

2ND EDITION

Mastering React Test-Driven Development

Build simple and maintainable web apps with
React, Redux, and GraphQL

**DANIEL IRVINE**

Foreword by Justin Searls, VP of Engineering at Test Double

Mastering

React Test-Driven Development

Second Edition

Build simple and maintainable web apps with React, Redux,
and GraphQL

Daniel Irvine



BIRMINGHAM—MUMBAI

Mastering React Test-Driven Development

Second Edition

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Bhavya Rao

Senior Editor: Aamir Ahmed

Senior Content Development Editor: Feza Shaikh

Technical Editor: Saurabh Kadave

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Aparna Bhagat

Marketing Coordinator: Anamika Singh and Marylou De Mello

First published: May 2019

Second edition: September 2022

Production reference: 2130922

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80324-712-0

www.packtpub.com

To Nige, my first mentor, who taught me the importance of slowing down.

– Daniel Irvine

Foreword

For as long as **test-driven development (TDD)** has existed, there has been debate over whether TDD is feasible for user interface work—even among its staunchest proponents. When people built tools to prove that it’s possible, the debate shifted to whether TDD is as valuable for UIs, considering that programmers can get rapid feedback by seeing and interacting with an interface. Even when TDD’s value is evidenced by smaller, more consistent, single-responsibility units of UI code, some would question whether the same could have been accomplished with less effort by simply following established design patterns.

Is TDD worth your time? Will it result in fewer bugs in your code? Will it improve the design of your system? Will it make future maintenance easier? Maybe.

In the fifteen-or-so years that I’ve been learning, practicing, and teaching test-driven development I have oscillated back-and-forth. Sometimes I relentlessly pursue 100% code coverage, and other times I’ll build worrisomely large applications with no automated tests at all. What I’ve found—and what might surprise some readers—is that my code basically turns out the same whether I practice TDD or not: a similar frequency of bugs, the same idiosyncratic design, and no more or less a burden to maintain.

(At this point, you’d be right to start wondering what this ambivalent foreword is doing in a book designed to teach you TDD.)

The reason that TDD itself doesn’t impact my code very much is because my years of practice have utterly and irrevocably changed me.

I write small single-purpose units, because I’ve felt the sheer exhaustion of writing tests of sprawling unfocused objects.

I avoid mixing levels of abstraction, because I’ve been hopelessly lost in mazes of mock objects that combine testing logic with specifying interactions.

I segregate code coupled to frameworks from feature logic, because I’ve contorted too many tests to fit dependencies that weren’t meant to be tested.

Rigorously practicing TDD transformed my career. Not because it’s the One True Way to program, but because it forces you to ceaselessly ask “how would we test that?” TDD is incredibly challenging at first, but patterns gradually emerge that result in easy-to-test code. And code that’s easy to test, is easy to write. And use. And maintain.

Wherever you are in your journey, I hope this book brings you closer to a similar destination.

*Justin Searls
VP of Engineering at Test Double*

testdouble.com

Contributors

About the author

Daniel Irvine is a software consultant based in London. He works with a variety of languages including C#, Clojure, JavaScript, and Ruby. He's a mentor and coach for junior developers and runs TDD and XP workshops and courses. When he's not working, he spends time cooking and practicing yoga. He co-founded the Queer Code London meetup and is an active member of the European software craft community.

I would like to thank the technical reviewer of this edition of the book, Emmanuel Demey, for spotting the weakest points of the text and helping me improve them. I continue to be grateful for my friend and technical reviewer from the first edition, Raimo Radczewski, who is also the author of the expect-redux package that is used in this book.

The team at Packt has been supportive and professional throughout, giving me exactly what I needed to finish the book. For this second edition, I am thankful for the detailed and thoughtful editing that Feza Shaikh has undertaken, helping to deliver the book ahead of schedule. My technical editor, Saurabh Kadave, spotted plenty of mistakes that would have otherwise gone unnoticed. Aamir Ahmed, Bhavya Rao, and Manthan Patel were also instrumental in the production of the second edition.

From the first edition, I am thankful for the hard work of Keagan Carneiro, Sachin Sunilkumar from Packt, and also my friends and colleagues Charlotte Payne, Dan Pelensky, Isidro López, Makis Otman, Sam Szreter, Zach Shaw, Brendan Murphy, and Lucy Monie Hall.

Finally, thank you to all the readers of the first edition who took the time to send me comments and feedback, all of which were invaluable. This second edition is better because of your help.

About the reviewer

Emmanuel Demey works with the JavaScript ecosystem every day, and he spends his time sharing his knowledge with anybody. His first goal at work is to help the people that he works with. He speaks at French conferences (Devfest Nantes, Devfest Toulouse, Sunny Tech, Devoxx France, and others) about anything related to the web platform: JavaScript frameworks (Angular, React.js, Vue.js), accessibility, Nest.js, and so on. He has been a trainer for 10 years at Worldline and Zenika (two French consulting companies). He is also a co-leader of the Google Developer of Lille group and a co-organizer of the Devfest Lille conference.

Table of Contents

Preface

xv

Part 1 – Exploring the TDD Workflow

1

First Steps with Test-Driven Development		3	
Technical requirements	4	Triangulating to remove hardcoding	20
Creating a new React project from scratch	4	Backtracking on ourselves	22
Installing NPM	5	Refactoring your work	25
Creating a new Jest project	5	Sharing setup code between tests	25
Bringing in React and Babel	7	Extracting methods	26
Displaying data with your first test	8	Writing great tests	27
Writing a failing test	8	Red, green, refactor	28
Make it pass	16	Streamlining your testing process	29
Making use of the act test helper	18	Summary	30
		Further reading	30

2

Rendering Lists and Detail Views		33	
Technical requirements	34	Selecting data to view	43
Sketching a mock-up	34	Initial selection of data	43
Creating the new component	35	Adding events to a functional component	45
Specifying list item content	40	Manually testing our changes	49
		Adding an entry point	49
		Putting it all together with webpack	51

Summary	54	Further reading	55
Exercises	54		

3

Refactoring the Test Suite	57
-----------------------------------	-----------

Technical requirements	58	Extracting DOM helpers	70
Pulling out reusable rendering logic	58	Summary	73
Creating a Jest matcher using TDD	61	Exercises	73
		Further reading	74

4

Test-Driving Data Input	75
--------------------------------	-----------

Technical requirements	75	Duplicating tests for multiple form fields	94
Adding a form element	76		
Accepting text input	78	Nesting describe blocks	94
Submitting a form	84	Generating parameterized tests	95
Submitting without any changes	84	Solving a batch of tests	98
Preventing the default submit action	87	Modifying handleChange so that it works with multiple fields	100
Submitting changed values	90	Testing it out	100
		Summary	101
		Exercises	101

5

Adding Complex Form Interactions	103
---	------------

Technical requirements	103	Adding a header column	113
Choosing a value from a select box	104	Adding a header row	117
Providing select box options	105	Test-driving radio button groups	120
Preselecting a value	108	Hiding input controls	122
Constructing a calendar view	111	Selecting a radio button in a group	127
Adding the table	112		

Handling field changes through a component hierarchy	129	Extracting a test props object	140
Reducing effort when constructing components	136	Summary	144
Extracting test data builders for time and date functions	137	Exercises	145
		Further reading	145

6

Exploring Test Doubles **147**

Technical requirements	148	Stubbing fetch responses	167
What is a test double?	148	Upgrading spies to stubs	168
Learning to avoid fakes	149	Acting on the fetch response	168
Submitting forms using spies	149	Displaying errors to the user	174
Untangling AAA	150	Grouping stub scenarios in nested describe contexts	177
Making a reusable spy function	152		
Using a matcher to simplify spy expectations	154	Migrating to Jest's built-in test double support	178
Spying on the fetch API	157	Using Jest to spy and stub	178
Replacing global functions with spies	158	Migrating the test suite to use Jest's test double support	179
Test-driving fetch argument values	159	Extracting fetch test functionality	182
Reworking existing tests with the side-by-side implementation	162		
Improving spy expectations with helper functions	166	Summary	184
		Exercises	184
		Further reading	184

7

Testing `useEffect` and Mocking Components **185**

Technical requirements	185	Fetching data on mount with <code>useEffect</code>	191
Mocking child components	186	Understanding the <code>useEffect</code> hook	191
How to mock components, and why?	186	Adding the <code>renderAndWait</code> helper	193
Testing the initial component props	188	Adding the <code>useEffect</code> hook	194
		Testing the <code>useEffect</code> dependency list	198

Building matchers for component mocks	200	Checking multiple instances of the rendered component	205
Variants of the jest.mock call	204	Alternatives to module mocks	206
Removing the spy function	205	Summary	207
Rendering the children of mocked components	205	Exercises	207
		Further reading	207

8

Building an Application Component	209		
Technical requirements	209	Making use of callback values	225
Formulating a plan	210	Summary	227
Using state to control the active view	211	Exercises	228
Test-driving callback props	219		

Part 2 – Building Application Features

9

Form Validation	231		
Technical requirements	231	Handling server errors	252
Performing client-side validation	232	Indicating form submission status	254
Validating a required field	233	Testing state before promise completion	255
Generalizing validation for multiple fields	237	Refactoring long methods	258
Submitting the form	246	Summary	258
Extracting non-React functionality into a new module	249	Exercises	259
		Further reading	259

10

Filtering and Searching Data	261		
Technical requirements	262	Paging through a large dataset	268
Displaying tabular data fetched from an endpoint	263	Adding a button to move to the next page	268
		Adjusting the design	272

Adding a button to move to the previous page	274	Performing actions with render	
Forcing design changes using tests	277	props	284
Filtering data	279	Testing render props in additional render contexts	287
Refactoring to simplify the component design	283	Summary	290
		Exercises	291

11

Test-Driving React Router	293
----------------------------------	------------

Technical requirements	293	Using the Routes component to replace a switch statement	297
Designing React Router applications from a test-first perspective	294	Using intermediate components to translate URL state	300
A list of all the React Router pieces	294	Testing router links	304
Splitting tests when the window location changes	294	Checking the page for hyperlinks	305
Up-front design for our new routes	295	Mocking the Link component	306
Testing components within a router	296	Testing programmatic navigation	310
The Router component and its test equivalent	296	Summary	312
		Exercise	312
		Further reading	312

12

Test-Driving Redux	313
---------------------------	------------

Technical requirements	314	Test-driving a saga	324
Up-front design for a reducer and a saga	314	Using expect-redux to write expectations	325
Why Redux?	314	Making asynchronous requests with sagas	328
Designing the store state and actions	315	Switching component state for Redux state	334
Test-driving a reducer	316	Submitting a React form by dispatching a Redux action	334
Pulling out generator functions for reducer actions	322	Making use of store state within a component	337
Setting up a store and an entry point	323	Navigating router history in a Redux saga	341

Summary	344	Further reading	344
Exercise	344		

13

Test-Driving GraphQL	345
-----------------------------	------------

Technical requirements	346	Test-driving a singleton instance of Environment	355
Compiling the schema before you begin	346	Fetching GraphQL data from within a component	356
Testing the Relay environment	346	Summary	369
Building a performFetch function	347	Exercises	369
Test-driving the Environment object construction	351	Further reading	370

Part 3 – Interactivity

14

Building a Logo Interpreter	373
------------------------------------	------------

Technical requirements	373	Changing keyboard focus	402
Studying the Spec Logo user interface	374	Writing the reducer	402
Undoing and redoing user actions in Redux	376	Focusing the prompt	405
Building the reducer	376	Requesting focus in other components	408
Building buttons	390	Summary	409
		Further reading	409
Saving to local storage via Redux middleware	393		
Building middleware	394		

15

Adding Animation	411
-------------------------	------------

Technical requirements	412	Designing animation	412
-------------------------------	-----	----------------------------	-----

Building an animated line component	414	Canceling animations with cancelAnimationFrame	429
Animating with requestAnimationFrame	417	Varying animation behavior	431
		Summary	437
		Exercises	438

16

Working with WebSockets	439		
Technical requirements	439	Streaming events with redux-saga	454
Designing a WebSocket interaction	440	Updating the app	460
The sharing workflow	440	Summary	463
The new UI elements	442	Exercises	463
Splitting apart the saga	442	Further reading	463
Test-driving a WebSocket connection	443		

Part 4 – Behavior-Driven Development with Cucumber

17

Writing Your First Cucumber Test	467		
Technical requirements	469	Writing your first Cucumber test	470
Integrating Cucumber and Puppeteer into your code base	469	Using data tables to perform setup	478
		Summary	483

18

Adding Features Guided by Cucumber Tests	485		
Technical requirements	487	Fixing Cucumber tests by test-driving production code	496
Adding Cucumber tests for a dialog box	487	Adding a dialog box	496
		Updating sagas to a reset or replay state	502

Avoiding timeouts in test code	505	Updating step definitions to use waitForSelector	508
Adding HTML classes to mark animation status	506	Summary	511
		Exercise	511

19

Understanding TDD in the Wider Testing Landscape	513		
Test-driven development as a testing technique	513	System tests and end-to-end tests	519
		Acceptance tests	519
Best practices for your unit tests	514	Property-based and generative testing	519
Improving your technique	515	Snapshot testing	520
		Canary testing	521
Manual testing	516	Not testing at all	521
Demonstrating software	516	When quality doesn't matter	521
Testing the whole product	517	Spiking and deleting code	522
Exploratory testing	517		
Debugging in the browser	518	Summary	523
Automated testing	518	Further reading	523
Integration tests	518		
Index			525
Other Books You May Enjoy			538

Preface

This is a book about dogma. *My* dogma. It is a set of principles, practices, and rituals that I have found to be extremely beneficial when building React applications. I try to apply these ideas in my daily work, and I believe in them so much that I take every opportunity to teach others about them. That's why I've written this book: to show you the ideas that have helped me be successful in my own career.

