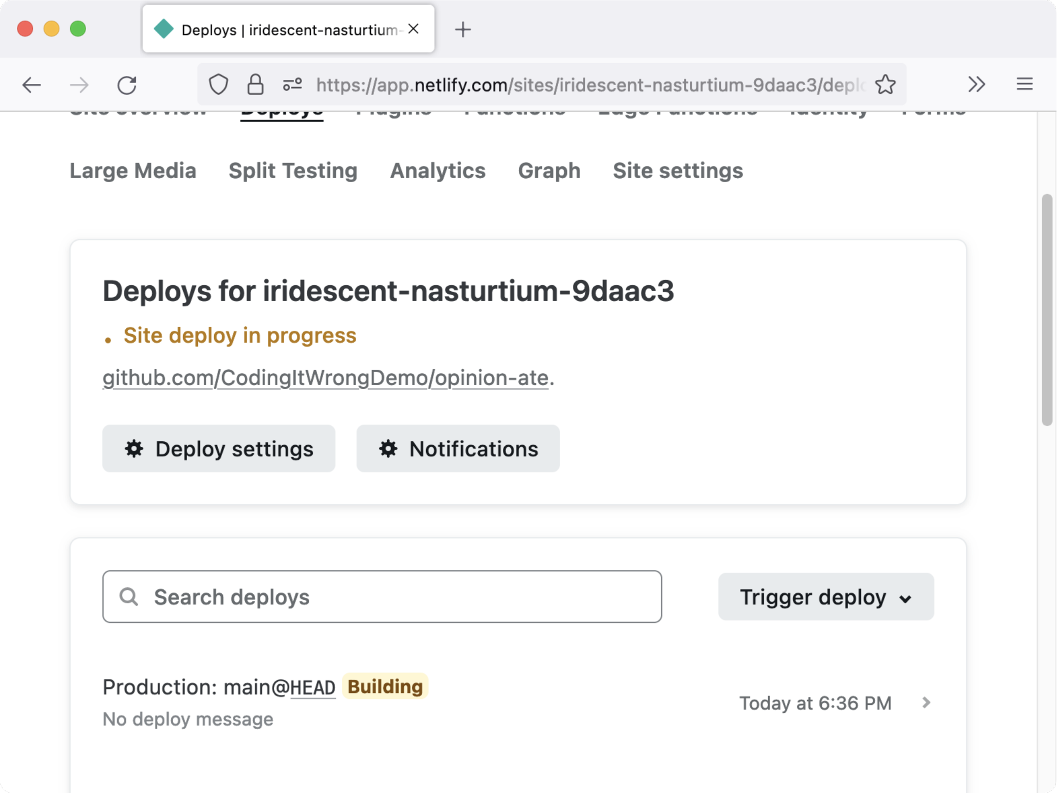
iridescent-nasturtium-9daac3

• Site deploy in progress

Deploys from GitHub. Created at 6:36 PM.

Netlify site deploy in progress

Click the "Site deploy in progress" link and you'll be taken to the Deploys page. In a list at the bottom you'll see "Production: main@HEAD Building".



Deploys | iridescent-nasturtium-
https://app.netlify.com/sites/iridescent-nasturtium-9daac3/deplo
Large Media
Split Testing
Analytics
Graph
Site settings

Deploys for iridescent-nasturtium-9daac3

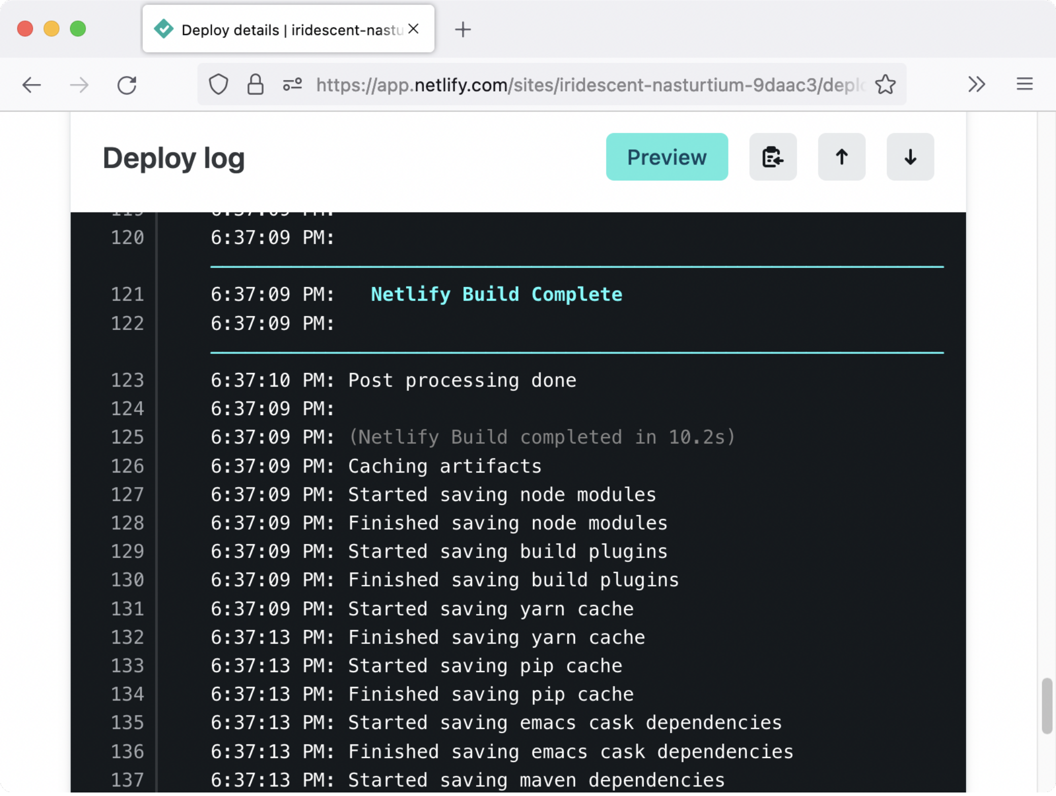
- Site deploy in progress

github.com/CodingItWrongDemo/opinion-ate.

Production: main@HEAD Building
No deploy message
Today at 6:36 PM

Netlify deploys page

Click "Production: main@HEAD Building". You'll see a console log of output as the site is being built. Eventually you'll see a "Netlify Build Complete" message.

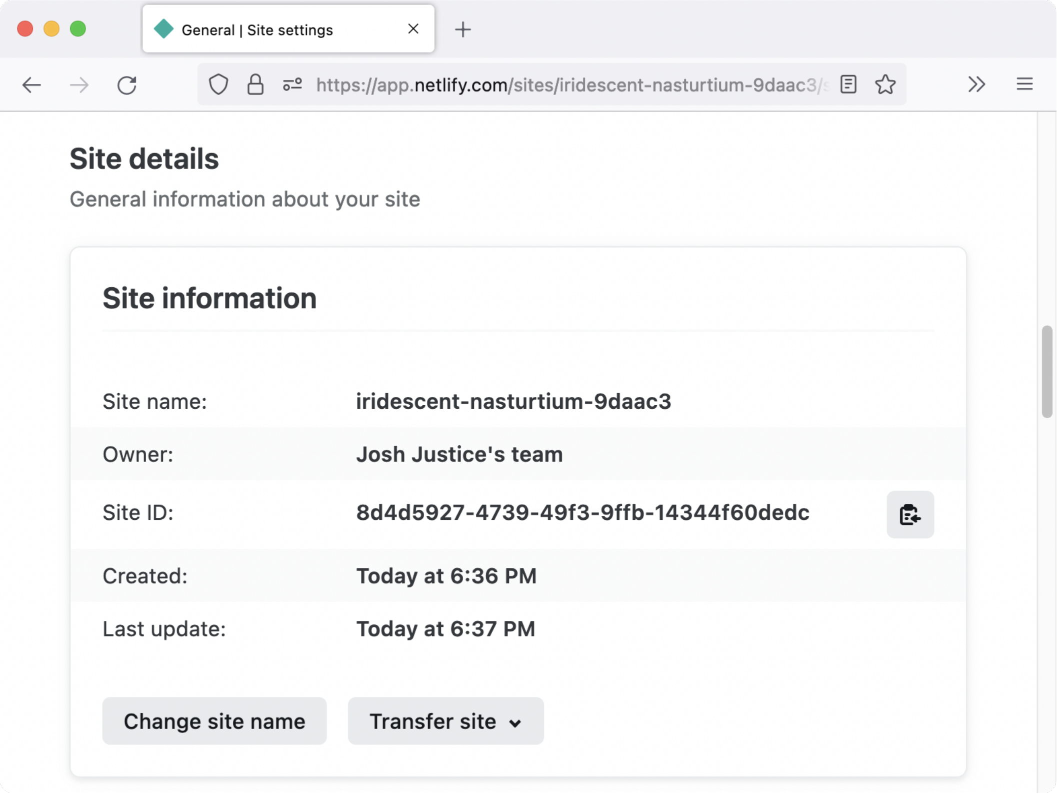

Deploy details | iridescent-nastu
https://app.netlify.com/sites/iridescent-nasturtium-9daac3/depl
Deploy log
Preview

```
6:37:09 PM:
────────────────────────────────────────────
6:37:09 PM:   Netlify Build Complete
6:37:09 PM:
────────────────────────────────────────────
6:37:10 PM: Post processing done
6:37:09 PM:
6:37:09 PM: (Netlify Build completed in 10.2s)
6:37:09 PM: Caching artifacts
6:37:09 PM: Started saving node modules
6:37:09 PM: Finished saving node modules
6:37:09 PM: Started saving build plugins
6:37:09 PM: Finished saving build plugins
6:37:09 PM: Started saving yarn cache
6:37:13 PM: Finished saving yarn cache
6:37:13 PM: Started saving pip cache
6:37:13 PM: Finished saving pip cache
6:37:13 PM: Started saving emacs cask dependencies
6:37:13 PM: Finished saving emacs cask dependencies
6:37:13 PM: Started saving maven dependencies
```

Netlify deploy log page

Scroll to the top of the window and click "< Deploys" to go back to the Deploys tab. If you waited for the deployment to complete, at the top under "Deploys for..." you'll see the URL of your site, with an automatically-assigned set of nonsense words and characters. For example, mine was `iridescent-nasturtium-9daac3`. Click the URL link. You should get the "Learn React" page that is what our application was configured with when we created it.

Now let's rename the Netlify site to be a bit easier to remember. Go back to Netlify, then click the "Site overview" tab, then "Site settings" button. Under "General" > "Site details" > "Site information", click "Change site name".



General | Site settings
https://app.netlify.com/sites/iridescent-nasturtium-9daac3/s

Site details

General information about your site

Site information

Site name: iridescent-nasturtium-9daac3

Owner: Josh Justice's team

Site ID: 8d4d5927-4739-49f3-9ffb-14344f60dedc

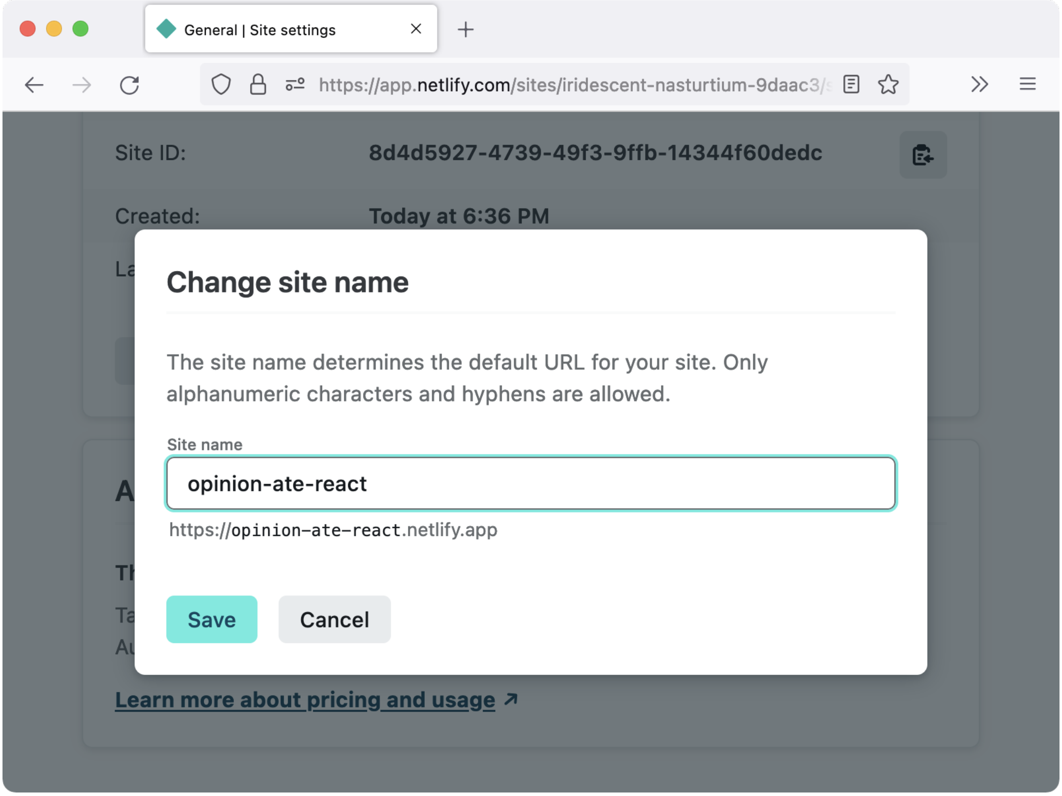
Created: Today at 6:36 PM

Last update: Today at 6:37 PM

Change site name
Transfer site

Netlify site details page

In the dialog that appears, enter any site name you like and click Save. For example, I changed mine to `opinion-ate-react`. You'll need to choose something different since I already took that one!



General | Site settings
https://app.netlify.com/sites/iridescent-nasturtium-9daac3/

Site ID: 8d4d5927-4739-49f3-9ffb-14344f60dedc

Created: Today at 6:36 PM

Change site name

The site name determines the default URL for your site. Only alphanumeric characters and hyphens are allowed.

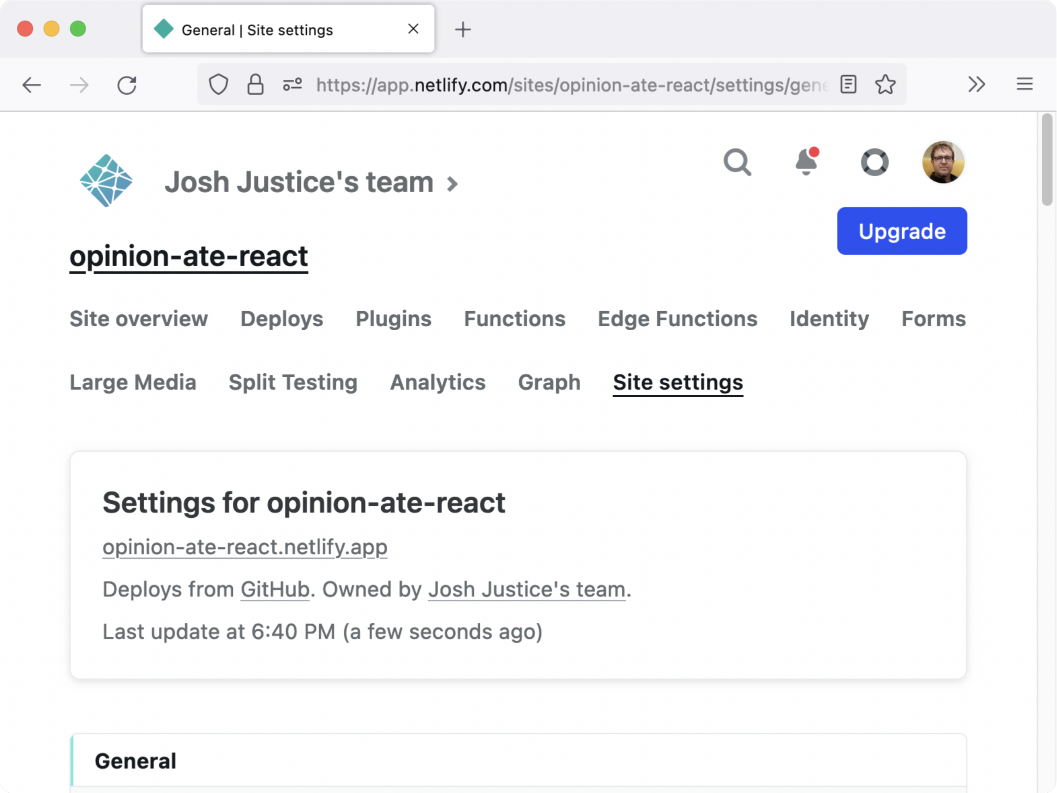
Site name: opinion-ate-react

https://opinion-ate-react.netlify.app

Save
Cancel
Learn more about pricing and usage ↗

Netlify change site name modal

At the top of this screen, under “Settings for”, your updated site URL will appear in gray. Mine is `opinion-ate-react.netlify.app`.



Settings for opinion-ate-react

opinion-ate-react.netlify.app

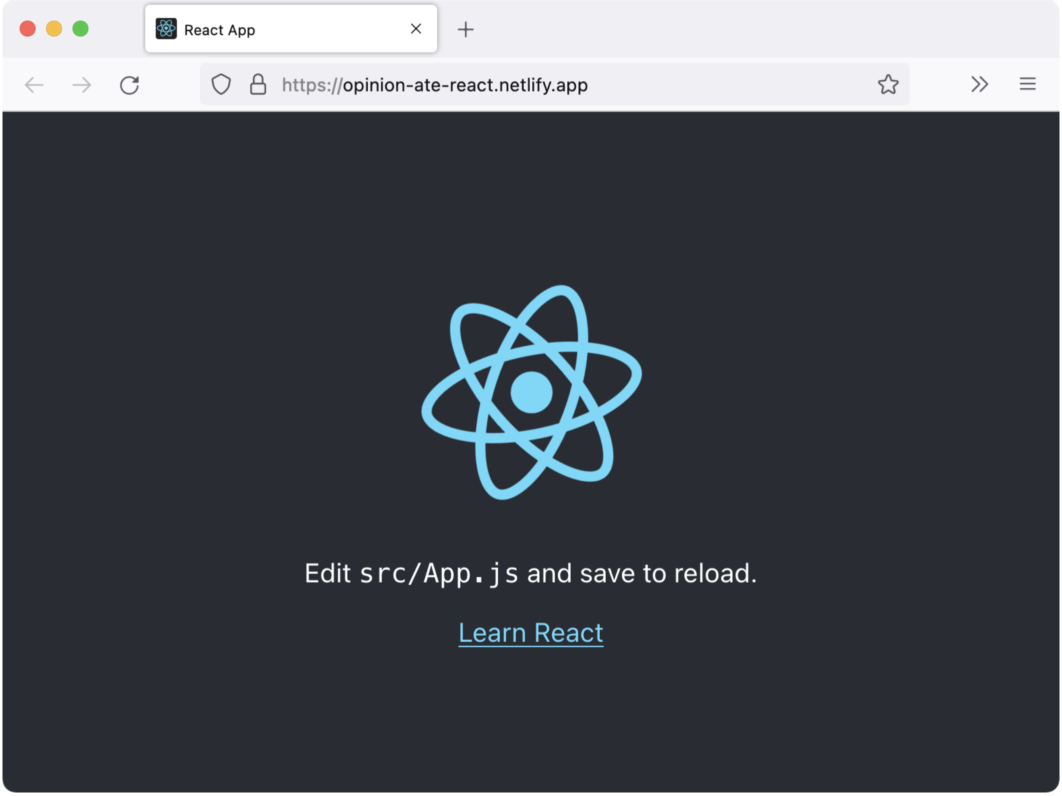
Deploys from GitHub. Owned by Josh Justice's team.

Last update at 6:40 PM (a few seconds ago)

General

Netlify new site name

Click on your new site URL to be taken there, and confirm your site is working at that URL.



Edit `src/App.js` and save to reload.

Learn React

Netlify site with custom name

With this, we can drag “Set Up Automatic Deployment” to “Done” in Trello.

We’ve made great progress so far! Our app is configured to run its tests on CI and automatically deploy to production. We had a little setup work to do, but the defaults provided by Create React App, GitHub Actions, and Netlify went a long way toward simplifying the process.

Filling In the Readme

Our final setup task before we begin developing features is “Fill In Readme”. Drag it to “In Progress” in Trello.

It’s important to write down useful information to help future developers (including yourself) work on the app. Open `README.md` and see what Create React App created for us by default. It’s a straightforward readme that lists the NPM scripts available, as well as links to learn more about Create React App. Any time you're working on a project with a readme that doesn't already have commands listed for how to run, test, and build the app, I would recommend adding them.

Let's add a description of the project and link to production, filling in your Netlify domain:

```
~~# Getting Started with Create React App~~
# opinion-ate

An app for tracking reviews of dishes at different restaurants.

Production: <https://your-netlify-domain.netlify.app>

This project was bootstrapped with
[Create React App](https://github.com/facebook/create-react-app).
```

Something important to note in the readme is that, if another developer uses `npm` instead of `yarn`, it won't check the lockfile and so they won't get the right versions of dependencies. Let's make a note about this:

```
Production: <https://your-netlify-domain.netlify.com/>

Dependencies are locked with a `yarn.lock` file, so please use `yarn` instead
of `npm` to install them.

This project was bootstrapped with
[Create React App](https://github.com/facebook/create-react-app).
```

Commit these README changes to git and push them up to GitHub.

With this, we can drag "Fill In Readme" to "Done" in Trello.

What's Next

Now all our project setup is done and we're ready to work on our first feature! In the next chapter we'll get to the meat of the exercise: building a feature using the outside-in TDD loop.

7. Vertical Slice

In this chapter we'll build our first application feature. We'll follow the practice of outside-in test driven development: write a failing end-to-end test, watch it fail, then build out the functionality with unit tests using multiple inner red-green-refactor cycles. We'll also see the principle of "write the code you wish you had" in action.

In the last chapter we made it through a number of stories in Trello, each involving application setup. Our next story in Trello is the first one that involves building an application feature: "List Restaurants". Drag it to "In Progress".

We chose this story as our first feature story because it allows us to build out a **vertical slice** of our application. It touches all layers of our code: it has a user interface aspect (the list screen), a data layer aspect (where the restaurants are loaded and stored), and an API client aspect (the HTTP request to load the restaurants). It also minimizes other work: we aren't building authentication now, and we aren't handling restaurant loading edge cases yet in this story. The point of a vertical slice is to get something built out that touches all the layers of your application. This ensures that all the layers work together, and it provides a framework for adding in future features.

Setup

We'll do all our work from this feature on a branch. Create a new one:

```
$ git checkout -b list-restaurants
```

To get a clean start, let's delete out the sample content Create React App created with our app. Delete the following files and folders:

- `cypress/e2e/smoke.cy.js`
- `src/App.css`
- `src/App.test.js`
- `src/index.css`
- `src/logo.svg`
- `src/reportWebVitals.js`

Make the following changes to `src/index.js`:

```
import React from 'react';
import ReactDOM from 'react-dom';
~~import './index.css';~~
import App from './App';
~~import reportWebVitals from './reportWebVitals';~~

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
);

~~// If you want to start measuring performance in your app, pass a function~~
~~// to log results (for example: reportWebVitals(console.log))~~
~~// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals~~
~~reportWebVitals();~~
```

Replace the contents of `App.js` with the following minimal content:

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

In `public/index.html`, find the `<title>` tag and see that the page has the default title "React App". Update it:

```
  Learn how to configure a non-root public URL by running `npm run build`.
  -->
~~  <title>React App</title>~~
  <title>Opinion Ate</title>
</head>
```

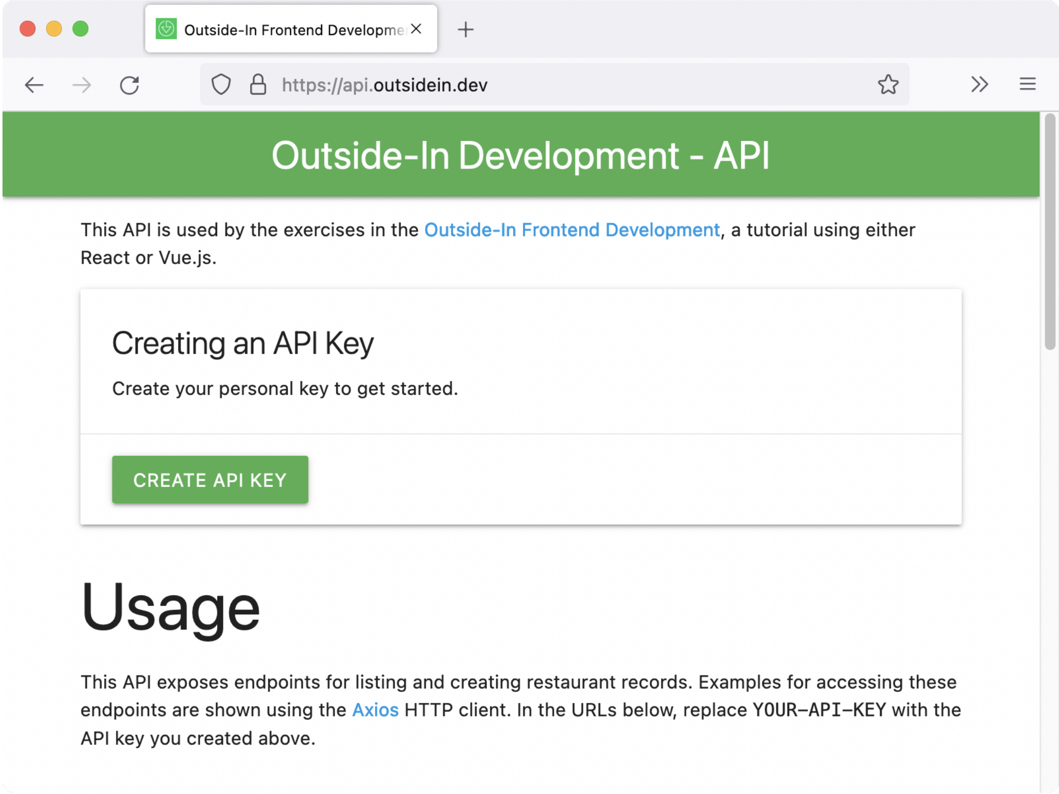
Commit these changes to git:

```
$ git add .
$ git commit -m "Delete sample content"
```

Reviewing the Back-End

This exercise will focus exclusively on our front-end codebase, so we won't be building the corresponding back-end; we'll use one that has already been built. Let's take a look at that back-end and see how we can load our restaurant data from it. It's accessible at https://api.outsidein.dev. Rather than using username-and-password authentication as we might do for a real system, for simplicity we'll just use an API key instead. This will allow you to access your own personal data on the server, so you can edit it without seeing or modifying others' data.

Go to https://api.outsidein.dev in a browser.



Outside-In Development - API

This API is used by the exercises in the Outside-In Frontend Development, a tutorial using either React or Vue.js.

Creating an API Key

Create your personal key to get started.

CREATE API KEY

Usage

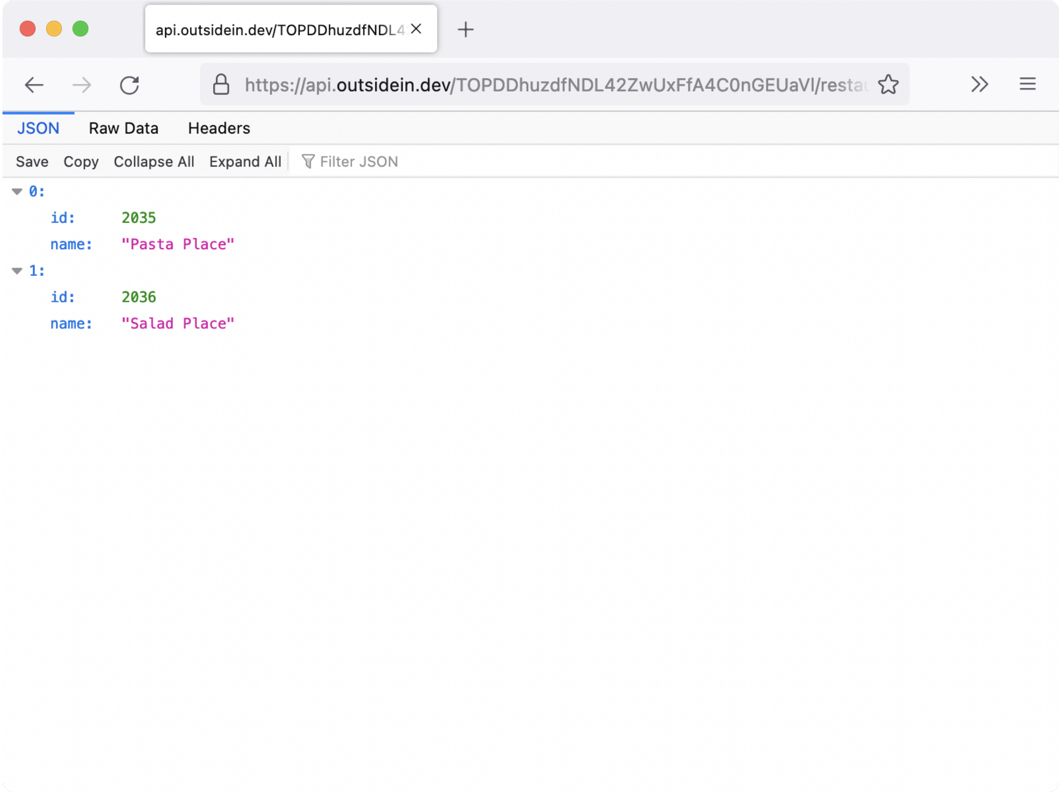
This API exposes endpoints for listing and creating restaurant records. Examples for accessing these endpoints are shown using the Axios HTTP client. In the URLs below, replace YOUR-API-KEY with the API key you created above.

Outside In Development API home page

Click the "Create API Key" button. You'll be given a new API key that is a random sequence of letters and numbers. Copy it and save it someplace safe—you won't be able to get back to it again.

Next, go to `https://api.outsidein.dev/YOUR-API-KEY/restaurants` in a browser, filling in your API key in place of `YOUR-API-KEY`. You should see the following JSON data, including a few default restaurants that were created when your API key was created. It may be formatted differently depending on your browser and extensions, and the IDs will differ:

```
[
  {
    "id": 31,
    "name": "Pasta Place"
  },
  {
    "id": 32,
    "name": "Salad Place"
  }
]
```


api.outsidein.dev/TOPDDhuzdfNDL4
https://api.outsidein.dev/TOPDDhuzdfNDL42ZwUxFfA4C0nGEUaVl/resta
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
▼ 0:
    id:      2035
    name:    "Pasta Place"
▼ 1:
    id:      2036
    name:    "Salad Place"
```

Outside In Development API JSON response displayed in browser

This is the web service endpoint our first story will use; we'll send a `GET` request to retrieve the list of restaurants.

You can also send a `POST` request to that endpoint to create a new restaurant. The API web site provides details on how that works; feel free to try it out if you like.

Now, to start building the front-end.

End-to-End Test

When performing outside-in TDD, our first step is to **create an end-to-end test describing the feature we want users to be able to do**.

In the `cypress/e2e` folder, create a `listing-restaurants.cy.js` file and add the following:

```
describe('Listing Restaurants', () => {
  it('shows restaurants from the server', () => {
    const sushiPlace = 'Sushi Place';
    const pizzaPlace = 'Pizza Place';

    cy.intercept('GET', 'https://api.outsidein.dev/*/restaurants', [
      {id: 1, name: sushiPlace},
      {id: 2, name: pizzaPlace},
    ]);

    cy.visit('/');
    cy.contains(sushiPlace);
    cy.contains(pizzaPlace);
  });
});
```

First, we create variables with a few restaurant names, because we'll use them several times.

Next, we call `cy.intercept()` to stub a back-end request to a URL—in this case, the `https://api.outsidein.dev/YOUR-API-KEY/restaurants` URL we just tested out. (Note that we don't hard-code your API key here; we use a `*` to match any value in the API key spot of the path.) When the app attempts to send a `GET` request to a matching URL, Cypress will intercept that request and return the specified response. We pass the method an array of two restaurant objects. Cypress will convert that array of objects into a JSON string and return that from the stubbed network call.

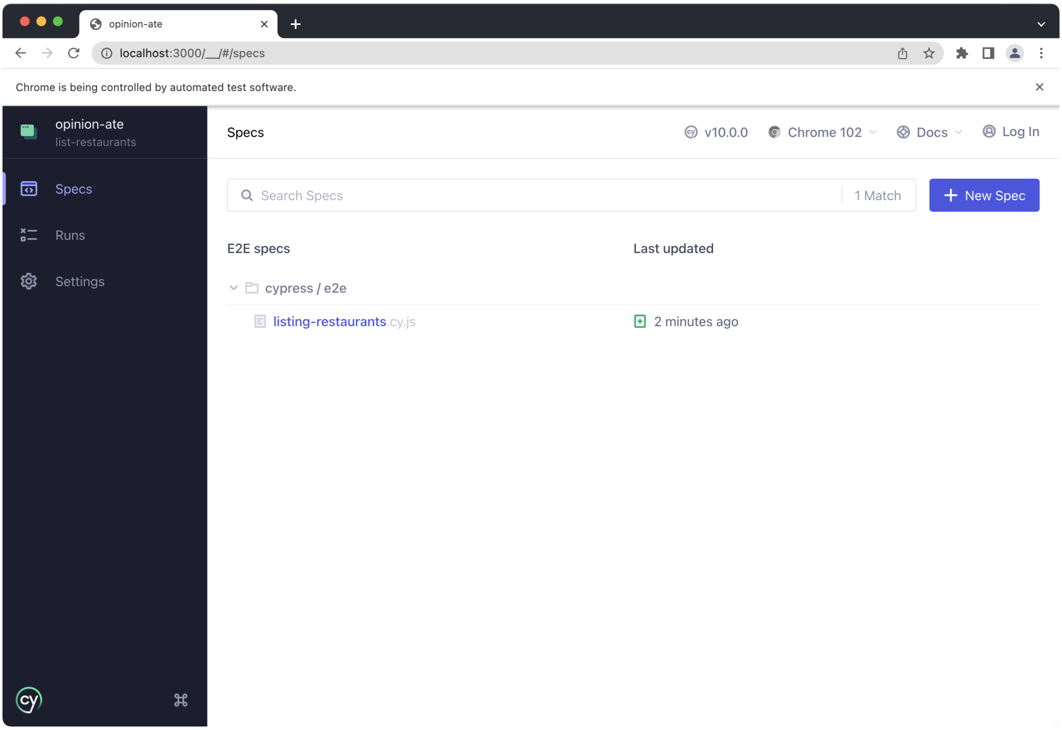
Why stub our back-end request? This allows us to test our front-end application in isolation from the back-end. Doing so means that even if the back-end is slow, unreachable, or its data has changed, our front-end E2E test will continue to pass, confirming that our front-end app behaves as expected. In particular, isolating our front-end from our back-end is important to prevent flaky test failures on CI that would prevent merging our pull requests.

Next, we call `cy.visit()` to load our application in the browser. In the previous chapter we configured the `baseUrl` to `http://localhost:3000`, so we don't need to pass in the full URL, just the path on the server. We pass a path of `/` to load the root of our app.

Finally, we confirm that the page contains both restaurant names. This will ensure that the app successfully retrieved them from the back-end and displayed them on the page.

After we've created our test, the next step in TDD is to **run the test and watch it fail**. This test will fail (be "red") at first because we haven't yet implemented the functionality.

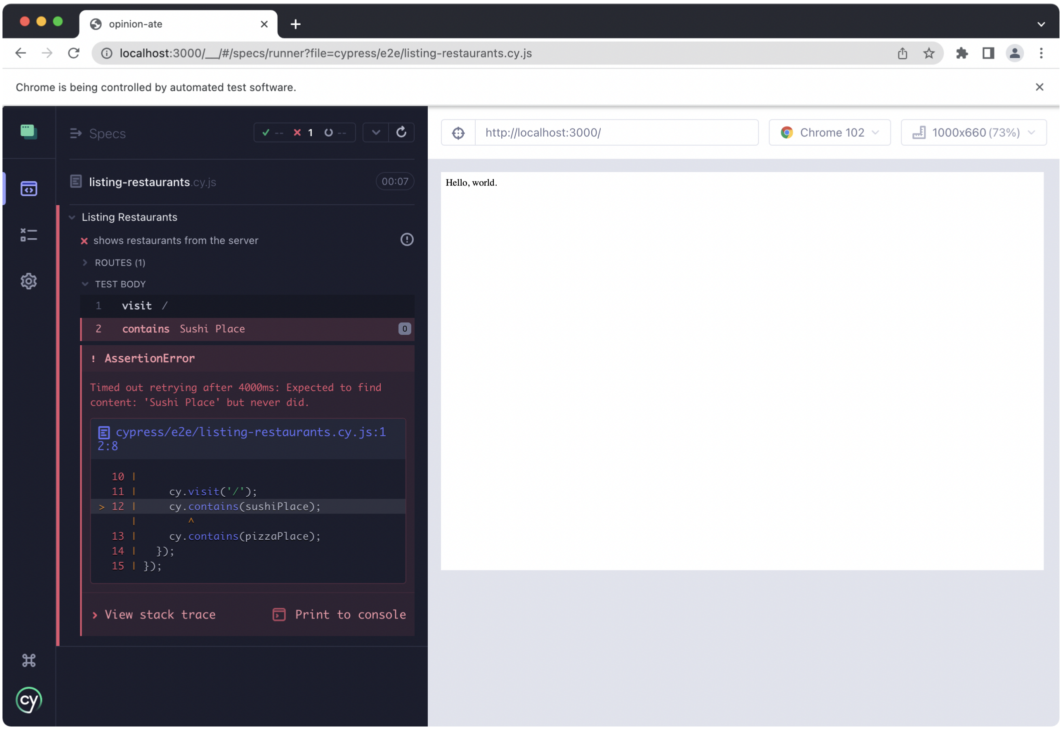
To run our test, run the app with `yarn start` and leave it running, then, in another terminal, run `yarn cypress`. After a few seconds Cypress should open. In Cypress, click "E2E Testing", then "Start E2E Testing In Chrome". Chrome should open, and a list of tests should show, showing only one test, `listing-restaurants.cy.js`.



Chrome is being controlled by automated test software.

Cypress displaying listing-restaurants.cy.js test

Click `listing-restaurants.cy.js` and the test should run. It is able to visit the root of our app, but when it attempts to find “Sushi Place” on the page, it fails.