As with any dogma, you are free to make your own mind up about it. There are people who will dislike everything about this book. There are those who will love everything about this book. Yet more people will absorb some things and forget others. All of these are fine. The only thing I ask is that you maintain an open mind while you follow along and prepare to have your own dogmas challenged.

Test-driven development (TDD) did not originate in the JavaScript community. However, it is perfectly possible to test-drive JavaScript code. And although TDD is not common in the React community, there's no reason why it shouldn't be. In fact, React as a user interface platform is a good fit for TDD because of its elegant model of functional components and state.

So, what is TDD, and why should you use it? TDD is a process for writing software that involves writing tests, or specifications, before writing any code. Its practitioners follow it because they believe that it helps them build and design higher-quality software with longer lifespans, at a lower cost. They believe it offers a mechanism for communicating about design and specification that *also* doubles up as a rock-solid regression suite. There isn't much empirical data available that *proves* any of that to be true, so the best you can do is try it out yourself and make your own mind up.

Perhaps most importantly for me, I find that TDD removes the *fear* of making changes to my software and makes my working days much less stressful than they used to be. I don't worry about introducing bugs or regressions into my work because the tests protect me from that.

TDD is often taught with toy examples: to-do lists, temperature converters, tic-tac-toe, and so on. This book teaches two real-world applications. Often, the tests get hairy. We will hit many challenging scenarios and come up with solutions for all of them. There are over 500 tests contained in this book, and each one will teach you something.

Before we begin, a few words of advice.

This is a book about **first principles**. I believe that learning TDD is about understanding the process in exceptional detail. For that reason, we will *not* use React Testing Library. Instead, we will build our own test helpers. I am not suggesting that you should avoid these tools in your daily work – I use them myself – but I am suggesting that going without them while you learn is a worthwhile adventure. The benefit of doing so is a deeper understanding and awareness of what those testing libraries are doing for you.

The JavaScript and React landscape changes at such a pace that I can't claim that this book will remain current for very long. That is another reason why I use a first-principles approach. My hope is that when things *do* change, you'll still be able to use this book and apply what you've learned to those new scenarios.

Another theme in this book is **systematic refactoring**, which can come across as rather laborious but is a cornerstone of TDD and other good design practices. I have provided many examples of that within these pages, but for brevity, I sometimes jump straight to a final, refactored solution. For example, I sometimes choose to extract methods *before* they are written, whereas, in the real world, I would usually write methods inline and only extract when the containing method (or test) becomes too long.

Yet another theme is that of **cheating**, which you won't find mentioned in many TDD books. It's an acknowledgment that the TDD workflow is a scaffold around which you can build your own rules. Once you've learned and practiced the strict version of TDD for a while, you can learn what cheats you can use to cut corners. What tests won't provide much value in the long run? How can you speed up repetitive tests? So, a **cheat** is almost like saying you cut a corner in a way that wouldn't be obvious to an observer if they came to look at your code tomorrow. Maybe, for example, you implement three tests at once, rather than one at a time.

In this second edition of the book, I have doubled down on teaching TDD over React features. Beyond updating the code samples to work with React 18, there are few usages of new React features. Instead, the tests have been vastly improved; they are simpler, smaller, and utilize custom Jest matchers (which are themselves test-driven). Readers of the first edition will notice that I've changed my approach to component mocks; this edition relies on module mocks via the `jest.mock` function. The book no longer teaches shallow rendering. There are other smaller changes too, such as the avoidance of the `ReactTestUtils.Simulate` module. Chapter organization has been improved too, with some of the earlier chapters split up and streamlined. I hope you'll agree that this edition is leaps and bounds better than the first.

Who this book is for

If you're a React programmer, this book is for you. I aim to show you how TDD can improve your work.

If you're already knowledgeable about TDD, I hope there's still a lot you can learn from comparing your own process with mine.

If you don't already know React, you will benefit from spending some time running through the *Getting Started* guide on the React website. That being said, TDD is a wonderful platform for explaining new technologies, and it's entirely plausible that you'll be able to pick up React simply by following this book.

What this book covers

Chapter 1, First Steps with Test-Driven Development, introduces Jest and the TDD cycle.

Chapter 2, Rendering Lists and Detail Views, uses the TDD cycle to build a simple page displaying customer information.

Chapter 3, Refactoring the Test Suite, introduces some of the basic ways in which you can simplify tests.

Chapter 4, Test-Driving Data Input with React, covers using React component state to manage the display and saving of text input fields.

Chapter 5, Adding Complex Form Interactions, looks at a more complex form setup with dropdowns and radio buttons.

Chapter 6, Exploring Test Doubles, introduces various types of test doubles that are necessary for testing collaborating objects, and how to use them to test-drive form submission.

Chapter 7, Testing useEffect and Mocking Components, looks at using test doubles to fetch data when components are mounted, and how to use module mocks to block that behavior when testing parent components.

Chapter 8, Building an Application Component, ties everything together with a “root” component that threads together a user journey.

Chapter 9, Form Validation, continues with form building by adding client- and server-side validation and adding an indicator to show that data is being submitted.

Chapter 10, Filtering and Searching Data, shows how to build a search component with some complex interaction requirements, in addition to complex fetch request requirements.

Chapter 11, Test-Driving React Router, introduces the React Router library to simplify navigation within our user journeys.

Chapter 12, Test-Driving Redux, introduces Redux into our application.

Chapter 13, Test-Driving GraphQL, introduces the Relay library to communicate with a GraphQL endpoint that’s provided by our application backend.

Chapter 14, Building a Logo Interpreter, introduces a fun application that we will begin to explore by building out features across both React components and Redux middleware: undo/redo, persisting state across browser sessions with the LocalStorage API, and programmatically managing field focus.

Chapter 15, Adding Animation, covers adding animations to our application using the browser requestAnimationFrame API, all with a test-driven approach.

Chapter 16, Working with WebSockets, adds support for WebSocket communication with our application backend.

Chapter 17, Writing Your First Cucumber Test, introduces Cucumber and Puppeteer, which we will use to build BDD tests for existing functionality.

Chapter 18, Adding Features Guided by Cucumber Tests, integrates acceptance testing into our development process by first building BDD tests with Cucumber, before dropping down to unit tests.

Chapter 19, Understanding TDD in the Wider Testing Landscape, finishes the book by looking at how what you've learned fits in with other test and quality practices.

To get the most out of this book

There are two ways to read this book.

The first is to use it as a reference when you are faced with specific testing challenges. Use the index to find what you're after and move to that page.

The second, and the one I'd recommend starting with, is to follow the walk-throughs step by step, building your own code base as you go along. The companion GitHub repository has a directory for each chapter (such as Chapter01) and then, within that, three directories:

- **Start**, which is the starting point for the chapter, and you should start here if you're following along.
- **Exercises**, which is the point at the end of the chapter where the exercises begin. You should start here if you're attempting the exercises in each chapter. (Note that not every chapter has exercises.)
- **Complete**, which contains completed solutions to all the exercises.

You will need to be at least a little proficient with **Git**; a basic understanding of the branch, checkout, clone, commit, diff, and merge commands should be sufficient.

Take a look at the README . md file in the GitHub repository for more information and instructions on working with the code base.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <http://github.com/packtpublishing/Mastering-React-Test-Driven-Development-Second-Edition/>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: <https://packt.link/5dqQx>.

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “In the first test, change the word `appendChild` to `replaceChildren`.”

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “The presenter clicks the **Start sharing** button.”

Tips or important notes

Appear like this.

Code snippet conventions

A block of code is set as follows:

```
it("renders the customer first name", () => {
  const customer = { firstName: "Ashley" };
  render(<Appointment customer={customer} />);
  expect(document.body.textContent).toContain("Ashley");
});
```

There are two important things to know about the code snippets that appear in this book.

The first is that some code samples show modifications to existing sections of code. When this happens, the changed lines appear in **bold**, and the other lines are simply there to provide context:

```
export const Appointment = ({ customer }) => (
  <div>{customer.firstName}</div>
);
```

The second is that, often, some code samples will skip lines in order to keep the context clear. When this occurs, you'll see this marked by a line with three dots:

```
if (!anyErrors(validationResult)) {  
    ...  
} else {  
    setValidationErrors(validationResult);  
}
```

Sometimes, this happens for function parameters too:

```
if (!anyErrors(validationResult)) {  
    setSubmitting(true);  
    const result = await window.fetch(...);  
    setSubmitting(false);  
    ...  
}
```

Any command-line input or output is written as follows:

```
npx relay-compiler
```

JavaScript conventions

The book almost exclusively uses arrow functions for defining functions. The only exceptions are when we write generator functions, which must use the standard function's syntax. If you're not familiar with arrow functions, they look like this, which defines a single-argument function named `inc`:

```
const inc = arg => arg + 1;
```

They can appear on one line or be broken into two:

```
const inc = arg =>  
    arg + 1;
```

Functions that have more than one argument have the arguments wrapped in brackets:

```
const add = (a, b) => a + b;
```

If a function has multiple statements, then the body is wrapped in curly braces:

```
const dailyTimeSlots = (salonOpensAt, salonClosesAt) => {  
    ...
```

```
    return timeIncrements(totalSlots, startTime, increment);  
};
```

If the function returns an object, then that object must be wrapped in brackets so that the runtime doesn't think it's executing a block:

```
setAppointment(appointment => ({  
  ...appointment,  
  [name]: value  
});
```

This book makes liberal use of destructuring techniques to keep the code base as concise as possible. As an example, object destructuring generally happens for function parameters:

```
const handleSelectBoxChange = (  
  { target: { value, name } }  
) => {  
  ...  
};
```

This is equivalent to saying this:

```
const handleSelectBoxChange = (event) => {  
  const target = event.target;  
  const value = target.value;  
  const name = target.name;  
  ...  
};
```

Return values can also be destructured in the same way:

```
const [customer, setCustomer] = useState({});
```

This is equivalent to the following:

```
const customerState = useState({});  
const customer = customerState[0];  
const setCustomer = customerState[1];
```

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Mastering React Test-Driven Development*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1 – Exploring the TDD Workflow

Part 1 introduces all of the basic techniques you'll need to test-drive React applications. As you build more of the application, you will create a set of library functions that help to simplify and accelerate your testing. The goal is to give you both theoretical and practical advice that will help you apply the test-driven development workflow to your daily work.

This part includes the following chapters:

- *Chapter 1, First Steps with Test-Driven Development*
- *Chapter 2, Rendering Lists and Detail Views*
- *Chapter 3, Refactoring the Test Suite*
- *Chapter 4, Test-Driving Data Input*
- *Chapter 5, Adding Complex Form Interactions*
- *Chapter 6, Exploring Test Doubles*
- *Chapter 7, Testing useEffect and Mocking Components*
- *Chapter 8, Building an Application Component*

1

First Steps with Test-Driven Development

This book is a walk-through of building React applications using a test-driven approach. We'll touch on many different parts of the React experience, including building forms, composing interfaces, and animating elements. Perhaps more importantly, we'll do that all while learning a whole range of testing techniques.

You might have already used a React testing library such as React Testing Library or Enzyme, but this book doesn't use them. Instead, we'll be working from *first principles*: building up our own set of test functions based directly on our needs. That way, we can focus on the key ingredients that make up all great test suites. These ingredients—ideas such as super-small tests, test doubles, and factory methods—are decades old and apply across all modern programming languages and runtime environments. That's why this book doesn't use a testing library; there's really no need. What you'll learn will be useful to you no matter which testing libraries you use.

On the other hand, **Test-Driven Development (TDD)** is an effective technique for learning new frameworks and libraries. That makes this a very well-suited book for React and its ecosystem. This book will allow you to explore React in a way that you may not have experienced before as well as to make use of React Router and Redux and build out a GraphQL interface.

If you're new to the TDD process, you might find it a bit heavy-handed. It is a meticulous and disciplined style of developing software. You'll wonder why we're going to such Herculean efforts to build an application. For those that master it, there is tremendous value to be gained in specifying our software in this way, as follows:

- By being crystal clear about our product specifications, we gain the ability to adapt our code without fear of change.
- We gain automated regression testing by default.
- Our tests act as comments for our code, and those comments are verifiable when we run them.
- We gain a method of communicating our decision-making process with our colleagues.

You'll soon start recognizing the higher level of trust and confidence you have in the code you're working on. If you're anything like us, you'll get hooked on that feeling and find it hard to work without it.

Parts 1 and *2* of this book involve building an appointment system for a hair salon – nothing too revolutionary, but as sample applications go, it offers plenty of scope. We'll get started with that in this chapter. *Parts 3* and *4* use an entirely different application: a logo interpreter. Building that offers a fun way to explore more of the React landscape.

The following topics will be covered in this chapter:

- Creating a new React project from scratch
- Displaying data with your first test
- Refactoring your work
- Writing great tests

By the end of the chapter, you'll have a good idea of what the TDD process looks like when building out a simple React component. You'll see how to write a test, how to make it pass, and how to refactor your work.

Technical requirements

Later in this chapter, you'll be required to install **Node Package Manager (npm)** together with a whole host of packages. You'll want to ensure you have a machine capable of running the Node.js environment.

You'll also need access to a terminal.

In addition, you should choose a good editor or **Integrated Development Environment (IDE)** to work with your code.

The code files for this chapter can be found at the following link: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter01>.

Creating a new React project from scratch

In this section, we'll assemble all of the necessary pieces that you'll need to write a React application with TDD.

You may have come across the `create-react-app` package, which many people use to create an initial React project, but we won't be using that. The very first TDD principle you're going to learn is **You Ain't Gonna Need It (YAGNI)**. The `create-react-app` package adds a whole bunch of boilerplate that isn't relevant to what we're doing here—things such as a `favicon.ico` file, a sample logo, and CSS files. While these are undoubtedly useful, the basic idea behind YAGNI is that if it doesn't meet a needed specification, then it doesn't go in.

The thinking behind YAGNI is that anything unnecessary is simply *technical debt* – it's stuff that's just sitting there, unused, getting in your way.

Once you see how easy it is to start a React project from scratch, you won't ever use `create-react-app` again!

In the following subsections, we'll install NPM, Jest, React, and Babel.

Installing npm

Following the TDD process means running tests frequently—*very* frequently. Tests are run on the command line using the `npm test` command. So, let's start by getting npm installed.

You can find out if you already have it installed on your machine by opening a terminal window (or Command Prompt if you're on Windows) and typing the following command:

```
npm -v
```

If the command isn't found, head on over to the Node.js website at <https://nodejs.org> for installation instructions.

If you've already got npm installed, we recommend you ensure you're on the latest version. You can do this on the command line by typing the following command:

```
npm install npm@latest -g
```

Now you're all set. You can use the `npm` command to create your project.

Creating a new Jest project

Now that With npm installed, we can create our project by performing the following steps:

1. If you're following along with the book's Git repository, open a terminal and navigate to the repository directory that you've cloned. Otherwise, simply navigate to where you normally store your work projects.
2. Create a new directory using `mkdir appointments` and then set it as your current directory, using `cd appointments`.
3. Enter the `npm init` command. This initializes a new npm project by generating a template `package.json` file. You'll be prompted to enter some information about your project, but you can just accept all of the defaults *except* for the `test` command question, for which you should type in `jest`. This will enable you to run tests by using the `npm test` shortcut command.

Editing the package.json file by hand

Don't worry if you miss the prompt for the test command while you work through the instructions; you can set it afterwards by adding "test": "jest" to the scripts section of the generated package.json file.

4. Now go ahead and install Jest using `npm install --save-dev jest`. NPM will then download and install everything. Once completed, you should see a message like the following:

```
added 325 packages, and audited 326 packages in 24s
```

Alternatives to Jest

The TDD practices you'll study in this book will work for a wide variety of test runners, not just Jest. An example is the Mocha test runner. If you're interested in using Mocha with this book, take a look at the guidance at <https://reacttdd.com/migrating-from-jest-to-mocha>.

Commit early and often

Although we've just started, it's time to commit what you've done. The TDD process offers natural stopping points to commit – each time you see a new test pass, you can commit. This way, your repository will fill up with lots of tiny commits. You might not be used to that—you may be more of a “one commit per day” person. This is a great opportunity to try something new!

Committing *early and often* simplifies commit messages. If you have just one test in a commit, then you can use the test description as your commit message. No thinking is required. Plus, having a detailed commit history helps you backtrack if you change your mind about a particular implementation.

So, get used to typing `git commit` when you've got a passing test.

As you approach the end of a feature's development, you can use `git rebase` to squash your commits so that your Git history is kept tidy.

Assuming you're using Git to keep track of your work, go ahead and enter the following commands to commit what you've done so far:

```
git init
echo "node_modules" > .gitignore
git add .
git commit -m "Blank project with Jest dependency"
```

You've now “banked” that change and you can safely put it out of your mind and move on to the following two dependencies, which are React and Babel.

Bringing in React and Babel

Let's install React. That's two packages that can be installed with this next command:

```
npm install --save react react-dom
```

Next, we need Babel, which transpiles a few different things for us: React's **JavaScript Syntax Extension (JSX)** templating syntax, module mocks (which we'll meet in *Chapter 7, Testing useEffect and Mocking Components*), and various draft ECMAScript constructs that we'll use.

Important note

The following information is accurate for Babel 7. If you're using a later version, you may need to adjust the installation instructions accordingly.

Now, Jest already includes Babel—for the aforementioned module mocks—so we just need to install presets and plugins as follows:

```
npm install --save-dev @babel/preset-env @babel/preset-react
npm install --save-dev @babel/plugin-transform-runtime
npm install --save @babel/runtime
```

A Babel preset is a set of plugins. Each plugin enables a specific feature of the ECMAScript standards or a preprocessor such as JSX.

Configuring Babel

The `env` preset should usually be configured with target execution environments. It's not necessary for the purposes of this book. See the *Further reading* section at the end of this chapter for more information.

We need to enable the packages we've just installed. Create a new file, `.babelrc`, and add the following code:

```
{
  "presets": ["@babel/env", "@babel/react"],
  "plugins": ["@babel/transform-runtime"]
}
```

Both Babel and React are now ready for use.

Tip

You may wish to commit your source code to Git at this point.

In this section, you've installed NPM, primed your new Git repository, and you've installed the package dependencies you'll need to build your React app with TDD. You're all set to write some tests.

Displaying data with your first test

Now we'll use the **TDD cycle** for the first time, which you'll learn about as we go through each step of the cycle.

We'll start our application by building out an appointment view, which shows the details of an appointment. It's a React component called `Appointment` that will be passed in a data structure that represents an appointment at the hair salon. We can imagine it looks a little something like the following example:

```
{  
  customer: {  
    firstName: "Ashley",  
    lastName: "Jones",  
    phoneNumber: "(123) 555-0123"  
  },  
  stylist: "Jay Speares",  
  startsAt: "2019-02-02 09:30",  
  service: "Cut",  
  notes: ""  
}
```

We won't manage to get all of this information displayed by the time we complete the chapter; in fact, we'll only display the customer's `firstName`, and we'll make use of the `startsAt` timestamp to order a list of today's appointments.

In the following few subsections, you'll write your first Jest test and go through all of the necessary steps to make it pass.

Writing a failing test

What exactly *is* a test? To answer that, let's write one. Perform the following steps:

1. In your project directory, type the following commands:

```
mkdir test  
touch test/Appointment.test.js
```

- Open the `test/Appointment.test.js` file in your favorite editor or IDE and enter the following code:

```
describe("Appointment", () => {  
}) ;
```

The `describe` function defines a *test suite*, which is simply a set of tests with a given name. The first argument is the name of the unit you are testing. It could be a React component, a function, or a module. The second argument is a function inside of which you define your tests. The purpose of the `describe` function is to describe how this named “thing” works—whatever the thing is.

Global Jest functions

All of the Jest functions (such as `describe`) are already required and available in the global namespace when you run the `npm test` command. You don’t need to import anything.

For React components, it’s good practice to give `describe` blocks the same name as the component itself.

Where should you place your tests?

If you do try out the `create-react-app` template, you’ll notice that it contains a single unit test file, `App.test.js`, which exists in the same directory as the source file, `App.js`.

We prefer to keep our test files separate from our application source files. Test files go in a directory named `test` and source files go in a directory named `src`. There is no real objective advantage to either approach. However, do note that it’s likely that you won’t have a one-to-one mapping between production and test files. You may choose to organize your test files differently from the way you organize your source files.

Let’s go ahead and run this with Jest. You might think that running tests now is pointless, since we haven’t even written a test yet, but doing so gives us valuable information about what to do next. With TDD, it’s normal to run your test runner at every opportunity.

On the command line, run the `npm test` command again. You will see this output:

```
No tests found, exiting with code 1  
Run with `--passWithNoTests` to exit with code 0
```

That makes sense—we haven’t written any tests yet, just a `describe` block to hold them. At least we don’t have any syntax errors!

Tip

If you instead saw the following:

```
> echo "Error: no test specified" && exit 1
```

You need to set Jest as the value for the test command in your package.json file. See *Step 3* in *Creating a new Jest project* above.

Writing your first expectation

Change your describe call as follows:

```
describe("Appointment", () => {
  it("renders the customer first name", () => {
    });
});
```

The `it` function defines a single test. The first argument is the description of the test and always starts with a present-tense verb so that it reads in plain English. The `it` in the function name refers to the noun you used to name your test suite (in this case, `Appointment`). In fact, if you run tests now, with `npm test`, the output (as shown below) will make good sense:

```
PASS test/Appointment.test.js
  Appointment
    ✓ renders the customer first name (1ms)
```

You can read the `describe` and `it` descriptions together as one sentence: *Appointment renders the customer first name*. You should aim for all of your tests to be readable in this way.

As we add more tests, Jest will show us a little checklist of passing tests.

Jest's test function

You may have used the `test` function for Jest, which is equivalent to `it`. We prefer `it` because it reads better and serves as a helpful guide for how to succinctly describe our test.

You may have also seen people start their test descriptions with “*should...*”. I don’t really see the point in this, it’s just an additional word we have to type. Better to just use a well-chosen verb to follow the “`it`”.

Empty tests, such as the one we just wrote, always pass. Let’s change that now. Add an *expectation* to our test as follows:

```
it("renders the customer first name", () => {
```

```
expect(document.body.textContent).toContain("Ashley");
});
```

This `expect` call is an example of a fluent API. Like the test description, it reads like plain English. You can read it like this:

I expect document.body.textContent toContain the string Ashley.

Each expectation has an **expected value** that is compared against a **received value**. In this example, the expected value is `Ashley` and the received value is whatever is stored in `document.body.textContent`. In other words, the expectation passes if `document.body.textContent` has the word `Ashley` anywhere within it.

The `toContain` function is called a **matcher** and there are a whole lot of different matchers that work in different ways. You can (and should) write your own matchers. You'll discover how to do that in *Chapter 3, Refactoring the Test Suite*. Building matchers that are specific to your own project is an essential part of writing clear, concise tests.