```
! AssertionError

Timed out retrying after 4000ms: Expected to find
content: 'Sushi Place' but never did.

cypress/e2e/listing-restaurants.cy.js:12:8

  10 |
  11 |     cy.visit('/');
> 12 |     cy.contains(sushiPlace);
     |        ^
  13 |     cy.contains(pizzaPlace);
  14 |   });
  15 | });
```

Cypress test failing because restaurant is not found

Let’s go ahead and commit this E2E test to git. It won’t pass until the end of the branch, but as outside-in TDDers we don’t mind having our current E2E test failing: it tells us that we still have more work to do before our current feature is done. Committing it now allows us to have focused commits going forward for each step of unit functionality we add toward completing the feature.

```
$ git add .
$ git commit -m "Specify app should list restaurants"
```

Now it’s time for us to start writing the code to make the E2E test pass. Let’s think about how we want to structure our code. We’re going to have three layers:

- **Components** that display the user interface,

- **A Redux store** that stores our data and lets us interact with it, and
- **An API client** that allows us to make requests to the back-end.

With outside-in testing, we build the outside first, which in this case is our user interface components. And a common principle is to **write the code you wish you had**. What does that mean in our case? Well, when we created our app, we were given an `App` component. Do we want to put our user interface directly in there? No, it's best to save the `App` component for app-wide concerns such as a title bar that we'll add soon. Instead, it would be great if we had a `RestaurantScreen` component that would contain everything specific to our restaurants. We wish we had it, so let's add it to `App.js`:

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

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

We've referenced the `RestaurantsScreen` component that we wish we had, so our next step is to actually create it. In `src`, create a `components` folder, then inside it create a `RestaurantScreen.js` file. For the moment let's add just enough implementation to render a static header. Add the following to `RestaurantScreen.js`:

```
export default function RestaurantScreen() {
  return (
    <div>
      <h1>Restaurants</h1>
    </div>
  );
}
```

Rerun the E2E test and you should get the same test failure as before. But on the right you should see the "Restaurants" text displayed, and we're beginning to set up the structure of the code that we'll soon use to get past the test failure.

What do we do next? Well, what do we want to do on the `RestaurantScreen`? For the current story, we want to display a restaurant list. But we also have an upcoming story where we want to add new restaurants. Those are two different responsibilities, so instead of adding both responsibilities directly to `RestaurantScreen`, let's plan to eventually have a child component for each. Since our current story is limited to listing restaurants, we'll just create the one child component for the restaurant list.

Let's start by writing the code we wish we had again. In `RestaurantScreen.js`, add the following:

```
import RestaurantList from './RestaurantList';

export default function RestaurantScreen() {
  return (
    <div>
      <h1>Restaurants</h1>
      <RestaurantList />
    </div>
  );
}
```

Now let's implement this `RestaurantList` component we wish we had. Create a `RestaurantList.js` file in `src/components` and again add some minimal static content:

```
export default function RestaurantList() {
  return <div>RestaurantList</div>;
}
```

Now we finally have `RestaurantList` where we'll put logic UI for this story. Let's commit the changes we have. They're a nice, small unit of work: we've added the structure of components that we'll add the behavior to next.

```
$ git add .
$ git commit -m "Add RestaurantScreen and RestaurantList"
```

Stepping Down to a Unit Test

So far our components haven't done much: `App` just renders `RestaurantScreen`, and `RestaurantScreen` just renders `RestaurantList`. This wasn't any significant application *logic*: it was just code *structure*. Because of this, there would have been no real benefit to stepping down to a unit test: unit tests are for driving out *logic*. This is why we wrote this structural code directly under the guidance of the E2E test.

But now that we have `RestaurantList`, we finally have a place where our *logic* can live. The component needs to:

- Request for the restaurants to be loaded
- Display the restaurants once they're returned

Instead of adding this logic directly, let's **step down from the "outside" level of end-to-end tests to an "inside" component test.** This allows us to more precisely specify the behavior of each piece. This unit test will also be helpful in a future story as we add more edge cases to this component. If instead we tried to end-to-end testing every edge case, the tests would be slower to run and harder to understand.

Let's get started on that unit test. In `src/components`, create a file named `RestaurantList.spec.js`. The first bit of functionality we need is to load the restaurants, so that's what we'll test first. We'll start with the structure of the test suite:

```
describe('RestaurantList', () => {
  it('loads restaurants on first render', () => {
  });
});
```

In production our component will be connected to our Redux store, with a dispatch function passed as a prop. We don't want to connect it that way in our unit tests, though. Instead, we'll create a mock function that is passed as a prop in place of the Redux dispatch function; then we can run an expectation on that mock function.

Our component needs to ask our store to load the restaurants, so that means we need a `loadRestaurants` function to pass in:

```
it('loads restaurants on first render', () => {
  const loadRestaurants = jest.fn().mockName('loadRestaurants');
});
```

`jest.fn()` creates a Jest mock function that will allow us to check that it was called. We assign it to the `loadRestaurants` variable. We also chain a call to

`.mockName('loadRestaurants')` to give the function a name that is visible to Jest; this will make our error messages more readable.

Now, we're ready to render our component:

```
import {render} from '@testing-library/react';
import RestaurantList from './RestaurantList';

describe('RestaurantList', () => {
  it('loads restaurants on first render', () => {
    const loadRestaurants = jest.fn().mockName('loadRestaurants');

    render(<RestaurantList loadRestaurants={loadRestaurants} />);
  });
});
```

We import the `RestaurantList` component, then we use React Testing Library's `render()` function to render it. We pass JSX to the `render()` function just like we'd write in production code, and we pass the `loadRestaurants` function as a component prop.

Finally, we're ready to run an expectation to confirm that the component loads restaurants on first render. We check that our mock function was called:

```
it('loads restaurants on first render', () => {
  const loadRestaurants = jest.fn().mockName('loadRestaurants');

  render(<RestaurantList loadRestaurants={loadRestaurants} />);

  expect(loadRestaurants).toHaveBeenCalled();
});
```

Now we're ready to run our unit test and watch it fail. Remember, the TDD loop is red-green-refactor, and the first step is to write a failing test. Once we see it fail in the way we expect, then we'll make it pass.

Run `yarn test` and leave it running for the remainder of this chapter. Jest will run our unit test, and we'll get the following error:

```
FAIL  src/components/RestaurantList.spec.js
  RestaurantList
    □ loads restaurants on first render (19ms)

  □ RestaurantList › loads restaurants on first render

    expect(loadRestaurants).toHaveBeenCalled()


    Expected number of calls: >= 1
    Received number of calls:    0

       8 |     render(<RestaurantList loadRestaurants={loadRestaurants} />);
       9 |
    > 10 |     expect(loadRestaurants).toHaveBeenCalled();
         |                             ^
```

Our test says we expected the `loadRestaurants()` function to have been called at least once, but it wasn't called. This is the error we would expect to see, because we haven't hooked up the first-render functionality yet.

Now that we have a red test, it's time to move on to the second step of the TDD loop and make it green. To call a function once when our component renders, we'll use the React `useEffect` hook. Call the `loadRestaurants` function inside a `useEffect`:

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

~~export default function RestaurantList() {~~
export default function RestaurantList({loadRestaurants}) {
  useEffect(() => {
    loadRestaurants();
  }, [loadRestaurants]);

  return <div>RestaurantList</div>;
}
```

This is the typical `useEffect` pattern to run an effect only once. The dependency array we pass to `useEffect` contains `loadRestaurants`, so the effect will run once each time `loadRestaurants` changes. In our test (and in our real application) `loadRestaurants` will never change, so the effect just runs once when the component first renders.

Save the file and Jest will automatically rerun our unit test. Sure enough, our test is green. We've passed our first unit test! Let's commit the unit test and production code that makes it pass in one commit:

```
$ git add .
$ git commit -m "Load restaurants upon first rendering RestaurantList"
```

This gives us one of the behaviors we want our `RestaurantList` to have: loading the restaurants when it is first rendered. Now it's time to write a test for the second behavior: displaying the restaurants. Let's add another `it()` block inside the `describe()`, with the following contents:

```
it('displays the restaurants', () => {
  const noop = () => {};
  const restaurants = [
    {id: 1, name: 'Sushi Place'},
    {id: 2, name: 'Pizza Place'},
  ];

  render(<RestaurantList loadRestaurants={noop} restaurants={restaurants} />);
});
```

So far this test is pretty similar to our previous one. There are just a few differences:

- Instead of creating a Jest mock function for `loadRestaurants`, we create a `noop` function that does nothing ("no operation"). We still need a function our component can call, but because in this test we don't need to run an expectation on it, a noop function is all we need.
- We define a `restaurants` variable that contains an array of two restaurant objects.

Now, instead of running an expectation that `loadRestaurants` was called, we use the `screen.getByText` function to check what our component renders:

```
~~import {render} from '@testing-library/react';~~
import {render, screen} from '@testing-library/react';
import {RestaurantList} from './RestaurantList';
...
  render(<RestaurantList loadRestaurants={noop} restaurants={restaurants} />);

  expect(screen.getByText('Sushi Place')).toBeInTheDocument();
  expect(screen.getByText('Pizza Place')).toBeInTheDocument();
});
```

`screen.getByText` finds an element containing the passed-in text. We pass in the name of each of the two restaurants. If found, `getByText` returns a reference to the element, and the `.toBeInTheDocument()` matcher will pass. If not found, we will get a test error that the element could not be found.

Why did we split this unit test out from the first one? We are following the testing principle to **check one behavior per test in unit tests**. In our first test we checked the loading behavior, and in this test we are checking the restaurant-display behavior. Having separate test cases for each behavior of the component makes it easy to understand what the component does, and if one of the assertions fails it will be easy to see what went wrong. This principle is sometimes phrased "one assertion per test", but in this test we have two assertions. We're following the spirit of the principle, though, because those two assertions are very closely related: they're checking for two analogous bits of text on the page.

When to check *multiple* behaviors per test

While we're discussing this principle, we should note that it only applies to unit tests, not end-to-end tests. When your feature has multiple behaviors that need to be checked, you should **check *multiple* behaviors per test in end-to-end tests**. Why? End-to-end tests are slower, so the overhead of the repeating the setup steps would significantly slow down our end-to-end test suite as it grows. Also, in outside-in TDD we think of an end-to-end test as defining a single feature in our app, so the test should run through an entire sequence necessary to define the feature, even if it requires multiple assertions.

Our first feature happens to only have one set of closely-related assertions in the end-to-end test. The second feature we'll build in, in a later chapter, will make assertions on multiple behaviors in one end-to-end test.

When we save the file, our test runs, and it’s red, as we expect. We get the following error:

```
  □ RestaurantList › displays the restaurants

    TestingLibraryElementError: Unable to find an element with the text: Sushi
    Place. This could be because the text is broken up by multiple elements. In
    this case, you can provide a function for your text matcher to make your
    matcher more flexible.
```

So no element with the text “Sushi Place” is found. We could hard-code an element with that text, but it’s better to go ahead and pull it from the props:

```
~~export default function RestaurantList({loadRestaurants}) {~~
export default function RestaurantList({loadRestaurants, restaurants}) {
  useEffect(() => {
    loadRestaurants();
  }, [loadRestaurants]);

~~  return <div>RestaurantList</div>;~~
  return (
    <ul>
      {restaurants.map(restaurant => (
        <li key={restaurant.id}>{restaurant.name}</li>
      ))}
    </ul>
  );
};
```

When we save the file, our test of the output passes, but now our first test fails:

```
□ RestaurantList › loads restaurants on first render

  TypeError: Cannot read property 'map' of undefined

     8 |   return (
     9 |     <ul>
  > 10 |       {restaurants.map(restaurant => (
       |                    ^
```

We’re mapping over the restaurants, but in our first test we didn’t pass in a restaurants prop. Let’s update the test to pass in an empty array, since that test doesn’t care if any restaurants are displayed:

```
it('loads restaurants on first render', () => {
  const loadRestaurants = jest.fn().mockName('loadRestaurants');
  const restaurants = [];

~~  render(<RestaurantList loadRestaurants={loadRestaurants} />);~~
  render(
    <RestaurantList
      loadRestaurants={loadRestaurants}
      restaurants={restaurants}
    />,
  );

  expect(loadRestaurants).toHaveBeenCalled();
});
```

Save and now both tests are passing. We’ve now successfully defined both behaviors of our RestaurantList!

Go ahead and commit your changes again. From here on out, we won’t remind you to make small commits as we go, but I would encourage you to do so.

In the TDD cycle, **whenever the tests go green**, **look for opportunities to refactor**, both in production code and test code. Our production code is pretty simple and no improvements jump out at me. But our test code has some duplication. Now that we see which parts are duplicated between the two tests, let’s extract that duplication.

Let’s set up some shared data and a helper function to render the component:

```
describe('RestaurantList', () => {
  const restaurants = [
    {id: 1, name: 'Sushi Place'},
    {id: 2, name: 'Pizza Place'},
  ];
  let loadRestaurants;

  function renderComponent() {
    loadRestaurants = jest.fn().mockName('loadRestaurants');

    render(
      <RestaurantList
        loadRestaurants={loadRestaurants}
        restaurants={restaurants}
      />,
    );
  }

  it('loads restaurants on first render', () => {
```

You may be wondering about why we are setting up `restaurants` and `loadRestaurants` in different ways. We assign `restaurants` once to a `const`, but we declare `loadRestaurants` as a `let` and assign a new mock function to it each time `renderComponent()` is called. The reason for this difference is to make sure we have a fresh starting point for each of our tests. Jest mock functions have internal state that is mutated to track when they are called. If you reuse the same mock function for multiple tests, the mock will report that it was called in a previous test, and you can end up with false positives or negatives. There are ways to reset the state of a mock function, but as a general rule it's safest to recreate any mutable state at the start of each test. Since `restaurants` is not mutated, though, it's safe to create it statically.

You may also wonder why we're creating a mock function and array with data when both are not used in both of our tests. Only one test needs `loadRestaurants` to be a mock, and only the other test needs `restaurants` to have elements. That's true, but our test code would be more complex if we left it as-is with separate setup for each. As written, our `renderComponent()` sets up our component in a good default state, so each test be written to focus only on what it wants to assert.

Now we can replace the duplicated code from the individual tests with a call the `renderComponent()` function:

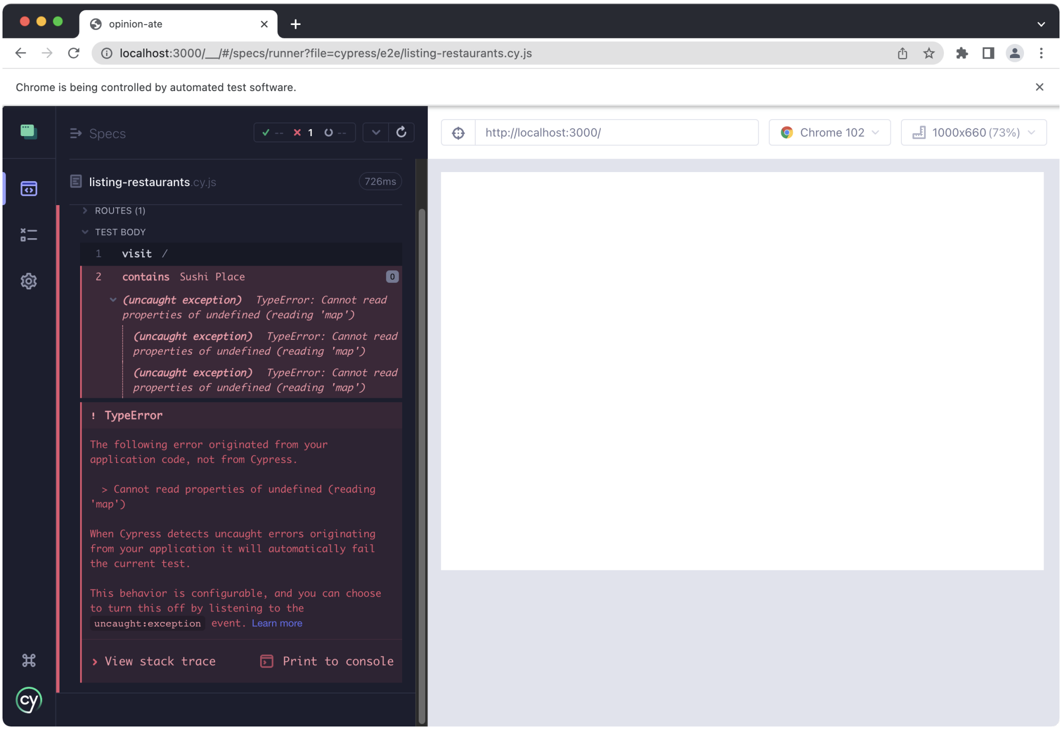
```
it('loads restaurants on first render', () => {
~~  const loadRestaurants = jest.fn().mockName('loadRestaurants');~~
~~  const restaurants = [];~~

~~  render(~~
~~    <RestaurantList~~
~~      loadRestaurants={loadRestaurants}~~
~~      restaurants={restaurants}~~
~~    />,~~
~~  );~~

  renderComponent();
  expect(loadRestaurants).toHaveBeenCalled();
});

it('displays the restaurants', () => {
~~  const noop = () => {};~~
~~  const restaurants = [~~
~~    {id: 1, name: 'Sushi Place'},~~
~~    {id: 2, name: 'Pizza Place'},~~
~~  ];~~

~~  render(<RestaurantList loadRestaurants={noop} restaurants={restaurants} />);~~

  renderComponent();
  expect(screen.getByText('Sushi Place')).toBeInTheDocument();
  expect(screen.getByText('Pizza Place')).toBeInTheDocument();
});
```

Save the file and our tests should still pass. With this, our test blocks are much shorter: all they contain is the expectations. This is good because it keeps our tests focused and very easy to read.

Stepping Back Up

We've now specified the behavior of our `RestaurantList` component, so our unit test is complete. The next step in outside-in TDD is to **step back up to the end-to-end test and see our next failure**. Rerun the test in Chrome and we see:

> TypeError: Cannot read properties of undefined (reading 'map')


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ↻ --
listing-restaurants.cy.js
726ms
ROUTES (1)
TEST BODY
View stack trace
Print to console
http://localhost:3000/
Chrome 102
1000x660 (73%)

```
1  visit  /
2  contains  Sushi Place
   (uncaught exception)  TypeError: Cannot read
   properties of undefined (reading 'map')
     (uncaught exception)  TypeError: Cannot read
     properties of undefined (reading 'map')
     (uncaught exception)  TypeError: Cannot read
     properties of undefined (reading 'map')

! TypeError

The following error originated from your
application code, not from Cypress.

  > Cannot read properties of undefined (reading
'map')

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more
```

Cypress test failing because map is undefined

This should make sense from what we just built: we called the `map` function on the `restaurants` array, but in our application code we aren't yet passing a `restaurants` array. How do we want our component to get that array? We want it to be provided by the Redux store. It's time to write the code we wish we had, and hook our restaurant list up to Redux.

Add `redux` and `react-redux` dependencies:

```
$ yarn add \
    redux@4.2.0 \
    react-redux@8.0.4
```

Next, connect the `RestaurantList` component to the appropriate state. This is what will ultimately fix our Cypress error:

```
import {useEffect} from 'react';
import {connect} from 'react-redux';

~~export default function RestaurantList({loadRestaurants, restaurants}) {~~
function RestaurantList({loadRestaurants, restaurants}) {
  useEffect(() => {
...
}

const mapStateToProps = state => ({
  restaurants: state.restaurants.records,
});

export default connect(mapStateToProps)(RestaurantList);
```

Choosing between `connect()` function and hooks

Note that we are using the React-Redux `connect()` function rather than the newer hooks-based API. Redux maintainer Mark Erikson writes about the trade-offs between the two React-Redux APIs[a], and there is not a strong recommendation to use one or the other.

Because our component testing approach involves passing props to a component that is unaware of Redux, the `connect()` function is a more natural fit. We could accomplish the same by creating a "connected" component that uses React-Redux hooks and passes the props down to the unconnected component, but there are few benefits to that approach over using `connect()`.

[a] https://blog.isquaredsoftware.com/2019/07/blogged-answers-thoughts-on-hooks/

Save `RestaurantList.js`. If you haven't left your Jest tests running, run `yarn test`. Notice that we are now getting a failure:

```
 RestaurantList › loads restaurants on first render

  Could not find "store" in the context of "Connect(RestaurantList)". Either
  wrap the root component in a <Provider>, or pass a custom React context
  provider to <Provider> and the corresponding React context consumer to
  Connect(RestaurantList) in connect options.
```

How do we want to provide the Redux store in our component test? Well, we don't: that integration will be tested as part of our E2E test, not our component test. For the latter, we want to test the component in isolation, to answer the question "*assuming* Redux passes in the correct data to the component, does it behave correctly?"

To arrange our component test to answer this question, we can follow a technique where we export *two* components:

- We export the Redux-connected component as the default export, and
- We export the unconnected component as a named export.

We'll use the connected component in our production code, and we'll use the unconnected component for testing.

To make this change, add the `export` keyword in front of our component function:

```
~~function RestaurantList({loadRestaurants, restaurants}) {~~
**export function RestaurantList({loadRestaurants, restaurants}) {**
  useEffect(() => {
```

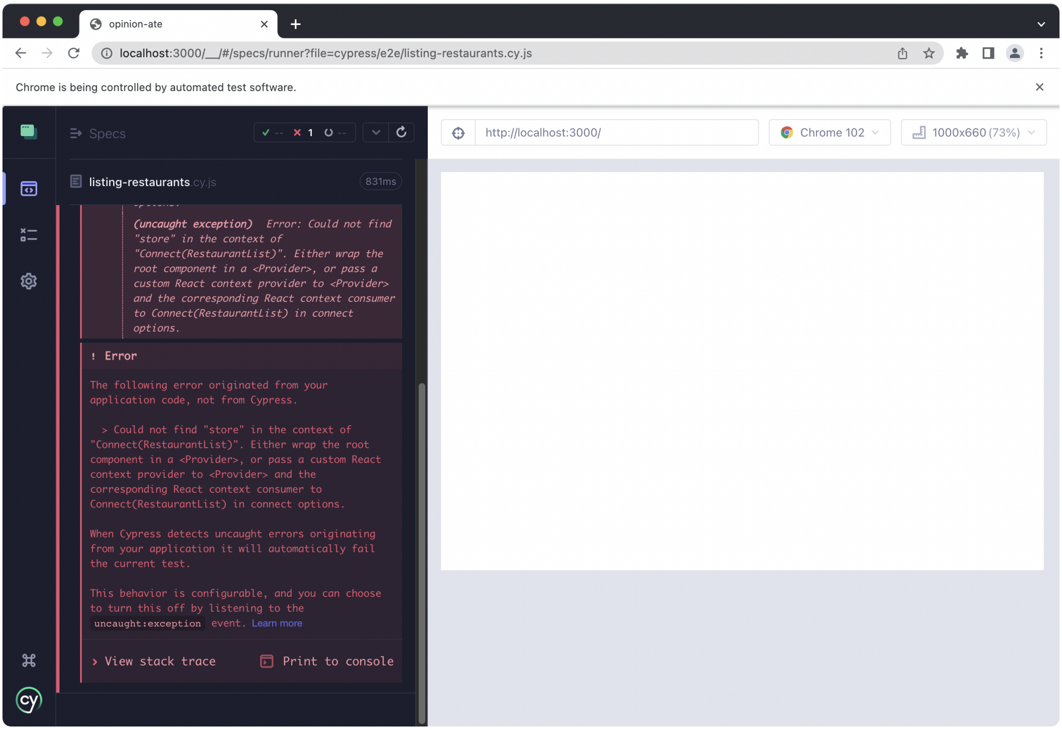
Next, update the test to use the named import:

```
import {render, screen} from '@testing-library/react';
~~import RestaurantList from './RestaurantList';~~
**import {RestaurantList} from './RestaurantList';**

describe('RestaurantList', () => {
```

The tests should automatically rerun and pass again.

If you've used Redux before you know we have more setup steps to do. But let's rerun the E2E test to let it drive us to do so. The error we get is:

> Error: Could not find "store" in the context of "Connect(RestaurantList)".


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ↻ --
listing-restaurants.cy.js
831ms
http://localhost:3000/
Chrome 102
1000x660 (73%)

```
(uncaught exception) Error: Could not find
"store" in the context of
"Connect(RestaurantList)". Either wrap the
root component in a <Provider>, or pass a
custom React context provider to <Provider>
and the corresponding React context consumer
to Connect(RestaurantList) in connect
options.

! Error

The following error originated from your
application code, not from Cypress.

  > Could not find "store" in the context of
"Connect(RestaurantList)". Either wrap the root
component in a <Provider>, or pass a custom React
context provider to <Provider> and the
corresponding React context consumer to
Connect(RestaurantList) in connect options.

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

› View stack trace        Print to console
```

Cypress test failing because there is no provided store

This is the same error we saw in our Jest tests when our component wasn't hooked up to a Redux store. But because this is our real application code now, we *do* want to hook it up to a Redux store. Let's do that now, in `App.js`.

We wish we had a Redux store, so we write the code we wish we had, importing a `./store` module. We pass it to the Redux `Provider` component:

```
import {Provider} from 'react-redux';
import store from './store';
import RestaurantScreen from './components/RestaurantScreen';

export default function App() {
  return (
~~    <div>~~
    <Provider store={store}>
      <RestaurantScreen />
~~    </div>~~
    </Provider>
  );
)
```

Now let's define that store. Under `src/`, create a `store` folder, then an `index.js` inside it. Add the following contents:

```
import {createStore} from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);

export default store;
```

Now we'll need a root reducer. Create a `src/store/reducers.js` file. Add the following contents:

```
import {combineReducers} from 'redux';
import restaurants from './restaurants/reducers';

export default combineReducers({restaurants});
```

Using `combineReducers` is typical in Redux, but right now it's a bit unnecessary because we're combining a *single* reducer into a larger one. The reason we do this is that we're still following the principle of "writing the code we wish we had:" we wish our reducer tree was organized into child reducers, so that it would be easy to add additional reducers in the future. So we go ahead and write our code that way.

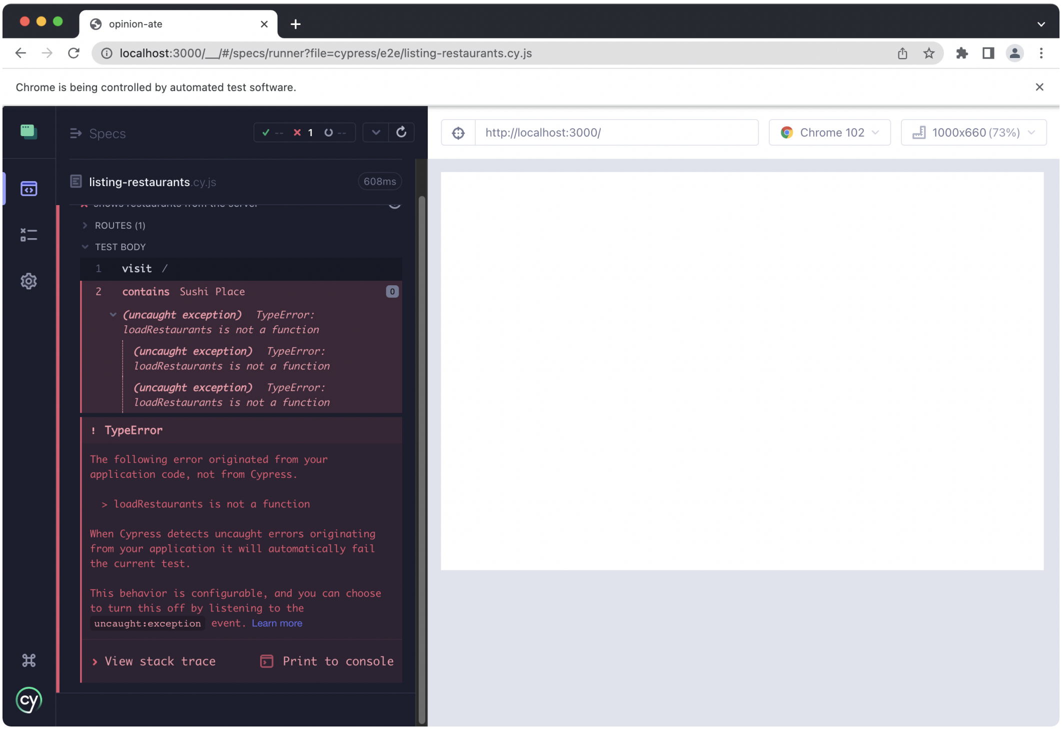
Now we need to create the restaurant reducer. Create a `src/store/restaurants` folder, then a `reducers.js` file inside it. Add the following contents:

```
import {combineReducers} from 'redux';

function records() {
  return [];
}

export default combineReducers({
  records,
});
```

Once again we go ahead and use `combineReducers` since we will ultimately have multiple restaurant reducers, for values like loading and error flags. For now, to get past the current E2E test error, we just need the one `records` reducer returning a hard-coded empty array.

This should fix our E2E test error, so rerun the Cypress test. Now we get a new error:

> TypeError: loadRestaurants is not a function



opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ○ --
listing-restaurants.cy.js
608ms
ROUTES (1)
TEST BODY
http://localhost:3000/
Chrome 102
1000x660 (73%)

```
1  visit  /
2  contains  Sushi Place
   (uncaught exception)  TypeError:
   loadRestaurants is not a function
     (uncaught exception)  TypeError:
     loadRestaurants is not a function
     (uncaught exception)  TypeError:
     loadRestaurants is not a function

! TypeError

The following error originated from your
application code, not from Cypress.

  > loadRestaurants is not a function

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

> View stack trace        Print to console
```

Cypress test failing because loadRestaurants is not a function

How do we want this missing `loadRestaurants` function to work? We want it to be an asynchronous Redux action. To make that work, it's time to add `redux-thunk`:

```
$ yarn add redux-thunk@2.4.1
```

Hook it up in `src/store/index.js`:

```
~~import {createStore} from 'redux';~~
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

~~const store = createStore(rootReducer);~~
const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;
```

In RestaurantList.js, write the code we wish we had to import a loadRestaurants action and provide it to the component:

```
import {connect} from 'react-redux';
import {loadRestaurants} from '../store/restaurants/actions';

export function RestaurantList({loadRestaurants, restaurants}) {
...
const mapStateToProps = state => ({
  restaurants: state.restaurants.records,
});

const mapDispatchToProps = {loadRestaurants};

~~export default connect(mapStateToProps)(RestaurantList);~~
export default connect(mapStateToProps, mapDispatchToProps)(RestaurantList);
```

Next, we need to implement the actions module we just used. In src/store/restaurants, create actions.js. We're still focused on writing the minimal code to fix the current E2E test error, so let's just export a loadRestaurants thunk that does nothing:

```
export const loadRestaurants = () => () => {};
```

Rerun the E2E test. We are no longer getting any errors in the application code; instead, we are back to the failure that the text "Sushi Place" is never shown. But we've made progress in the structure of our code. Now our component is dispatching the loadRestaurants async action and is reading the restaurants from the store. The only thing remaining is that our async action isn't loading those records from the API yet. That's logic we need to implement, meaning it's time to step back down to a unit test—this time for our Redux store.

Unit Testing the Store

To test our store, we're going to create a real Redux store, configure it with our reducer, then dispatch our actions against it. After we finish test-driving our store code, we'll look at the advantages this approach gives us.

Under src/store/, create a restaurants.spec.js file. Add the following structure:

```
describe('restaurants', () => {
  describe('loadRestaurants action', () => {
    it('stores the restaurants', async () => {
    });
  });
});
```

We create a `describe` block for our `loadRestaurants` action, which right now just has one test: that it stores the restaurants. Note that the test function is `async`, because the network request will be asynchronous. Now let's fill that test out.