Before we run this test, spend a minute thinking about the code. You might have guessed that the test will fail. The question is, how will it fail?

Run the `npm test` command and find out:

```
FAIL  test/Appointment.test.js
  Appointment
    × renders the customer first name (1 ms)

● Appointment > renders the customer first name

  The error below may be caused by using the
  wrong test environment, see https://jestjs.io/docs/
  configuration#testenvironment-string.

  Consider using the "jsdom" test environment.

  ReferenceError: document is not defined

  1 | describe("Appointment", () => {
  2 |   it("renders the customer first name", () => {
  > 3 |     expect(document.body.textContent).
  toContain("Ashley");
      |
      ^
  4 |   });
}
```

```
 5 |  })
 6 |  
  
  at Object.<anonymous> (test/Appointment.test.js:3:12)
```

We have our first failure!

It's probably not the failure you were expecting. Turns out, we still have some setup to take care of. Jest helpfully tells us what it thinks we need, and it's correct; we need to specify a test environment of `jsdom`.

A **test environment** is a piece of code that runs before and after your test suite to perform setup and teardown. For the `jsdom` test environment, it instantiates a new `JSDOM` object and sets global and document objects, turning Node.js into a browser-like environment.

`jsdom` is a package that contains a headless implementation of the **Document Object Model (DOM)** that runs on Node.js. In effect, it turns Node.js into a browser-like environment that responds to the usual DOM APIs, such as the `document` API we're trying to access in this test.

Jest provides a pre-packaged `jsdom` test environment that will ensure our tests run with these DOM APIs ready to go. We just need to install it and instruct Jest to use it.

Run the following command at your command prompt:

```
npm install --save-dev jest-environment-jsdom
```

Now we need to open `package.json` and add the following section at the bottom:

```
{  
  ...,  
  "jest": {  
    "testEnvironment": "jsdom"  
  }  
}
```

Then we run `npm test` again, giving the following output:

```
FAIL test/Appointment.test.js  
 Appointment  
   X renders the customer first name (10ms)  
  
 ● Appointment > renders the customer first name
```

```
expect(received).toContain(expected)

Expected substring: "Ashley"
Received string:    ""

1 | describe("Appointment", () => {
2 |   it("renders the customer first name", () => {
> 3 |     expect(document.body.textContent).
toContain("Ashley");
|     ^
|   });
4 | });
5 | });
6 | 

at Object.toContain (test/Appointment.test.js:3:39)
```

There are four parts to the test output that are relevant to us:

- The name of the failing test
- The expected answer
- The actual answer
- The location in the source where the error occurred

All of these help us to pinpoint why our tests failed: `document.body.textContent` is empty. That's not surprising given we haven't written any React code yet.

Rendering React components from within a test

In order to make this test pass, we'll have to write some code above the expectation that will call into our production code.

Let's work backward from that expectation. We know we want to build a React component to render this text (that's the `Appointment` component we specified earlier). If we imagine we already have that component defined, how would we get React to render it from within our test?

We simply do the same thing we'd do at the entry point of our own app. We render our root component like this:

```
ReactDOM.createRoot(container).render(component);
```

The preceding function replaces the DOM container element with a new element that is constructed by React by rendering our React component, which in our case will be called `Appointment`.

The `createRoot` function

The `createRoot` function is new in React 18. Chaining it with the call to `render` will suffice for most of our tests, but in *Chapter 7, Testing `useEffect` and Mocking Components*, you'll adjust this a little to support re-rendering in a single test.

In order to call this in our test, we'll need to define both `component` and `container`. The test will then have the following shape:

```
it("renders the customer first name", () => {
  const component = ???;
  const container = ???;
  ReactDOM.createRoot(container).render(component);
  expect(document.body.textContent).toContain("Ashley");
});
```

The value of `component` is easy; it will be an instance of `Appointment`, the component under test. We specified that as taking a `customer` as a prop, so let's write out what that might look like now. Here's a JSX fragment that takes `customer` as a prop:

```
const customer = { firstName: "Ashley" };
const component = <Appointment customer={customer} />;
```

If you've never done any TDD before, this might seem a little strange. Why are we writing test code for a component we haven't yet built? Well, that's partly the point of TDD – we let the test drive our design. At the beginning of this section, we formulated a verbal specification of what our `Appointment` component was going to do. Now, we have a concrete, written specification that can be automatically verified by running the test.

Simplifying test data

Back when we were considering our design, we came up with a whole object format for our appointments. You might think the definition of a `customer` here is very sparse, as it only contains a first name, but we don't need anything else for a test about customer names.

We've figured out `component`. Now, what about `container`? We can use the DOM to create a `container` element, like this:

```
const container = document.createElement("div");
```

The call to `document.createElement` gives us a new HTML element that we'll use as our rendering root. However, we also need to attach it to the current document body. That's because certain DOM events will only register if our elements are part of the document tree. So, we also need to use the following line of code:

```
document.body.appendChild(container);
```

Now our expectation should pick up whatever we render because it's rendered as part of `document.body`.

Warning

We won't be using `appendChild` for long; later in the chapter, we'll be switching it out for something more appropriate. We would not recommend using `appendChild` in your own test suites for reasons that will become clear!

Let's put it all together:

1. Change your test in `test/Appointments.test.js` as follows:

```
it("renders the customer first name", () => {
  const customer = { firstName: "Ashley" };
  const component = (
    <Appointment customer={customer} />
  );
  const container = document.createElement("div");
  document.body.appendChild(container);

  ReactDOM.createRoot(container).render(component);

  expect(document.body.textContent).toContain(
    "Ashley"
  );
}) ;
```

2. As we're using both the `ReactDOM` namespace and JSX, we'll need to include the two standard React imports at the top of our test file for this to work, as shown below:

```
import React from "react";
import ReactDOM from "react-dom/client";
```

3. Go ahead and run the test; it'll fail. Within the output, you'll see the following code:

```
ReferenceError: Appointment is not defined

      5 |   it("renders the customer first name", () => {
      6 |     const customer = { firstName: "Ashley" };
>  7 |     const component = (
      8 |       <Appointment customer={customer} />
      |
      9 |     );

```

This is subtly different from the test failure we saw earlier. This is a runtime exception, not an expectation failure. Thankfully, though, the exception is telling us exactly what we need to do, just as a test expectation would. It's finally time to build `Appointment`.

Make it pass

We're now ready to make the failing test pass. Perform the following steps:

1. Add a new `import` statement to `test/Appointment.test.js`, below the two React imports, as follows:

```
import { Appointment } from "../src/Appointment";
```

2. Run tests with `npm test`. You'll get a different error this time, with the key message being this:

```
Cannot find module '../src/Appointment' from
'Appointment.test.js'
```

Default exports

Although `Appointment` was defined as an export, it wasn't defined as a *default* export. That means we have to import it using the curly brace form of `import { ... }`. We tend to avoid using default exports as doing so keeps the name of our component and its usage in sync. If we change the name of a component, then every place where it's imported will break until we change those, too. This isn't the case with default exports. Once your names are out of sync, it's harder to track where components are used—you can't simply use text search to find them.

3. Let's create that module. Type the following code in your command prompt:

```
mkdir src
touch src/Appointment.js
```

- In your editor, add the following content to `src/Appointment.js`:

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

Why have we created a shell of `Appointment` without actually creating an implementation? This might seem pointless, but another core principle of TDD is *always do the simplest thing to pass the test*. We could rephrase this as *always do the simplest thing to fix the error you're working on*.

Remember when we mentioned that we listen carefully to what the test runner tells us? In this case, the test runner said `Cannot find module Appointment`, so what was needed was to create that module, which we've done, and then immediately stopped. Before we do anything else, we need to run our tests to learn what's the next thing to do.

Running `npm test` again, you should get this test failure:

```
● Appointment > renders the customer first name

  expect(received).toContain(expected)

    Expected substring: "Ashley"
    Received string:   ""

  12 |     ReactDOM.createRoot(...).render(component);
  13 |
> 14 |     expect(document.body.textContent).toContain(
  |           ^
  15 |       "Ashley"
  16 |     );
  17 |   });

  at Object.<anonymous> (test/Appointment.test.js:14:39)
```

To fix the test, let's change the `Appointment` definition as follows:

```
export const Appointment = () => "Ashley";
```

You might be thinking, “*That’s not a component! There’s no JSX.*” Correct. “*And it doesn’t even use the customer prop!*” Also correct. But React will render it anyway, and theoretically, it should make the test pass; so, in practice, it’s a good enough implementation, at least for now.

We always write the minimum amount of code that makes a test pass.

But does it pass? Run `npm test` again and take a look at the output:

```
● Appointment > renders the customer first name

    expect(received).toContain(expected)

      Expected substring: "Ashley"
      Received string:   ""

12 |     ReactDOM.createRoot(...).render(component);
13 |
> 14 |     expect(document.body.textContent).toContain(
15 |           ^
16 |         "Ashley"
17 |       );
|   };
```

No, it does not pass. This is a bit of a headscratcher. We *did* define a valid React component. And we did tell React to render it in our container. What's going on?

Making use of the `act` test helper

In a React testing situation like this, often the answer has something to do with the `async` nature of the runtime environment. Starting in React 18, the `render` function is **asynchronous**: the function call will return before React has modified the DOM. Therefore, the expectation will run *before* the DOM is modified.

React provides a helper function for our tests that pauses until asynchronous rendering has completed. It's called `act` and you simply need to wrap it around any React API calls. To use `act`, perform the following steps:

1. Go to the top of `test/Appointment.test.js` and add the following line of code:

```
import { act } from "react-dom/test-utils";
```

2. Then, change the line with the `render` call to read as follows:

```
act(() =>
  ReactDOM.createRoot(container).render(component)
);
```

- Now rerun your test and you should see a passing test, but with an odd warning printed above it, like this:

```
> jest

  console.error
    Warning: The current testing environment is not
    configured to support act(...)

      at printWarning (node_modules/react-dom/cjs/react-
        dom.development.js:86:30)
```

React would like us to be explicit in our use of `act`. That's because there are use cases where `act` does not make sense—but for unit testing, we almost certainly want to use it.

Understanding the `act` function

Although we're using it here, the `act` function is not required for testing React. For a detailed discussion on this function and how it can be used, head to <https://reacttdd.com/understanding-act>.

- Let's go ahead and enable the `act` function. Open `package.json` and modify your `jest` property to read as follows:

```
{
  ...
  "jest": {
    "testEnvironment": "jsdom",
    "globals": {
      "IS_REACT_ACT_ENVIRONMENT": true
    }
  }
}
```

- Now run your test again with `npm test`, giving the output shown:

```
> jest

PASS  test/Appointment.test.js
  Appointment
```

```
✓ renders the customer first name (13 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.355 s
Ran all test suites.
```

Finally, you have a passing test, with no warnings!

In the following section, you will discover how to remove the hardcoded string value that you've introduced by adding a second test.

Triangulating to remove hardcoding

Now that we've got past that little hurdle, let's think again about the problems with our test. We did a bunch of strange acrobatics just to get this test passing. One odd thing was the use of a hardcoded value of `Ashley` in the React component, even though we'd gone to the trouble of defining a customer prop in our test and passing it in.

We did that because we want to stick to our rule of only doing the simplest thing that will make a test pass. In order to get to the real implementation, we need to add more tests.

This process is called **triangulation**. We add more tests to build more of a *real* implementation. The more specific our tests get, the more general our production code needs to get.

Ping pong programming

This is one reason why pair programming using TDD can be so enjoyable. Pairs can play *ping pong*. Sometimes, your pair will write a test that you can solve trivially, perhaps by hardcoding, and then you force them to do the hard work of both tests by triangulating. They need to remove the hardcoding and add the generalization.

Let's triangulate by performing the following steps:

1. Make a copy of your first test, pasting it just under the first test, and change the test description and the name of `Ashley` to `Jordan`, as follows:

```
it("renders another customer first name", () => {
  const customer = { firstName: "Jordan" };
  const component = (
    <Appointment customer={customer} />
  );
}
```

```
const container = document.createElement("div");
document.body.appendChild(container);

act(() =>
  ReactDOM.createRoot(container).render(component)
);

expect(document.body.textContent).toContain(
  "Jordan"
);
}) ;
```

2. Run tests with `npm test`. We expect this test to fail, and it does. But examine the code carefully. Is this what you expected to see? Take a look at the value of `Received` string in the following code:

```
FAIL test/Appointment.test.js
Appointment
  ✓ renders the customer first name (18ms)
  ✗ renders another customer first name (8ms)

● Appointment > renders another customer first name

  expect(received).toContain(expected)

    Expected substring: "Jordan"
    Received string:    "AshleyAshley"
```

The document body has the text `AshleyAshley`. This kind of repeated text is an indicator that our tests are not *independent* of one another. The component has been rendered twice, once for each test. That's correct, but the document isn't being cleared between each test run.

This is a problem. When it comes to unit testing, we want all tests to be independent of one other. If they aren't, the output of one test could affect the functionality of a subsequent test. A test might pass because of the actions of a previous test, resulting in a false positive. And even if the test did fail, having an unknown initial state means you'll spend time figuring out if it was the initial state of the test that caused the issue, rather than the test scenario itself.

We need to change course and fix this before we get ourselves into trouble.

Test independence

Unit tests should be independent of one another. The simplest way to achieve this is to not have any shared state between tests. Each test should only use variables that it has created itself.

Backtracking on ourselves

We know that the **shared state** is the problem. Shared state is a fancy way of saying “shared variables.” In this case, it’s `document`. This is the single global document object that is given to us by the `jsdom` environment, which is consistent with how a normal web browser operates: there’s a single `document` object. But unfortunately, our two tests use `appendChild` to add into that single `document` that’s shared between them. They don’t each get their own separate instance.

A simple solution is to replace `appendChild` with `replaceChildren`, like this:

```
document.body.replaceChildren(container);
```

This will clear out everything from `document.body` before doing the append.

But there’s a problem. We’re in the middle of a *red* test. We should never refactor, rework, or otherwise change course while we’re red.

Admittedly, this is all highly contrived—we could have used `replaceChildren` right from the start. But not only are we proving the need for `replaceChildren`, we are also about to discover an important technique for dealing with just this kind of scenario.

What we’ll have to do is *skip* this test we’re working on, fix the previous test, then re-enable the skipped test. Let’s do that now by performing the following steps:

1. In the first test you’ve just written, change `it` to `it.skip`. Do that now for the second test as follows:

```
it.skip("renders another customer first name", () => {
  ...
});
```

2. Run tests. You’ll see that Jest ignores the second test and the first one still passes, as follows:

```
PASS test/Appointment.test.js
Appointment
  ✓ renders the customer first name (19ms)
  ○ skipped 1 test
```

```
Test Suites: 1 passed, 1 total
Tests: 1 skipped, 1 passed, 2 total
```

3. In the first test, change `appendChild` to `replaceChildren` as follows:

```
it("renders the customer first name", () => {
  const customer = { firstName: "Ashley" };
  const component = (
    <Appointment customer={customer} />
  );
  const container = document.createElement("div");
  document.body.replaceChildren(container);

  ReactDOM.createRoot(container).render(component);

  expect(document.body.textContent).toContain(
    "Ashley"
  );
}) ;
```

4. Rerun the tests with `npm test`. It should still be passing.

It's time to bring the skipped test back in by removing `.skip` from the function name.

5. Perform the same update in this test as in the first: change `appendChild` to `replaceChildren`, like this:

```
it("renders another customer first name", () => {
  const customer = { firstName: "Jordan" };
  const component = (
    <Appointment customer={customer} />
  );
  const container = document.createElement("div");
  document.body.replaceChildren(container);

  act(() =>
    ReactDOM.createRoot(container).render(component)
) ;
```

```
    expect(document.body.textContent).toContain(
      "Jordan"
    );
}) ;
```

6. Running tests now should give us the error that we were originally expecting. No more repeated text content, as you can see:

```
FAIL test/Appointment.test.js
Appointment
  ✓ renders the customer first name (18ms)
  ✗ renders another customer first name (8ms)

● Appointment > renders another customer first name

  expect(received).toContain(expected)

  Expected substring: "Jordan"
  Received string:    "Ashley"
```

7. To make the test pass, we need to introduce the prop and use it within our component. Change the definition of Appointment to look as follows, destructuring the function arguments to pull out the customer prop:

```
export const Appointment = ({ customer }) => (
  <div>{customer.firstName}</div>
);
```

8. Run tests. We expect this test to now pass:

```
PASS test/Appointment.test.js
Appointment
  ✓ renders the customer first name (21ms)
  ✓ renders another customer first name (2ms)
```

Great work! We're done with our passing test, and we've successfully triangulated to remove hardcoding.

In this section, you've written two tests and, in the process of doing so, you've discovered and overcome some of the challenges we face when writing automated tests for React components.

Now that we've got our tests working, we can take a closer look at the code we've written.

Refactoring your work

Now that you've got a green test, it's time to refactor your work. Refactoring is the process of adjusting your code's structure without changing its functionality. It's crucial for keeping a code base in a fit, maintainable state.

Sadly, the refactoring step is the step that always gets forgotten. The impulse is to rush straight into the next feature. We can't stress how important it is to take time to simply stop and *stare* at your code and think about ways to improve it. Practicing your refactoring skills is a sure-fire way to level up as a developer.

The adage "more haste; less speed" applies to coding just as it does in life. If you make a habit of skipping the refactoring phase, your code quality will likely deteriorate over time, making it harder to work with and therefore slower to build new features.

The TDD cycle helps you build good personal discipline and habits, such as consistently refactoring. It might take more effort upfront, but you will reap the rewards of a code base that remains maintainable as it ages.

Don't Repeat Yourself

Test code needs as much care and attention as production code. The number one principle you'll be relying on when refactoring your tests is **Don't Repeat Yourself (DRY)**. *Drying up tests* is a phrase all TDDers repeat often.

The key point is that you want your tests to be as concise as possible. When you see repeated code that exists in multiple tests, it's a great indication that you can pull that repeated code out. There are a few different ways to do that, and we'll cover just a couple in this chapter.

You will see further techniques for drying up tests in *Chapter 3, Refactoring the Test Suite*.

Sharing setup code between tests

When tests contain identical setup instructions, we can promote those instructions into a shared `beforeEach` block. The code in this block is executed before each test.

Both of our tests use the same two variables: `container` and `customer`. The first one of these, `container`, is initialized identically in each test. That makes it a good candidate for a `beforeEach` block.

Perform the following steps to introduce your first `beforeEach` block:

1. Since `container` needs to be accessed in the `beforeEach` block and each of the tests, we must declare it in the outer `describe` scope. And since we'll be setting its value in the `beforeEach` block, that also means we'll need to use `let` instead of `const`. Just above the first test, add the following line of code:

```
let container;
```

2. Below that declaration, add the following code:

```
beforeEach(() => {
  container = document.createElement("div");
  document.body.replaceChildren(container);
});
```

3. Delete the corresponding two lines from each of your two tests. Note that since we defined `container` in the scope of the `describe` block, the value set in the `beforeEach` block will be available to your test when it executes.

Use of let instead of const

Be careful when you use `let` definitions within the `describe` scope. These variables are not cleared by default between each test execution, and that shared state will affect the outcome of each test. A good rule of thumb is that any variable you declare in the `describe` scope should be assigned to a new value in a corresponding `beforeEach` block, or in the first part of each test, just as we've done here.

For a more detailed look at the use of `let` in test suites, head to <https://reacttdd.com/use-of-let>.

In *Chapter 3, Refactoring the Test Suite*, we'll look at a method for sharing this setup code between multiple test suites.

Extracting methods

The call to `render` is the same in both tests. It's also quite lengthy given that it's wrapped in a call to `act`. It makes sense to extract this entire operation and give it a more meaningful name.

Rather than pull it out as is, we can create a new function that takes the `Appointment` component as its parameter. The explanation for why this is useful will come after, but now let's perform the following steps:

1. Above the first test, write the following definition. Note that it still needs to be within the `describe` block because it uses the `container` variable:

```
const render = component =>
  act(() =>
    ReactDOM.createRoot(container).render(component)
);
```

- Now, replace the call to `render` in each test with the following line of code:

```
render(<Appointment customer={customer} />);
```

- In the preceding step, we *inlined* the JSX, passing it directly into `render`. That means you can now delete the line starting with `const component`. For example, your first test should end up looking as follows:

```
it("renders the customer first name", () => {
  const customer = { firstName: "Ashley" };
  render(<Appointment customer={customer} />);
  expect(document.body.textContent).toContain(
    "Ashley"
  );
}) ;
```

- Rerun your tests and verify that they are still passing.

Highlighting differences within your tests

The parts of a test that you want to highlight are the parts that differ between tests. Usually, some code remains the same (such as `container` and the steps needed to render a component) and some code differs (`customer` in this example). Do your best to hide away whatever is the same and highlight what differs. That way, it makes it obvious what a test is specifically testing.

This section has covered a couple of simple ways of refactoring your code. As the book progresses, we'll look at many different ways that both production source code and test code can be refactored.

Writing great tests

Now that you've written a couple of tests, let's step away from the keyboard and discuss what you've seen so far.

Your first test looks like the following example:

```
it("renders the customer first name", () => {
  const customer = { firstName: "Ashley" };
  render(<Appointment customer={customer} />);
  expect(document.body.textContent).toContain("Ashley");
}) ;
```

This is concise and clearly readable.

A *good* test has the following three distinct sections:

- **Arrange:** Sets up test dependencies
- **Act:** Executes production code under test
- **Assert:** Checks that expectations are met

This is so well understood that it is called the **Arrange, Act, Assert (AAA)** pattern, and all of the tests in this book follow this pattern.

A *great* test is not just good but is also the following:

- Short
- Descriptive
- Independent of other tests
- Has no side effects

In the remainder of this section, we'll discuss the TDD cycle, which you've already used, and also how to set up your development environment for easy TDD.

Red, green, refactor

TDD, at its heart, is the red, green, refactor cycle that we've just seen.

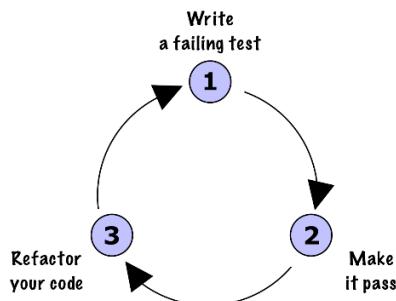


Figure 1.1 – The TDD cycle

The steps of the TDD cycle are:

1. **Write a failing test:** Write a short test that describes some functionality you want. Execute your test and watch it fail. If it doesn't fail, then it's an unnecessary test; delete it and write another.
2. **Make it pass:** Make the test green by writing the simplest production code that will work. Don't worry about finding a neat code structure; you can tidy it up later.

3. **Refactor your code:** Stop, slow down, and resist the urge to move on to the next feature. Work hard to make your code—both production and test code—as clean as it can be.

That's all there is to it. You've already seen this cycle in action in the preceding two sections, and we'll continue to use it throughout the rest of the book.

Streamlining your testing process

Think about the effort you've put into this book so far. What actions have you been doing the most? They are the following:

- Switching between `src/Appointment.js` and `test/Appointment.test.js`
- Running `npm test` and analyzing the output

Make sure you can perform these actions quickly.

For a start, you should use split-screen functionality in your editor. If you aren't already, take this opportunity to learn how to do it. Load your production module on one side and the corresponding unit test file on the other.

Here's a picture of our setup; we use `nvim` and `tmux`:

```

 0 1 2 ~%2
tmux
 1 import React from "react";
 2 import ReactDOM from "react-dom";
 3 import { act } from "react-dom/test-utils";
 4
 5 import { Appointment } from "../src/Appointment";
 6
 7 describe("Appointment", () => {
 8   let container;
 9   let customer;
10
11   beforeEach(() => {
12     container = document.createElement("div");
13   });
14
15   const render = (component) =>
16     act(() => ReactDOM.createRoot(container).render(component));
17
18   it("renders the customer first name", () => {
19     customer = { firstName: "Ashley" };
20     render();
21     expect(container.textContent).toMatch("Ashley");
22   });
23
test/Appointment.test.js      1/29  N...  src/Appointment.js  1/5

```

```

> jest
PASS  test/Appointment.test.js
Appointment
  ✓ renders the customer first name (12 ms)
  ✓ renders another customer first name (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        0.88 s
Ran all test suites.
work/react-tdd-2 %

```

Figure 1.2 – A typical TDD setup running tmux and vim in a terminal

You can see that we also have a little test window at the bottom for showing test output.