We will need some records to be returned by our stubbed API:

```
it('stores the restaurants', async () => {
  const records = [
    {id: 1, name: 'Sushi Place'},
    {id: 2, name: 'Pizza Place'},
  ];
});
```

As we said earlier, our app will consist of three layers:

- The UI components
- The Redux store
- The API client

So we won't make an HTTP request directly in our Redux store. Instead, we'll delegate to an API client object that we pass in. Let's design the interface of that object now:

```
it('stores the restaurants', async () => {
  const records = [
    {id: 1, name: 'Sushi Place'},
    {id: 2, name: 'Pizza Place'},
  ];

  const api = {
    loadRestaurants: () => Promise.resolve(records),
  };
});
```

Giving the `api` object a descriptive `loadRestaurants()` method seems good. We are stubbing out the API here in the test, so we just implement that method to return a promise that resolves to our hard-coded records.

Now, to set up our Redux store. We'll use a real Redux store to run our tests through. That way we are testing our thunks, action creators, and reducers in integration.

Start with the initial state of the reducer:

```
  const api = {
    loadRestaurants: () => Promise.resolve(records),
  };

  const initialState = {
    records: [],
  };
});
```

Now we'll create the store itself. Unlike in the full application, we will only pass in the restaurant reducer. The full application may have other reducers, but our test will be focused only on the restaurant reducer.

```
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';
import restaurantsReducer from './restaurants/reducers';

describe('restaurants', () => {
...
      const initialState = {
        records: [],
      };

      const store = createStore(
        restaurantsReducer,
        initialState,
        applyMiddleware(
          thunk.withExtraArgument(api),
        ),
      );
    });
```

There's something slightly different between how we're setting up `thunk` here in our test and how we did so in the application code: now we're chaining a call to `.withExtraArgument()`. This method allows you to pass an additional argument at setup time that will be available to all thunk functions. Our unit test has driven us to use it because now we have an `api` we need to make available to our thunk.

Dependency Injection

`.withExtraArgument()` isn't the only way we can make a mock `api` available to our thunks. If we were following a more typical JavaScript approach and directly `import`ing the `api` in our thunks, we could use Jest module mocking[a] to replace that real import with a mocked import.

`.withExtraArgument()` is a `thunk` implementation of an approach called "dependency injection," where instead of hard-coding production dependencies, you pass them in to your code as arguments. Dependency injection makes it easier to use test doubles for your dependencies in unit tests.

In general, dependency injection is safer than overriding a hard-coded dependency using something like Jest module mocking because you won't accidentally use the production dependency in tests. If you forget to inject the dependency in your tests, you'll just get an error that the dependency is null. I recommend taking advantage of dependency injection whenever you're using a library that provides a way to inject dependencies and whenever you can write your own code to allow injecting dependencies.

[a] https://jestjs.io/docs/mock-functions#mocking-modules

Now that our store is set up, we can dispatch the `loadRestaurants` action, then check the state of the store afterward:

```
import restaurantsReducer from './restaurants/reducers';
import {loadRestaurants} from './restaurants/actions';

describe('restaurants', () => {
...
        applyMiddleware(
          thunk.withExtraArgument(api),
        ),
      );

      await store.dispatch(loadRestaurants());

      expect(store.getState().records)
        .toEqual(records);
    });
```

The test fails, showing an empty array as the received value:

```
▯ restaurants › loadRestaurants action › stores the restaurants

  expect(received).toEqual(expected) // deep equality

  - Expected  - 10
  + Received  + 1

  - Array [
  -   Object {
  -     "id": 1,
  -     "name": "Sushi Place",
  -   },
  -   Object {
  -     "id": 2,
  -     "name": "Pizza Place",
  -   },
  - ]
  + Array []
```

Now we're ready to implement our `loadRestaurants` thunk to retrieve the records from the `api` and dispatch an action to store them.

First, update the `loadRestaurants` function in `actions.js`:

```
~~export const loadRestaurants = () => () => {};~~
export const STORE_RESTAURANTS = 'STORE_RESTAURANTS';

export const loadRestaurants = () => async (dispatch, getState, api) => {
  const records = await api.loadRestaurants();
  dispatch(storeRestaurants(records));
};

const storeRestaurants = records => ({
  type: STORE_RESTAURANTS,
  records,
});
```

We define a new `STORE_RESTAURANTS` action type. Then, in the function `loadRestaurants()` returns, we call `.loadRestaurants()` on the passed-in `api` that we configured when we set up the store. When the call resolves with the returned records, we dispatch a new `storeRestaurants()` action, passing it the records. We define a `storeRestaurants` action creator to create the correct action object.

Save the file and the test failure is the same, because the `records` reducer doesn't yet store the restaurants in response to the `STORE_RESTAURANTS` action. Update it to do so:

```
import {combineReducers} from 'redux';
import {STORE_RESTAURANTS} from './actions';

~~function records() {~~
~~  return [];~~
function records(state = [], action) {
  switch (action.type) {
    case STORE_RESTAURANTS:
      return action.records;
    default:
      return state;
  }
}

export default combineReducers({
  records,
});
```

With this, our test passes.

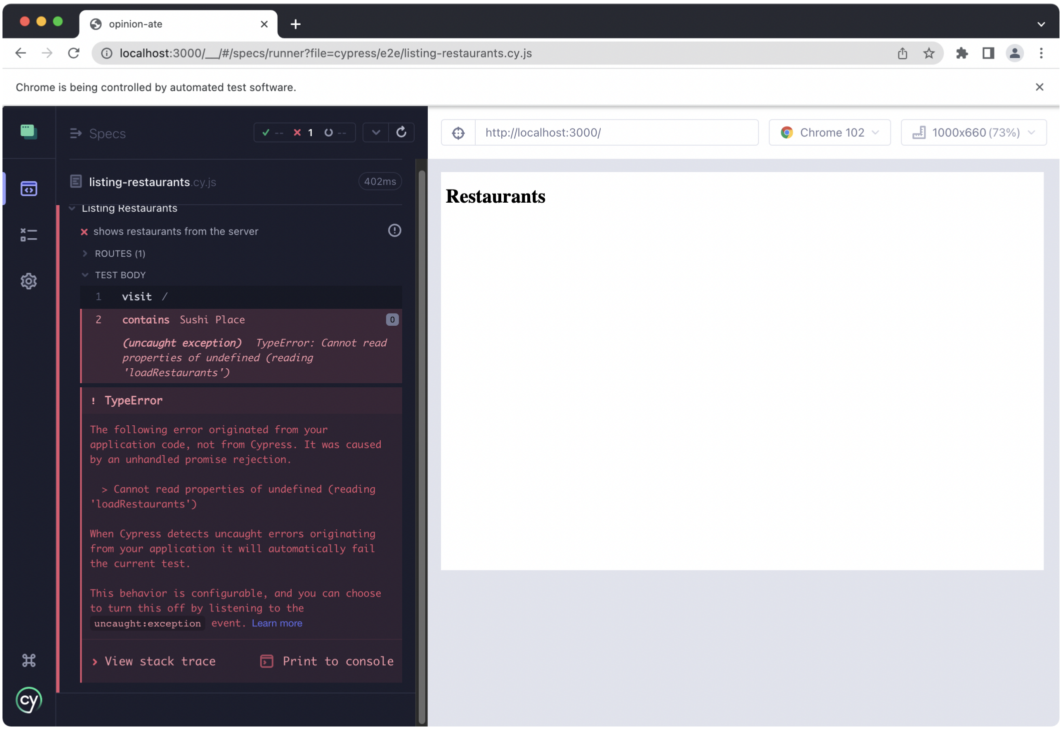
Now that our test is passing and our store code for this feature is complete, let's talk about the benefits that come from testing the store in integration. Our test interacts with the store the way the rest of our application does: by dispatching async actions and then receiving updated state. Just like the rest of our application, our test doesn't know or care about the `STORE_RESTAURANTS` action type; it treats it as an implementation detail. This gives us greater flexibility to refactor our store; for example, we could change the way the actions that `loadRestaurants` dispatches are set up. Our tests would continue to pass as long as the action type and state stayed the same—which is the contract that the rest of our application relies on as well.

Another benefit of testing the store from the outside is that it ensures that all the pieces work together. If we were testing the `loadRestaurants` async action, the `storeRestaurants` action creator, and the reducer separately from one another, they might work individually but not together. For example, maybe the names of properties in the action object returned by `storeRestaurants` aren't the same names as the properties the reducer looks for. Our test exercises the async action, action creator, and reducer in integration, ensuring that if they aren't working together, a unit test will fail. If we weren't testing this way, only an E2E test would catch this problem—and then only if the problem is in one of the main flows that our E2E test covers, not our edge cases.

Creating the API Client

Now that our unit test is passing, it's time to step back up to the E2E test. It fails with a new error:

> TypeError: Cannot read properties of undefined (reading 'loadRestaurants')



Chrome is being controlled by automated test software.

✕ shows restaurants from the server

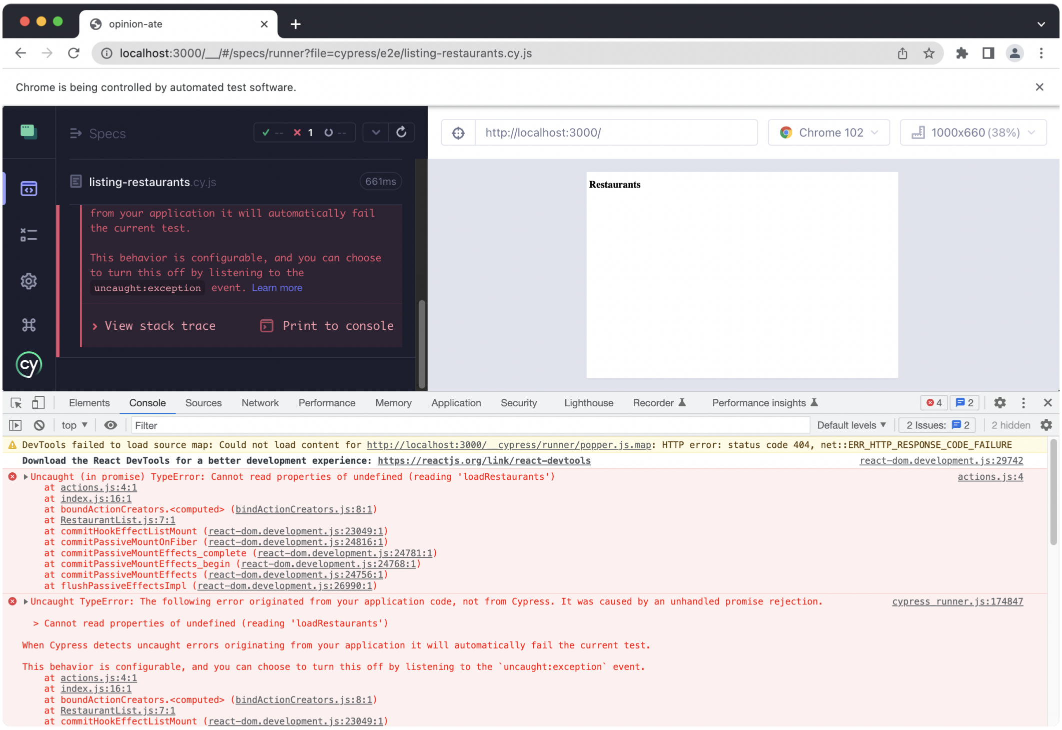
```
1  visit /
2  contains Sushi Place
   (uncaught exception) TypeError: Cannot read
   properties of undefined (reading
   'loadRestaurants')
```

```
! TypeError

The following error originated from your
application code, not from Cypress. It was caused
by an unhandled promise rejection.

  > Cannot read properties of undefined (reading
'loadRestaurants')

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more
```

Cypress test failing because it cannot read properties of undefined

It's not obvious what is going wrong from the output we can see by default in Cypress. To figure it out, we need to use our web developer console. Assuming you chose Chrome to run your tests, choose View > Developer > Developer Tools to open the developer tools, then click the Console tab. Click the circle icon with an arrow through it to clear out any existing console entries. Then click the circular arrow icon at the top of the left column to rerun the Cypress test.

After the test reruns, you'll see the same error outputted in the console:

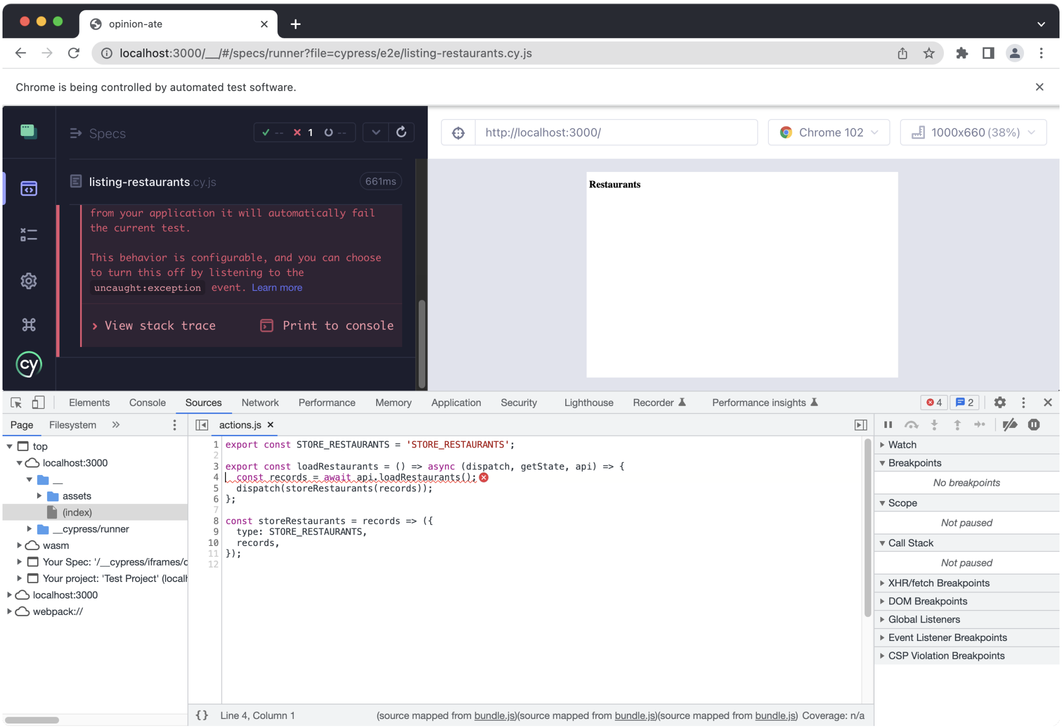


```
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
 uncaught:exception  event. Learn more

> View stack trace        Print to console
```

```
DevTools failed to load source map: Could not load content for http://localhost:3000/__cypress/runner/popper.js.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
Download the React DevTools for a better development experience: https://reactjs.org/link/react-devtools          react-dom.development.js:29742
▸Uncaught (in promise) TypeError: Cannot read properties of undefined (reading 'loadRestaurants')                  actions.js:4
    at actions.js:4:1
    at index.js:16:1
    at boundActionCreators.<computed> (bindActionCreators.js:8:1)
    at RestaurantList.js:7:1
    at commitHookEffectListMount (react-dom.development.js:23049:1)
    at commitPassiveMountOnFiber (react-dom.development.js:24816:1)
    at commitPassiveMountEffects_complete (react-dom.development.js:24781:1)
    at commitPassiveMountEffects_begin (react-dom.development.js:24768:1)
    at commitPassiveMountEffects (react-dom.development.js:24756:1)
    at flushPassiveEffectsImpl (react-dom.development.js:26990:1)
▸Uncaught TypeError: The following error originated from your application code, not from Cypress. It was caused by an unhandled promise rejection.     cypress_runner.js:174847

  > Cannot read properties of undefined (reading 'loadRestaurants')

When Cypress detects uncaught errors originating from your application it will automatically fail the current test.

This behavior is configurable, and you can choose to turn this off by listening to the `uncaught:exception` event.
    at actions.js:4:1
    at index.js:16:1
    at boundActionCreators.<computed> (bindActionCreators.js:8:1)
    at RestaurantList.js:7:1
    at commitHookEffectListMount (react-dom.development.js:23049:1)
```

Cypress test runner with browser console showing an error

Click the line that says "at actions.js:4:1" (your numbers might be slightly different if you typed the code differently). This should open `actions.js` in the Sources tab, showing the line that the error originated from:



```
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

› View stack trace        Print to console
```

```
export const STORE_RESTAURANTS = 'STORE_RESTAURANTS';

export const loadRestaurants = () => async (dispatch, getState, api) => {
  const records = await api.loadRestaurants();
  dispatch(storeRestaurants(records));
};

const storeRestaurants = records => ({
  type: STORE_RESTAURANTS,
  records,
});
```

Line 4, Column 1 (source mapped from bundle.js)(source mapped from bundle.js)(source mapped from bundle.js) Coverage: n/a

Cypress test runner with browser sources tool showing a line with an error

Now that we can see the source of the error, we can tell that this isn't referring the `loadRestaurants` action, but to the `loadRestaurants` method of the `api`. Our component and store are built; now we just need to build our API client.

You may be surprised to hear that we aren't going to unit test our API client at all; instead, we're going to let the E2E test drive the implementation. Let's go through the process first, then we'll discuss why we didn't unit test it.

To fix the current E2E test failure, we need to create an API object and provide it to `redux-thunk`. We'll go ahead and create a `loadRestaurants()` method on the object, too, since we can see from the error that we'll need it.

Create an `api.js` file under `src`, and add the following code:

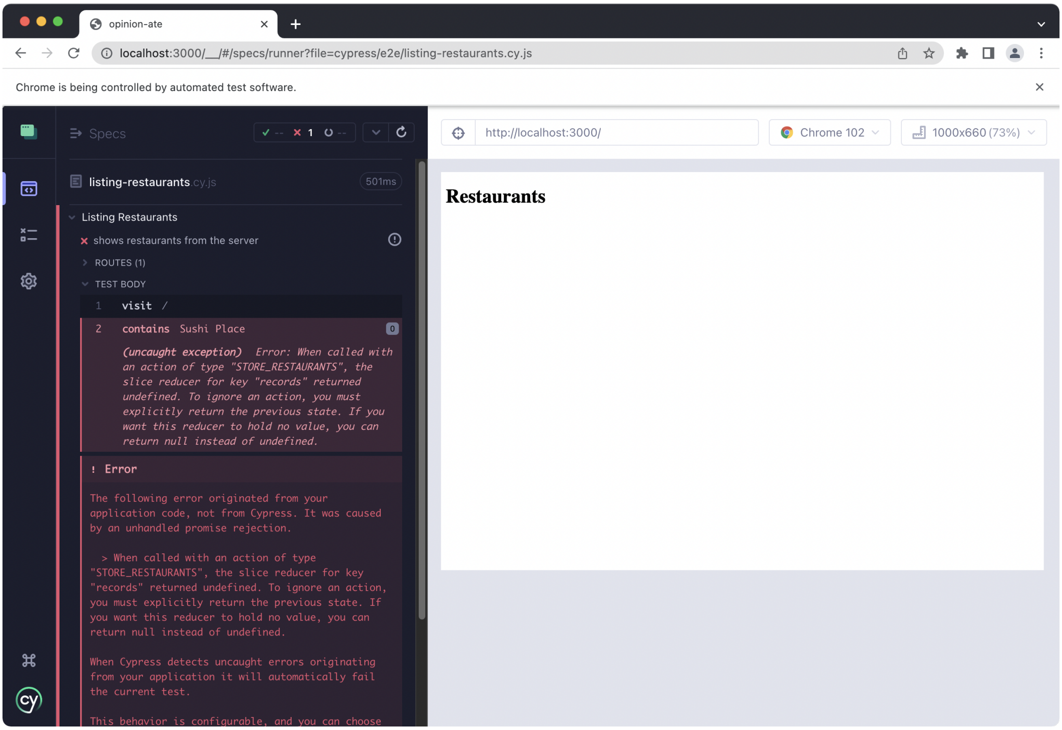
```
const api = {
  loadRestaurants() {},
};

export default api;
```

Next, let's wire the API object up our store. Update `src/store/index.js`:

```
import rootReducer from './reducers';
import api from '../api';

~~const store = createStore(rootReducer, applyMiddleware(thunk));~~
const store = createStore(
  rootReducer,
  applyMiddleware(
    thunk.withExtraArgument(api),
  ),
);
```

Rerun the E2E test and we get a new error:

> When called with an action of type "STORE_RESTAURANTS", the slice reducer for key "records" returned undefined.


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ○ --
http://localhost:3000/
Chrome 102
1000x660 (73%)
listing-restaurants.cy.js
501ms
Listing Restaurants
✕ shows restaurants from the server
ROUTES (1)
TEST BODY
Restaurants

```
1  visit  /
2  contains  Sushi Place
   (uncaught exception)  Error: When called with
   an action of type "STORE_RESTAURANTS", the
   slice reducer for key "records" returned
   undefined. To ignore an action, you must
   explicitly return the previous state. If you
   want this reducer to hold no value, you can
   return null instead of undefined.
```

```
! Error

The following error originated from your
application code, not from Cypress. It was caused
by an unhandled promise rejection.

  > When called with an action of type
"STORE_RESTAURANTS", the slice reducer for key
"records" returned undefined. To ignore an action,
you must explicitly return the previous state. If
you want this reducer to hold no value, you can
return null instead of undefined.

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
```

Cypress failure because a reducer returned undefined

What's going on here? The return value of `records()` for action `STORE_RESTAURANTS` is `undefined`. We're returning the `action.records` field, so this means `action.records` is `undefined`. This is because we still aren't making the HTTP request that we need to retrieve this data from the back-end. Let's fix that now.

We'll use the popular Axios library to make our HTTP requests. Add it to your project:

```
$ yarn add axios@0.27.2
```

What about `fetch()`?
Instead of Axios, we could have used the browser's built-in `fetch()` function. But there are still users on older iPhones and IE 11 whose browsers don't have `fetch()`, and in my opinion Axios provides some nice usability improvements over `fetch()`.

Next, use Axios to make an HTTP request to the correct endpoint:

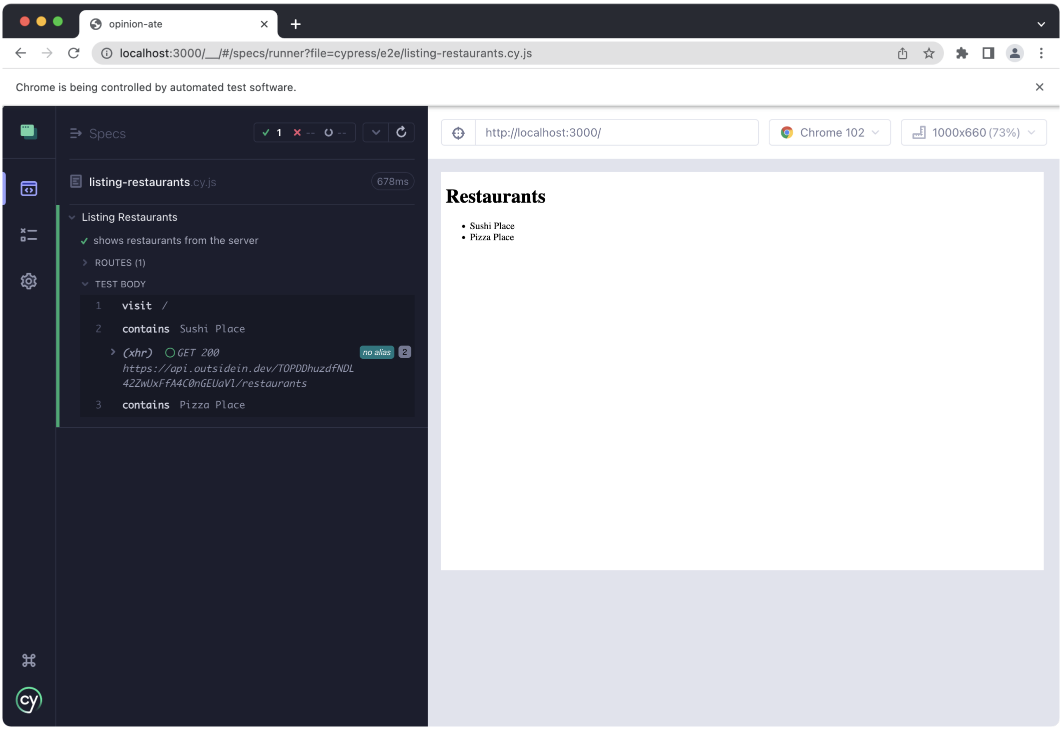
```
import axios from 'axios';

const client = axios.create({
  baseURL: 'https://api.outsidein.dev/YOUR-API-KEY',
});

const api = {
~~  loadRestaurants() {}~~
  async loadRestaurants() {
    const response = await client.get('/restaurants');
    return response.data;
  },
};

export default api;
```

In `baseURL`, replace `YOUR-API-KEY` with the API key you created earlier. (Note that in a real application you wouldn't hard-code authentication like this, but this is just to keep our exercise simple.)

We import Axios, then call its `create()` method to create a new Axios instance configured with a base URL pointing to the back-end, including your personal API key. Then we implement our `loadRestaurants()` method by calling the Axios client's `get()` method to make an HTTP `GET` request to the path `/restaurants` under our base URL.

Axios' `get()` method returns a promise that resolves to an Axios response object. That object includes a `data` field containing the response body. In cases like ours where the response is JSON data, Axios will automatically parse it into a JavaScript data structure and give that to us instead of the raw text. That's the value we need, so we return `response.data`.

Rerun the E2E test one more time. The test should confirm that "Sushi Place" and "Pizza Place" are loaded and displayed on the page. Our E2E test is passing!


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ 1 ✗ -- ○ --
http://localhost:3000/
Chrome 102
1000x660 (73%)

```
listing-restaurants.cy.js                         678ms
Listing Restaurants
  ✓ shows restaurants from the server
    ROUTES (1)
    TEST BODY
    1  visit  /
    2  contains  Sushi Place
         (xhr)  ○GET 200                       no alias  2
         https://api.outsidein.dev/TOPDDhuzdfNDL
         42ZwUxFfA4C0nGEUaVl/restaurants
    3  contains  Pizza Place
```

Restaurants

- Sushi Place
- Pizza Place

Cypress test for loading restaurants passing

Now, let's step back to a question that came up when we started building the API client: why didn't we unit test it? It would be possible for us to set up the code so that we could pass in a test double for `axios` to allow us to test it. But I hesitate to do so because of a unit testing principle: **don't mock what you don't own**. The principle applies equally well to using any kind of test doubles for code you don't own, not just mocks. There are a few reasons for this principle:

- If you mock third party code but you get the functionality wrong, then your tests will pass against your mock but won't work with the real third-party library. This can happen if the mock initially matches the library but, later, the behavior of the library changes.
- Some of the value of unit tests is in allowing you to design the API of your dependencies, but since you can't control the API of the third-party library, you don't get the opportunity to affect the API. (Pull requests to open-source projects notwithstanding!)

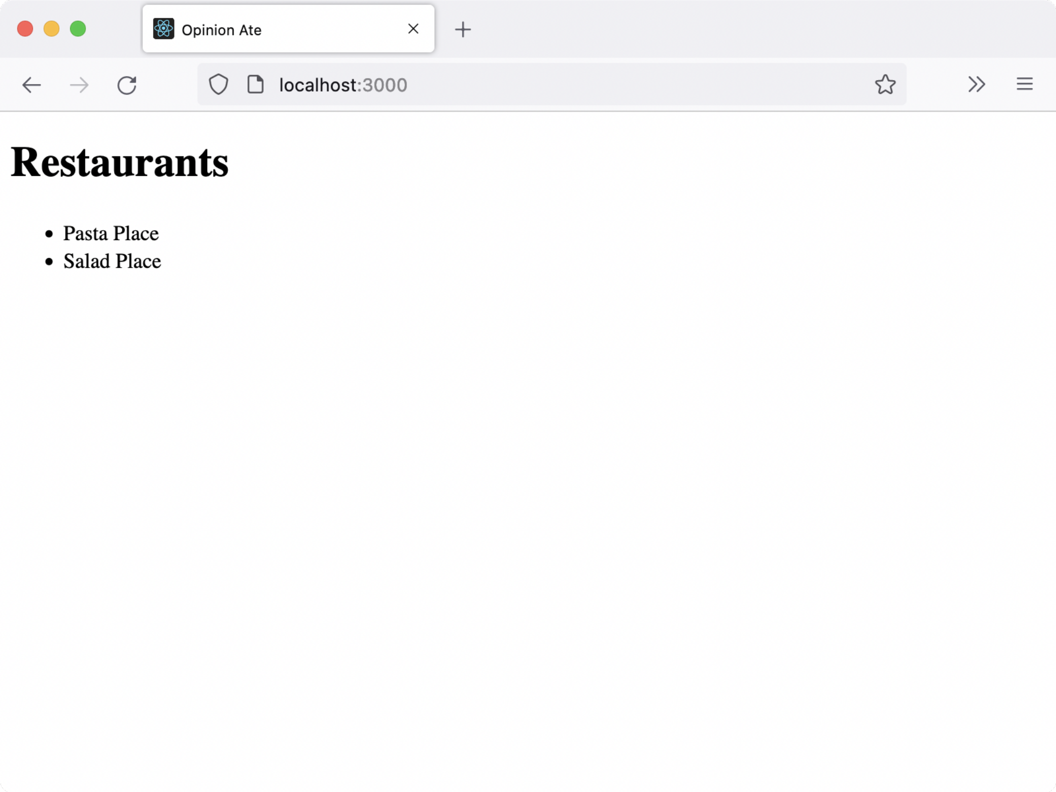
If you shouldn't mock third-party dependencies, how can you test code that uses them? The alternative is to do what we did earlier in this chapter: **wrap the third-party code with**

your *own* interface that you do control, and mock that. In our case, we created an `api` interface with a `loadRestaurants()` method that returns our array of restaurants directly, not nested in a `response` object. The module that wraps the third-party library should be very simple, with as little logic as possible—ideally without any conditionals. That way, you won't even feel the need to unit test it.

Let's think about what would happen if we *did* try to write a unit test for `api`. We would need to replace `axios` with a test double, stubbing `axios.create()` to return another test double. The second test double would need to stub `axios.get()` to return a test response. And we would need to ensure the correct arguments were passed to each of these two methods. At that point almost the only thing the test is doing is repeating the production code, so it isn't adding a lot of value. And if Axios made a change that caused `api` to break, these unit tests wouldn't catch it.

Instead, it's better to test your wrapper in integration with the third-party library. In our case, our E2E tests serve this function. Because Cypress stubs out the actual HTTP request, our E2E test will only pass if `api` and Axios work together correctly to send the right request and provide the response data to the rest of the app. No more testing of the `api` is needed than that.

Now let's see our app working against the real back-end. Go to your React app at `http://localhost:3000`. You should see the default "Pasta Place" and "Salad Place" records loaded from the API.



Restaurants

- Pasta Place
- Salad Place

App running locally displaying restaurants from the back-end

We successfully implemented our first feature with outside-in test-driven development!

Double HTTP Requests in Development Mode

If you check the Network tab of your web developer tools, you'll notice that the HTTP request to load the restaurants is sent *twice!* It doesn't hurt anything, but it's not ideal. Why does this happen?

By default, in development mode React 18 automatically unmounts and remounts components when they mount for the first time, resulting in effects being run twice. This doesn't occur in production builds, which you can verify once we've deployed to Netlify later in this chapter.

This isn't a bug in React: it's a feature intended to make components more resilient[a]. The reason this was implemented despite the problem of multiple data fetches is that, despite how common it is for React developers to load data in a `useEffect`, it's not actually recommended to do so. Unfortunately, there isn't currently any recommended way to load data when a component is first mounted built in to React. The planned future recommended approach is React Suspense for data fetching, but it is not yet ready to be used directly[b].

In the meantime, if this double-request behavior isn't causing problems for you in development mode, you can ignore it. If you'd like to prevent it, see the discussion in the React GitHub issue on duplicate effects[c] for options, including using a React-specific data fetching library like React Query.

[a] https://reactjs.org/docs/strict-mode.html#ensuring-reusable-state
[b] https://reactjs.org/blog/2022/03/29/react-v18.html#suspense-in-data-frameworks
[c] https://github.com/facebook/react/issues/24502

Pull Request Workflow

Our feature is working locally, and now we need to get it into our main git repository and deployed. We'll do this with a pull request.

If you have any uncommitted changes, commit them to git.

Next, push up your branch to the origin:

```
$ git push -u origin HEAD
```

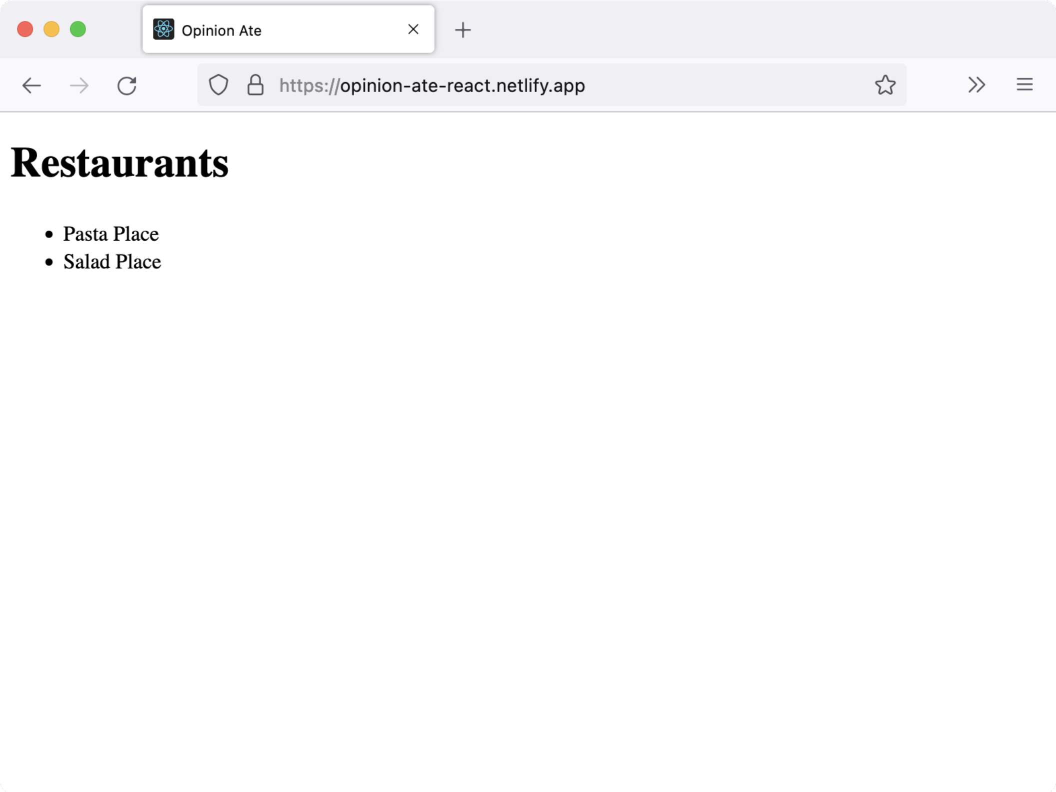
Click the link that GitHub provides to open a pull request. Title the pull request "List restaurants". You can leave the description field blank for this exercise; in a team context you would describe the change you made, how to manually test it, and other important information about decisions or trade-offs you made. Go ahead and click "Create pull request".