Jest can also watch your files and auto-run tests when they change. To enable this, change the `test` command in `package.json` to `jest --watchAll`. This reruns all of your tests when it detects any changes.

Watching files for changes

Jest's watch mode has an option to run only the tests in files that have changed, but since your React app will be composed of many different files, each of which are interconnected, it's better to run everything as breakages can happen in many modules.

Summary

Tests act like a safety harness in our learning; we can build little blocks of understanding, building on top of each other, up and up to ever-greater heights, without fear of falling.

In this chapter, you've learned a *lot* about the TDD experience.

To begin with, you set up a React project from scratch, pulling in only the dependencies you need to get things running. You've written two tests using Jest's `describe`, `it`, and `beforeEach` functions. You discovered the `act` helper, which ensures all React rendering has been completed before your test expectations execute.

You've also seen plenty of testing ideas. Most importantly, you've practiced TDD's red-green-refactor cycle. You've also used triangulation and you learned about the **Arrange, Act, Assert** pattern.

And we threw in a couple of design principles for good measure: DRY and YAGNI.

While this is a great start, the journey has only just begun. In the following chapter, we'll test drive a more complex component.

Further reading

Take a look at the Babel web page to discover how to correctly configure the Babel `env` preset. This is important for real-world applications, but we skipped over it in this chapter. You can find it at the following link:

<https://babeljs.io/docs/en/babel-preset-env>.

React's `act` function was introduced in React 17 and has seen updates in React 18. It is deceptively complex. See this blog post for some more discussion on how this function is used at the following link: <https://reacttdd.com/understanding-act>.

This book doesn't make much use of Jest's `watch` functionality. In recent versions of Jest, this has seen some interesting updates, such as the ability to choose which files to watch. If you find rerunning tests a struggle, you might want to try it out. You can find more information at the following link: <https://jestjs.io/docs/en/cli#watch>.

2

Rendering Lists and Detail Views

The previous chapter introduced the core TDD cycle: red, green, refactor. You had the chance to try it out with two simple tests. Now, it's time to apply that to a bigger React component.

At the moment, your application displays just a single item of data: the customer's name. In this chapter, you'll extend it so that you have a view of all appointments for the current day. You'll be able to choose a time slot and see the details for the appointment at that time. We will start this chapter by sketching a mock-up to help us plan how we'll build out the component. Then, we'll begin implementing a list view and showing appointment details.

Once we've got the component in good shape, we'll build the entry point with webpack and then run the application in order to do some manual testing.

The following topics will be covered in this chapter:

- Sketching a mock-up
- Creating the new component
- Specifying list item content
- Selecting data to view
- Manually testing our changes

By the end of this chapter, you'll have written a decent-sized React component using the TDD process you've already learned. You'll also have seen the app running for the first time.

Technical requirements

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter02>.

Sketching a mock-up

Let's start with a little more up-front design. We've got an `Appointment` component that takes an appointment and displays it. We will build an `AppointmentsDayView` component around it that takes an array of appointment objects and displays them as a list. It will also display a single `Appointment`: the appointment that is currently selected. To select an appointment, the user simply clicks on the time of day that they're interested in.

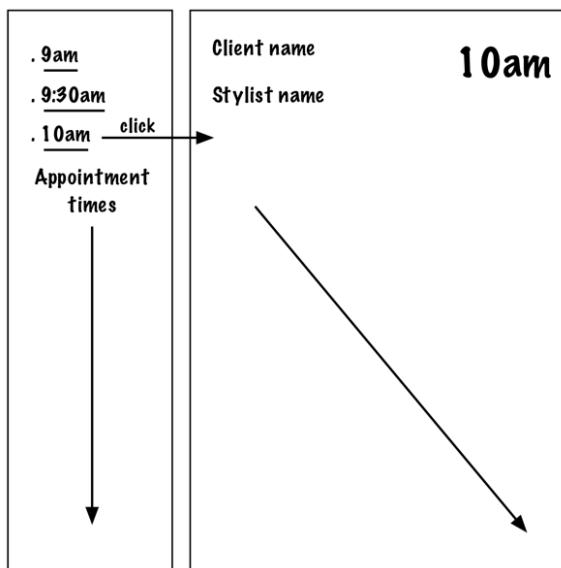


Figure 2.1 – A mock-up of our appointment system UI

Up-front design

When you're using TDD to build new features, it's important to do a little up-front design so that you have a general idea of the direction your implementation needs to take.

That's all the design we need for now; let's jump right in and build the new `AppointmentsDayView` component.

Creating the new component

In this section, we'll create the basic form of `AppointmentsDayView`: a list of appointment times for the day. We won't build any interactive behavior for it just yet.

We'll add our new component into the same file we've been using already because so far there's not much code in there. Perform the following steps:

Placing components

We don't always need a new file for each component, particularly when the components are short functional components, such as our `Appointment` component (a one-line function). It can help to group related components or small sub-trees of components in one place.

1. In `test/Appointment.test.js`, create a new `describe` block under the first one, with a single test. This test checks that we render a `div` with a particular ID. That's important in this case because we load a CSS file that looks for this element. The expectations in this test use the DOM method, `querySelector`. This searches the DOM tree for a single element with the tag provided:

```
describe("AppointmentsDayView", () => {
  let container;

  beforeEach(() => {
    container = document.createElement("div");
    document.body.replaceChildren(container);
  });

  const render = (component) =>
    act(() =>
      ReactDOM.createRoot(container).render(component)
    );

  it("renders a div with the right id", () => {
    render(<AppointmentsDayView appointments={[]}>);
    expect(
      document.querySelector(
        "div#appointmentsDayView"
      )
    ).not.toBeNull();
  });
}
```

```
    }) ;
}) ;
```

Note

It isn't usually necessary to wrap your component in a `div` with an ID or a class. We tend to do it when we have CSS that we want to attach to the entire group of HTML elements that will be rendered by the component, which, as you'll see later, is the case for `AppointmentsDayView`.

This test uses the exact same `render` function from the first `describe` block as well as the same `let container` declaration and `beforeEach` block. In other words, we've introduced duplicated code. By duplicating code from our first test suite, we're making a mess straight after cleaning up our code! Well, we're allowed to do it when we're in the first stage of the TDD cycle. Once we've got the test passing, we can think about the right structure for the code.

2. Run `npm test` and look at the output:

```
FAIL test/Appointment.test.js
  Appointment
    ✓ renders the customer first name (18ms)
    ✓ renders another customer first name (2ms)
  AppointmentsDayView
    ✗ renders a div with the right id (7ms)

● AppointmentsDayView > renders a div with the right id

ReferenceError: AppointmentsDayView is not defined
```

Let's work on getting this test to pass by performing the following steps:

1. To fix this, change the last `import` statement in your test file to read as follows:

```
import {
  Appointment,
  AppointmentsDayView,
} from "../src/Appointment";
```

2. In `src/Appointment.js`, add this functional component below `Appointment` as shown:

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

- Run your tests again. You'll see output like this:

```
● AppointmentsDayView > renders a div with the right id
```

```
expect(received).not.toBeNull()
```

- Finally, a test failure! Let's get that div in place as follows:

```
export const AppointmentsDayView = () => (
  <div id="appointmentsDayView"></div>
);
```

- Your test should now be passing. Let's move on to the next test. Add the following text, just below the last test in `test/Appointment.test.js`, still inside the `AppointmentsDayView` describe block:

```
it("renders an ol element to display appointments", () =>
{
  render(<AppointmentsDayView appointments={ [] } />);
  const listElement = document.querySelector("ol");
  expect(listElement).not.toBeNull();
});
```

- Run your tests again and you'll see output matching the text shown below:

```
● AppointmentsDayView > renders an ol element to display
appointments
```

```
expect(received).not.toBeNull()
```

```
Received: null
```

- To make that pass, add the `ol` element as follows:

```
export const AppointmentsDayView = () => (
  <div id="appointmentsDayView">
    <ol />
  </div>
);
```

8. Alright, now let's fill that `ol` with an item for each appointment. For that, we'll need (at least) two appointments to be passed as the value of the `appointments` prop, as. Add the next test, as shown:

```
it("renders an li for each appointment", () => {
  const today = new Date();
  const twoAppointments = [
    { startsAt: today.setHours(12, 0) },
    { startsAt: today.setHours(13, 0) },
  ];

  render(
    <AppointmentsDayView
      appointments={twoAppointments}
    />
  );

  const listChildren =
    document.querySelectorAll("ol > li");
  expect(listChildren).toHaveLength(2);
});
```

Testing dates and times

In the test, the `today` constant is defined to be `new Date()`. Each of the two records then uses this as a base date. Whenever we're dealing with dates, it's important that we base all events on the same moment in time, rather than asking the system for the current time more than once. Doing that is a subtle bug waiting to happen.

9. Run `npm test` again and you'll see this output:

```
● AppointmentsDayView > renders an li for each appointment

expect(received).toHaveLength(expected)

Expected length: 2
Received length: 0
Received object: []
```

10. To fix this, we map over the provided appointments prop and render an empty li element:

```
export const AppointmentsDayView = (
  { appointments }
) => (
  <div id="appointmentsDayView">
    <ol>
      {appointments.map(() => (
        <li />
      ))}
    </ol>
  </div>
);
```

Ignoring unused function arguments

The map function will provide an appointment argument to the function passed to it. Since we don't use the argument (yet), we don't need to mention it in the function signature—we can just pretend that our function has no arguments instead, hence the empty brackets. Don't worry, we'll need the argument for a subsequent test, and we'll add it in then.

11. Great, let's see what Jest thinks. Run `npm test` again:

```
console.error
Warning: Each child in a list should have a unique
"key" prop.

Check the render method of AppointmentsDayView.
...
PASS test/Appointment.test.js
Appointment
  ✓ renders the customer first name (19ms)
  ✓ renders another customer first name (2ms)
AppointmentsDayView
  ✓ renders a div with the right id (7ms)
  ✓ renders an ol element to display appointments
(16ms)
  ✓ renders an li for each appointment (16ms)
```

12. Our test passed, but we got a warning from React. It's telling us to set a key value on each child element. We can use `startsAt` as a key, like this:

```
<ol>
  {appointments.map(appointment => (
    <li key={appointment.startsAt} />
  )));
</ol>
```

Testing keys

There's no easy way for us to test key values in React. To do it, we'd need to rely on internal React properties, which would introduce a risk of tests breaking if the React team were to ever change those properties.

The best we can do is set a key to get rid of this warning message. In an ideal world, we'd have a test that uses the `startsAt` timestamp for each `li` key. Let's just imagine that we have that test in place.

This section has covered how to render the basic structure of a list and its list items. Next, it's time to fill in those items.

Specifying list item content

In this section, you'll add a test that uses an array of example appointments to specify that the list items should show the time of each appointment, and then you'll use that test to support the implementation.

Let's start with the test:

1. Create a fourth test in the new `describe` block as shown:

```
it("renders the time of each appointment", () => {
  const today = new Date();
  const twoAppointments = [
    { startsAt: today.setHours(12, 0) },
    { startsAt: today.setHours(13, 0) },
  ];

  render(
    <AppointmentsDayView
      appointments={twoAppointments}
```

```
/>
);

const listChildren =
  document.querySelectorAll("li");
expect(listChildren[0].textContent).toEqual(
  "12:00"
);
expect(listChildren[1].textContent).toEqual(
  "13:00"
);
}) ;
```

Jest will show the following error:

```
● AppointmentsDayView > renders the time of each appointment

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

Expected: "12:00"
Received: ""
```

The `toEqual` matcher

This matcher is a stricter version of `toContain`. The expectation only passes if the text content is an exact match. In this case, we think it makes sense to use `toEqual`. However, it's often best to be as loose as possible with your expectations. Tight expectations have a habit of breaking any time you make the slightest change to your code base.

2. Add the following function to `src/Appointment.js`, which converts a Unix timestamp (which we get from the return value from `setHours`) into a time of day. It doesn't matter where in the file you put it; we usually like to define constants before we use them, so this would go at the top of the file:

```
const appointmentTimeOfDay = (startsAt) => {
  const [h, m] = new Date(startsAt)
    .toTimeString()
    .split(":");
```

```

        return `${h}:${m}`;
    }
}

```

Understanding syntax

This function uses *destructuring assignment* and *template literals*, which are language features that you can use to keep your functions concise.

Having good unit tests can help teach advanced language syntax. If we're ever unsure about what a function does, we can look up the tests that will help us figure it out.

3. Use the preceding function to update `AppointmentsDayView` as follows:

```

<ol>
  {appointments.map(appointment => (
    <li key={appointment.startsAt}>
      {appointmentTimeOfDay(appointment.startsAt)}
    </li>
  )));
</ol>

```

4. Running tests should show everything as green:

```

PASS test/Appointment.test.js
Appointment
  ✓ renders the customer first name (19ms)
  ✓ renders another customer first name (2ms)
AppointmentsDayView
  ✓ renders a div with the right id (7ms)
  ✓ renders an ol element to display appointments
(16ms)
  ✓ renders an li for each appointment (6ms)
  ✓ renders the time of each appointment (3ms)

```

This is a great chance to refactor. The last two `AppointmentsDayView` tests use the same `twoAppointments` prop value. This definition, and the `today` constant, can be lifted out into the `describe` scope, the same way we did with `customer` in the `Appointment` tests. This time, however, it can remain as `const` declarations as they never change.

5. To do that, move the `today` and `twoAppointments` definitions from one of the tests to the top of the `describe` block, above `beforeEach`. Then, delete the definitions from both tests.

That's it for this test. Next, it's time to focus on adding click behavior.

Selecting data to view

Let's add in some dynamic behavior to our page. We'll make each of the list items a link that the user can click on to view that appointment.

Thinking through our design a little, there are a few pieces we'll need:

- A button element within our li
- An onClick handler that is attached to that button element
- Component state to record which appointment is currently being viewed

When we test React actions, we do it by observing the consequences of those actions. In this case, we can click on a button and then check that its corresponding appointment is now rendered on the screen.

We'll break this section into two parts: first, we'll specify how the component should initially appear, and second, we'll handle a click event for changing the content.

Initial selection of data

Let's start by asserting that each li element has a button element:

1. We want to display a message to the user if there are no appointments scheduled for today. In the AppointmentsDayView describe block, add the following test:

```
it("initially shows a message saying there are no
appointments today", () => {
  render(<AppointmentsDayView appointments={[]}>);
  expect(document.body.textContent).toContain(
    "There are no appointments scheduled for today."
  );
});
```

2. Make the test pass by adding in a message at the bottom of the rendered output. We don't need a check for an empty appointments array just yet; we'll need another test to triangulate to that. The message is as follows:

```
return (
  <div id="appointmentsDayView">
  ...
  <p>There are no appointments scheduled for today.</p>
```

```
    </div>
  ) ;
```

3. When the component first loads, we should show the first appointment of the day. A straightforward way to check that happens is to look for the customer's first name is shown on the page. Add the next test which does just that, shown below:

```
it("selects the first appointment by default", () => {
  render(
    <AppointmentsDayView
      appointments={twoAppointments}
    />
  ) ;
  expect(document.body.textContent).toContain(
    "Ashley"
  );
}) ;
```

4. Since we're looking for the customer's name, we'll need to make sure that's available in the `twoAppointments` array. Update it now to include the customer's first name as follows:

```
const twoAppointments = [
  {
    startsAt: today.setHours(12, 0),
    customer: { firstName: "Ashley" },
  },
  {
    startsAt: today.setHours(13, 0),
    customer: { firstName: "Jordan" },
  },
];
```

5. Make the test pass by modifying the `Appointment` component. Change the last line of the `div` component to read as follows:

```
<div id="appointmentsDayView">
  ...
  {appointments.length === 0 ? (
    <p>There are no appointments scheduled for today.</p>
  ) : (
```

```
<Appointment {...appointments[0]} />
)
</div>
```

Now we're ready to let the user make a selection.

Adding events to a functional component

We're about to add *state* to our component. The component will show a button for each appointment. When the button is clicked, the component stores the array index of the appointment that it refers to. To do that, we'll use the `useState` hook.

What are hooks?

Hooks are a feature of React that manages various non-rendering related operations. The `useState` hook stores data across multiple renders of your function. The call to `useState` returns both the current value in storage and a setter function that allows it to be set.

If you're new to hooks, check out the *Further reading* section at the end of this chapter. Alternatively, you could just follow along and see how much you can pick up just by reading the tests!

We'll start by asserting that each `li` element has a `button` element:

1. Add the following test below the last one you added. The second expectation is peculiar in that it is checking the `type` attribute of the `button` element to be `button`. If you haven't seen this before, it's idiomatic when using `button` elements to define its role by setting the `type` attribute as shown in this test:

```
it("has a button element in each li", () => {
  render(
    <AppointmentsDayView
      appointments={twoAppointments}
    />
  );

  const buttons =
    document.querySelectorAll("li > button");
  expect(buttons).toHaveLength(2);
  expect(buttons[0].type).toEqual("button");
});
```

Testing element positioning

We don't need to be pedantic about checking the content or placement of the `button` element within its parent. For example, this test would pass if we put an empty `button` child at the end of `li`. But, thankfully, doing the right thing is just as simple as doing the wrong thing, so we can opt to do the right thing instead. All we need to do to make this test pass is wrap the existing content in the new tag.

2. Make the test pass by wrapping the appointment time with a `button` element in the `AppointmentsDayView` component, as follows:

```
...
<li key={appointment.startsAt}>
  <button type="button">
    {appointmentTimeOfDay(appointment.startsAt) }
  </button>
</li>
...
```

3. We can now test what happens when the button is clicked. Back in `test/Appointment.test.js`, add the following as the next test. This uses the `click` function on the DOM element to raise a DOM click event:

```
it("renders another appointment when selected", () => {
  render(
    <AppointmentsDayView
      appointments={twoAppointments}
    />
  );
  const button =
    document.querySelectorAll("button") [1];
  act(() => button.click());
  expect(document.body.textContent).toContain(
    "Jordan"
  );
}) ;
```

Synthetic events and Simulate

An alternative to using the `click` function is to use the `Simulate` namespace from React's test utilities to raise a `synthetic` event. While the interface for using `Simulate` is somewhat simpler than the DOM API for raising events, it's also unnecessary for testing. There's no need to use extra APIs when the DOM API will suffice. Perhaps more importantly, we also want our tests to reflect the real browser environment as much as possible.

4. Go ahead and run the test. The output will look like this:

```
● AppointmentsDayView > renders appointment when selected

    expect(received).toContain(expected)

      Expected substring: "Jordan"
      Received string:    "12:0013:00Ashley"
```

Notice the full text in the received string. We're getting the text content of the list too because we've used `document.body.textContent` in our expectation rather than something more specific.

Specificity of expectations

Don't be too bothered about *where* the customer's name appears on the screen. Testing `document.body.textContent` is like saying "*I want this text to appear somewhere, but I don't care where.*" Often, this is enough for a test. Later on, we'll see techniques for expecting text in specific places.

There's a lot we now need to get in place in order to make the test pass. We need to introduce state and we need to add the handler. Perform the following steps:

1. Update the import at the top of the file to pull in the `useState` function as follows:

```
import React, { useState } from "react";
```

2. Wrap the constant definition in curly braces, and then return the existing value as follows:

```
export const AppointmentsDayView = (
  { appointments }
) => {
  return (
    <div id="appointmentsDayView">
```

```
    ...
  </div>
);
},
```

3. Add the following line of code above the `return` statement:

```
const [selectedAppointment, setSelectedAppointment] =
  useState(0);
```

4. We can now use `selectedAppointment` rather than hardcoding an index selecting the right appointment. Change the return value to use this new state value when selecting an appointment, like this:

```
<div id="appointmentsDayView">
  ...
<Appointment
  {...appointments[selectedAppointment]}>
</Appointment>
</div>
```

5. Change the `map` call to include an index in its arguments. Let's just name that `i` as shown here:

```
{appointments.map((appointment, i) => (
  <li key={appointment.startsAt}>
    <button type="button">
      {appointmentTimeOfDay(appointment.startsAt)}
    </button>
  </li>
))}
```

6. Now call `setSelectedAppointment` from within the `onClick` handler on the `button` element as follows:

```
<button
  type="button"
  onClick={() => setSelectedAppointment(i)}>
```

7. Run your tests, and you should find they're all green:

```
PASS test/Appointment.test.js
  Appointment
    ✓ renders the customer first name (18ms)
    ✓ renders another customer first name (2ms)
  AppointmentsDayView
    ✓ renders a div with the right id (7ms)
    ✓ renders multiple appointments in an ol element
      (16ms)
    ✓ renders each appointment in an li (4ms)
    ✓ initially shows a message saying there are no
      appointments today (6ms)
    ✓ selects the first element by default (2ms)
    ✓ has a button element in each li (2ms)
    ✓ renders another appointment when selected (3ms)
```

We've covered a lot of detail in this section, starting with specifying the initial state of the view through to adding a `button` element and handling its `onClick` event.

We now have enough functionality that it makes sense to try it out and see where we're at.

Manually testing our changes

The words *manual testing* should strike fear into the heart of every TDDer because it takes up *so* much time. Avoid it when you can. Of course, we can't avoid it entirely – when we're done with a complete feature, we need to give it a once-over to check we've done the right thing.

As it stands, we can't yet run our app. To do that, we'll need to add an entry point and then use webpack to bundle our code.

Adding an entry point

React applications are composed of a hierarchy of components that are rendered at the root. Our application entry point should render this root component.

We tend to *not* test-drive entry points because any test that loads our entire application can become quite brittle as we add more and more dependencies into it. In *Part 4, Behavior-Driven Development with Cucumber*, we'll look at using Cucumber tests to write some tests that *will* cover the entry point.

Since we aren't test-driving it, we follow a couple of general rules:

- Keep it as brief as possible
- Only use it to instantiate dependencies for your root component and to call `render`

Before we run our app, we'll need some sample data. Create a file named `src/sampleData.js` and fill it with the following code:

```
const today = new Date();

const at = (hours) => today.setHours(hours, 0);

export const sampleAppointments = [
  { startsAt: at(9), customer: { firstName: "Charlie" } },
  { startsAt: at(10), customer: { firstName: "Frankie" } },
  { startsAt: at(11), customer: { firstName: "Casey" } },
  { startsAt: at(12), customer: { firstName: "Ashley" } },
  { startsAt: at(13), customer: { firstName: "Jordan" } },
  { startsAt: at(14), customer: { firstName: "Jay" } },
  { startsAt: at(15), customer: { firstName: "Alex" } },
  { startsAt: at(16), customer: { firstName: "Jules" } },
  { startsAt: at(17), customer: { firstName: "Stevie" } },
];
```

Important note

The `Chapter02/Complete` directory in the GitHub repository contains a more complete set of sample data.

This list also doesn't need to be test-driven for the following couple of reasons:

1. It's a list of static data with no behavior. Tests are all about specifying behavior, and there's none here.
2. This module will be removed once we begin using our backend API to pull data.

Tip

TDD is often a pragmatic choice. Sometimes, not test-driving is the right thing to do.

Create a new file, `src/index.js`, and enter the following code:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { AppointmentsDayView } from "./Appointment";
import { sampleAppointments } from "./sampleData";

ReactDOM.createRoot(
  document.getElementById("root")
).render(
  <AppointmentsDayView appointments={sampleAppointments} />
);
```

That's all you'll need.

Putting it all together with webpack

Jest uses Babel to transpile all our code when it's run in the test environment. But what about when we're serving our code via our website? Jest won't be able to help us there.

That's where webpack comes in, and we can introduce it now to help us do a quick manual test as follows:

1. Install webpack using the following command:

```
npm install --save-dev webpack webpack-cli babel-loader
```

2. Add the following code to the `scripts` section of your `package.json` file:

```
"build": "webpack",
```

3. You'll also need to set some configuration for webpack. Create the `webpack.config.js` file in your project root directory with the following content:

```
const path = require("path");
const webpack = require("webpack");

module.exports = {
  mode: "development",
  module: {
    rules: [
      {
        test: /\.(\.js|jsx)$/,
```

```
        exclude: /node_modules/,  
        loader: "babel-loader",  
    },  
],  
},  
};
```

This configuration works for webpack in development mode. Consult the webpack documentation for information on setting up production builds.

4. In your source directory, run the following commands:

```
mkdir dist  
touch dist/index.html
```

5. Add the following content to the file you just created:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Appointments</title>  
  </head>  
  <body>  
    <div id="root"></div>  
    <script src="main.js"></script>  
  </body>  
</html>
```

6. You're now ready to run the build using the following command:

```
npm run build
```

You should see output such as the following:

```
modules by path ./src/*.js 2.56 KiB  
./src/index.js 321 bytes [built] [code generated]  
./src/Appointment.js 1.54 KiB [built] [code generated]  
./src/sampleData.js 724 bytes [built] [code generated]  
webpack 5.65.0 compiled successfully in 1045 ms
```

7. Open `dist/index.html` in your browser and behold your creation!

The following screenshot shows the application once the *Exercises* are completed, with added CSS and extended sample data. To include the CSS, you'll need to pull `dist/index.html` and `dist/styles.css` from the `Chapter02/Complete` directory.