The other thing that would happen at this point in a team context is that your team members would review the pull request and provide feedback. Often reviewers limit this feedback to pointing out problems, but there is a lot more you can do in code review. You can ask questions to better understand the author's intent. You can encourage them about decisions they made that you like or have learned from. You can make proposals for changes that you consider optional, allowing the author the freedom to choose whether to take or leave them. All of these create a code review culture that feels encouraging and motivating.

When you open the pull request, you can see CI running at the bottom. If it fails, click “Details” and check the output to see what went wrong. Try running the tests locally to see if you get the same problem; if so, fix it and push up the fix.

When CI passes, merge the pull request.

Now Netlify should automatically be deploying the updated version of our site. Go to https://app.netlify.com and check the build progress. When it completes, go to your site and see it successfully listing restaurants. It’s exciting to see it live!



Restaurants

- Pasta Place
- Salad Place

App running on Netlify displaying restaurants from the back-end

Some real production systems deploy on every merge like this. If you’re practicing TDD, you might decide that the test coverage it provides gives you enough confidence to deploy your system on every merge. Others will not actually deploy quite as often, but they’re *able* to if needed. An agile team is equipped to deploy as often as the business wants, which allows delivering value and getting feedback as quickly as possible.

Now we can drag our “List Restaurants” story to “Done” in Trello.

Then, locally, switch back to the `main` branch and pull in the the changes that have been merged in from the branch:

```
$ git checkout main
$ git pull
```

What’s Next

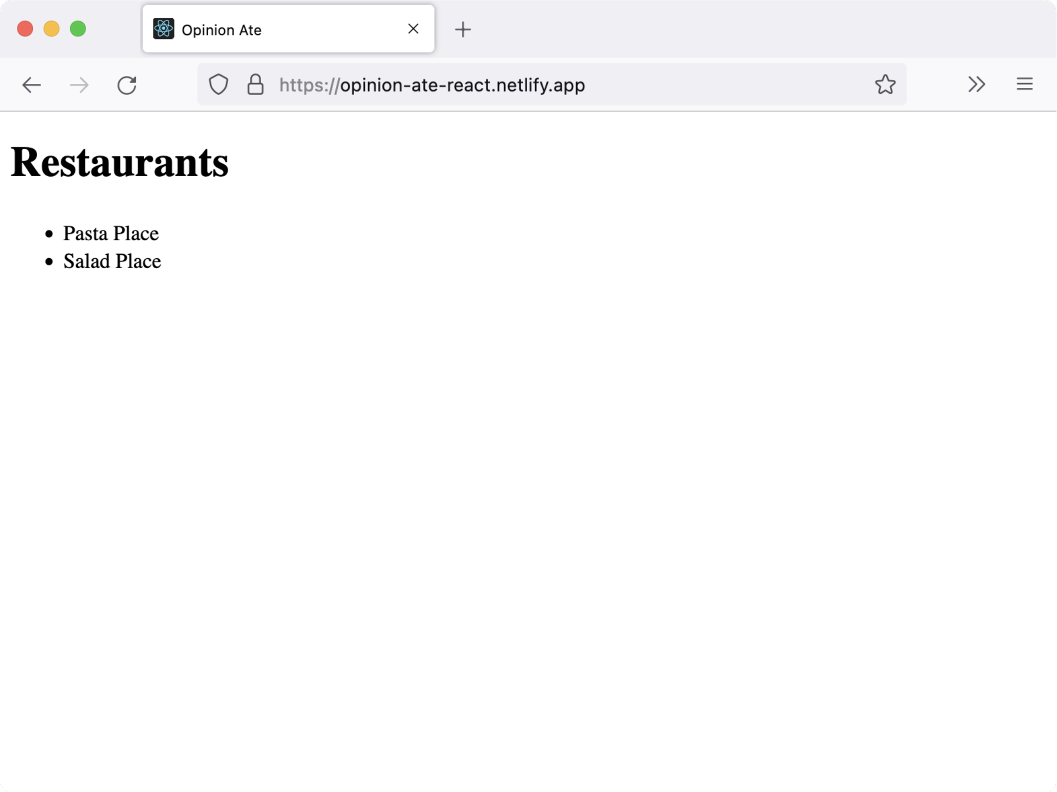
Now our first feature is working, and TDD has helped us to start with a minimal implementation. In the next chapter we’ll iterate on this implementation by adding a UI component library and improving the styling of the app.

8. Refactoring Styles

In this chapter we'll see one example of the kind of refactoring you can do when you have the thorough test coverage that TDD provides: restyling. We'll update our application from using plain unstyled HTML elements to using styled elements provided by a UI component library. TDD allows us to separate the process of getting the application working from the process of making it look good, so we can focus on one thing at a time.

Our next story in Trello is "Style App with Material Design"; drag it to "In Progress".

This work is definitely needed. By following TDD and writing only the minimal code to pass the tests, our app certainly isn't going to win any awards for visual design:



Restaurants

- Pasta Place
- Salad Place

App with little visual design

But one of the benefits of the thorough test suite that TDD provides is that you can make changes to the look and feel of your app with confidence that the functionality still works.

First, let's confirm our tests are passing. Run `yarn test`. You may get the message:

```
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

If you get this message, press `a` to run all the tests. They should pass. Keep the unit test process running throughout this chapter.

In another terminal, run `yarn start`, and in a third, run `yarn cypress`. Click “E2E Testing” then “Start E2E Testing in Chrome”, then `listing-restaurants.cy.js`. Wait to make sure it passes.

Now that we know all our tests pass, we’re ready to update the look-and-feel of the app. We’re going to use MUI, a popular React component library that follows Google’s Material Design.

Create a new branch for this story:

```
$ git checkout -b material-design
```

Install the MUI packages:

```
$ yarn add \
    @mui/material@5.10.7 \
    @emotion/react@11.10.4 \
    @emotion/styled@11.10.4
```

Then, in `public/index.html`, add the following `link` tag to load Roboto, the font used by MUI:

```
  Learn how to configure a non-root public URL by running `npm run build`.
-->
<link
  rel="stylesheet"
  href="https://fonts.googleapis.com/css?
    family=Roboto:300,400,500,700&display=swap"
/>
<title>Opinion Ate</title>
```

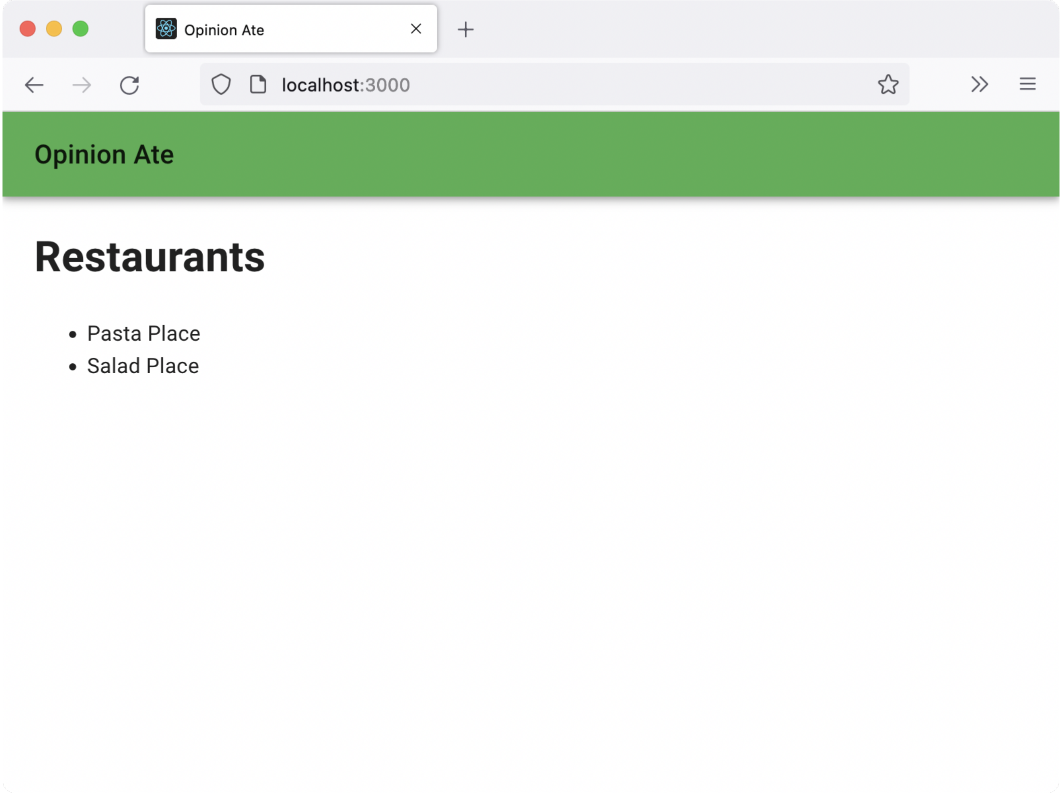
The line break between `css?` and `family=` is just to make the URL fit on one page in the book; in your code put `css?family=` right next to one another.

Now we’re ready to begin styling our app. We’ll begin by styling the `App` component to give it a title bar and some theme-standard layout.

In `App.js`, let’s add a number of MUI components around our `RestaurantScreen` component:

```
import {Provider} from 'react-redux';
import {createTheme} from '@mui/material/styles';
import {green} from '@mui/material/colors';
import {ThemeProvider} from '@mui/material/styles';
import CssBaseline from '@mui/material/CssBaseline';
import AppBar from '@mui/material/AppBar';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import store from './store';
import RestaurantScreen from './components/RestaurantScreen';

const theme = createTheme({
  palette: {
    primary: green,
  },
});

export default function App() {
  return (
    <Provider store={store}>
      <ThemeProvider theme={theme}>
        <CssBaseline />
        <AppBar position="static">
          <Toolbar>
            <Typography variant="h6">Opinion Ate</Typography>
          </Toolbar>
        </AppBar>
        <Container>
          <RestaurantScreen />
        </Container>
      </ThemeProvider>
    </Provider>
  );
}
```

Rerun the E2E test. It still passes, and notice we now have a nice green toolbar, and there’s some padding on the left and right on the content area.



Opinion Ate

Restaurants

- Pasta Place
- Salad Place

App styled with MUI

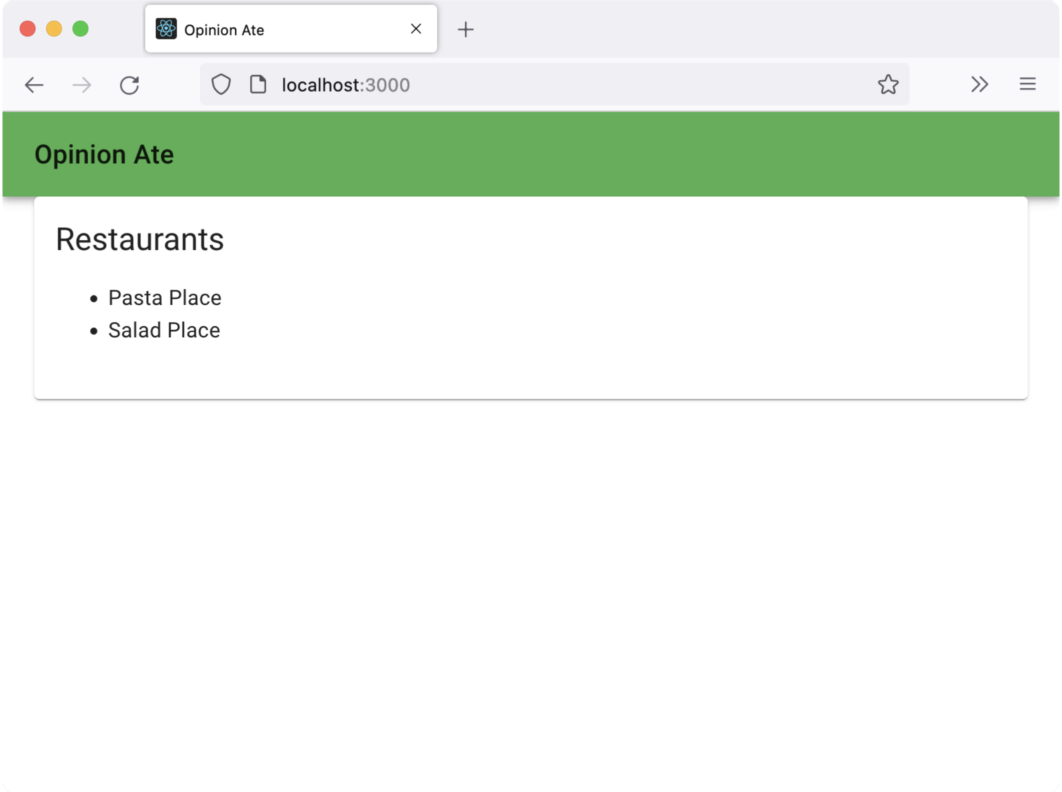
Here's what each of these MUI components does to help achieve this look:

- `createTheme()` allows us to create a theme, including setting the `primary` color of our app to the `green` color defined by MUI.
- `ThemeProvider` allows you to make the theme you created with `createThem()` available to the whole app. It's an optional component that isn't needed if you aren't customizing the theme.
- `CssBaseline` applies some default page-wide CSS styles.
- `AppBar` and `Toolbar` work together to provide the top title bar.
- `Typography` formats text to styles that are part of the Material Design system. MUI components are designed with certain typographic styles in mind, so it's best to check the docs for full examples of what typography settings are recommended for text displayed inside other components.
- `Container` centers your content horizontally to provide some padding and keeps the content from stretching too much in very wide browser windows.

Next let's style `RestaurantScreen`. A common UI element in Material Design is a card, which is a box around some content. Let's style the `RestaurantScreen` as a card:

```
import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import Typography from '@mui/material/Typography';
import RestaurantList from './RestaurantList';

export default function RestaurantScreen() {
  return (
    ~~<div>~~
      ~~<h1>Restaurants</h1>~~
    <Card>
      <CardContent>
        <Typography variant="h5">Restaurants</Typography>
        <RestaurantList />
    ~~</div>~~
      </CardContent>
    </Card>
  );
}
```

Reload the E2E test and notice there's a box around the content now.



Opinion Ate

Restaurants

- Pasta Place
- Salad Place

Card styled with MUI

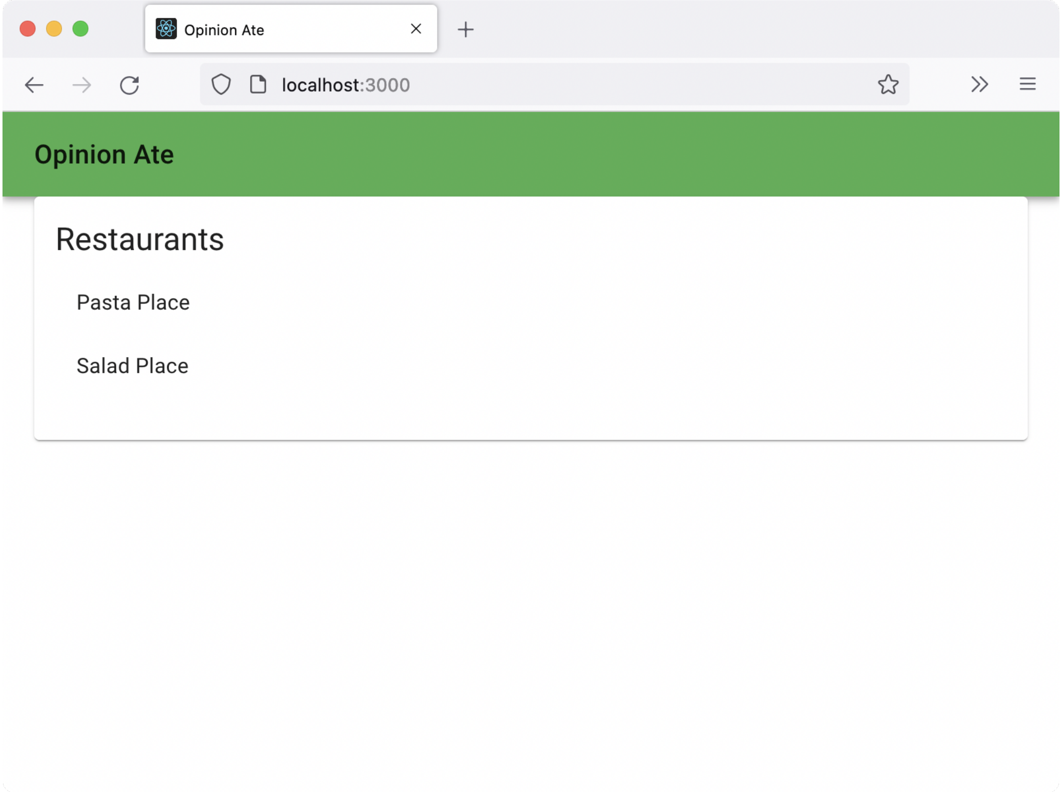
Here's what the components do:

- `Card` is the wrapper for the card and provides the outline.
- `CardContent` provides appropriate padding around the content area of a card.
- We saw `Typography` earlier for styling text. In this case, the recommended variant for a card title is `h5`.

Finally, let's style the list of the restaurants. Material Design includes the concept of a list, and its design will look nice here:

```
import {useEffect} from 'react';
import {connect} from 'react-redux';
import List from '@mui/material/List';
import ListItem from '@mui/material/ListItem';
import ListItemText from '@mui/material/ListItemText';
import {loadRestaurants} from '../store/restaurants/actions';

export function RestaurantList({loadRestaurants, restaurants}) {
...
  return (
    ~~<ul>~~
    <List>
      {restaurants.map(restaurant => (
        ~~<li key={restaurant.id}>{restaurant.name}</li>~~
        <ListItem key={restaurant.id}>
          <ListItemText>{restaurant.name}</ListItemText>
        </ListItem>
      ))}
    ~~</ul>~~
    </List>
  );
```

Rerun the E2E test to see the changes:



Opinion Ate

Restaurants

Pasta Place

Salad Place

List styled with MUI

Here's what these components do:

- `List` wraps a list and provides appropriate outer styling.
- `ListItem` is the wrapping component for the list item.
- `ListItemText` is the primary title in the list item.

Our E2E and unit tests still pass. With that, we've completed restyling our app. Because we have a thorough test suite, we can have a high degree of confidence that we haven't broken anything. As a result, we didn't feel the need to balance visual improvements with the cost of manual retesting or the risk of breaking anything; we were able to do as many visual improvements as we liked.

Don’t Specify Markup and Styles in Tests

Some front-end TDD approaches recommend specifying every detail of your markup and styling in your component tests. TDD traditionally says you shouldn’t write any production code without a test driving you to do it. These folks argue that because in React markup and styling are part of your code, therefore you shouldn’t write complex markup and CSS without a test for them.

I think that’s a bad idea. Here’s why:

- Tests of detailed markup and styles don’t add a much value; they are just repeating what is in the production code.
- Behavioral tests aren’t well-suited to visuals. Test-driving the markup and CSS won’t ensure the component looks right; it just ensures that you typed in the HTML tag you intended to.
- Such tests are incredibly highly-coupled to the production code. Every change to the production code would require a change to the test. That’s a sign that they aren’t testing the interface, but rather the implementation.
- Such tests prevent refactoring. You wouldn’t be able to do the visual changes we did in this chapter under test; you would need to change the tests at the same time, so the tests would not provide a high degree of confidence that the behavior is still working.

This kind of extreme view, and the downsides that come from it, convince folks not to try TDD or give them a bad experience if they do try it.

Here’s what I recommend as a better approach: don’t write any *behavioral* production code without a failing test driving you to do so, but *do* write markup and styling without specifying it in a test. Keep your component tests focused on the behavior of the component, and consider markup and styling specifics to be implementation details.

If you have any uncommitted changes, commit them to git. Push up your branch to the origin and open a pull request. Wait for CI to complete, then merge the pull request. Now we can drag our “Style App with Material Design” story to “Done” in Trello.

What’s Next

In this chapter we improved the look-and-feel of our first feature. That isn’t the only improvement it needs: there are some functionality improvements it could benefit from as well. In the next chapter we’ll add test-drive the code for edge cases related to loading and error states.

9. Edge Cases

In this chapter we'll look at how to test-drive edge case functionality using unit tests. This relates to our next story in Trello, "Show Loading and Error States". Drag it to "In Progress".

Our first story was "List Restaurants", and we kept it minimal so that we could build out a vertical slice quickly and get all the parts of our app talking together. Our next story will make the restaurant loading functionality more robust, providing a loading indicator and an error message in case of problems.

Create a new branch for this story:

```
$ git checkout -b edge-cases
```

You could theoretically write an E2E test for this functionality, confirming the loading indicator and error message appear at the appropriate times. But if you write too many E2E tests, your test suite will get slow. That slowdown will cause you to run it locally less and less frequently over time, and slow CI runs will slow down your ability to merge PRs.

To prevent this from happening, it's best to write fewer E2E tests and more unit tests. But how should you decide exactly what ratio to use? Outside-in TDD provides an answer to this question. In outside-in TDD, you write an E2E test for each main flow of your application, as well as the unit tests to help implement that flow. Then, for more detailed or edge-case functionality, you only write the unit tests.

In our case, we're considering the loading indicator and error message to be detailed, edge-case functionality. We'll TDD them at the unit level but forego any E2E tests for them.

Loading Indicator

First let's add the loading indicator. Although we aren't writing an E2E test, we can still start from the "outside" in the sense of the user interface: the `RestaurantList` component. First we'll TDD the loading indicator in the UI, then we'll move "inside" to TDD the store functionality to support it.

Component Layer

In `RestaurantList.spec.js` we render our component using a `renderComponent()` helper function. This is working well, but now we need a way to set up the props slightly differently for different tests. We want a test where a loading flag prop is set. To do this, let’s refactor our tests for more flexibility.

First, start the unit tests with `yarn test` and keep them running for the duration of this chapter.

Next, let’s change the `renderComponent` function to allow passing in props to override the defaults:

```
~~function renderComponent() {~~
  ~~loadRestaurants = jest.fn().mockName('loadRestaurants');~~

  ~~render(~~
    ~~<RestaurantList~~
      ~~loadRestaurants={loadRestaurants}~~
      ~~restaurants={restaurants}~~
    ~~/>,~~
  ~~);~~
function renderComponent(propOverrides = {}) {
  const props = {
    loadRestaurants: jest.fn().mockName('loadRestaurants'),
    restaurants,
    ...propOverrides,
  };
  loadRestaurants = props.loadRestaurants;

  render(<RestaurantList {...props} />);
}
```

Here’s what’s going on:

- `renderComponent` now takes an optional `propOverrides` argument. If it’s not passed, it defaults to an empty object.
- We create a local `props` variable and assign an object to it. By default, that object contains values for the `loadRestaurants` and `restaurants` props. But we use the object

spread operator to take any passed-in propOverrides and assign those values to the object, potentially overriding the defaults.

- Whatever the final value of the loadRestaurants property is (either the default Jest mock function or a passed-in override) we assign that value to a variable so it can be accessed in the tests.
- We render the component, passing it all the props.

Save the changes and the tests should still pass.

Now we're ready to write our a new test for when the store is in a loading state. First, let's render the component, passing in a new loading prop indicating that the restaurants are currently loading:

```
it('displays the loading indicator while loading', () => {
  renderComponent({loading: true});
});
```

Now, how can we check that a loading indicator is shown? Sometimes apps will show the text "Loading…" as a loading indicator, but say we want to indicate it visually instead. For example, now that we're using MUI, say that we check its docs and see the CircularProgress component, an animated circular indicator. If no text is shown, how can we check for it in our test? This isn't just a problem for tests: it's also a problem for screen readers, tools that allow visually impaired users to interact with software. Screen readers will read out textual content, but how do they decide what out when there's an element with no textual content?

One way screen readers can interpret elements without text is using ARIA roles[23], which are descriptions of what an element is meant to represent. One such role is "progressbar", which fits what we want. And MUI provides good screen reader support here by automatically giving its CircularProgress component an ARIA role of "progressbar". Because of this, we can look for an element with the ARIA role of "progressbar":

```
it('displays the loading indicator while loading', () => {
  renderComponent({loading: true});
  expect(screen.getByRole('progressbar'))
    .toBeInTheDocument();
});
```

[23] https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles

Save the test. In good TDD style, our test fails, because the element isn't yet present.

Next, we want to follow the TDD practice of making the smallest possible change to get the test to pass. The smallest change in this case would be to hard-code the loading indicator to show *all* the time. We import MUI's `CircularProgress` spinner and render it:

```
import {connect} from 'react-redux';
import CircularProgress from '@mui/material/CircularProgress';
import List from '@mui/material/List';
...
  return (
    <>
      <CircularProgress />
      <List>
        {restaurants.map(restaurant => (
          <ListItem key={restaurant.id}>
            <ListItemText>{restaurant.name}</ListItemText>
          </ListItem>
        ))}
      </List>
    </>
  );
```

Save and the test passes.

This last step might make you feel uncomfortable. We don't want the loading indicator to *always* show, right? Shouldn't we go ahead and put a conditional on it?

No, and here's why: if we add the conditional now, *the conditional is not tested*. This is because our tests pass whether or not there is a conditional in place. It's good that we want the conditional, but **we also need to implement the tests that will confirm the conditional is working**. After we make the tests pass in the easiest way possible, if we see that there is more logic we need, we should think: what is the additional application behavior that we need to write a test for?

In this case, the behavior we need is that the loading indicator should *not* show when the data is *not* being loaded. To build that behavior, our first step is to write a test for it:

```
it('does not display the loading indicator while not loading', () => {
  renderComponent({loading: false});
  expect(screen.queryByRole('progressbar'))
    .not.toBeInTheDocument();
});
```

Note that we used `queryByRole` instead of `getByRole` here. `query` methods return a `null` if an element is not found, whereas `get` methods throw an error. Because we *expect* to not find the element here, a `query` method is necessary for our assertion to succeed.

This test fails, of course, because we are not yet hiding the loading indicator the way we want. And now that we have two tests, this will force us to implement the conditional to get them both to pass:

```
import {loadRestaurants} from '../store/restaurants/actions';

~~export default function RestaurantList({loadRestaurants, restaurants}) {~~
export default function RestaurantList({
  loadRestaurants,
  restaurants,
  loading,
}) {
...
  return (
    <>
~~      <CircularProgress />~~
      {loading && <CircularProgress />}
      <List>
```

Save the file and all tests pass.

Now, let's think about refactoring. We're getting a lot of tests in our `RestaurantList.spec.js`. A few of them are more closely related than others: both the "don't display loading indicator" test and the "display restaurants" test relate to the situation when the loading is done. We can use a `describe` block to represent this. Wrap the two of them in the following:

```
describe('when loading succeeds', () => {
//...
});
```

Save and confirm the tests still pass.

Now, look at the "does not display the loading indicator" test. In it, we pass a `loading: false` prop to `renderComponent`. But conceptually "not loading" will be the default state of our application. To represent this, let's set up `renderComponent` with `loading: false` as a default prop:

```
const props = {
  loadRestaurants: jest.fn().mockName('loadRestaurants'),
  loading: false,
  restaurants,
  ...propOverrides,
};
```

Now we don't need to pass a prop override in the test of the loading indicator hiding:

```
it('does not display the loading indicator while not loading', () => {
~~  renderComponent({loading: false});~~
  renderComponent();
```

Save and the tests should pass.

Now both of our tests in the "when loading succeeds" group call `renderComponent()` with no argument. There is one other test that calls `renderComponent()` with no argument: the test that it "loads restaurants on first render." Should we group that test together to remove duplication? I wouldn't recommend it. Although the call is the same, conceptually the situation is different. That test is focused on the situation when loading restaurants is first kicked off, and the other tests are focused on the situation when loading completes. It just so happens that the state of the store is the same in both cases, but conceptually it's describing a different situation. Keeping them separate makes the test suite easier to understand, and it can prevent test failures if the state of the store later needs to be different in those two different situations.

Data Layer

Now that we've implemented the loading indicator in the UI, we need to drive out the loading flag in the store itself. Open `src/store/restaurants.spec.js`. Our store test will have the same separation as our component test between the scenarios of starting loading and loading succeeding. Because of this, let's proactively group our existing "stores the restaurants" test in a describe:

```
describe('when loading succeeds', () => {
  it('stores the restaurants', async () => {
//...
  });
});
```

Now let's add a new group for while loading is happening, at the same level as "when loading succeeds":

```
describe('while loading', () => {
});
```

Inside that describe block, add the test:

```
it('sets a loading flag', () => {
  const api = {
    loadRestaurants: () => new Promise(() => {}),
  };

  const initialState = {};

  const store = createStore(
    restaurantsReducer,
    initialState,
    applyMiddleware(
      thunk.withExtraArgument(api),
    ),
  );

  store.dispatch(loadRestaurants());
  expect(store.getState().loading)
    .toEqual(true);
});
```

Here’s what’s going on:

- We define a stubbed API with a `loadRestaurants` method that returns a promise. The function passed to the promise never calls its arguments, so the promise never resolves or rejects. This is the best way to create the scenario we want for this test, because we want to test what happens *before* the promise resolves.
- We set up a store with the `restaurantsReducer` and our stubbed API.
- We dispatch the `loadRestaurants` async action. This time we don’t want to `await` it because we want to check the state of the store immediately upon the action returning, *before* it resolves. (And our promise will ensure it never resolves, anyway.)
- We check the store’s `loading` flag to confirm that it’s `true` after we initiate a load.

Our test fails, as we expect:

```
 □ restaurants › loadRestaurants action › while loading › sets a loading flag

    expect(received).toEqual(expected) // deep equality

    Expected: true
    Received: undefined
```

We’re getting back `undefined` because the `loading` flag isn’t even defined yet.

Following the principle of making the test green in the easiest way possible, we set up a `loading` child reducer of the restaurants reducer that always returns `true`:

```
};

function loading() {
  return true;
}

export default combineReducers({
  records,
  loading,
});
```

As before, we know this won't be our final implementation, but we want to write tests that drive us to handle all the scenarios.

When we save the file, the test passes.

So why is it not enough to have a `loading` flag that is always `true`? Well, we want it to be `false` once loading is complete. We already have another test that makes an assertion when loading is complete. Let's extract out the setup code we will need for both tests.

First, the `records` array is never modified by the test, so we can move it outside the test to the `describe`:

```
describe('when loading succeeds', () => {
  const records = [
    {id: 1, name: 'Sushi Place'},
    {id: 2, name: 'Pizza Place'},
  ];

  it('stores the restaurants', async () => {
~~    const records = [~~
~~      {id: 1, name: 'Sushi Place'},~~
~~      {id: 2, name: 'Pizza Place'},~~
~~    ];~~
    const api = {
```

Next, we will need to access the `store` from both tests, so make it a `let` variable defined outside the individual test:

```
];

let store;

it('stores the restaurants', async () => {
  const api = {
    loadRestaurants: () => Promise.resolve(records),
  };

  const initialState = {
    records: [],
  };

~~  const store = createStore(~~
  store = createStore(
    restaurantsReducer,
```

Finally, we move the code that sets up the `api`, the `store`, and dispatches the action to a `beforeEach` block:

```
let store;

beforeEach(async () => {
  const api = {
    loadRestaurants: () => Promise.resolve(records),
  };

  const initialState = {
    records: [],
  };

  store = createStore(
    restaurantsReducer,
    initialState,
    applyMiddleware(
      thunk.withExtraArgument(api),
    ),
  );
  await store.dispatch(loadRestaurants());
});

it('stores the restaurants', async () => {
  ~~const api = {~~
    ~~loadRestaurants: () => Promise.resolve(records),~~
  ~~};~~

  ~~const initialState = {~~
    ~~records: [],~~
  ~~};~~

  ~~store = createStore(~~
    ~~restaurantsReducer,~~
    ~~initialState,~~
    ~~applyMiddleware(~~
      ~~thunk.withExtraArgument(api),~~
    ~~),~~
  ~~);~~

  ~~await store.dispatch(loadRestaurants());~~

  expect(store.getState().records)
    .toEqual(records);
});
```

A `beforeEach` block will automatically run before every test in the surrounding `describe`. As an alternative to the `beforeEach` block, another option would have been to create a helper function that we explicitly call inside each test, as we did with `renderComponent` in our component test. There are pros and cons of each approach. React Testing Library discourages rendering a component in a `beforeEach` block, so we used a helper function for that case. But our store test is just plain JavaScript and has no such discouragement. One benefit of a `beforeEach` block is that you don't have to remember to call the same helper function in each test. Another benefit is that the block communicates "this is the situation that we are testing across all the tests in this group." Feel free to try both helper methods and `beforeEach` blocks in your non-component unit tests and decide when you prefer each.

Now that our setup code has been moved to a `beforeEach` block, we can simplify it a bit. Since the `beforeEach` function only has one promise it `awaits` and that promise is returned by the final statement, we can just return the promise instead of `await`ing it. If we do, the function no longer needs to be an `async` function:

```
~~beforeEach(async () => {~~
beforeEach(() => {
  const api = {
    loadRestaurants: () => Promise.resolve(records),
  };
...
~~  await store.dispatch(loadRestaurants());~~
  return store.dispatch(loadRestaurants());
});
```

Test Asynchrony and Undocumented Behavior

If you remove the `await` keyword but don't add the `return` keyword, you'll notice that the tests still pass. As of this writing, it seems like Jest is still waiting for the promise to resolve whether we return it or not. Why is that?

It has to do with the way Jest uses promises internally in running tests. This won't work for just any promise, though: for example, if our promise was really hitting an external service, or was set up with a `setTimeout()`, Jest would move on and not wait for the promise to resolve before running the tests.

The fact that Jest works this way is an implementation detail and isn't documented as a behavior you can rely on. This means it could change at any time without warning. It's safer to rely on the documented behavior[a] that if we want Jest to wait for a promise in a `beforeEach` block to resolve, we should return it. This is one way to make your tests as robust as possible, avoiding mysterious breakages in the future.

[a] https://jestjs.io/docs/en/setup-teardown#repeating-setup-for-many-tests

Our test function has the `async` keyword, but it turns out that all the asynchronous functionality was moved to the `beforeEach` block. As a result, we can remove the `async` keyword from the test function:

```
~~it('stores the restaurants', async () => {~~
it('stores the restaurants', () => {
  expect(store.getState().records)
    .toEqual(records);
});
```

With this refactoring done, we're ready to add a second test in that describe block: a test to check the loading state.

```
it('clears the loading flag', () => {
  expect(store.getState().loading)
    .toEqual(false);
});
```

Our test fails, as we expect, and now we need to actually clear the loading flag. We can do this in the `loading` reducer:

```
~~function loading() {~~
~~  return true;~~
function loading(state = true, action) {
  switch (action.type) {
    case STORE_RESTAURANTS:
      return false;
    default:
      return state;
  }
}
```

Save the file and our test passes.

Is our implementation complete? Well, the `loading` flag starts as `true`. That's *almost* correct, because right now we happen to dispatch the `loadRestaurants` action as soon as our app starts. But conceptually it makes more sense for the `loading` flag to start `false` and for the `loadRestaurants` action to set it to `true`. That way we aren't relying on the coincidence that we start the loading when the app starts, and if we ever change when the loading begins, our `loading` flag will be ready for that change.

We don't just want to make this change to the `loading` flag directly, though—we want to specify it! Add a new `describe` block directly inside the top-level "restaurants" block:

```
describe('initially', () => {
  it('does not have the loading flag set', () => {
    const initialState = {};

    const store = createStore(
      restaurantsReducer,
      initialState,
      applyMiddleware(thunk),
    );

    expect(store.getState().loading)
      .toEqual(false);
  });
});
```

In this case we don't need to pass an `api` to `thunk` at all, because we won't be making any calls to the `api`. We just create a store with empty initial state, then we assert that the starting value of `loading` is `false`.

Our test fails, as we expect. Let's change the initial `loading` flag:

```
~~function loading(state = true, action) {~~
function loading(state = false, action) {
  switch (action.type) {
```

Now our test of the initial state passes, but we get another test failure: the test for while loading is happening. Previously the reason the `loading` flag was `true` while loading was that that was the flag's initial value. Now that its initial value is `false`, we need to *change* the value when `loadRestaurants` is called. We can do this by adding a new action that is dispatched immediately in the `loadRestaurants` thunk, before the API is called:

```
export const START_LOADING = 'START_LOADING';
export const STORE_RESTAURANTS = 'STORE_RESTAURANTS';

export const loadRestaurants = () => async (dispatch, getState, api) => {
  dispatch(startLoading());
  const records = await api.loadRestaurants();
  dispatch(storeRestaurants(records));
};

const startLoading = () => ({type: START_LOADING});

const storeRestaurants = records => ({
```

Then, in the `loading` reducer, we return a state of `true` when `START_LOADING` happens:

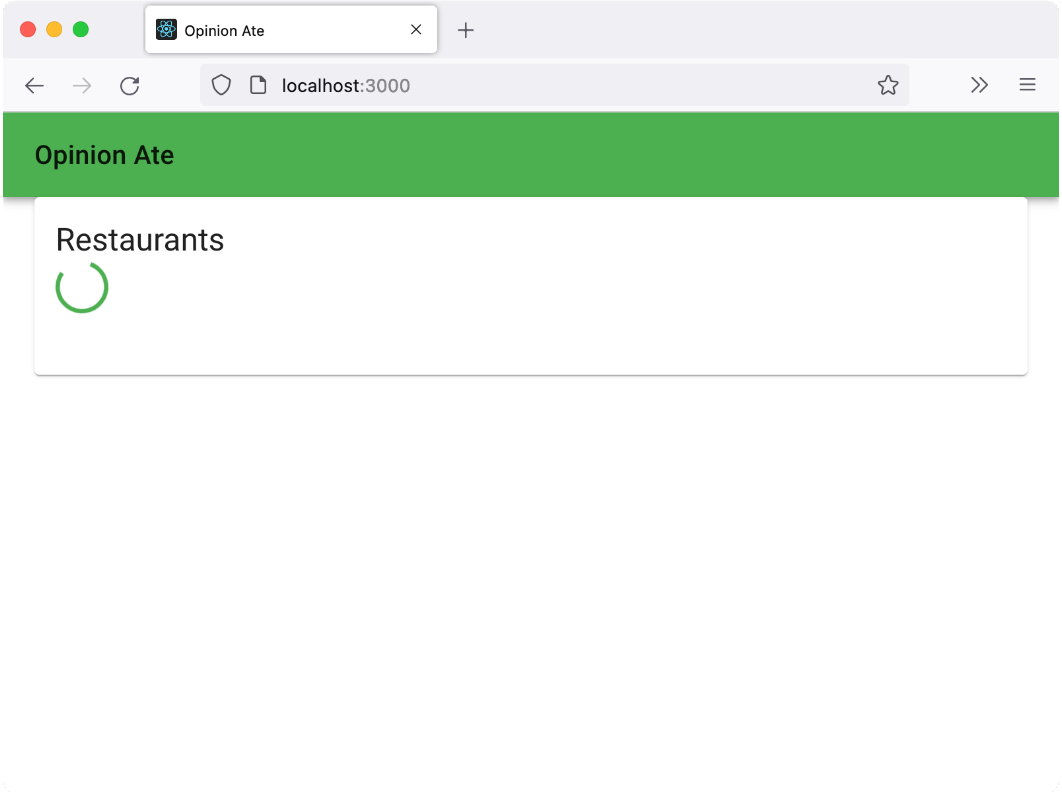
```
import {combineReducers} from 'redux';
~~import {STORE_RESTAURANTS} from './actions';~~
import {START_LOADING, STORE_RESTAURANTS} from './actions';
...
function loading(state = false, action) {
  switch (action.type) {
    case START_LOADING:
      return true;
    case STORE_RESTAURANTS:
      return false;
```

Save the files, and all our tests pass. We now have a loading flag that starts `false`, is set to `true` when loading begins, and is set back to `false` when loading ends.

Our unit tests are passing, and all we need to do now is hook up the `loading` state to our component. In `RestaurantList.js`:

```
const mapStateToProps = state => ({
  restaurants: state.restaurants.records,
  loading: state.restaurants.loading,
});
```

With this, our loading functionality should be complete. Run the app with `yarn start`, then load it in the browser. To help make it easy to see the loading state, our back-end is set up with a hard-coded one-second delay before returning the restaurant list. So regardless of your internet connection speed you should see the loading spinner for at least one second before the results appear. Our loading flag is working!



Restaurant list with loading spinner

Run our E2E tests and note that they still pass. They don't care whether or not a loading flag is shown; they just ensure that the restaurant names eventually appear.

Error Flag

The other edge case we want to handle is displaying an error message if the API call fails. This will be implemented using a very similar process to the loading flag. If you like, you can try to test-drive this functionality yourself, then compare the steps you took with the steps below. Remember to always start with a failing test, and write only the minimum code to pass the test!

Note that instead of implementing both flags in the component, then implementing both in the store, we got one flag working entirely before we moved on to the second. This ensures that we could ship the loading flag to our customers even before the error flag is ready.

Component Layer

Let's once again start with the test for the component. We are describing a new situation, when loading fails, so let's put our test in a new `describe` block:

```
describe('when loading fails', () => {
  it('displays the error message', () => {
    renderComponent({loadError: true});
    expect(
      screen.getByText('Restaurants could not be loaded.'),
    ).toBeInTheDocument();
  });
});
```

We decide we want to indicate the error state with a flag named `loadError`, so we set it up as a prop set to `true`. We check for a new error message on the page. Our test fails because the element is not found.

Fix it the simplest way possible by hard-coding the error message to show. MUI has an `Alert` component that will work well. Add it:

```
import {connect} from 'react-redux';
import Alert from '@mui/material/Alert';
import CircularProgress from '@mui/material/CircularProgress';
...
      {loading && <CircularProgress />}
      <Alert severity="error">Restaurants could not be loaded.</Alert>
      <List>
```

Save the file and our test passes. Now, specify that the error does *not* show when loading succeeds:

```
describe('when loading succeeds', () => {
  it('does not display the loading indicator while not loading', () => {
    renderComponent();
    expect(screen.queryByRole('progressbar'))
      .not.toBeInTheDocument();
  });

  it('does not display the error message', () => {
    renderComponent();
    expect(screen.queryByText('Restaurants could not be loaded.'))
      .not.toBeInTheDocument();
  });

  it('displays the restaurants', () => {
```

Make this test pass by making the display of the error alert conditional on the `loadError` prop that we set up in our test:

```
export function RestaurantList({
  loadRestaurants,
  restaurants,
  loading,
  loadError,
}) {
  useEffect(() => {
...
      {loading && <CircularProgress />}
      {loadError && (
        <Alert severity="error">Restaurants could not be loaded.</Alert>
      )}
      <List>
```

Now both tests pass. Our component is working; on to the store.

Store Layer

In `restaurants.spec.js`, create a new `describe` block after "when loading succeeds" for the error scenario. Let's go ahead and do the setup in a `beforeEach` block, assuming we will need to have other expectations too:

```
describe('when loading fails', () => {
  let store;

  beforeEach(() => {
    const api = {
      loadRestaurants: () => Promise.reject(),
    };

    const initialState = {};

    store = createStore(
      restaurantsReducer,
      initialState,
      applyMiddleware(
        thunk.withExtraArgument(api),
      ),
    );

    return store.dispatch(loadRestaurants());
  });

  it('sets an error flag', () => {
    expect(store.getState().loadError)
      .toEqual(true);
  });
});
```

In writing this test we had to decide how our `api` would report that an HTTP request failed. We decided that in this case the promise `api.loadRestaurants()` returns will reject. This is common practice for JavaScript HTTP clients.

When we run our test, it fails with the following error:

```
thrown: undefined
```

This means that there was an error thrown that as not caught. We configured our `api.loadRestaurants()` stub to throw an error, but our async `loadRestaurants` function doesn't catch it. Since it's a good practice to handle promise rejections in general, let's set up our action to catch a rejected promise. We won't do anything with the catch for now; maybe our tests will drive us to do something with it later.

```
export const loadRestaurants = () => async (dispatch, getState, api) => {
  try {
    dispatch(startLoading());
    const records = await api.loadRestaurants();
    dispatch(storeRestaurants(records));
  } catch {}
};
```

This fixes the error, and now we see a failing expectation:

```
  □ restaurants › loadRestaurants action › when loading fails › sets an error
    flag

    expect(received).toEqual(expected) // deep equality

    Expected: true
    Received: undefined

       96 |       it('sets an error flag', async () => {
    >  97 |         expect(store.getState().loadError)
       98 |           .toEqual(true);
          |            ^
          |       });
```

Our `loadError` flag contains an `undefined` value because we haven't defined it yet.

We fix this failing test the simplest way possible, adding a `loadError` reducer that always returns `true`:

```
    default:
      return state;
  }
}

function loadError() {
  return true;
}

export default combineReducers({
  records,
  loading,
  loadError,
});
```

Save and the tests pass.

Now we specify that the `loadError` flag should actually start out as `false`, and *only* be set to true upon a failing load. First, extract the setup in the "initially" block:

```
describe('initially', () => {
  let store;

  beforeEach(() => {
    const initialState = {};

    store = createStore(
      restaurantsReducer,
      initialState,
      applyMiddleware(thunk),
    );
  });

  it('does not have the loading flag set', () => {
~~    const initialState = {};~~

~~    const store = createStore(~~
~~      restaurantsReducer,~~
~~      initialState,~~
~~      applyMiddleware(thunk),~~
~~    );~~

    expect(store.getState().loading)
      .toEqual(false);
  });
});
```

Then add a test for the `loadError` flag:

```
describe('initially', () => {
...
  it('does not have the loading flag set', () => {
    expect(store.getState().loading)
      .toEqual(false);
  });

  it('does not have the error flag set', () => {
    expect(store.getState().loadError)
      .toEqual(false);
  });
});
```

The test fails. Make it pass while keeping the other tests passing by setting `loadError` to an initial value of `false`, then setting it to true when a new loading error action is dispatched. In `actions.js`:

```
export const START_LOADING = 'START_LOADING';
export const STORE_RESTAURANTS = 'STORE_RESTAURANTS';
export const RECORD_LOADING_ERROR = 'RECORD_LOADING_ERROR';

export const loadRestaurants = () => async (dispatch, getState, api) => {
  try {
    dispatch(startLoading());
    const records = await api.loadRestaurants();
    dispatch(storeRestaurants(records));
~~  } catch {}~~
  } catch {
    dispatch(recordLoadingError());
  }
};

const startLoading = () => ({type: START_LOADING});

const storeRestaurants = records => ({
  type: STORE_RESTAURANTS,
  records,
});
```

```
const recordLoadingError = () => ({type: RECORD_LOADING_ERROR});
```

And in `reducers.js`:

```
~~import {START_LOADING, STORE_RESTAURANTS} from './actions';~~
import {
  RECORD_LOADING_ERROR,
  START_LOADING,
  STORE_RESTAURANTS,
} from './actions';
...
~~function loadError() {~~
~~  return true;~~
function loadError(state = false, action) {
  switch (action.type) {
    case RECORD_LOADING_ERROR:
      return true;
    default:
      return state;
  }
}
```

Save the file and all tests should pass.

We also want to make sure that if the restaurants are loaded again later, the error flag is cleared out, since a new request is being made. This test should go in the "loadRestaurants action > while loading" group, so extract the setup from the "sets a loading flag" test:

```
describe('while loading', () => {
  let store;

  beforeEach(() => {
    const api = {
      loadRestaurants: () => new Promise(() => {}),
    };

    const initialState = {};
    store = createStore(
      restaurantsReducer,
      initialState,
      applyMiddleware(
        thunk.withExtraArgument(api),
      ),
    );

    store.dispatch(loadRestaurants());
  });

  it('sets a loading flag', () => {
    ~~const api = {~~
      ~~loadRestaurants: () => new Promise(() => {}),~~
    ~~};~~

    ~~const initialState = {};~~

    ~~const store = createStore(~~
      ~~restaurantsReducer,~~
      ~~initialState,~~
      ~~applyMiddleware(~~
        ~~thunk.withExtraArgument(api),~~
      ~~),~~
    ~~);~~

    ~~store.dispatch(loadRestaurants());~~

    expect(store.getState().loading)
      .toEqual(true);
  });
});
```

Save and the tests should still pass.

Now we're finally ready to set up our expectation that the `loadError` should be reset to `false` after starting a load operation. To represent this scenario, update the `initialState` in our `beforeEach` block to set `loadError` to `true`:

```
const api = {
  loadRestaurants: () => new Promise(() => {}),
};

~~const initialState = {};~~
const initialState = {loadError: true};

store = createStore(
```

Next, add the following test after the “sets a loading flag” test:

```
it('clears the error flag', () => {
  expect(store.getState().loadError)
    .toEqual(false);
});
```

Save the file and the new test should fail. Fix it by updating the `loadError` reducer to return `false` upon the `START_LOADING` action:

```
function loadError(state = false, action) {
  switch (action.type) {
    case START_LOADING:
      return false;
    case RECORD_LOADING_ERROR:
      return true;
    default:
      return state;
  }
}
```

Now that we are handling the error state, there’s one more bit of functionality we could add: currently the `loading` flag is not cleared when the request errors. Let’s add a test for that:

```
describe('when loading fails', () => {
...
  it('sets an error flag', () => {
    expect(store.getState().loadError)
      .toEqual(true);
  });

  it('clears the loading flag', () => {
    expect(store.getState().loading)
      .toEqual(false);
  });
});
```

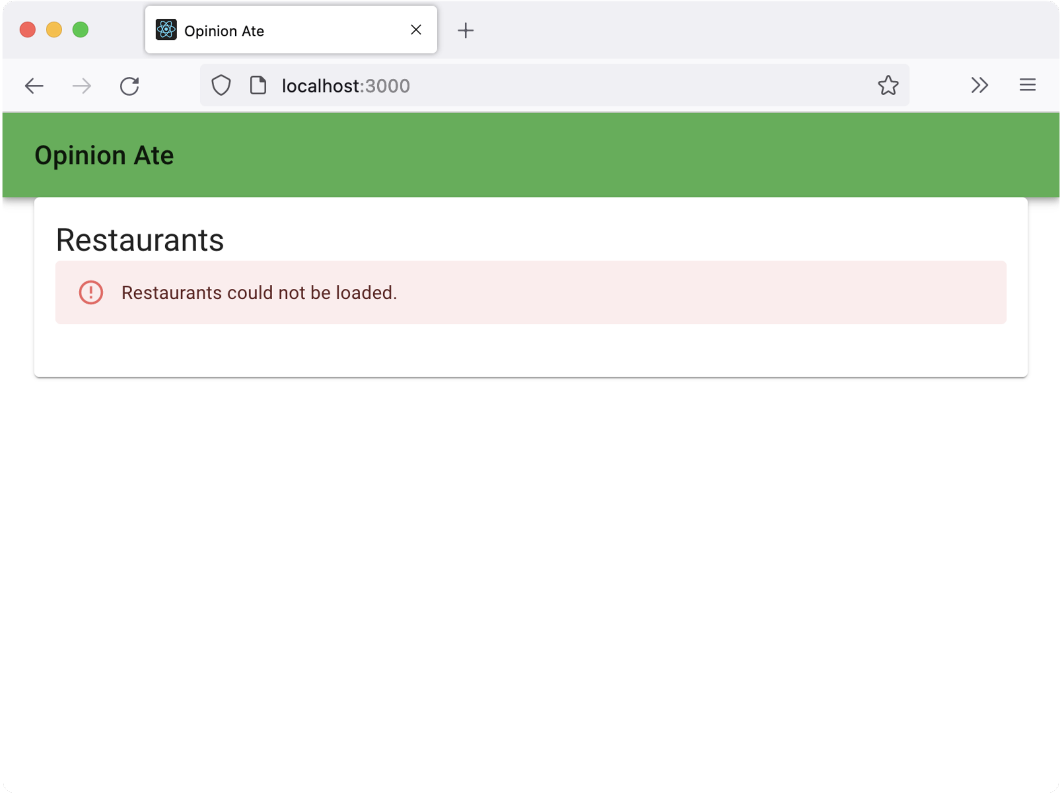
To make it pass, just return `false` from the `loading` reducer upon `RECORD_LOADING_ERROR`:

```
function loading(state = false, action) {
  switch (action.type) {
    case START_LOADING:
      return true;
    case STORE_RESTAURANTS:
    case RECORD_LOADING_ERROR:
      return false;
    default:
      return state;
  }
}
```

With this, all our tests are passing. Our code has error state functionality added, and now we just need to wire up our `RestaurantList` component to our new Redux state value:

```
const mapStateToProps = state => ({
  restaurants: state.restaurants.records,
  loading: state.restaurants.loading,
  loadError: state.restaurants.loadError,
});
```

To see this error state in action, we need to force the API requests in our running app to fail. Let's do that by putting in an incorrect API key. In `src/api.js`, in the `baseURL` property for the Axios instance, change the API key to an incorrect value. This will result in the server returning a 404 Not Found response code. Reload the web app and you should see a nice red "Restaurants could not be loaded" error box.



Opinion Ate

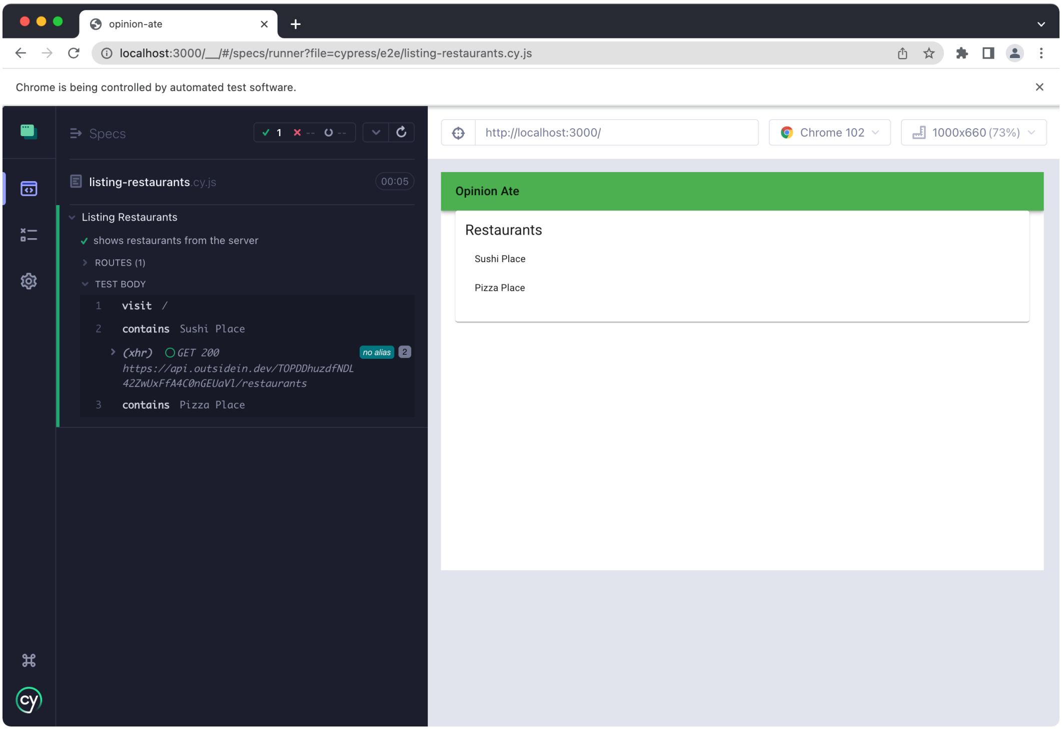
Restaurants

Restaurants could not be loaded.

Loading error message

Restore the correct API key value, then reload the page. You should see the loading spinner, then our results.

Run the E2E test one more time to make sure it's still passing—it should be.


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/listing-restaurants.cy.js
Chrome is being controlled by automated test software.
Specs
✓ 1 ✕ -- ↻ --
listing-restaurants.cy.js
00:05
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place
Pizza Place

```
Listing Restaurants
  ✓ shows restaurants from the server
    ROUTES (1)
    TEST BODY
    1  visit  /
    2  contains  Sushi Place
       (xhr)  ○ GET 200                         no alias  2
       https://api.outsidein.dev/TOPDDhuzdfNDL
       42ZwUxFfA4C0nGEUaVl/restaurants
    3  contains  Pizza Place
```

Cypress test passing with MUI design

With that, our edge case functionality is done. If you have any uncommitted changes, commit them to git. Push up your branch to the origin and open a pull request. Wait for CI to complete, then merge the pull request. Now we can drag our "Show Loading and Error States" story to "Done" in Trello.

What's Next

With these edge cases handled, our restaurant-loading functionality is complete. We've handled main cases, edge cases, and made it look good. In the next chapter we'll take everything we've learned and apply it to a second feature: creating a new restaurant.

10. Writing Data

In this chapter we'll move on to our next new feature. We'll follow the process of outside-in TDD once again, with an outer and inner red-green-refactor loop. We'll also see how to tackle some of the unique challenges that arise when testing forms and when saving data to an API.

This work is represented by the next story in Trello, "Add Restaurants". Drag it to "In Progress".

When we did the "Load Restaurants" story we saved the edge cases for a separate story, as a way to take smaller steps. Now that we have some practice with these techniques, we'll handle both the main functionality and edge cases under the same story.

Main Functionality

Create a new branch for this story:

```
$ git checkout -b creating-a-restaurant
```

It's been a while since we wrote a new E2E test because the last few chapters were all focused on a single user-facing features. But now that we're building a new feature, we follow the outside-in TDD loop, starting with creating an E2E test that specifies that feature.

End-to-End Test

Create a file `cypress/e2e/creating-a-restaurant.cy.js` and add the following:

```
describe('Creating a Restaurant', () => {
  it('allows adding restaurants', () => {
    const restaurantId = 27;
    const restaurantName = 'Sushi Place';

    cy.intercept('GET', 'https://api.outsidein.dev/*/restaurants', []);

    cy.intercept('POST', 'https://api.outsidein.dev/*/restaurants', {
      id: restaurantId,
      name: restaurantName,
    }).as('addRestaurant');

    cy.visit('/');

    cy.get('[placeholder="Add Restaurant"]').type(restaurantName);
    cy.contains('Add').click();

    cy.wait('@addRestaurant')
      .its('request.body')
      .should('deep.equal', {name: restaurantName});

    cy.contains(restaurantName);
  });
});
```

As in our previous E2E test, we stub the GET request to load the restaurants. This time we don't need any pre-existing restaurants for the test, so we return an empty array.

We also configure Cypress to intercept a POST request, which is the request we'll use to create a restaurant. From that request, we return an object containing the data for the new restaurant that is created. After we call `cy.intercept()` we chain on a call to `.as()`, which allows us to give the request the name `addRestaurant`. We'll see why this name is useful in a moment.

We visit the home page, and this time we interact with the page:

- We find an element with a placeholder of "Add Restaurant" (so, presumably a text input), and we type a restaurant name into it.
- We find an element "Add" and click it.

Next, we call `cy.wait()`, which waits for an HTTP request to be sent. We pass the name of the request we want to wait for, prepending an `@` to it—our `addRestaurant` request in this case. The reason we call `cy.wait()` here is to make an assertion on the request that was sent: checking that the restaurant name is correctly sent in the body of the request. In Cypress, this kind of check is done with a declarative approach: chaining the `.its()` method call to retrieve the property `'request.body'`, then chaining the `.should()` method call to do a `'deep.equal'` comparison. (To learn more, check out Cypress's docs[24] for `its()`, `should()`, and BDD Assertions.)

The reason we're making an assertion on the request body is because, for API calls that change data on the back-end, it's not enough to stub the request to return the data the application expects. If the application sent the wrong data to the back-end then it would not be saved correctly, so we need to confirm our app is sending the *right* data.

Don't Fear `cy.wait()`

If you've used other E2E testing tools, you may have bad memories of `wait` statements. Older test tools require you to `wait` to prevent timing issues, and those `wait` statements clutter up your test, slow it down, and can introduce flake. Cypress doesn't need `wait` statements for any of those reasons: Cypress will automatically retry commands up to a configured timeout to give network requests, animations, and other operations time to finish. As a result, Cypress tests tend to be less cluttered, faster, and more robust than those of older E2E tools.

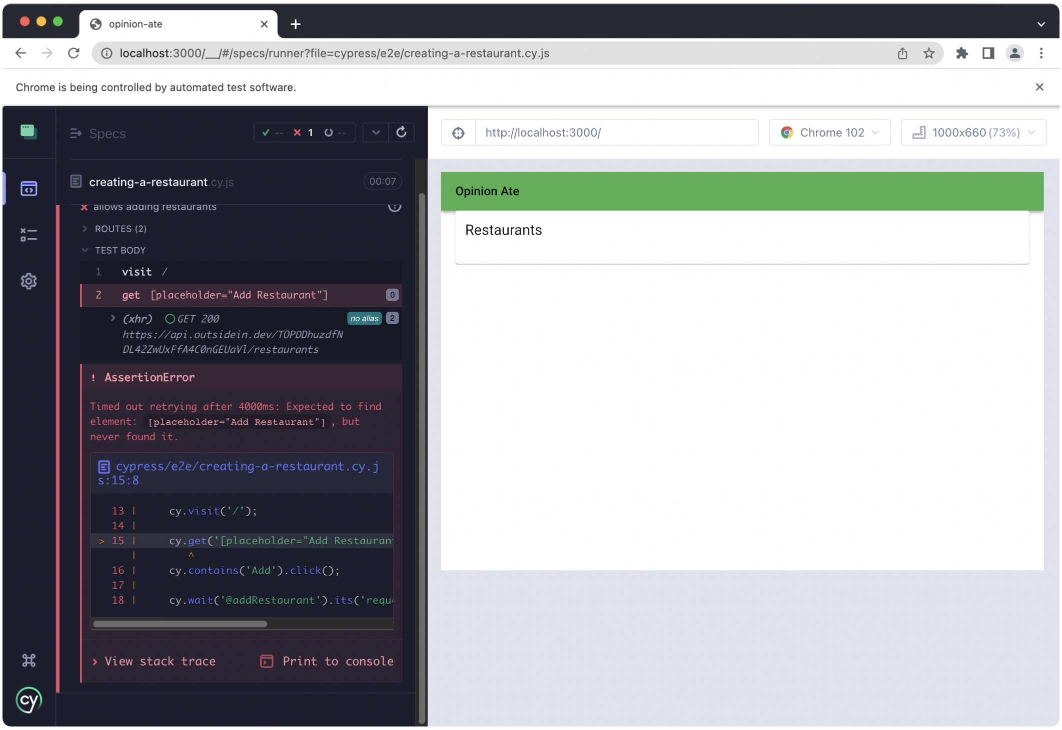
In this case, for example, even if we removed the `cy.wait()` statement, the `cy.contains(restaurantName)` check would still succeed even if it takes a few seconds for a network request to finish. We don't need `cy.wait()` to give the network request time; we only need it to get access to the request to make an assertion, as described above.

Finally, we confirm that the restaurant name is shown on the page, showing that the restaurant has been added to the list.

Start your app with `yarn start`, then start Cypress with `yarn cypress`. Click "E2E Testing" then "Start E2E Testing in Chrome", then `creating-a-restaurants.cy.js`. It fails, showing the first bit of functionality we need to implement:

> Timed out retrying after 4000ms: Expected to find element: [placeholder="Add Restaurant"], but never found it.

[24] https://docs.cypress.io



opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✗ 1 --
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants

creating-a-restaurant.cy.js 00:07

✗ allows adding restaurants

ROUTES (2)

TEST BODY

```
1  visit  /
2  get  [placeholder="Add Restaurant"]                        0
   (xhr) ○ GET 200                                  no alias  2
   https://api.outsidein.dev/TOPDDhuzdfN
   DL42ZwUxFfA4C0nGEUaVl/restaurants
```

```
! AssertionError

Timed out retrying after 4000ms: Expected to find
element: [placeholder="Add Restaurant"] , but
never found it.

cypress/e2e/creating-a-restaurant.cy.j
s:15:8

  13 |     cy.visit('/');
  14 |
> 15 |     cy.get('[placeholder="Add Restauran
     |        ^
  16 |     cy.contains('Add').click();
  17 |
  18 |     cy.wait('@addRestaurant').its('requ

> View stack trace          Print to console
```

Cypress test where the text field is not found

We need an “Add Restaurant” text input. What component should it be in? We discussed in “Vertical Slice” that `RestaurantScreen` would hold both the restaurant list and new restaurant form. The text input should live in the New Restaurant Form, so it’s time to create that component.

Create the file `src/components/NewRestaurantForm.js`, and add the following:

```
import TextField from '@mui/material/TextField';

export function NewRestaurantForm() {
  return (
    <form>
      <TextField placeholder="Add Restaurant" fullWidth variant="filled" />
    </form>
  );
}
```

```
export default NewRestaurantForm;
```

Note a few things:

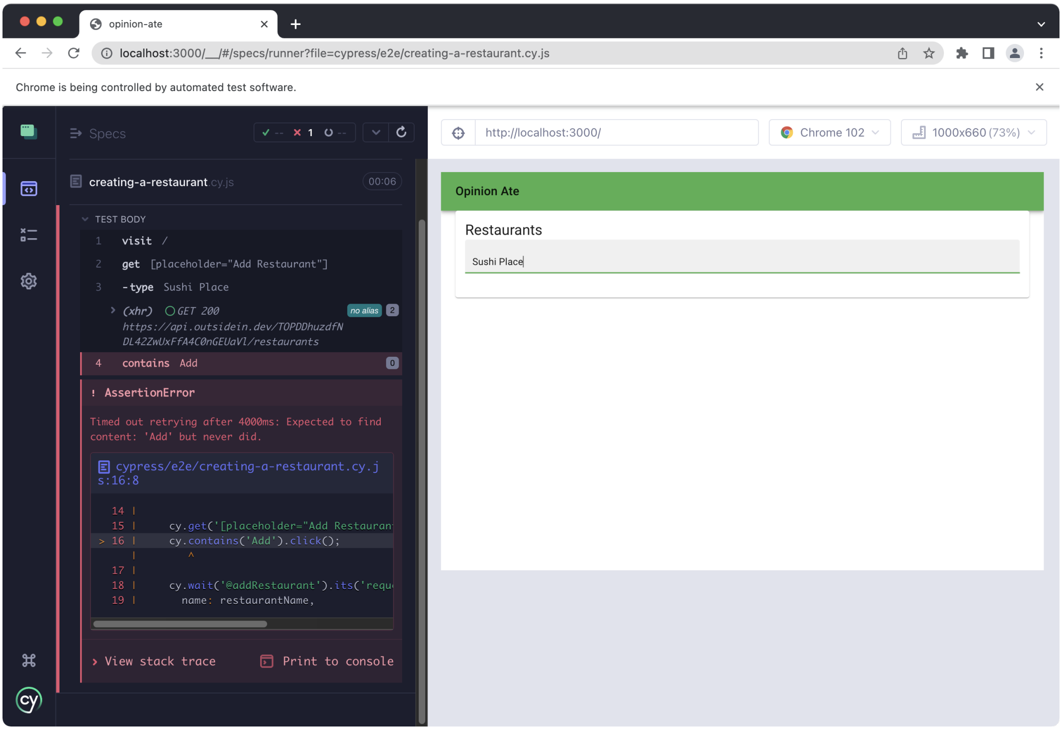
- We're using MUI's `TextField` component instead of a plain `<input type="text" />`
- We're exporting the form both as a named and default export. This is because we'll be connecting the default export to Redux, and we know we'll want the unconnected component for testing

Next, add the form to the `RestaurantScreen` component:

```
import RestaurantList from './RestaurantList';
import NewRestaurantForm from './NewRestaurantForm';

export default function RestaurantScreen() {
  return (
    <Card>
      <CardContent>
        <Typography variant="h5">Restaurants</Typography>
        <NewRestaurantForm />
        <RestaurantList />
```

Rerun the E2E tests and they should successfully find and type into the "Add Restaurant" input. The next error is:

> Timed out retrying after 4000ms: Expected to find content: 'Add' but never did.


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ↻ --
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place

creating-a-restaurant.cy.js 00:06

TEST BODY

```
1  visit  /
2  get  [placeholder="Add Restaurant"]
3  -type  Sushi Place
   (xhr)  ○ GET 200  https://api.outsidein.dev/TOPDDhuzdfNDL42ZwUxFfA4C0nGEUaVl/restaurants   no alias 2
4  contains  Add   0
```

```
! AssertionError

Timed out retrying after 4000ms: Expected to find
content: 'Add' but never did.

cypress/e2e/creating-a-restaurant.cy.js:16:8

  14 |
  15 |     cy.get('[placeholder="Add Restauran
> 16 |     cy.contains('Add').click();
     |        ^
  17 |
  18 |     cy.wait('@addRestaurant').its('requ
  19 |       name: restaurantName,
```

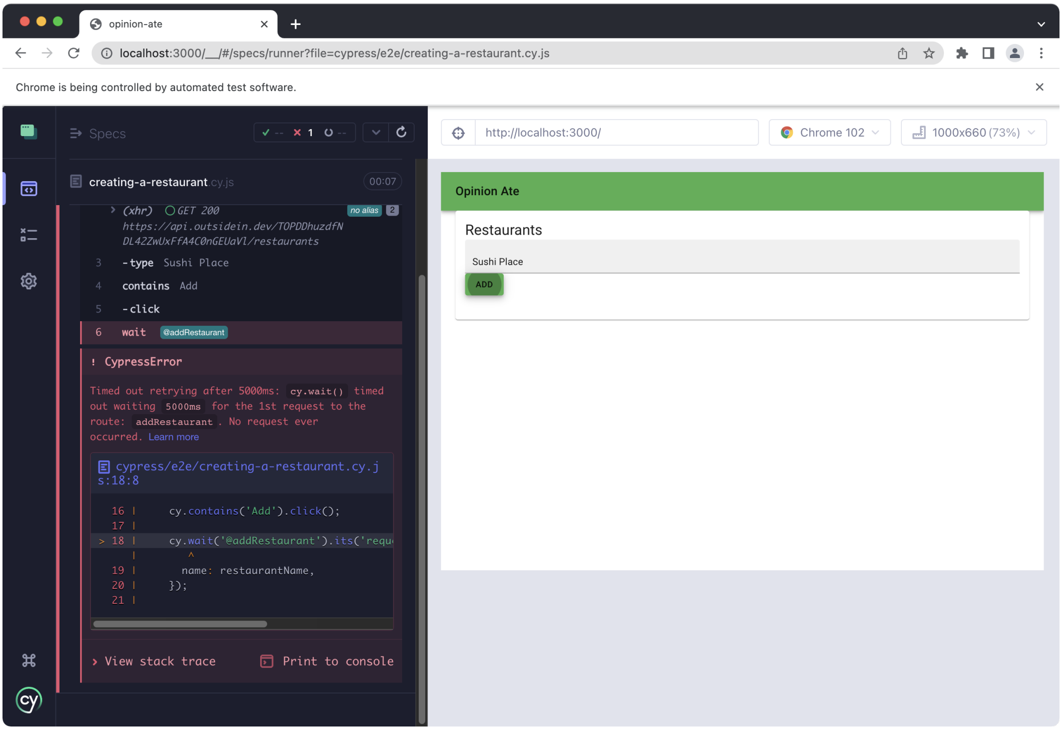
Cypress test where the button is not found

To fix this error, we add a button to `NewRestaurantForm` but don't wire it up to anything yet:

```
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';

export function NewRestaurantForm() {
  return (
    <form>
      <TextField placeholder="Add Restaurant" fullWidth variant="filled" />
      <Button variant="contained" color="primary">
        Add
      </Button>
    </form>
  );
```

Rerun the E2E tests and we get this failure:

> Timed out retrying after 5000ms: cy.wait() timed out waiting 5000ms for the 1st request to the route: 'addRestaurant'. No request ever occurred.


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ○ --
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place
ADD

```
creating-a-restaurant.cy.js                         00:07
   > (xhr)  ○ GET 200                         no alias  2
     https://api.outsidein.dev/TOPDDhuzdfN
     DL42ZwUxFfA4C0nGEUaVl/restaurants
  3  -type  Sushi Place
  4  contains  Add
  5  -click
  6  wait  @addRestaurant

! CypressError

Timed out retrying after 5000ms: cy.wait() timed
out waiting 5000ms for the 1st request to the
route: addRestaurant. No request ever
occurred. Learn more

cypress/e2e/creating-a-restaurant.cy.js:18:8

  16 |     cy.contains('Add').click();
  17 |
> 18 |     cy.wait('@addRestaurant').its('requ
     |        ^
  19 |       name: restaurantName,
  20 |     });
  21 |

> View stack trace          Print to console
```

Cypress test where cy.wait() times out

So now we need to send the request is our back-end service. This is missing logic, so we will want to step down to unit tests to add it. How will it work?

- The `NewRestaurantForm` component will dispatch an asynchronous Redux action.
- The action will call a function in our API client.
- The API client will make an HTTP `POST` request.

Unit Testing the Component

Starting from the outside as usual, we'll start with the `NewRestaurantForm` component. We want to reproduce the E2E test's failure at the unit level. We should specify that when you click the send button a function prop is called—a function which in production will be wired to an action in our store. Now, the E2E test failure didn't tell us that we need to send along the restaurant name entered in the form, but we can go ahead and specify that, too.

Create the file `src/components/NewRestaurantForm.spec.js` and start out by setting up the component and a mock function in a `renderComponent` helper function:

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

describe('NewRestaurantForm', () => {
  const restaurantName = 'Sushi Place';

  let user;
  let createRestaurant;

  function renderComponent() {
    user = userEvent.setup();
    createRestaurant = jest.fn().mockName('createRestaurant');
    render(<NewRestaurantForm createRestaurant={createRestaurant} />);
  }
});
```

Notice the use of `userEvent`. This component involves user interaction, and `userEvent` will help us simulate it. We call `userEvent.setup()` to get a `user` object we can use for interactions.

Next, let's try to proactively organize our test file. Since we're taking the approach of having one behavior per test, it's likely that we will ultimately have multiple tests for each situation. So let's group situations with a `describe` block with a situation-specific setup helper function, even there there will only be one expectation at first. Add the following:

```
describe('when filled in', () => {
  async function fillInForm() {
    renderComponent();
    await user.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    await user.click(screen.getByText('Add'));
  }

  it('calls createRestaurant with the name', async () => {
    await fillInForm();
    expect(createRestaurant)
      .toHaveBeenCalledWith(restaurantName);
  });
});
```

We describe the situation when the form is filled in. We enter a restaurant name into a text field, then click the submit button. Note that all methods on `user` require `await`ing.

Note that we now have two levels of helper functions: our test calls `fillInForm()`, which in turn calls `renderComponent()`. After rendering is complete, `fillInForm()` simulates the user actions of filling in the form, then our test runs an expectation. We'll see below that other `describe` blocks will have other helper functions, each of which calls `renderComponent()`. In cases like this with nontrivial setup, using explicit helper functions instead of `beforeEach` blocks can make it easier to follow what's going on.

In `RestaurantList` we didn't pass any arguments to our function prop, so all we had to confirm was that it was called. But in this test we need to ensure the restaurant name is passed as an argument to the action function, and we can do that with the `.toHaveBeenCalledWith()` matcher. We pass one argument to it, confirming that the correct `restaurantName` is passed through.

Save the file and we get an assertion failure:

```
□ NewRestaurantForm › when filled in › calls createRestaurant with the name

  expect(createRestaurant)
    .toHaveBeenCalledWith(...expected)

  Expected: "Sushi Place"

  Number of calls: 0

    28 |       await fillInForm();
  > 29 |       expect(createRestaurant)
    30 |         .toHaveBeenCalledWith(restaurantName);
       |          ^
```

The test failure reports the action wasn't called at all. This is because our button isn't currently hooked up to anything. The typical way to set this up in HTML forms is to make the button a `submit` button, so it submits the form:

```
<form>
  <TextField placeholder="Add Restaurant" fullWidth variant="filled" />
  ~~<Button variant="contained" color="primary">~~
  <Button type="submit" variant="contained" color="primary">
    Add
  </Button>
</form>
```

Now, write just enough production code to get past the current test failure, let's just call the action without any arguments:

```
import Button from '@mui/material/Button';

~~export function NewRestaurantForm() {~~
export function NewRestaurantForm({createRestaurant}) {
  return (
    ~~<form>~~
    <form onSubmit={() => createRestaurant()}>
      <TextField placeholder="Add Restaurant" fullWidth variant="filled" />
```

We set up an `onSubmit` prop for the form tag, passing an arrow function that calls `createRestaurant`. Why not just pass the `createRestaurant` function to `onSubmit` directly? The reason is that React passes the browser event object as an argument to the `onSubmit` function. We don't want that argument to be passed, and in fact our `.toHaveBeenCalledWith()` matcher would detect that argument, fail, and show it to us in verbose error output. By wrapping `createRestaurant` in an arrow function we can ensure it's called with no arguments.

Save the file and now, in addition to the test failure, we get this error outputted:

```
Error: Not implemented: HTMLFormElement.prototype.submit
    at module.exports
```

It isn't obvious what's going on at first glance, but the problem is that the HTML form is attempting use the default browser mechanism of submitting to the server and refreshing the page. (The reason for this default behavior is that HTML forms predate using JavaScript to make HTTP requests.)

To prevent this default form submission behavior, we need to call the `preventDefault()` method on the event sent to the `onSubmit` event. We can do this by extracting a handler function:

```
export function NewRestaurantForm({createRestaurant}) {
  function handleSubmit(e) {
    e.preventDefault();
    createRestaurant();
  }

  return (
    ~~<form onSubmit={() => createRestaurant()}>~~
    <form onSubmit={handleSubmit}>
      <TextField placeholder="Add Restaurant" fullWidth variant="filled" />
```

Save the file and the error is gone, and now we just have the test failure:

```
  □ NewRestaurantForm › when filled in › calls createRestaurant with the name

    expect(createRestaurant)
      .toHaveBeenCalledWith(...expected)

    Expected: "Sushi Place"
    Received: called with 0 arguments

    Number of calls: 1
```

Now the `createRestaurant` function is successfully called—note the "Number of calls: 1". The problem is that the function didn't receive the argument it expected: it wanted "Sushi Place", but it didn't receive any arguments.

To pass the restaurant name to `createRestaurant`, first we're going to need to set up a state item to track that name:

```
import {useState} from 'react';
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';

export function NewRestaurantForm({createRestaurant}) {
  const [name, setName] = useState('');

  function handleSubmit(e) {
```

Then, we'll make `TextField` a controlled component, reading its value from the `name` state item and writing changes back using `setName` in the normal React way:

```
return (
  <form onSubmit={handleSubmit}>
~~    <TextField placeholder="Add Restaurant" fullWidth variant="filled" />~~
    <TextField
      value={name}
      onChange={e => setName(e.target.value)}
      placeholder="Add Restaurant"
      fullWidth
      variant="filled"
    />
   <Button type="submit" variant="contained" color="primary">
```

Finally, now that the entered text is stored in the `name` variable, we'll pass that as the argument to `createRestaurant()`:

```
function handleSubmit(e) {
  e.preventDefault();
~~  createRestaurant();~~
  createRestaurant(name);
};
```

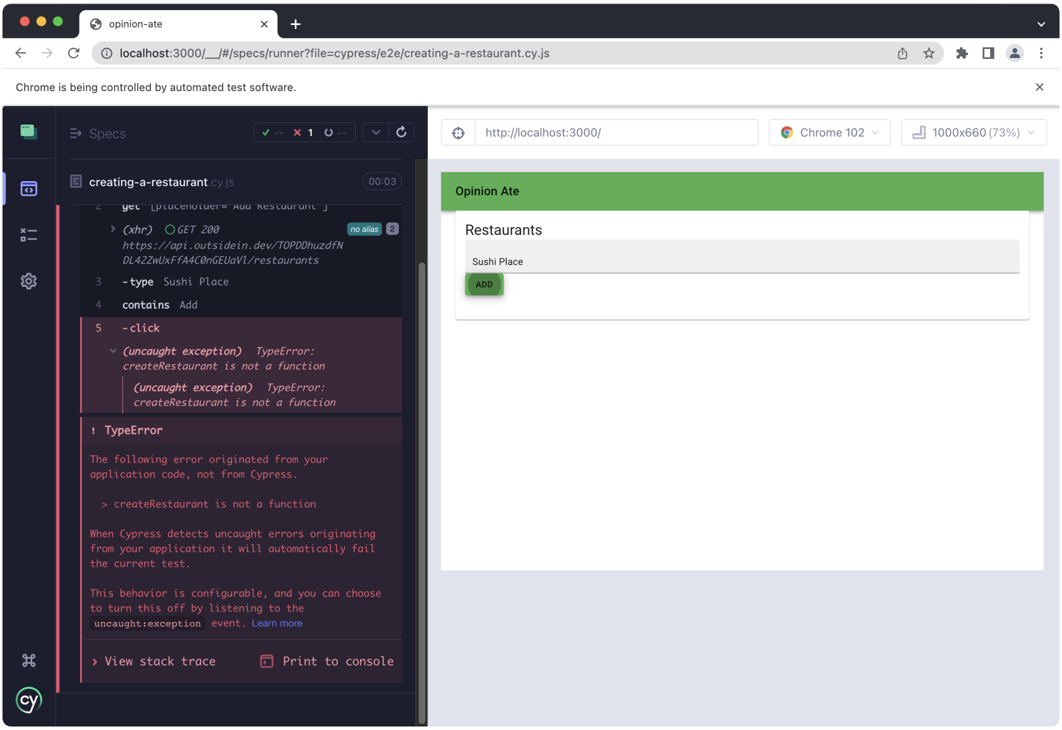
Save the file and the test passes.

Stepping Back Up

We'll want to add some more edge case functionality to the form at some point, but not right now. Remember the two-level outside-in TDD loop: we've finished test-driving the functionality the E2E test drove us to, so it's time to step back up to the E2E test to see what functionality it directs us to implement next.

Rerun the E2E test and you should see the following failure:

> TypeError: createRestaurant is not a function



```
creating-a-restaurant.cy.js                     00:03
2   get [placeholder= Add Restaurant ]
    > (xhr)  GET 200                               no alias  2
      https://api.outsidein.dev/TOPDDhuzdfN
      DL42ZwUxFfA4C0nGEUaVl/restaurants
3   -type  Sushi Place
4   contains  Add
5   -click
    (uncaught exception)  TypeError:
    createRestaurant is not a function
      (uncaught exception)  TypeError:
      createRestaurant is not a function

! TypeError
The following error originated from your
application code, not from Cypress.

  > createRestaurant is not a function

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

> View stack trace          Print to console
```

opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✗ 1 --
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place
ADD

Cypress test failure because createRestaurant is not a function

`createRestaurant` is not defined because we aren't passing it in to `NewRestaurantForm` as a prop. This is just a structural error, not a logic error, so let's fix this error directly instead of stepping down to a unit test yet.

We want the `createRestaurant` prop to be provided by Redux and correspond to a `createRestaurant` action that we don't have yet. Let's write the code we wish we had, importing that `createRestaurant` action. We'll direct Redux to connect it to our component, then we'll make that connected component the default export:

```
import {useState} from 'react';
import {connect} from 'react-redux';
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';
import {createRestaurant} from '../store/restaurants/actions';

export function NewRestaurantForm({createRestaurant}) {
...
}

~~export default NewRestaurantForm;~~
const mapStateToProps = null;
const mapDispatchToProps = {createRestaurant};

export default connect(mapStateToProps, mapDispatchToProps)(NewRestaurantForm);
```

Next we need to define that `createRestaurant` action that we wish we had. Add it in `src/store/restaurants/actions.js`. Because we know this will be an async action, we can go ahead and implement it as a thunk:

```
export const createRestaurant = () => () => {};
```

Rerun the E2E test, and we're back to the error we saw previously:

> Timed out retrying after 5000ms: cy.wait() timed out waiting 5000ms for the 1st request to the route: 'addRestaurant'. No request ever occurred.

Now our component is correctly calling our `createRestaurant` async action, but that function isn't doing anything. We need it to make the appropriate call to the API, then dispatch an action that results in the reducer adding the new restaurant to the list. That's a logic error, so it's time to step down to a unit test to drive out our store functionality.

Unit Testing the Store

In `src/store/restaurants.spec.js`, below the "loadRestaurants action" group, add a "createRestaurant action" group, and write a test to confirm the API is called:

```
describe('createRestaurant action', () => {
  const newRestaurantName = 'Sushi Place';

  let api;
  let store;

  beforeEach(() => {
    api = {
      createRestaurant: jest.fn().mockName('createRestaurant'),
    };

    const initialState = {};

    store = createStore(
      restaurantsReducer,
      initialState,
      applyMiddleware(
        thunk.withExtraArgument(api),
      ),
    );
  });

  it('saves the restaurant to the server', () => {
    store.dispatch(
      createRestaurant(newRestaurantName));
    expect(api.createRestaurant)
      .toHaveBeenCalledWith(newRestaurantName);
  });
});
```

We'll need to add a second expectation shortly so we go ahead and put the setup in a `beforeEach`.

We also need to import `createRestaurant`:

```
import restaurantsReducer from './restaurants/reducers';
~~import {loadRestaurants} from './restaurants/actions';~~
import {loadRestaurants, createRestaurant} from './restaurants/actions';

describe('restaurants', () => {
```

Save the file, and the test fails because the API method was not called:

```
  □ restaurants › createRestaurant action › saves the restaurant to the server

    expect(createRestaurant)
      .toHaveBeenCalledWith(...expected)

    Expected: "Sushi Place"

    Number of calls: 0
```

Update the `createRestaurant` thunk to call it:

```
~~export const createRestaurant = () => () => {};~~
export const createRestaurant = () => async (dispatch, getState, api) => {
  await api.createRestaurant();
};
```

This changes the test failure. Now the method is called, but not with the right arguments:

```
□ restaurants › createRestaurant action › saves the restaurant to the server

    expect(createRestaurant)
      .toHaveBeenCalledWith(...expected)

    Expected: "Sushi Place"
    Received: called with 0 arguments

    Number of calls: 1
```

Our restaurant name is passed in as the first argument of the action, so we can pass it along to the API method:

```
~~export const createRestaurant = () => async (dispatch, getState, api) => {~~
~~  await api.createRestaurant();~~
export const createRestaurant = name => async (dispatch, getState, api) => {
  await api.createRestaurant(name);
};
```

Save the file and the test passes.

Now we need to specify one more thing that happens when the `create` action is dispatched: the returned restaurant from the API is appended to the restaurant list in the state. To write that test, we're going to need to add a little to the setup as well:

```
describe('createRestaurant action', () => {
  const newRestaurantName = 'Sushi Place';
  const existingRestaurant = {id: 1, name: 'Pizza Place'};
  const responseRestaurant = {id: 2, name: newRestaurantName};

  let api;
  let store;

  beforeEach(() => {
    api = {
      createRestaurant: jest.fn().mockName('createRestaurant'),
    };

~~    const initialState = {};~~
    const initialState = {records: [existingRestaurant]};

    store = createStore(
```

This adds a restaurant to the pre-existing list of restaurants in the store. Save the file and the tests should still pass.

Now we're ready to specify that the returned restaurant is added to the store. Let's add it in a "describe" block:

```
describe('when save succeeds', () => {
  beforeEach(() => {
    api.createRestaurant
      .mockResolvedValue(responseRestaurant);
    return store.dispatch(
      createRestaurant(newRestaurantName),
    );
  });

  it('stores the returned restaurant in the store', () => {
    expect(store.getState().records).toEqual([
      existingRestaurant,
      responseRestaurant,
    ]);
  });
});
```

We ensure that the existing restaurant is still in the store, and the restaurant record returned from the server is added after it. Save the file and the test fails:

```
□ restaurants › createRestaurant action › when save succeeds › stores the
returned restaurant in the store

  expect(received).toEqual(expected) // deep equality

  - Expected  - 4
  + Received  + 0

    Array [
      Object {
        "id": 1,
        "name": "Pizza Place",
      },
  -   Object {
  -     "id": 2,
  -     "name": "Sushi Place",
  -   },
    ]
```

The store only contains the restaurant it was initialized with, not the new one the server responds with. Let's update `createRestaurant` to handle the server response data:

```
export const RECORD_LOADING_ERROR = 'RECORD_LOADING_ERROR';
export const ADD_RESTAURANT = 'ADD_RESTAURANT';

export const loadRestaurants = () => async (dispatch, getState, api) => {
...
export const createRestaurant = name => async (dispatch, getState, api) => {
~~  await api.createRestaurant(name);~~
  const record = await api.createRestaurant(name);
  dispatch(addRestaurant(record));
};

const addRestaurant = record => ({
  type: ADD_RESTAURANT,
  record,
});
```

After `createRestaurant()` resolves, we take the record the API returns to us and dispatch a new `addRestaurant()` action. Now let's handle that action in the reducer:

```
import {
  ADD_RESTAURANT,
  RECORD_LOADING_ERROR,
...
function records(state = [], action) {
  switch (action.type) {
    case STORE_RESTAURANTS:
      return action.records;
    case ADD_RESTAURANT:
      return [...state, action.record];
    default:
      return state;
  }
};
```

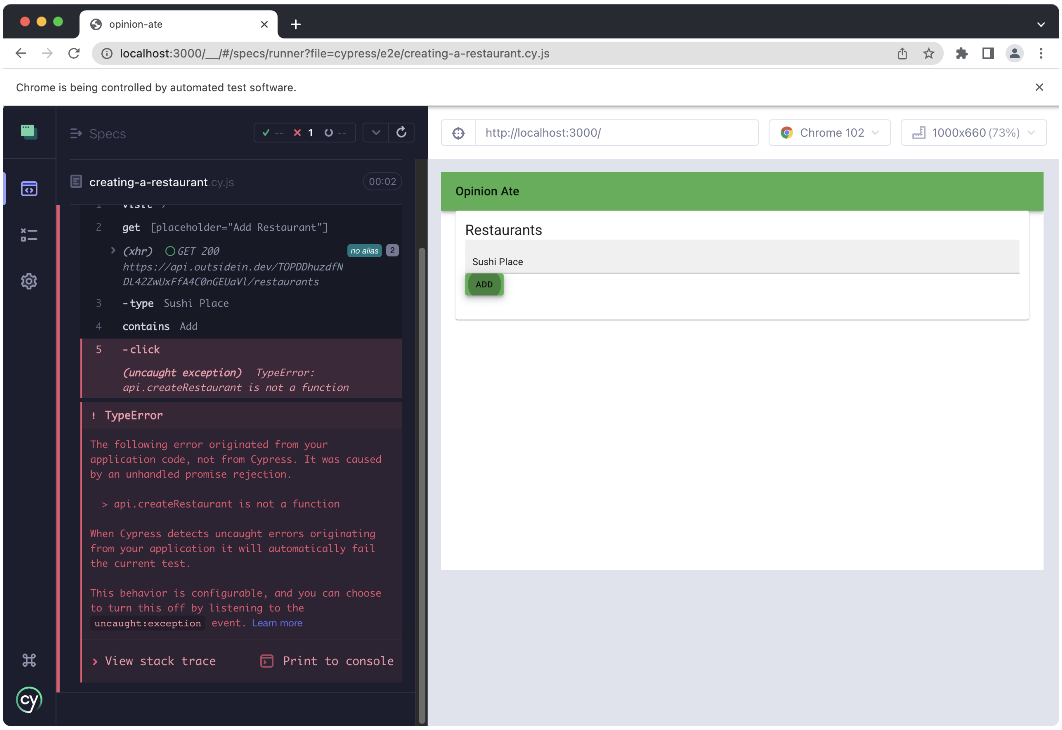
When `ADD_RESTAURANT` is dispatched we set records to a new array including the previous array, plus the new record on the end.

Save and all unit tests pass. Our store should now successfully handle creating a new restaurant.

Creating the API Method

Let's step back up to the E2E level and see if the E2E test has gotten past the previous failure. When we rerun it, we get a new error:

> TypeError: api.createRestaurant is not a function


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ -- ✕ 1 ↻ --
creating-a-restaurant.cy.js
00:02
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place
ADD

```
2   get  [placeholder="Add Restaurant"]
    › (xhr)  ○ GET 200                           no alias  2
    https://api.outsidein.dev/TOPDDhuzdfN
    DL42ZwUxFfA4C0nGEUaVl/restaurants
3   -type  Sushi Place
4   contains  Add
5   -click
    (uncaught exception)  TypeError:
    api.createRestaurant is not a function

! TypeError

The following error originated from your
application code, not from Cypress. It was caused
by an unhandled promise rejection.

  > api.createRestaurant is not a function

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

› View stack trace        Print to console
```

Cypress test failure because api.createRestaurant is not a function

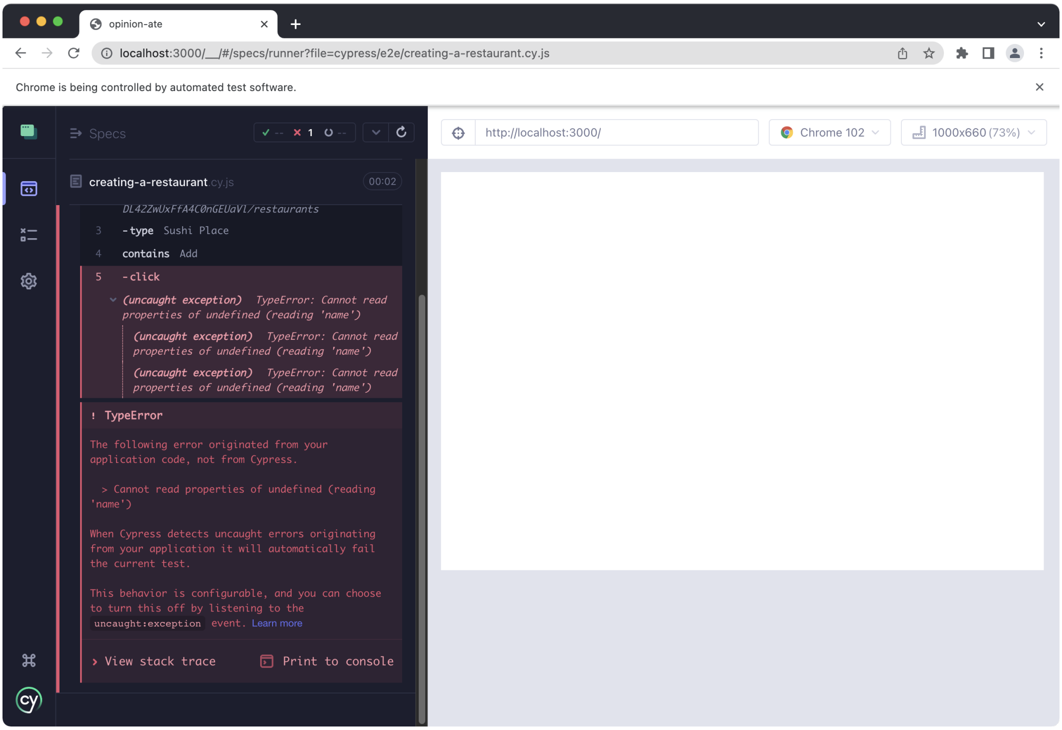
Our component is successfully dispatching the action to the store, which is successfully calling `api.createRestaurant()`, but we haven't implemented it yet. Let's do that now. Remember, as we discussed in "Vertical Slice", we're implementing our API client directly in response to the E2E test, instead of unit testing it.

Let's start by fixing the immediate error by defining an empty `createRestaurant()` method:

```
const api = {
  async loadRestaurants() {
    const response = await client.get('/restaurants');
    return response.data;
  },
  async createRestaurant() {},
};
```

When we rerun the E2E test, we get another error:

> TypeError: Cannot read properties of undefined (reading 'name')


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Specs
✓ -- ✕ 1 ↻ --
creating-a-restaurant.cy.js
00:02
http://localhost:3000/
Chrome 102
1000x660 (73%)

Chrome is being controlled by automated test software.

```
DL42ZwUxFfA4C0nGEUaVl/restaurants
3  -type  Sushi Place
4  contains  Add
5  -click
   (uncaught exception)  TypeError: Cannot read
   properties of undefined (reading 'name')
     (uncaught exception)  TypeError: Cannot read
     properties of undefined (reading 'name')
     (uncaught exception)  TypeError: Cannot read
     properties of undefined (reading 'name')

! TypeError

The following error originated from your
application code, not from Cypress.

  > Cannot read properties of undefined (reading
'name')

When Cypress detects uncaught errors originating
from your application it will automatically fail
the current test.

This behavior is configurable, and you can choose
to turn this off by listening to the
uncaught:exception event. Learn more

> View stack trace        Print to console
```

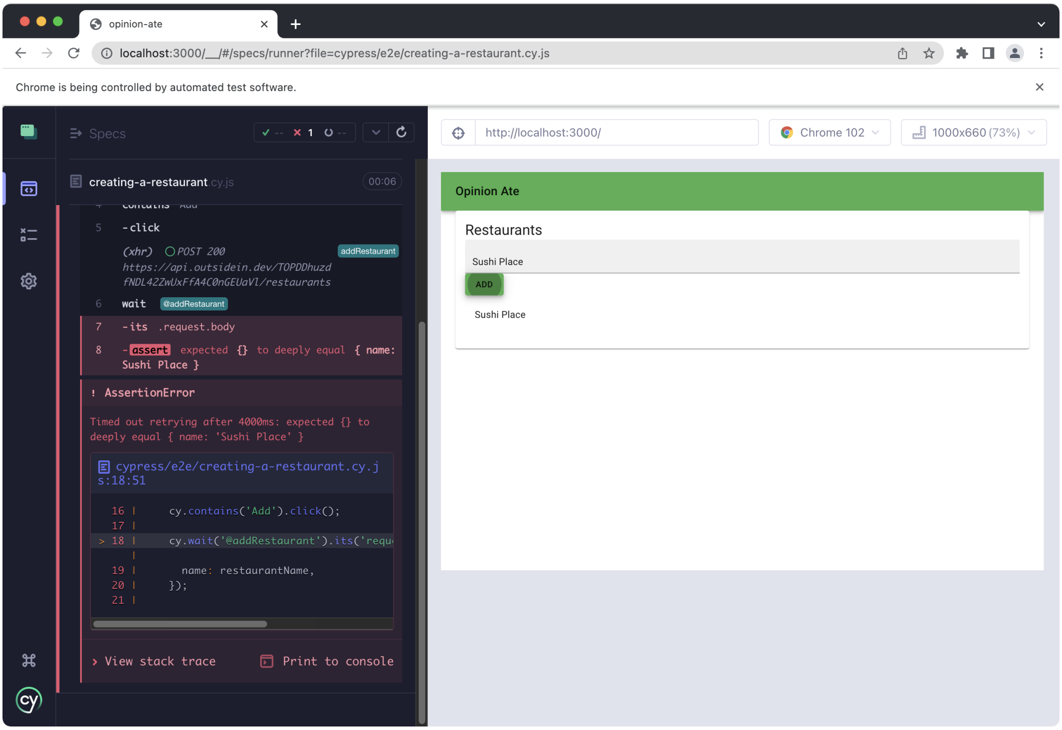
Cypress test failure reading property 'name'

We aren't getting a name value back from the function, because we still aren't making the HTTP request that kicked off this whole sequence. Fixing this will move us forward better, so let's actually make the HTTP request in the API:

```
  },
~~  async createRestaurant() {},~~
  async createRestaurant() {
    const response = await client.post('/restaurants', {});
    return response.data;
  },
};
```

Now the `POST` request is made, and we get an error on the assertion we made about the request's body:

> assert: expected {} to deeply equal { name: Sushi Place }



Chrome is being controlled by automated test software.

creating-a-restaurant.cy.js

```
5  -click
   (xhr) ○ POST 200
   https://api.outsidein.dev/TOPDDhuzd
   fNDL42ZwUxFfA4C0nGEUaVl/restaurants
6  wait @addRestaurant
7  -its .request.body
8  -assert expected {} to deeply equal { name:
   Sushi Place }
```

```
! AssertionError

Timed out retrying after 4000ms: expected {} to
deeply equal { name: 'Sushi Place' }
```

cypress/e2e/creating-a-restaurant.cy.js:18:51

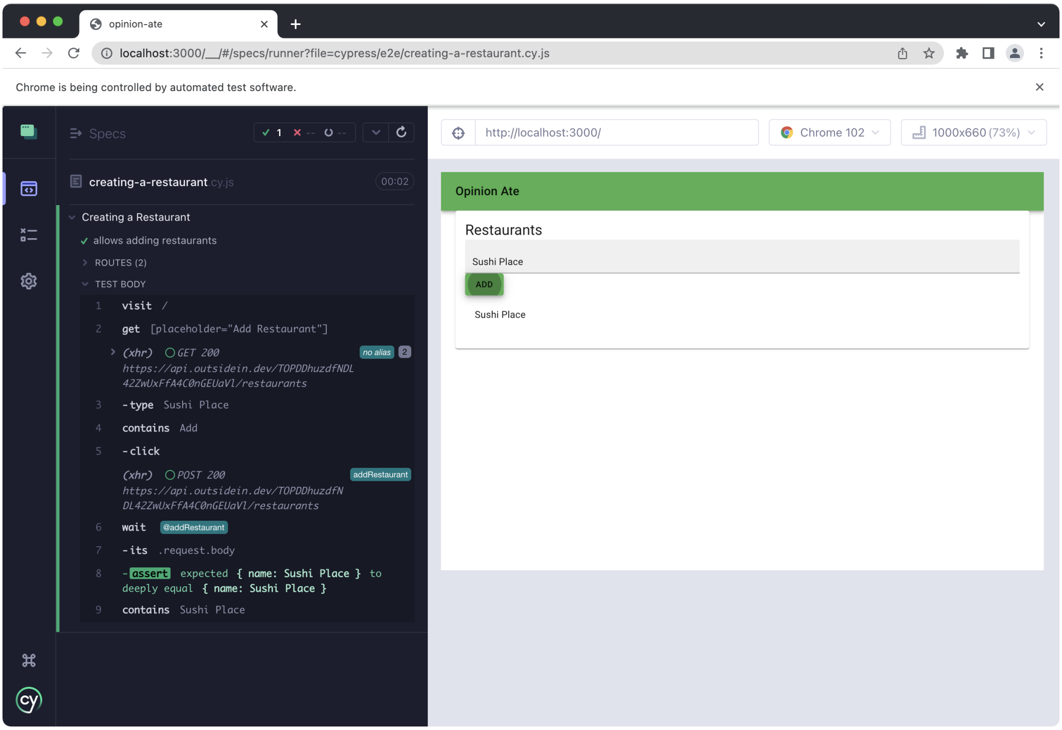
```
      cy.contains('Add').click();

>     cy.wait('@addRestaurant').its('requ

        name: restaurantName,
      });

```

Cypress test failure because name is missing

So we aren't passing the restaurant name in the `POST` body. That's easy to fix by passing it along from the argument to the method:

```
async createRestaurant() {
  const response = await client.post('/restaurants', {});
  return response.data;
},
~~async createRestaurant() {~~
~~  const response = await client.post('/restaurants', {});~~
async createRestaurant(name) {
  const response = await client.post('/restaurants', {name});
  return response.data;
},
```

Rerun the E2E test and it passes, and we see Sushi Place added to the restaurant list. Our feature is complete!


opinion-ate
localhost:3000/__/#/specs/runner?file=cypress/e2e/creating-a-restaurant.cy.js
Chrome is being controlled by automated test software.
Specs
✓ 1 ✕ -- ↻ --
http://localhost:3000/
Chrome 102
1000x660 (73%)
Opinion Ate
Restaurants
Sushi Place
ADD
Sushi Place

creating-a-restaurant.cy.js 00:02

Creating a Restaurant

✓ allows adding restaurants

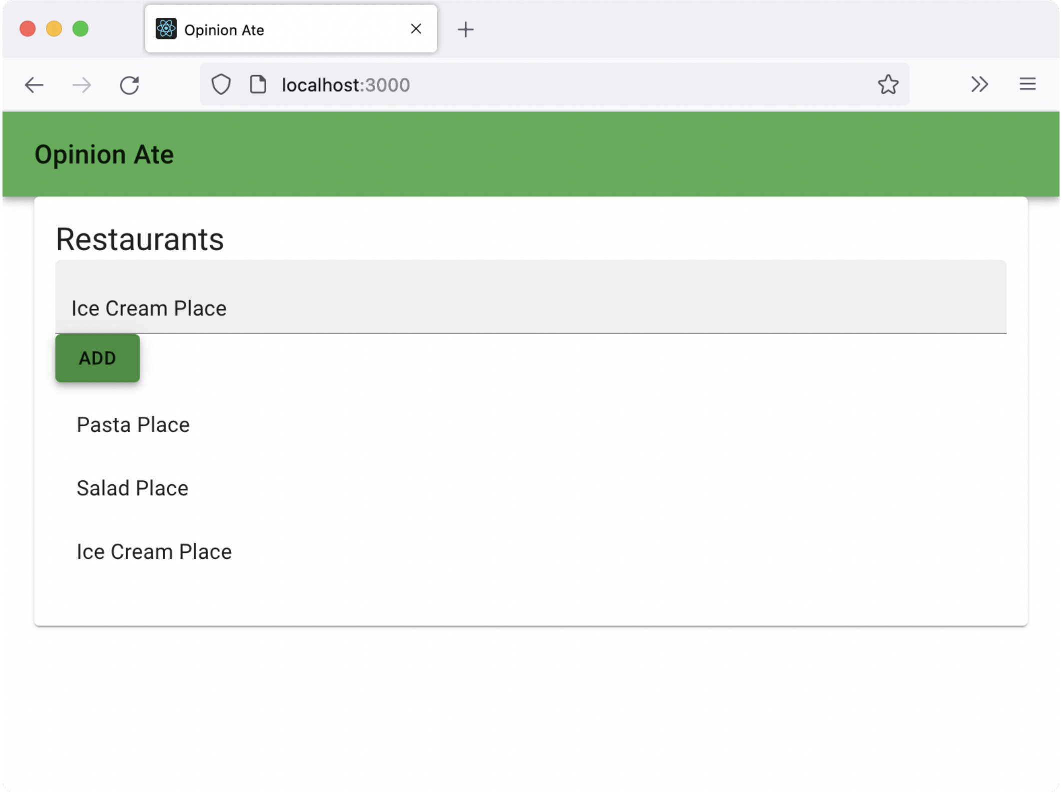
ROUTES (2)

TEST BODY

```
1  visit  /
2  get  [placeholder="Add Restaurant"]
   (xhr)  ○ GET 200                                  no alias 2
   https://api.outsidein.dev/TOPDDhuzdfNDL
   42ZwUxFfA4C0nGEUaVl/restaurants
3  -type  Sushi Place
4  contains  Add
5  -click
   (xhr)  ○ POST 200                                 addRestaurant
   https://api.outsidein.dev/TOPDDhuzdfN
   DL42ZwUxFfA4C0nGEUaVl/restaurants
6  wait  @addRestaurant
7  -its  .request.body
8  -assert  expected { name: Sushi Place } to
   deeply equal { name: Sushi Place }
9  contains  Sushi Place
```

Cypress test for creating a restaurant passing

Open your app in the browser and try out creating a restaurant for real. Reload the page to make sure it's really persisted to the server.



Restaurant created

Edge Cases

Now let's think about what edge cases we need to handle. Here are a few:

- The form should clear out the text field after you save a restaurant.
- If the form is submitted with an empty restaurant name, it should show a validation error, and not submit to the server.
- If the request to the server fails an error message should be shown, and the restaurant name should not be cleared.

Clearing the Text Field

First, let's implement the form clearing out the text field after saving. In `NewRestaurantForm.spec.js`, add a new test:

```
describe('when filled in', () => {
...
  it('calls createRestaurant with the name', async () => {
    await fillInForm();
    expect(createRestaurant)
      .toHaveBeenCalledWith(restaurantName);
  });

  it('clears the name', async () => {
    await fillInForm();
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue('');
  });
});
```

Save the test, and we get a test failure confirming that the text field is not yet cleared:

```
  □ NewRestaurantForm › when filled in › clears the name

    expect(element).toHaveValue()

    Expected the element to have value:

    Received:
      Sushi Place
```

To make this pass, we need to clear the text field—but where exactly should we write the code to do that? We could clear it before or after calling `createRestaurant()`.

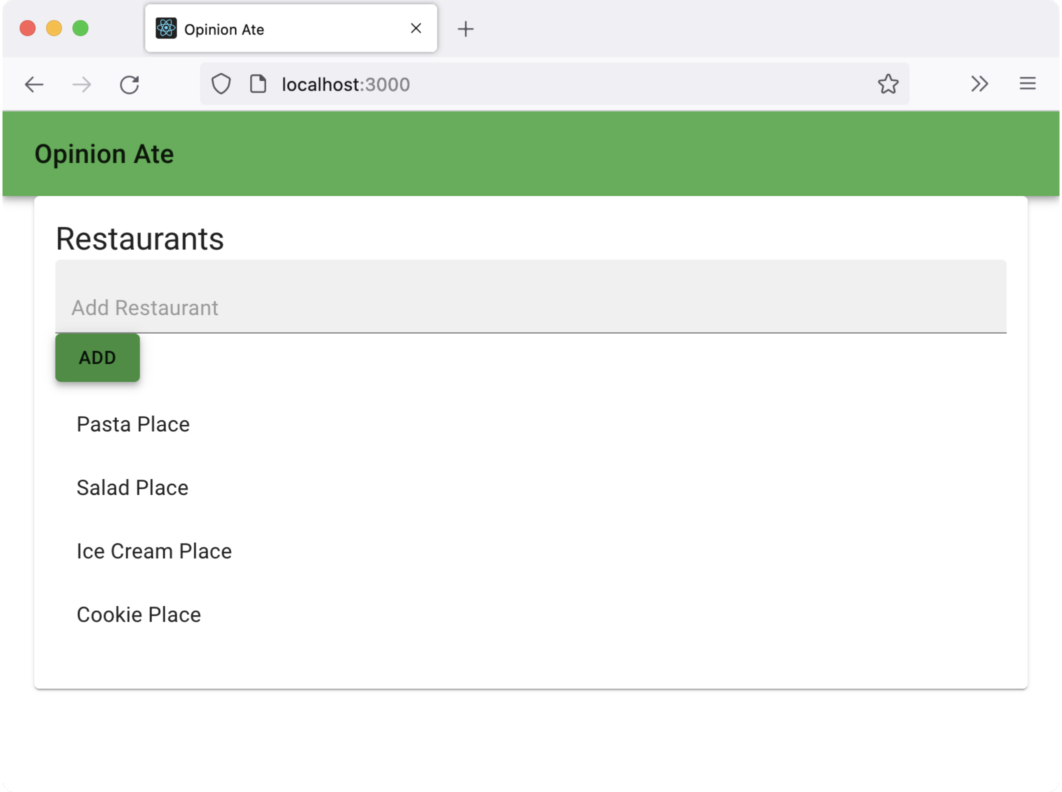
Let's think about what we know right now. We wrote down another upcoming edge case scenario that says the name should *not* be cleared if the web service call fails. But of course we won't know if that call failed or not until after the call is made. That suggests we should clear the text field *after* the call to `createRestaurant`, not before. We don't want to actually add that check for failure until a future test drives us to it, but it *is* okay for us to use that knowledge to decide where to put the code to clear the text field.

Make this change in `NewRestaurantForm.js`:

```
~~function handleSubmit(e) {~~
async function handleSubmit(e) {
  e.preventDefault();
~~  createRestaurant(name);~~
  await createRestaurant(name);
  setName('');
}
```

Save the file and the test passes.

If you add a new restaurant in the browser, now you'll see the name field cleared out afterward:



Pasta Place

Salad Place

Ice Cream Place

Cookie Place

Name field cleared after submission

Validation Error

The next edge case for us to handle is:

- If the form is submitted with an empty restaurant name, it should show a validation error, and not submit to the server.

We'll start with the component test. Create a new `describe` block for this situation, below the "when filled in" describe block. Let's write one assertion at a time. First, we'll confirm a validation error message is shown:

```
describe('when empty', () => {
  async function submitEmptyForm() {
    renderComponent();

    await user.click(screen.getByText('Add'));
  }

  it('displays a validation error', async () => {
    await submitEmptyForm();
    expect(screen.getByText('Name is required'))
      .toBeInTheDocument();
  });
});
```

Note that we don't type into the text field; this ensures it will keep the default value, which is the empty string.

Save the file and the test fails, because the validation error message is not found:

```
  □ NewRestaurantForm › when empty › displays a validation error

    TestingLibraryElementError: Unable to find an element with the text: Name
    is required.
```

Let's fix this error in the simplest way possible by adding the validation error unconditionally:

```
import Button from '@mui/material/Button';
import Alert from '@mui/material/Alert';
import {createRestaurant} from './store/restaurants/actions';
...
  return (
    <form onSubmit={handleSubmit}>
      <Alert severity="error">Name is required</Alert>
      <TextField
```

The tests pass. Now how can we write a test to drive out hiding that validation error in other circumstances? Well, we can check that it's not shown when the form is first rendered.

In preparation, let's move the validation error text we're searching for to a constant directly under our top-level `describe`:

```
describe('NewRestaurantForm', () => {
  const restaurantName = 'Sushi Place';
  const requiredError = 'Name is required';

  let createRestaurant;
...
    it('displays a validation error', async () => {
      await submitEmptyForm();
~~      expect(screen.getByText('Name is required'))~~
~~        .toBeInTheDocument();~~
      expect(screen.getByText(requiredError))
        .toBeInTheDocument();
    });
```

Save and confirm the tests still pass.

Next, add a new `describe` above the "when filled in" one:

```
describe('initially', () => {
  it('does not display a validation error', () => {
    renderComponent();
    expect(screen.queryByText(requiredError))
      .not.toBeInTheDocument();
  });
});
```

The test fails because we are always showing the error right now:

```
  □ NewRestaurantForm › initially › does not display a validation error

    expect(element).not.toBeInTheDocument()

    expected document not to contain element, found
    <div class="MuiAlert-message css-acap47-MuiAlert-message">Name is
    required</div>
    instead
```

Time to add some logic around this error. First, we'll add state to record whether it should be shown:

```
export function NewRestaurantForm({createRestaurant}) {
  const [name, setName] = useState('');
  const [validationError, setValidationError] = useState(false);

  async function handleSubmit(e) {
...
  return (
    <form onSubmit={handleSubmit}>
      ~~<Alert severity="error">Name is required</Alert>~~
      {validationError && <Alert severity="error">Name is required</Alert>}
      <TextField
```

Now, what logic should we use to set the `validationError` flag? Our tests just specify that initially the error is not shown, and after submitting an invalid form it's shown—that's all. The simplest logic to pass this test is to always show the validation error after saving:

```
async function handleSubmit(e) {
  e.preventDefault();
  setValidationError(true);
  await createRestaurant(name);
```

Save the file and all tests pass.

This is not the correct final logic, which may feel obvious to you. That should drive us to consider what test we are missing. What should behave differently? Well, when we submit a form with a name filled in, the validation error should not appear. Let's add that test to the "when filled in" `describe` block:

```
it('does not display a validation error', async () => {
  await fillInForm();
  expect(screen.queryByText(requiredError))
    .not.toBeInTheDocument();
});
```

We can pass this test by adding a conditional around setting the `validationError` flag:

```
async function handleSubmit(e) {
  e.preventDefault();

  if (!name) {
    setValidationError(true);
  }

  await createRestaurant(name);
```

Save the file and all tests pass.

Now, is there any other time we would want to hide or show the validation error? Well, if the user submits an empty form, gets the error, then adds the missing name and submits it again, we would want the validation error cleared out. Let's create this scenario as another `describe` block, below the "when empty" one:

```
describe('when correcting a validation error', () => {
  async function fixValidationError() {
    renderComponent();
    createRestaurant.mockResolvedValue();

    await user.click(screen.getByText('Add'));

    await user.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    await user.click(screen.getByText('Add'));
  }

  it('clears the validation error', async () => {
    await fixValidationError();
    expect(screen.queryByText(requiredError))
      .not.toBeInTheDocument();
  });
});
```

Note that we repeat the steps from the helper functions from *both* other groups, first submitting the form empty and then submitting it filled in.

Test Independence

Sometimes it can be tempting for developers to try to make one test run “after” another test so they can reuse the first test’s end state. For example, in this case, we need to test out clearing a validation error, and we already have another test that causes a validation error. It seems duplicative to copy the validation-error-causing code here. Can we make this test run after the validation-error test?

No, we should not try to sequence our tests and reuse state, because it’s important for unit tests to be independent. Independent tests can be run by themselves without requiring other tests to have been run first. This independence increases test reliability and makes it easier to troubleshoot test failures. If you find you need to reuse duplicate code and speed up tests, there are other techniques you can reach for—but don’t compromise your tests’ independence!

Save the test file and we get an assertion failure:

```
 □ NewRestaurantForm › when correcting a validation error › clears the
 validation error

   expect(received).not.toBeInTheDocument()

   expected document not to contain element, found
   <div class="MuiAlert-message css-acap47-MuiAlert-message">Name is
   required</div>
   instead
```

We can fix this by clearing the `validationError` flag when the name is filled in:

```
if (!name) {
  setValidationError(true);
} else {
  setValidationError(false);
}
```

Note that we aren't waiting for the web service to return to clear the validation error flag, the way we clear out the name field. We know right away that the form is valid, so we can clear the validation error flag even before the web service call is made.

Save and the tests pass. Now that we have an `else` branch to that conditional, let's invert the boolean to make it easier to read. Refactor it to:

```
if (name) {
  setValidationError(false);
} else {
  setValidationError(true);
}
```

Save and the tests should still pass.

With that, we've implemented the first behavior we want when we submit an empty form: displaying a validation error. Now we can move on to the second behavior when we submit an empty form: *not* dispatching the action to save the restaurant to the server.

The test for that new behavior is straightforward. Add the following in the "when empty" `describe` block:

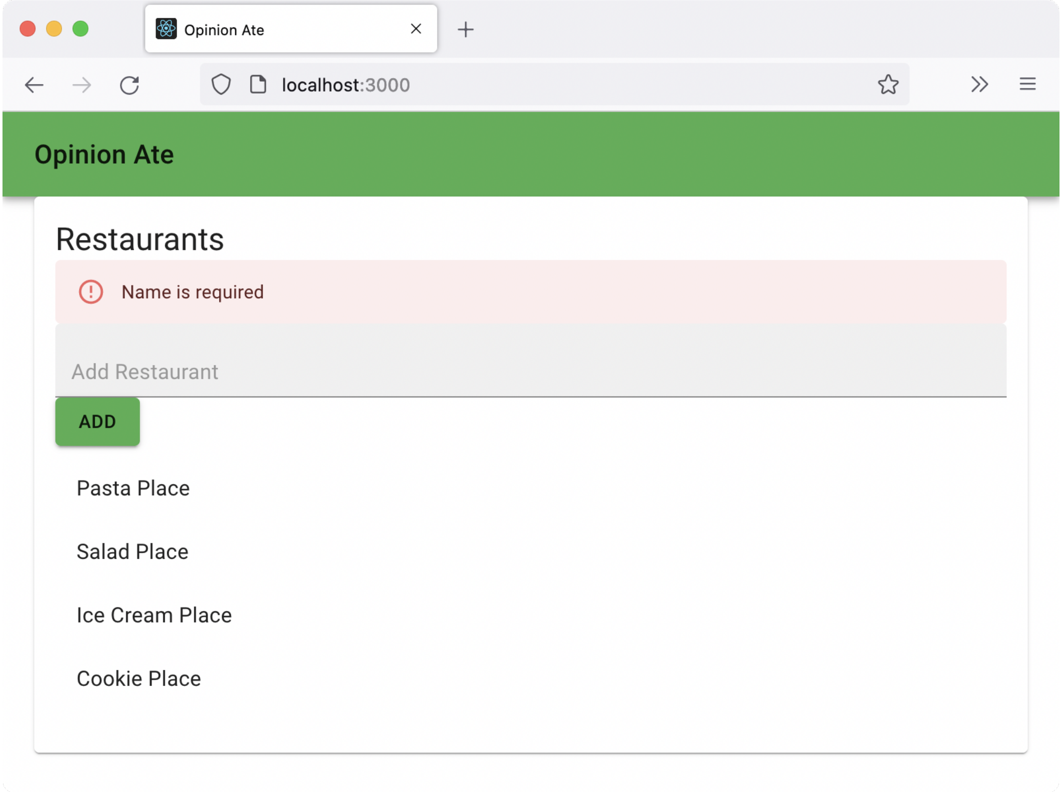
```
it('does not call createRestaurant', async () => {
  await submitEmptyForm();
  expect(createRestaurant)
    .not.toHaveBeenCalled();
});
```

Save and the test fails.

We can fix this error by moving the call to `createRestaurant()` inside the true branch of the conditional:

```
async function handleSubmit(e) {
  e.preventDefault();

  if (name) {
    setValidationError(false);
    await createRestaurant(name);
  } else {
    setValidationError(true);
  }

~~  await createRestaurant(name);~~
  setName('');
}
```

Save the file and the test passes. If you try to submit the form with an empty restaurant name in the browser, you'll see:



Restaurants

Name is required

Pasta Place

Salad Place

Ice Cream Place

Cookie Place

Name is required error

Server Error

Our third edge case is when the web service call fails. We want to display a server error message.

We'll want to check for the message in a few different places, so let's set it up as a constant in the uppermost `describe` block:

```
describe('NewRestaurantForm', () => {
  const restaurantName = 'Sushi Place';
  const requiredError = 'Name is required';
  const serverError = 'The restaurant could not be saved. Please try again.';

  let createRestaurant;
```

Since this is a new situation, let's set this up as yet another new `describe` block:

```
describe('when the store action rejects', () => {
  async function fillInForm() {
    renderComponent();
    createRestaurant.mockRejectedValue();

    await user.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    await user.click(screen.getByText('Add'));
  }

  it('displays a server error', async () => {
    await fillInForm();
    expect(screen.getByText(serverError))
      .toBeInTheDocument();
  });
});
```

This is almost the same as the successful submission case; the only difference is that the setup we call the `mockRejectedValue()` method of the Jest mock function `createRestaurant`. This means that when this function is called, it will return a promise that rejects. In our case we don't actually care about what error it rejects with, so we don't have to pass an argument to `mockRejectedValue()`.

Save and, in addition to an assertion failure, we also get an additional error:

```
  □ NewRestaurantForm › when the store action rejects › displays a server error

    thrown: undefined
```

What's happening is that our call to `createRestaurants()` is rejecting, but we aren't handling that promise rejection. Let's handle it with an empty `catch` block, just to silence this warning; we'll add behavior to that `catch()` function momentarily.

```
if (name) {
  setValidationError(false);
  try {
    await createRestaurant(name);
  } catch {}
} else {
  setValidationError(true);
}
```

Save the file and the "thrown" error goes away, leaving only the expectation failure:

```
  □ NewRestaurantForm › when the store action rejects › displays a server error

    TestingLibraryElementError: Unable to find an element with the text: The
    restaurant could not be saved. Please try again.
```

As usual, we'll first solve this by hard-coding the element into the component:

```
return (
  <form onSubmit={handleSubmit}>
    <Alert severity="error">
      The restaurant could not be saved. Please try again.
    </Alert>
    {validationError && <Alert severity="error">Name is required</Alert>}
```

Save and the test passes.

Now, when do we want the server message *not* to show? If you think it over, here are a few scenarios where we don't want the error message to show:

- When the component first renders
- When the server returns successfully
- When the server is retried after a failure, and succeeds

Let's test-drive these one at a time. First, confirming the server error doesn't display when the component first renders. Add another test to the "initially" describe block:

```
it('does not display a server error', () => {
  renderComponent();
  expect(screen.queryByText(serverError))
    .not.toBeInTheDocument();
});
```

Save and the test fails.

To make it pass, we'll add another bit of state to track whether the error should show. We'll start it out hidden and show it if the store action rejects:

```
export function NewRestaurantForm({createRestaurant}) {
  const [name, setName] = useState('');
  const [validationError, setValidationError] = useState(false);
  const [serverError, setServerError] = useState(false);

  async function handleSubmit(e) {
...
    if (name) {
      setValidationError(false);
      try {
        await createRestaurant(name);
~~      } catch {}~~
      } catch {
        setServerError(true);
      }
    } else {
...
  return (
    <form onSubmit={handleSubmit}>
      {serverError && (
        <Alert severity="error">
          The restaurant could not be saved. Please try again.
        </Alert>
      )}
      {validationError && <Alert severity="error">Name is required</Alert>}
```

Save and the tests pass.

Our next scenario was that the server error should not show when the server request returns successfully. In the "when filled in" describe block, add a similar test:

```
it('does not display a server error', async () => {
  await fillInForm();
  expect(screen.queryByText(serverError))
    .not.toBeInTheDocument();
});
```

Save and the test passes. This is another instance where the test doesn't drive new behavior, but it's helpful for extra assurance that the code is behaving the way we expect.

We have one more situation in which we don't want the server error to show: when we try to save, the server rejects, then we try again and it succeeds. This is a new situation, so let's create a new `describe` block for it:

```
describe('when retrying after the store rejects', () => {
  async function retrySubmittingForm() {
    renderComponent();
    createRestaurant
      .mockRejectedValueOnce()
      .mockResolvedValueOnce();

    await user.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    await user.click(screen.getByText('Add'));

    await user.click(screen.getByText('Add'));
  }
  it('clears the server error', async () => {
    await retrySubmittingForm();
    expect(screen.queryByText(serverError))
      .not.toBeInTheDocument();
  });
});
```

Save the file and you'll get the expected test failure:

```
  □ NewRestaurantForm › when retrying after a server error › clears the server
    error

    expect(element).not.toBeInTheDocument()

    expected document not to contain element, found
    <div class="MuiAlert-message css-acap47-MuiAlert-message">The restaurant
    could not be saved. Please try again.</div>
    instead
```

We should be able to make this test pass by just clearing the `serverError` flag when attempting to save:

```
if (name) {
  setValidationError(false);
  setServerError(false);
  try {
    await createRestaurant(name);
```

Save the file, but surprisingly, the test failure doesn't change–the server error is still shown! Why is that?

The way I ended up troubleshooting this is with some good old-fashioned `console.log` debugging. Let's temporarily add some log statements at various points in `handleSubmit` to help us visualize the sequence of code that is running.

Add a log statement each time `setServerError` is called, and after `createRestaurant` succeeds:

```
setValidationError(false);
setServerError(false);
console.log('cleared server error');
try {
  await createRestaurant(name);
  console.log('succeeded');
} catch {
  console.log('set server error');
  setServerError(true);
}
```

So that we don't see log output from other tests, we can use Jest's `only` functionality to only run one test from this file. Change `it(` to `it.only(` on the test that's failing:

```
~~it('clears the server error', async () => {~~
it.only('clears the server error', async () => {
  await retrySubmittingForm();
```

The output we hope to see is:

- "cleared server error" (after the first click)
- "set server error" (after the first response, which rejects)
- "cleared server error" (after the second click)
- "succeeded" (after the second response, which is resolves)

Save the test, and what we see instead is:

- "cleared server error"
- "set server error"

It looks like our second "cleared server error" isn't being reached. To see why, add a log statement in the validation error branch:

```
} else {
  console.log('invalid');
  setValidationError(true);
}
```

Save and, sure enough, our output is:

- "cleared server error"
- "set server error"
- "invalid"

Why is the form valid after the first click but invalid after the second? The only time our form is invalid is when the restaurant name is empty. The way we have our code written right now, whenever we submit the form, the restaurant name field is *always* cleared out, even if the server rejects.

We have it on our list to implement the behavior where the restaurant name field is *not* cleared out upon server error—but we haven't gotten to it yet! We've stumbled across a problem where a behavior we don't want is getting in the way of the current test. What do we do?

- We don't want to have to workaround this problem in the test by re-typing in the name of the restaurant, because that's not how we want our app to work; we want to *fix* that behavior.
- We don't want to write two tests at once; we want to focus on one test at a time and getting it passing. Seeing two failing tests is a distraction from the red-green-refactor loop.
- We don't want to discard the changes we've made for the current test, because we think we've made progress toward getting it working.

In a situation like this, one option is to use Jest's `skip` functionality to temporarily skip this test until we are ready to finish it. Change the `.only` to a `.skip`:

```
~~it.only('clears the server error', async () => {~~
it.skip('clears the server error', async () => {
  await retrySubmittingForm();
```

Save and the test suite reruns. This test is skipped, and all the other tests in the file are passing.

Now that that test is temporarily skipped, we can test-drive a fix to the root cause problem. We want to assert that when the server rejects, the restaurant name should *not* be cleared out.

Add another expectation to the "when the store action rejects" `describe` block:

```
it('does not clear the name', async () => {
  await fillInForm();
  expect(screen.getByPlaceholderText('Add Restaurant'))
    .toHaveValue(restaurantName);
});
```

Save and we get a test failure:

```
  □ NewRestaurantForm › when the store action rejects › does not clear the name

    expect(element).toHaveValue(Sushi Place)

    Expected the element to have value:
      Sushi Place
    Received:
```

To fix it, move the call to clear the name inside the `try` block, after the `createRestaurant()` call. This way the only situation where the name is cleared will be once we know we've successfully saved it to the server:

```
    try {
      await createRestaurant(name);
      setName('');
      console.log('succeeded');
    } catch {
      console.log('set server error');
      setServerError(true);
    }
  } else {
    console.log('invalid');
    setValidationError(true);
  }

~~  setName('');~~
}
```

Save and the test passes.

Now that the name isn't cleared upon server error, the problem it was causing for our other test should be resolved. Remove the `.skip` to get that test running again:

```
~~it.skip('clears the server error', async () => {~~
it('clears the server error', async () => {
  await retrySubmittingForm();
```

Save and the newly-unskipped test passes. The test can now run as we first intended it: we submit the form and get a server failure response, then we re-submit the form and get a server success response, and the server error message is hidden.

Now that the test is passing, you can remove the temporary `console.log` statements you added to `NewRestaurantForm.js`.

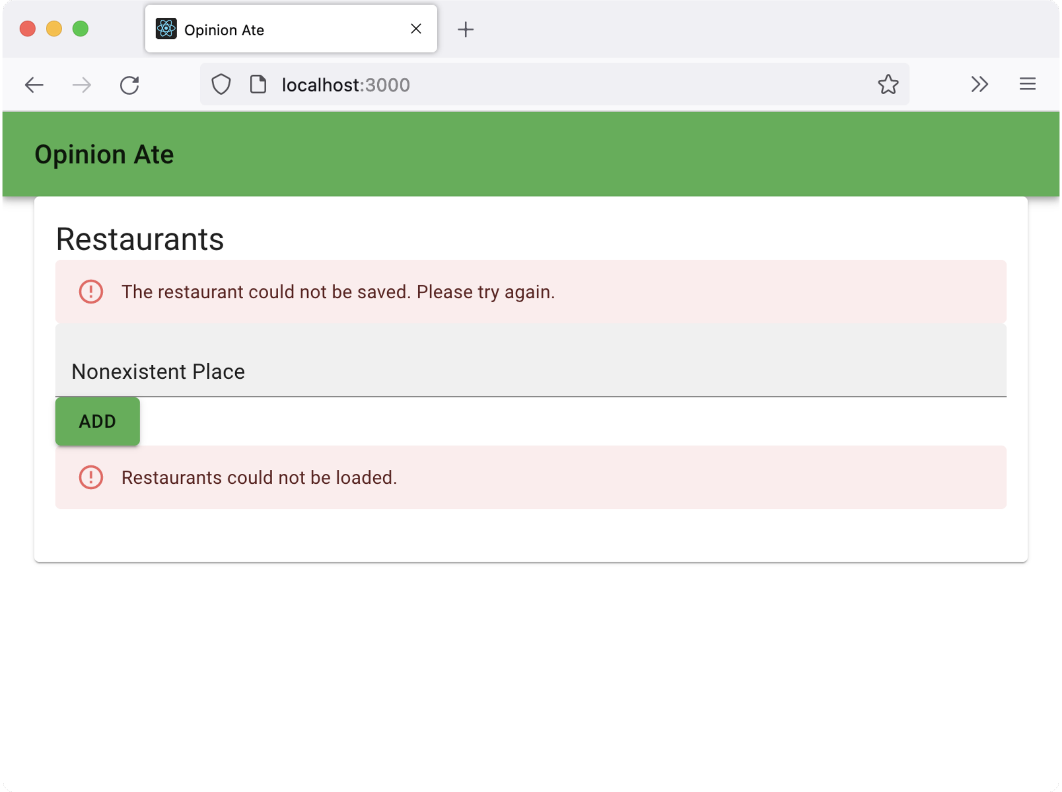
With that, our component tests for server errors are finally done. For the store, we have just one test we need to add. `NewRestaurantForm` is expecting the `createRestaurant` action to return a promise that rejects when there is a server error. Let's make sure this is happening. Add the following "describe" block inside "createRestaurant action" below "when save succeeds":

```
describe('when save fails', () => {
  it('rejects', () => {
    api.createRestaurant.mockRejectedValue();
    const promise = store.dispatch(
      createRestaurant(newRestaurantName),
    );
    return expect(promise).rejects.toBeUndefined();
  });
});
```

The chain `.rejects.toBeUndefined()` is a bit unintuitive. In Jest, when you test a promise with `.rejects`, you have to chain another matcher onto the end of it to test the rejected value. Typically you might say `.rejects.toEqual({error: 'Some message'})`. In our case, we didn't define a value that the promise rejects with, so the rejected value is `undefined`. `.rejects.toBeUndefined()` ensures both that the promise rejects (which we care about) and that the rejected value is `undefined` (which Jest requires us to check for, even though we don't care about it).

The test passes right away. Because our store returned the promise chain returned from the API, the rejection is passed along to the caller of `store.dispatch()`. But it's good to document this in a test, because it's part of the contract of the action that our component is relying on.

Now let's run our app in the browser and see it handle a server error. As you did in the last chapter, open `src/api.js` and put an incorrect API key in the `baseURL` value. Load up the front-end and ignore the error message for *loading* the restaurants. Enter a restaurant name and click "Add". You should see another red server error message, this time for saving:



Restaurants

The restaurant could not be saved. Please try again.

Nonexistent Place

Restaurants could not be loaded.

Server error message

Restore the correct API key value in `src/api.js`, reload the front-end, and make sure the app is working again. Rerun your E2E tests to make sure they still pass.

With that, we've finished implementing adding a restaurant! We had to handle a lot of edge cases, but in doing so we've added a lot of robustness to our form.

Imagine if we had relied on manual testing for all these edge cases. It would have been tedious to find ways to visualize the loading and error states, especially if we found we needed to change the implementation and had to start that testing all over again.

Or imagine if we had tried to handle all of these cases in E2E tests. We either would have had a lot of slow tests, or else one long test that ran through an extremely long sequence. Instead, our E2E tests cover our main functionality, and our unit tests cover all the edge cases thoroughly.

If you have any uncommitted changes, commit them to git. Push up your branch to the origin and open a pull request. Wait for CI to complete, then merge the pull request. Now we can drag our "Add Restaurants" story to "Done" in Trello.

What's Next

With this we've completed our second feature. We've also reached the end of this exercise! In the next chapter we'll look back at what we did over the course of the exercise and the benefits the outside-in test-driven development process gave us.

11. Exercise Wrap-Up

Congratulations, you've reached the end of the outside-in front-end development exercise! You've experienced enough outside-in TDD and other agile methodologies that you should have everything you need to put them into practice. One option you could take is to continue to build out more features of the example app on your own. Alternatively, you could go straight to applying these practices in your own front-end applications.

Let's reflect back on the agile development process we followed in the exercise and how it may have differed from the way you've built front-end apps in the past.

- *Our app has been deployed to a live environment since before we wrote our first feature.* This allows us to get user feedback as early as possible, even if it's only from business representatives on our team. It also avoids the situation where we *say* the app is complete, but then it takes weeks to get it running successfully in the "real" production environment.
- *Our code is the simplest possible implementation of the current feature set.* There is no unused code. For example, our data layer doesn't yet allow editing or deleting restaurants, because that functionality isn't yet exposed to the user. By not writing unused code, we avoid the cost of maintaining and updating code that isn't providing value.
- *Our app is thoroughly covered by tests: the main paths by end-to-end tests, and the edge cases by unit tests.* We know that the behavior of our app is fully specified because we only wrote production code in response to a failing test. This means that we can refactor and upgrade dependencies with confidence, knowing that if the tests pass, our application is working.
- *We are testing the interface, not the implementation, of our units.* We can change things like our HTML tags and styles, which component library we use if any, how we structure event-handling code, and the internals of our data store, our tests continuing to pass unchanged. This gives us greater confidence in our changes and reduces the cost of maintaining the tests, so their cost won't outweigh their value. It also encourages us to do smaller, more frequent refactorings.
- *We built one thing at a time.* Instead of building out every aspect of a component or store module at once, we built a vertical slice through all layers of our app. Instead of trying to get the visuals and functionality right at the same time, we got the

functionality working first, then refactored the visuals. Working on one thing at a time makes development feel less overwhelming, makes it less likely to miss details, provides lots of stopping points where working progress has been made, and allows us to deliver something useful to our customers earlier.

These differences have had a tremendous impact on my experience as a software developer. Before I started using TDD and other agile development practices, I was afraid to make changes to working code for fear of breaking it. This led me to live with code that was hard to understand and change, which slowed down the development process more and more over time. I had lost control over my code. TDD gives me back control, so I can change my code with confidence at any time.

Maybe you're already convinced of these benefits provided by TDD, together with the other agile development practices we've looked at. If so, give it a try on your projects!

Or maybe you aren't yet convinced TDD will make much of a difference. If that's the case, do you have an another way to keep your development speed fast, avoid production bugs and keep your code easy to understand in the face of unanticipated changes? If not, I would encourage you to give TDD a chance. Try it and see if it surprises you.

What's Next

We've now completed our exercise. In the final part of the book we'll do a deep dive into some supplemental material and talk about next steps you can take.

Part Three: Going Further

12. Integration Testing the API Client

In "Creating the API Client" we chose not to test the API client directly, relying instead on our E2E tests. Our client is a thin wrapper around Axios without any conditionals. As a result, it is fully exercised by our E2E test which mocks out the HTTP calls and then tests the application functionality that makes those calls. If the E2E tests pass, we know our API client works.

But what if we *did* want the extra assurance of testing our API directly? Or what if we were building an app in which we *didn't* think the API client was fully exercised by the E2E tests for some reason? How could we test the API client in that scenario?

We still want to follow the principle "don't mock what you don't own," so we shouldn't mock out the `axios` module itself. Instead, we'll write a test that integrates our wrapper with the third-party library it wraps. We'll call our functions, let them pass through to Axios, let Axios make the HTTP call, then we'll use a testing library to stub those HTTP calls.

Developers who write tests like these often call them "integration tests," because they test our code in integration with a third-party library that talks to external systems. As we discussed in "Testing Concepts", though, the term "integration tests" can be used in many ways. When you hear "integration tests" in other contexts, don't assume it refers to the same kind of test: ask to find out.

To make this kind of test work we need a library that can do HTTP stubbing within our integration tests. For this chapter we'll use a library called Nock[25]. It works well and is well-established, having been released 11 years ago at the time of this writing.

If we had decided to integration test our API client from the start we could have written it TDD style. But since the API client already exists, we'll be adding the test afterward. One challenge with test-after development is covering all code branches, but in this case that won't be too hard because there is no branching logic in the API client. Another risk is that the tests might not actually fail when we expect them to, giving us false positives. To make sure this doesn't happen we'll temporarily break our code and make sure we see the test fail.

[25] https://github.com/nock/nock

Our `api.js` file lives under `src`. Next to it, create an `api.spec.js` file. Let's test one method at a time, starting with `loadRestaurants`.

What behavior of `loadRestaurants` do we want to specify? Well, we want `loadRestaurants` to call the correct endpoint of the API and return the data that the server provides to it. Let's get that much written in a test to start:

```
import api from './api';

describe('api', () => {
  describe('loadRestaurants', () => {
    const restaurants = [{id: 1, name: 'Sushi Place'}];

    it('returns the response to the right endpoint', async () => {
      await expect(api.loadRestaurants())
        .resolves.toEqual(restaurants);
    });
  });
});
```

We know we'll need an array of restaurants, so we define one. We don't yet know how we'll mock out the API call, but we do know that when we call `api.loadRestaurants()` we should get back a promise that resolves to that array of restaurants.

If you save the test now, the test will fail, showing some output suspiciously similar to what you see when you run the app in the browser:

```
□ api › loadRestaurants › returns the response to the right endpoint

  expect(received).resolves.toEqual(expected) // deep equality

  - Expected  -  2
  + Received  + 14

    Array [
      Object {
  -     "id": 1,
  -     "name": "Sushi Place",
  +     "id": 2035,
  +  "name": "Pasta Place",
  + },
  + Object {
  +  "id": 2036,
  +  "name": "Salad Place",
  + },
  + Object {
  +  "id": 2039,
  +  "name": "Ice Cream Place",
  + },
  + Object {
  +  "id": 2040,
  +  "name": "Cookie Place",
      },
    ]
```

Wait a second, that *is* the data returned by the real back-end! Sure enough, the test is making a real call to `https://api.outsidein.dev` and retrieving results. You might be tempted to update the test to expect this data, but don't be fooled: as we mentioned in the "End-to-End Test" section of the "Vertical Slice" chapter, letting tests that run on CI hit an external service can result in test flake.

Instead, we need to stub out those external requests to make sure our test is reliable. That means it's time to install Nock.

Add `nock` as a dev dependency:

```
$ yarn add --dev nock@13.2.6
```

Next, we need to stub the request that is sent when the `api` module calls `client.get('/restaurants')`. Here's how we can do that with Nock:

```
import nock from 'nock';
import api from './api';

describe('api', () => {
  describe('loadRestaurants', () => {
    const restaurants = [{id: 1, name: 'Sushi Place'}];

    it('returns the response to the right endpoint', async () => {
      nock('https://api.outsidein.dev')
        .get(/^\/\w+\/restaurants$/)
        .reply(200, restaurants);
      await expect(api.loadRestaurants())
        .resolves.toEqual(restaurants);
    });
  });
```

We import the `nock` function from the module of the same name, then we call it in our test, passing the address of the root of our API server.

Then we chain a call to the `.get()` method to stub a `GET` request. We could have hard-coded our access token here, but since our test doesn't really care about the token, we allow our test to work with any token that is passed. Whereas Cypress used "glob"-style syntax for matching any string (`*`), Nock uses regular expressions. `/^\/\w+\/restaurants$/` will match any token, followed by the `/restaurants` path segment.

Understanding the Regular Expression

If you aren't familiar with regular expressions (regexes) or could use a refresher, here's what the bits of it mean:

- The beginning and ending `/` designate the start and end of the regex, in the same way quotes do for strings.
- `^` matches the start of the string and `$` matches the end of the string. Putting those at the start and end of the regex, respectively, mean we want to match against the *whole* path, not just a substring of it.
- Even though slashes are the delimiter for regexes, we also need slashes to appear within the regex because they're used for delimiters within URLs. To use slashes in the regex, we "escape" them by placing a backslash in front of each. So the sequence `\/` means that we will match an actual `/` in the path.

- `\w` matches any alphanumeric character (letters, numbers, or underscore), which is a good fit to match our API tokens. Appending the `+` after `\w` means we can match one or more such characters; for the sake of this test we don't care about the token length.
- `restaurants` matches against the exact string "restaurants".

Save the test and we get an error:

```
Error: Cross origin http://localhost forbidden
    at dispatchError
```

The phrase "cross origin" indicates that this error is related to the Cross-Origin Resource Sharing[26] security mechanism that web browsers use. Specifically, browsers look for HTTP headers returned by the server to see if the client code should be allowed to make this request to the server. If the request isn't coming from an "origin" that the server says is permitted, the browser won't permit the client code to see the response or to make data-modifying requests.

In our case, the `api.outsidein.dev` server is configured to return CORS headers that allow our app to access it, but our stubbed server provided by Nock doesn't return those headers by default. We need to add them ourselves. Here's how we can do so:

```
describe('api', () => {
  const responseHeaders = {'Access-Control-Allow-Origin': '*'};

  describe('loadRestaurants', () => {
    const restaurants = [{id: 1, name: 'Sushi Place'}];

    it('returns the response to the right endpoint', async () => {
      nock('https://api.outsidein.dev')
        .get(/^\/\w+\/restaurants$/)
~~        .reply(200, restaurants);~~
        .reply(200, restaurants, responseHeaders);
      await expect(api.loadRestaurants())
        .resolves.toEqual(restaurants);
```

[26] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

```
    });
  });
```

The * in the header means that all origins are allowed, which will work fine in our testing context.

Save this change and the test passes. Not only does this confirm that the method is returning the data it gets from the server, it *also* confirms that the request is being made to the right endpoint on the server. This is because we configured Nock to stub a specific endpoint, and it will only return the data if we correctly made a request to that endpoint.

Don't trust me on that, though—I don't want to trust myself either. Instead, let's confirm it. **When we write a test after the production code, we *still* want to see it go red to make sure it's ensuring what we *think* it's ensuring.** We can do this by temporarily breaking our production code and seeing whether the test catches that breakage.

First, let's try commenting out the server call entirely. Comment out the following lines in `api.js`;

```
async loadRestaurants() {
~~  const response = await client.get('/restaurants');~~
~~  return response.data;~~
  // const response = await client.get('/restaurants');
  // return response.data;
},
```

Save and we get the following failure:

```
□ api › loadRestaurants › returns the response to the right endpoint

  expect(received).resolves.toEqual(expected) // deep equality

  Expected: [{"id": 1, "name": "Sushi Place"}]
  Received: undefined

    14 |
    15 |       await expect(api.loadRestaurants())
  > 16 |         .resolves.toEqual(restaurants);
       |                   ^
```

The promise this method returns is resolving, but it doesn't resolve to the value we expect: it resolves to `undefined`. This makes sense because we've commented out the whole body of the function, so what we're getting is the default return value of an `async` function: a promise that resolves to its inner return value. Since the default inner return value of a function is `undefined`, that's what we get here.

So we've confirmed that the test will fail if we return the wrong data. Next, let's test a more subtle kind of bug. Uncomment the lines you commented in `api.js`, but change the path passed to the `.get()` method to something incorrect, like "/BAD-PATH":

```
async loadRestaurants() {
~~  // const response = await client.get('/restaurants');~~
~~  // return response.data;~~
  const response = await client.get('/BAD-PATH');
  return response.data;
},
```

Save and we get this test failure:

```
  □ api › loadRestaurants › returns the response to the right endpoint

    expect(received).resolves.toEqual()

    Received promise rejected instead of resolved
    Rejected to value: [AxiosError: Network Error]

      13 |         .reply(200, restaurants, responseHeaders);
      14 |
    > 15 |       await expect(api.loadRestaurants())
         |             ^
```

The test failure indicates that there was a network error, but it doesn't give many details. But we can see the details if we just scroll up in the test output to where it's logged to the console:

```
Error: Error: Nock: No match for request {
  "method": "GET",
  "url": "https://api.outsidein.dev/YOUR-API-KEY/BAD-PATH",
```

(In your output, "YOUR-API-KEY" will be replaced with your actual API key.)

Nock is giving us a pretty descriptive error: it saw a request for the path ending in "BAD-PATH", but Nock wasn't configured to handle that path. The way Nock responds to an unexpected path is to throw an error. This is great for precise tests to make sure we don't have unexpected HTTP requests going out.

To get our test back into working order, restore the path to the correct value, "/restaurants":

```
async loadRestaurants() {
~~  const response = await client.get('/BAD-PATH');~~
  const response = await client.get('/restaurants');
  return response.data;
},
```

Save and the test should pass again.

With that, our test for `api.loadRestaurants()` is complete. Now we just need to write the test for the `api.createRestaurant()` method. This one will go quicker as we apply what we've learned so far.

Add the following test to `api.spec.js` below `describe('loadRestaurants')`:

```
describe('createRestaurant', () => {
  const restaurantName = 'Sushi Place';
  const responseRestaurant = {id: 1, name: restaurantName};

  it('returns the response to the right endpoint', async () => {
    nock('https://api.outsidein.dev')
      .post(/^\/\w+\/restaurants$/, {name: restaurantName})
      .reply(200, responseRestaurant, responseHeaders);

    await expect(api.createRestaurant(restaurantName))
      .resolves.toEqual(responseRestaurant);
  });
});
```

This time we chain a call to `.post()` onto our `nock()` call, stubbing a POST request. We provide the same regex as before, since the POST goes to the same path as the GET.

We also provide a second argument to `.post()`: our expected POST body. Remember that, in our Cypress test, when we wanted to make sure the right data was POSTed to the server we had to use `cy.wait()` to confirm it after the request was sent. Nock, in contrast, allows you to configure the path and the required POST body at the same place in the test. If you pass an optional second argument, Nock will ensure that the request body matches it.

After this, we again chain a call to `.reply()` to configure the HTTP response. Then, with our stubbing complete, we call `api.createRestaurant` and confirm that it resolves to the same restaurant object returned by the server.

Save and the test passes. Great! …but don't trust it yet! Let's once again break our production code to make sure our test catches it.

As you did previously with `api.loadRestaurants()`, try commenting out the body of `api.createRestaurant()`; the test should fail. Then uncomment the body and change the `client.post()` path to something incorrect; the test should fail again. Restore the path to the correct value.

Finally, let's make sure we really have configured Nock to ensure the POST body is correct. Try removing the `name` from the post body in `api.js`:

```
async createRestaurant(name) {
~~  const response = await client.post('/restaurants', {name});~~
  const response = await client.post('/restaurants', {});
  return response.data;
},
```

Save, and we get a failing test and a Nock error again:

```
Error: Error: Nock: No match for request {
  "method": "POST",
  "url": "https://api.outsidein.dev/TOPDDhuzdfNDL42ZwUxFfA4C0nGEUaVl/
         restaurants",
  "headers": {
    ...
  },
  "body": "{}"
```

Look down below the `headers` entry and you'll see a `body` entry whose value is `{}`. Nock is saying that the body it received was a JSON structure representing an empty object. If we weren't sure why we were getting this error, we could compare this request to the one we configured in our test and see that the body we specified doesn't match. But in this case we know that the body is the problem because we just removed the `name` field from it—which means the test is ensuring the body is correct, just like we hoped.

Restore the `name` to the body in `api.js`, save, and the test should pass.

With this, we have an integration test of our API client confirming it's making the right requests and returning the responses in the appropriate way.

Now that the tests are written, let's assess them: are they adding enough value to be worth the cost of writing and maintaining them? Looking at these tests, I don't think they add much value beyond what our E2E tests already give us. If the API client were more complex the value might be higher, but then again it's best to *not* let wrappers of code that access external resources become complex.

But feel free to make your own decision on your projects. If you have some code that wraps third-party libraries and you find that testing it more directly would increase your confidence, integration tests like these are a good option for doing so.

13. Asynchrony in React Testing Library

In this book's exercise we had a pretty smooth experience implementing our React Testing Library tests. But when implementing your own projects you may run into more warnings and test failures related to asynchronous behavior. Let's take a look at why.

When we tested the `NewRestaurantForm`, we used the `@testing-library/user-event` library to simulate a user typing into the text field and clicking the button. In "Project Setup", when we created our app, version 13 of `user-event` was installed by default. We upgraded it to version 14 because that version included some significant improvements; specifically, it simulates asynchronous user interactions more realistically.

In a previous version of this book that used `user-event 13`, we needed a whole chapter to explore workarounds for asynchronous issues, but in `user-event 14` no workarounds are necessary. Instead of removing that workaround chapter from the book, though, I've rewritten it so that we still have a chance to learn from those asynchronous issues: we'll downgrade to `user-event` version 13, see the problems, and fix them.

This process is completely optional: for your own applications I recommend staying on the latest version of `user-event`. But going through this exercise will give you a better understanding of how asynchrony works in RTL, and it will give you practice addressing asynchronous issues that might crop up in your own tests, even in newer versions of `user-event`.

Downgrading to User Event 13

First, run your unit tests with `yarn test` and make sure they're passing. You can keep them running afterward.

Next, downgrade `user-event` to the last 13.x release:

```
$ yarn add @testing-library/user-event@13.5.0
```

Before we do anything else, we need to adjust our tests for some APIs that aren't present in version 13. In our code we currently call `userEvent.setup()` to get a `user` object to make our calls on, but in version 13 there is no such `setup()` method. Instead, we make all of our calls on `userEvent` directly.

We only have one test file that uses the `user-event` library: `NewRestaurantForm.spec.js`. In that file, remove the code that sets up the `user` object:

```
const serverError = 'The restaurant could not be saved. Please try again.';

~~let user;~~
let createRestaurant;

function renderComponent() {
  ~~user = userEvent.setup();~~
  createRestaurant = jest.fn().mockName('createRestaurant');
```

Next, everywhere there is a call to a method on `user`, we need to make two changes to it:

- We need to update the call to be on the `userEvent` object instead of `user`, and
- We need to remove the `await` keyword, because in version 13 these calls are synchronous rather than asynchronous.

Make those changes everywhere you see `user` in your test file. Here is one example of the changes (the full diff is omitted for brevity):

```
describe('when filled in', () => {
  async function fillInForm() {
    renderComponent();
    ~~await user.type(~~
    userEvent.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    ~~await user.click(screen.getByText('Add'));~~
    userEvent.click(screen.getByText('Add'));
  }
```

Note that as a result of removing the `await` keyword, some functions that are marked as `async` are no longer asynchronous. Leave them marked as `async` for now anyway; all of them will end up becoming asynchronous again over the course of this chapter.

Once you've replaced `await user` with `userEvent` everywhere in your tests, save the file to rerun the tests.

You should see two test failures and a lot of warnings. Ouch! We need to get those failing tests passing again, and it would be best if we can address the warnings as well.

Understanding `act()` Warnings

When you're changing tests, whenever possible it's best to avoid breaking more than one test at a time. But when you're changing the version of a dependency, it may be unavoidable. In a case like this a good troubleshooting technique is to narrow the focus to fixing one thing at a time. We can use two techniques to do so.

First, we can limit the test runner to just one file. If you ran the `yarn test` command Jest should keep your tests running in watch mode. Press the P key on your keyboard to filter by file name. Type "form" and press return. This limits our run to files that have "form" in the file name, which in our case is only `NewRestaurantForm.spec.js`.

We can narrow things down even further by using Jest's `.only` method to run just one group or test, allowing us to focus on fixing one scenario at a time. It can also make it easier to see the cause of warnings, because it's not always obvious which test is raising them in Jest's output.

Start by adding `.only` to the "initially" group:

```
~~describe('initially', () => {~~
describe.only('initially', () => {
  it('does not display a validation error', () => {
```

Save and the tests will rerun. You should see no errors or warnings, confirming that the "initially" block is working fine—nothing for us to do here.

Next, remove the `.only` from the "initially" group again and add it to the "when filled in" group. This time, you should see a test failure and four warnings, one for each of the tests in the group. The warning says:

```
Warning: An update to NewRestaurantForm inside a test was not wrapped in
act(...).

When testing, code that causes React state updates should be wrapped into
act(...):

act(() => {
  /* fire events that update state */
});

/* assert on the output */

This ensures that you're testing the behavior the user would see in the browser.
Learn more at https://reactjs.org/link/wrap-tests-with-act
```

What is this `act()` function the warning is talking about?

Under the hood, React Testing Library uses `react-dom/test-utils`, a set of official testing utilities provided by React. One such utility is `act()`. Its purpose (described in React's Testing Recipe for `act()`[27]) is to let React know that some interaction with the component is happening—either rendering or an event that can update state. When `act()` is called, React will ensure "all updates related to these 'units' have been processed and applied to the DOM before you make any assertions". In other words, using `act()` makes sure you won't run into a situation where a state change has been requested but it isn't yet visible to your test.

When you use React Testing Library, its test helpers *automatically* use `act()`, so you should almost never need to call it explicitly. But if RTL uses `act()` automatically, why are we getting a warning that something wasn't wrapped in `act()`?

When this warning occurs in an RTL test, it's usually because a state change happened after the end of your test. Usually, the warning output will tell which state it was that changed:

[27] https://reactjs.org/docs/testing-recipes.html#act

```
    at NewRestaurantForm
      (.../opinion-ate/src/components/NewRestaurantForm.js:8:36)

  19 |       try {
  20 |         await createRestaurant(name);
> 21 |         setName('');
     |         ^
```

After the test finishes running, `setName()` is called, setting the `name` to the empty string. Is this really a problem, though?

Kent C. Dodds, the creator and maintainer of React Testing Library, wrote a helpful blog post about this topic titled "Fix the 'not wrapped in act(…)' warning"[28]. In it he points out that this warning is beneficial from a test coverage standpoint. If a state update happens after the end of your test, that's behavior in your component that you aren't testing–behavior for which your tests won't protect you against regressions. Because of that, the recommended way to resolve `act()` warnings in RTL tests is to use RTL test helpers to make assertions about the state changes that happen. Not only does this wrap those changes in an `act()`, it also ensures they're covered by a test reflecting what the user experiences.

Waiting on Text Input Changes With `waitFor`

Now that we understand the `act()` warnings a bit better, we need to decide what to do first. It might be tempting to start out ignoring the warnings and focus on fixing the error. But compare the test failure and warnings. The failure is:

```
    Expected the element to have value:

    Received:
      Sushi Place

      45 |       await fillInForm();
      46 |       expect(screen.getByPlaceholderText('Add Restaurant'))
    > 47 |         .toHaveValue('');
         |          ^
```

[28] https://kentcdodds.com/blog/fix-the-not-wrapped-in-act-warning

The failure says the text input should have been cleared, but it was not. Remember what the warnings said: the text input was being cleared after the test finished. In this case, the warnings pinpoint the cause of our test failure! The test is finishing before the text input gets a chance to clear.

In this case, then, solving the test failure and solving the warnings are one and the same: we need our tests to wait until the text input is cleared. How can we do this?

To wait for the value of the text input to be set to the empty string, we can use RTL's `waitFor` function. To use `waitFor`, we pass it a function that it can call repeatedly. That function can contain an expectation; `waitFor` will rerun it until the expectation passes, at which point it allows the test to continue. If the expectation never passes within a configured timeout period of several seconds, the test will fail.

For our "when filled in" block, let's configure a `waitFor` function to look for the empty text input. Because all four tests are emitting the warning, let's put the `waitFor` into the `fillInForm()` setup helper function that each test calls:

```
async function fillInForm() {
  renderComponent();
  userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await waitFor(() =>
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue(''),
  );
}
```

Save and both the test failure and the warnings are resolved. Great!

With that, we're ready to investigate the next group for failures and warnings. Move the `.only` to the "when empty" group and save. No errors or warnings; that block is fine.

Move the `.only` to the next group again, this time to "when correcting a validation error". There is only one test in this group, and it passes but gives a warning about `setName` again.

Fix it in the same way you did in the "when filled in" group, by waiting for the text input to be cleared:

```
async function fixValidationError() {
  renderComponent();
  createRestaurant.mockResolvedValue();

  userEvent.click(screen.getByText('Add'));

  userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await waitFor(() =>
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue(''),
  );
}
```

Save and the warning should go away.

Waiting on Elements to Appear With `findBy*`

Next, move the `.only` to “when the store action rejects”. Save and one of the test fails, and both give the warning. This time, it’s not `setName()` that is triggering it:

```
  21 |         setName('');
  22 |       } catch {
> 23 |         setServerError(true);
     |         ^
```

So `setServerError` is what is being called after the end of the test.

To fix this, we want to wait for the server error message to be displayed. We could use `waitFor` again, but in cases where we’re waiting for an element to appear, there’s a simpler way: a `screen.findBy*()` method. These methods allow you to wait for a matching element to appear on the screen. In our case, `findByText` will work:

```
async function fillInForm() {
  renderComponent();
  createRestaurant.mockRejectedValue();

  userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await screen.findByText(serverError);
}
```

Save and the test failure and warnings should go away.

Finally, move the `.only` to the "when retrying after the store rejects" group. This group only has one test, and it doesn't fail, but does throw a warning triggered by `setServerError`.

This test submits to the server twice, and it's the first submission that errors, so wait for the error message at that point:

```
async function retrySubmittingForm() {
  renderComponent();
  createRestaurant.mockRejectedValueOnce().mockResolvedValueOnce();

  userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await screen.findByText(serverError);

  userEvent.click(screen.getByText('Add'));
}
```

Save, and notice that we are still getting a warning, but about a different state change: `setName`. This is triggered after the second submission to the server.

To fix this warning, we should wait for the result of the second state change as well. This change results in the text input being cleared, and we know how to wait on that:

```
  userEvent.click(screen.getByText('Add'));

  await screen.findByText(serverError);

  userEvent.click(screen.getByText('Add'));

  await waitFor(() =>
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue(''),
  );
}
```

Save and the warning is gone.

All our tests should now be corrected. Remove the `.only` to run the entire file of tests again. They should all pass without warnings!

Removing Duplication

Now that our test run is green, let's review the new state of our tests to see how we feel about it.

To start, look at the "when the store action rejects" group. Something is a bit confusing about them: after we wait for the error message with `findBy` in the setup function, we check for that message a second time in one of the test cases:

```
async function fillInForm() {
  //...
  await screen.findByText(serverError);
}

it('displays a server error', async () => {
  await fillInForm();
  expect(screen.getByText(serverError))
    .toBeInTheDocument();
});
```

That `expect()` should never be able to fail because, if it did, the `findBy` should have failed first and prevented us from reaching the `expect()`. It seems a little strange to have an expectation that will never fail. What are our options?

- We could remove that test since we don't need it; if the server error is missing the other test in this group will catch it. But this would mean our tests would no longer document our application's behavior as clearly: instead of a test case whose name explicitly documents that that error message will be present, the check is hidden in a setup helper function.
- We could leave that test to document the behavior of the code even though it is guaranteed to pass. But this may be confusing to readers of the test, and one of them may remove it in the future.
- We could change how our tests are organized by combining the two tests in this `describe` into a single test with multiple assertions.

Let's see what it would look like to combine our two tests into one. If we do, we can remove the first test entirely, because the call to `findBy` covers the purpose of it:

```
~~describe('when the store action rejects', () => {~~
~~  async function fillInForm() {~~
it('displays an error when the store action rejects', async () => {
    renderComponent();
    createRestaurant.mockRejectedValue();

    await userEvent.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    userEvent.click(screen.getByText('Add'));

    await screen.findByText(serverError);
~~  }~~

~~  it('displays a server error', async () => {~~
~~    await fillInForm();~~
~~    expect(screen.getByText(serverError))~~
~~      .toBeInTheDocument();~~
~~  });~~

~~  it('does not clear the name', async () => {~~
~~    await fillInForm();~~
    expect(screen.getByPlaceholderText('Add Restaurant'))
```

```
      .toHaveValue(restaurantName);
~~  });~~
});
```

Save and the test should pass.

Here’s the final code without the edits:

```
it('displays an error when the store action rejects', async () => {
  renderComponent();
  createRestaurant.mockRejectedValue();

  await userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await screen.findByText(serverError);

  expect(screen.getByPlaceholderText('Add Restaurant'))
    .toHaveValue(restaurantName);
});
```

Let’s think about what benefits we’ve gotten from combining the tests:

- It’s bit shorter in terms of amount of test code.
- It’s a bit easier to see the steps the test takes because you can follow them in order; there isn’t a helper function that you need to look up to see what it does.
- There’s a straightforward way for us to check for the error message only once, without introducing confusing duplication.

But there are some downsides to this form of the test too:

- The test name doesn’t describe one of the two assertions: the fact that the name field is not cleared. We could try to come up with a test name that does, but it would likely be long.

- We can no longer see a list of all the behaviors that are being checked at a glance. Previously when Jest outputted the results it included several tests, one for each behavior. But now it only outputs one test name, and if we want to figure out the behaviors we have to read through the test's implementation. What's more, the `findBy` isn't obviously an expectation of behavior, so it would be easy to miss.

While we mull over what we think about those trade-offs, let's move on to another test group: "when filled in". In this group, we run into the same test duplication as in our first `describe` block: the check we're waiting on in the helper is duplicated in one of the test cases:

```
async function fillInForm() {
  // ...
  await waitFor(() =>
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue(''),
  );
}

it('clears the name', async () => {
  await fillInForm();
  expect(screen.getByPlaceholderText('Add Restaurant'))
    .toHaveValue('');
});
```

To remove this duplication, let's combine the four tests into one test with multiple assertions:

~~describe('when filled in', () => {~~
~~ async function fillInForm() {~~

```
it('allows submitting the form', async () => {
    renderComponent();
    await userEvent.type(
      screen.getByPlaceholderText('Add Restaurant'),
      restaurantName,
    );
    userEvent.click(screen.getByText('Add'));

    await waitFor(() =>
      expect(screen.getByPlaceholderText('Add Restaurant'))
```

```
        .toHaveValue(''),
    );
~~  }~~

~~  it('does not display a validation error', async () => {~~
~~    await fillInForm();~~
    expect(screen.queryByText(requiredError))
      .not.toBeInTheDocument();
~~  });~~

~~  it('does not display a server error', async () => {~~
~~    await fillInForm();~~
    expect(screen.queryByText(serverError))
      .not.toBeInTheDocument();
~~  });~~

~~  it('calls createRestaurant with the name', async () => {~~
~~    await fillInForm();~~
    expect(createRestaurant)
      .toHaveBeenCalledWith(restaurantName);
~~  });~~

~~  it('clears the name', async () => {~~
~~    await fillInForm();~~
~~    expect(screen.getByPlaceholderText('Add Restaurant'))~~
~~      .toHaveValue('');~~
~~  });~~
});
```

Save and the test should pass with no warnings.

Here’s the final code without the edits:

```
it('allows submitting the form', async () => {
  renderComponent();
  await userEvent.type(
    screen.getByPlaceholderText('Add Restaurant'),
    restaurantName,
  );
  userEvent.click(screen.getByText('Add'));

  await waitFor(() =>
    expect(screen.getByPlaceholderText('Add Restaurant'))
      .toHaveValue(''),
  );

  expect(screen.queryByText(requiredError))
    .not.toBeInTheDocument();
  expect(screen.queryByText(serverError))
    .not.toBeInTheDocument();
  expect(createRestaurant)
    .toHaveBeenCalledWith(restaurantName);
});
```

What are the upsides of this form of the test?

- This time it's a *lot* shorter because we've combined four tests into one.
- Not having a helper function to follow is even more of an improvement, because it was previously called four times.

What about downsides?

- This time the test name was hard to write and is unclear. It says "allows submitting the form;" okay, but what *happens* when you submit the form? The multiple behaviors that happen can't easily be summarized in a short test name.
- This test has even more going on than the first one, making it harder to understand. There are four behaviors being checked, so it's even harder to see all of them at a glance.

Assessment

In his blog post mentioned earlier, Kent C. Dodds said that if a state update happens after the end of your test then there is a behavior that isn't covered by test. This is correct if you have one test that is making multiple assertions.

But if you follow the "one assertion per test" principle, the situation is different. For a given scenario we may have several tests, each of which makes one assertion about one aspect of that scenario. Hopefully we have one test that makes an assertion on that final state change, but not *all* the tests will.

So if we're following the "one assertion per test" principle, a state change after the end of the test doesn't guarantee we have untested behavior. But we can still silence the `act()` warnings, using one of several techniques we looked at in this chapter. We could add test code that waits for those state changes to happen, but those changes might make our tests less clear. Or, we could write switch to multiple assertions per test, taking on both the pros and the cons of that test style.

Thankfully, in this exercise and in many other cases, `user-event` version 14 solves this dilemma for us. Because version 14's user event simulation code involves asynchrony, this gives our code time for state updates to happen before the test proceeds. As a result, we can organize our tests whichever way we prefer: either "one assertion per test" as I've recommended or multiple assertions per test.

What should you do if you can't update to `user-event 14` yet, or if even in `user-event 14` you run into an `act()` warning? If that happens, take the warning as an opportunity to make sure you understand what's happening in the component. Figure out which state change is triggering the warning and what causes that state change. If the state change is visible in the UI, try to write a `findBy*()` or `waitFor()` call for it. And if the state change *isn't* visible in the UI, try to find a way to make it visible. This will not only make your code more testable, it could also be more informative to your users.

This investigation process can feel tedious when you're just trying to get a test to pass, but it's also a blessing in disguise. Asynchronous programming, especially asynchronous UI programming, is inherently complex and hard to follow; we developers need all the help we can get. React introduced `act()` warnings as a safety check to help us write more reliable interfaces. Instead of just trying to silence those warnings, try to learn from them—your application will be better for it.

There's one other architectural change that `act()` warnings can steer us toward: putting as much of our code as possible into plain JavaScript functions. Writing framework-agnostic code has a number of benefits, and easier testing is one of them. Think about our Redux tests: the interface to the data layer is just normal JavaScript function calls. We don't need any special APIs like `act()`; we just `await` returned promises and we're good to go. When you write code that uses standard JavaScript promises directly, your tests are often easier to write and understand.

14. Next Steps

You've finished this book and gotten a taste of outside-in test-driven development on the front-end. I wrote this book because I hope the principles that have helped me will help other developers. If you've benefited from it and/or have feedback, I would love to hear from you—please get in touch! https://outsidein.dev/connect has a number of ways connect with me, including my social media, email address, and mailing list.

If you choose to continue on in outside-in TDD, then this book will be only the beginning of your journey. Here are some suggestions for ways to take your learning further.

A Community of Practice

Agile development really sank in for me once I started doing it alongside other people. Fellow agile developers provides accountability to stick with your principles and insights when it's not clear what to do.

Here are some suggestions for how to find others to do agile development with:

- An easy first step is to join the book's online chat[29]. You can discuss what we've learned together and how we're applying it.
- Find a local meetup[30] focused on agile development practices. "Software crafters" or "software craftsmanship" are common terms for such meetups.
- Attend a workshop on TDD or refactoring. One instructor I would recommend is Sandi Metz[31], who is so amazing that it is *absolutely* worth learning a little Ruby programming to take her course.
- Give a copy of this book as a gift to someone you think would benefit from it, and talk together about your experiences with it.
- Start a book club with your coworkers to read through this book and work through the exercise together. I can provide team discounts to make this easier, so please reach out to me at https://outsidein.dev/connect

[29] https://link.outsidein.dev/chat

[30] https://www.meetup.com

[31] https://www.sandimetz.com/courses

- The next time you are looking for a new job, look for one that prioritizes agile development practices. Just hearing them say “we do agile” or “we do testing” isn’t enough: ask them what their agile process is like, and see if they have a thorough understanding of small stories, TDD, refactoring, and evolutionary design.

Testing Tool Documentation

In the same way that an experienced developer will learn their programming languages and frameworks deeply, it’s important to learn testing tools deeply as well. This will show you what features you can utilize in your tests, give you ideas for how to test, and prevent buggy tests due to misunderstanding the functionality of the testing tool. All of the tools we used have excellent guides and I would recommend reviewing through them thoroughly:

- Cypress web site[32]
- Jest web site[33]
- React Testing Library web site[34]

Books

If you get the opportunity to work on a team with someone who has helped push agile practice forward, take it. Unfortunately, we won’t all get the opportunity to do so, but many of them have recorded insights in books to share them with a broader audience. Here are some recommendations.

Outside-In TDD

Growing Object-Oriented Software, Guided by Tests

By Steve Freeman and Nat Pryce. This is the book that introduced outside-in test-driven development. The authors are also the inventors of mock objects, and over the course of the book they illustrate how mocking is intended to be used to isolate parts of your code, as we’ve done in our exercise. This book will help you develop an even deeper understanding of the way outside-in TDD addresses change in software by guiding you to code that has a high degree of both external and internal quality.

[32] https://www.cypress.io

[33] https://jestjs.io

[34] https://testing-library.com/react

Test Patterns

xUnit Test Patterns

As we went through the exercise there were many moments where we had to make a decision about how to organize our tests. Most of these decisions were informed by *xUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros. This book is a guide to creating high-quality tests in any programming language, and its principles apply as well to front-end JavaScript as anywhere else. This book also provides a *language* for talking about test patterns you may have seen, which can help make conversations about organizing tests more productive.

Refactoring

Over the course of this book we did a little refactoring, but not much: by and large the functionality we wrote was pretty straightforward. Real applications are different: as they grow and change, the code gets complex and it's essential to refactor to avoid getting bogged down. Here are two resources for learning about refactoring.

Refactoring: Improving the Design of Existing Code, Second Edition

By Martin Fowler. This is the original book that introduced refactoring as a disciplined process. It includes a comprehensive reference for different kinds of refactorings, helping you understand when you would want to apply them and how to do so. The second edition of the book has all of its examples written in JavaScript, so it's very easy for front-end developers to pick up.

99 Bottles of OOP

By Sandi Metz, Katrina Owen, and TJ Stankus. This book walks through one extended refactoring process step-by-step, giving you the experience of what refactoring over time looks like. It's available in JavaScript and several other programming languages. Don't let the "object-oriented" in the name fool you: even if you write your code in a functional-programming style, you'll be able to apply this book's principles of identifying code smells, listening to the code, and making small changes.

Agile Methodology

This book has briefly introduced agile development practices, but there is much more to consider about the broader scope of doing agile as a team. There are many books on agile development; here are two I would recommend.

Extreme Programming Explained: Embrace Change

By Kent Beck, the creator of Extreme Programming, one of the early agile methodologies. Beck is also the creator of test-driven development. Unlike some other agile methodologies, Extreme Programming is not agnostic about technical practices, but rather makes very specific recommendations. This is important because you can't deliver reliable software on a regular basis without applying technical practices that keep the software reliable and development speed consistent. *Extreme Programming Explained* also gets into the big-picture values behind Extreme Programming, such as the fact that humans have limited capacities and we should design software practices that acknowledge and support that, rather than deny it.

The Nature of Software Development

By Ron Jeffries. This is a recent attempt to restate the values and priorities of agile development in a methodology-agnostic way. Rather than teaching a complex approach, it lays out principles common to agile methods and makes a case for them.

Epilogue

I decided to write this book because I found that I couldn't stop thinking about, talking about, and advocating for outside-in TDD and other agile development practices. And the reason I couldn't stop is that I've seen them solve a problem that I haven't seen solved any other way: the problem of development slowdown over time due to code that is hard to work with.

New languages and frameworks don't fix this problem, because you can make a mess in any language. More process doesn't fix the problem if the process doesn't account for change. Trying harder doesn't fix the problem, because we're human and have limited capacity.

The reason agile development practices work is because they're based on a realistic view of the world in which software development occurs. Alternate approaches to development envision a world where requirements can be fully understood and perfectly executed. That world is appealing to programmers and businesspeople alike, but its call is a siren song, and if you follow it you'll suffer for it. Instead, agile development recognizes that in the world we live in change is inevitable and people have limited capacities. It provides practices that work with the forces of this world, not merely to weather them, but to thrive because of them.

These practices are the best way I've found to deliver value to my employers and to have a smooth and calm development experience. It's no exaggeration to say that they've had a life-changing effect on me. And now you have a foundation in these practices and a taste of the results as well. I hope you’ll try putting them to use in your development work. I think you’ll like the results.

Thanks for reading, and keep in touch!