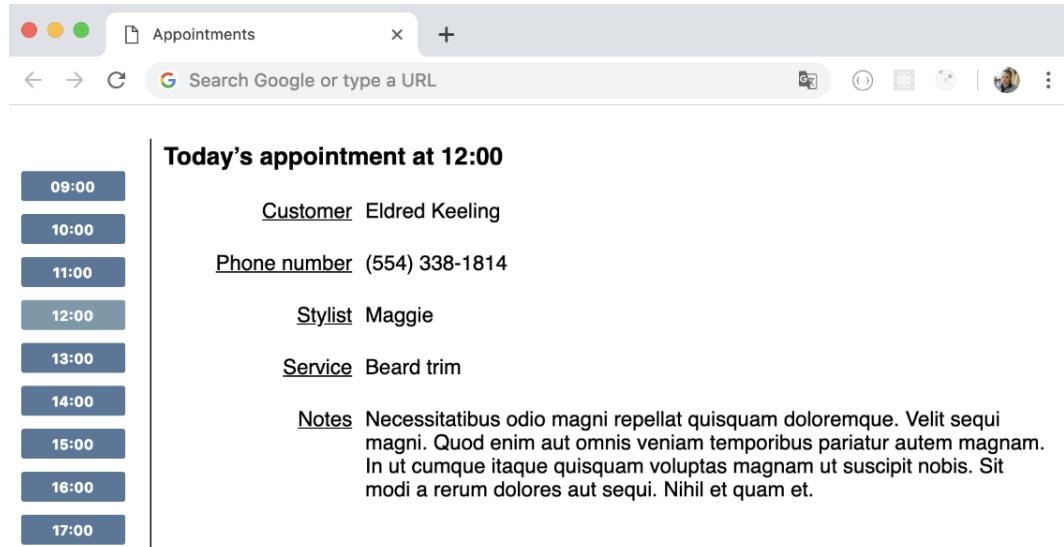


Figure 2.2 – The application so far

Before you commit your code into Git...

Make sure to add `dist/main.js` to your `.gitignore` file as follows:

```
echo "dist/main.js" >> .gitignore
```

The `main.js` file is generated by webpack, and as with most generated files, you shouldn't check it in.

You may also want to add `README.md` at this point to remind yourself how to run tests and how to build the application.

You've now seen how to put TDD aside while you created an entry point: since the entry point is small and unlikely to change frequently, we've opted not to test-drive it.

Summary

In this chapter, you've been able to practice the TDD cycle a few times and get a feel for how a feature can be built out using tests as a guide.

We started by designing a quick mock-up that helped us decide our course of action. We have built a container component (`AppointmentsDayView`) that displayed a list of appointment times, with the ability to display a single `Appointment` component depending on which appointment time was clicked.

We then proceeded to get a basic list structure in place, then extended it to show the initial `Appointment` component, and then finally added the `onClick` behavior.

This testing strategy, of starting with the basic structure, followed by the initial view, and finishing with the event behavior, is a typical strategy for testing components.

We've only got a little part of the way to fully building our application. The first few tests of any application are always the hardest and take the longest to write. We are now over that hurdle, so we'll move quicker from here onward.

Exercises

1. Rename `Appointment.js` and `Appointment.test.js` to `AppointmentsDayView.js` and `AppointmentsDayView.test.js`. While it's fine to include multiple components in one file if they form a hierarchy, you should always name the file after the root component for that hierarchy.
2. Complete the `Appointment` component by displaying the following fields on the page. You should use a `table` HTML element to give the data some visual structure. This shouldn't affect how you write your tests. The fields that should be displayed are the following:
 - Customer last name, using the `lastName` field
 - Customer telephone number, using the `phoneNumber` field
 - Stylist name, using the `stylist` field
 - Salon service, using the `service` field
 - Appointment notes, using the `notes` field
3. Add a heading to `Appointment` to make it clear which appointment time is being viewed.
4. There is some repeated sample data. We've used sample data in our tests, and we also have `sampleAppointments` in `src/sampleData.js`, which we used to manually test our application. Do you think it is worth drying this up? If so, why? If not, why not?

Further reading

Hooks are a relatively recent addition to React. Traditionally, React used classes for building components with state. For an overview of how hooks work, take a look at React's own comprehensive documentation at the following link:

<https://reactjs.org/docs/hooks-overview.html>.

3

Refactoring the Test Suite

At this point, you've written a handful of tests. Although they may seem simple enough already, they can be simpler.

It's extremely important to build a maintainable test suite: one that is quick and painless to build and adapt. One way to roughly gauge maintainability is to look at the number of lines of code in each test. To give some comparison to what you've seen so far, in the Ruby language, a test with more than *three* lines is considered a long test!

This chapter will take a look at some of the ways you can make your test suite more concise. We'll do that by extracting common code into a module that can be reused across all your test suites. We'll also create a custom Jest matcher.

When is the right time to pull out reusable code?

So far, you've written one module with two test suites within it. It's arguably too early to be looking for opportunities to extract duplicated code. Outside of an educational setting, you may wish to wait until the third or fourth test suite before you pounce on any duplication.

The following topics will be covered in this chapter:

- Pulling out reusable rendering logic
- Creating a Jest matcher using TDD
- Extracting DOM helpers

By the end of the chapter, you'll have learned how to approach your test suite with a critical eye for maintainability.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter03>.

Pulling out reusable rendering logic

In this section, we will extract a module that initializes a unique DOM container element for each test. Then, we'll build a render function that uses this container element.

The two test suites we've built both have the same `beforeEach` block that runs before each test:

```
let container;

beforeEach(() => {
  container = document.createElement("div");
  document.body.replaceChildren(container);
});
```

Wouldn't it be great if we could somehow tell Jest that any test suite that is testing a React component should *always* use this `beforeEach` block and make the `container` variable available to our tests?

Here, we will extract a new module that exports two things: the `container` variable and the `initializeReactContainer` function. This won't save us any typing, but it will hide the pesky `let` declaration and give a descriptive name to the call to `createElement`.

The importance of small functions with descriptive names

Often, it's helpful to pull out functions that contain just a single line of code. The benefit is that you can then give it a descriptive name that serves as a comment as to what that line of code does. This is preferable to using an actual comment because the name travels with you wherever you use the code.

In this case, the call to `document.createElement` could be confusing to a future maintainer of your software. Imagine that it is someone who has never done any unit testing of React code. They would be asking, "Why do the tests create a new DOM element for each and every test?" You can go some way to answer that by giving it a name, such as `initializeReactContainer`. It doesn't offer a complete answer as to why it's necessary, but it does allude to some notion of "initialization."

Let's go ahead and pull out this code:

1. Create a new file called `test/reactTestExtensions.js`. This file will ultimately contain a whole bunch of helper methods that we'll use in our React component tests.
2. Add the following content to the file. The function is implicitly updating the `container` variable within the module. That variable is then exported – our test suites can access this variable as if it were a “read-only” constant:

```
export let container;

export const initializeReactContainer = () => {
  container = document.createElement("div");
  document.body.replaceChildren(container);
}
```

3. Move to `test/AppointmentsDayView.test.js`. Add the following import just below the existing imports:

```
import {
  initializeReactContainer,
  container,
} from "./reactTestExtensions";
```

4. Now, replace the two `beforeEach` blocks – remember that there is one in each `describe` block – with the following code:

```
beforeEach(() => {
  initializeReactContainer();
});
```

5. Delete the `let container` definition from the top of both `describe` blocks.
6. Run `npm test` and verify that your tests are still passing.

Now, how about continuing with the `render` function? Let's move that into our new module. This time, it's a straight lift and replace job:

1. Copy the definition of `render` from one of the `describe` blocks.
2. Paste it into `reactTestExtensions.js`. For reference, here it is again:

```
export const render = (component) =>
  act(() =>
```

```
    ReactDOM.createRoot(container).render(component)
);
```

3. You'll also need to add these imports at the top of the file:

```
import ReactDOM from "react-dom/client";
import { act } from "react-dom/test-utils";
```

4. Back in your test file, you can now change the test extensions import so that it includes the new render function, and then remove the container import:

```
import {
  initializeReactContainer,
  render,
} from "./reactTestExtensions";
```

5. Delete the two render definitions from the two test suites.
6. Run `npm test` and verify that your tests are still passing.

So far, we've extracted two functions. We have one more to do: the `click` function. However, we have one more “action” function that we can create: `click`. Let's do that now:

1. Create the `click` function in your test extensions file, as shown here:

```
export const click = (element) =>
  act(() => element.click());
```

2. Back in your test file, adjust your import:

```
import {
  initializeReactContainer,
  container,
  render,
  click,
} from "./reactTestExtensions";
```

3. In your test suite, replace each invocation of the `click` function with the following line:

```
click(button);
```

4. The `act` import is no longer needed in your test suite. Go ahead and delete that import from your test file.
5. Run `npm test` and verify that your tests are still passing.

Avoiding the act function in your test code

The `act` function causes a fair amount of clutter in tests, which doesn't help in our quest for conciseness. Thankfully, we can push it out into our extensions module and be done with it.

Remember the **Arrange-Act-Assert** pattern that our tests should always follow? Well, we've now extracted everything we can from the **Arrange** and **Act** sections.

The approach we've taken here, of using an exported `container` variable, isn't the only approach worth exploring. You could, for example, build a wrapper function for `describe` that automatically includes a `beforeEach` block and builds a `container` variable that's accessible within the scope of that `describe` block. You could name it something like `describeReactComponent`.

An advantage of this approach is that it involves a lot less code – you won't be dealing with all those imports, and you could get rid of your `beforeEach` block in the test suites. The downside is that it's very *clever*, which is not always a good thing when it comes to maintainability. There's something a bit magical about it that requires a certain level of prior knowledge.

That being said, if this approach appeals to you, I encourage you to try it out.

In the next section, we'll start to tackle the **Assert** section of our tests.

Creating a Jest matcher using TDD

In our tests so far, we've used a variety of **matchers**. These functions tack on to the end of the `expect` function call:

```
expect(appointmentTable()).not.toBeNull();
```

In this section, you'll build a matcher using a test-driven approach to make sure it's doing the right thing. You'll learn about the Jest matcher API as you build your test suite.

You've seen quite a few matchers so far: `toBeNull`, `toContain`, `toEqual`, and `toHaveLength`. You've also seen how they can be negated with `not`.

Matchers are a powerful way of building expressive yet concise tests. You should take some time to learn all the matchers that Jest has to offer.

Jest matcher libraries

There are a lot of different matcher libraries available as npm packages. Although we won't use them in this book (since we're building everything up from first principles), you should make use of these libraries. See the *Further reading* section at the end of this chapter for a list of libraries that will be useful to you when testing React components.

Often, you'll want to build matchers. There are at least a couple of occasions that will prompt you to do this:

- An expectation you're writing is quite wordy, lengthy, or just doesn't read well in plain language.
- Some of the tests are repeating the same group of expectations again and again. This is a sign that you have a business concept that you can encode in a single matcher that will be specific to your project.

The second point is an interesting one. If you're writing the same expectations multiple times across multiple tests, you should treat it just like you would if it was repeated code in your production source code. You'd pull that out into a function. Here, the matcher serves the same purpose, except using a matcher instead of a function helps remind you that this line of code is a special statement of fact about your software: a specification.

One expectation per test

You should generally aim for just one expectation per test. "Future you" will thank you for keeping things simple! (In *Chapter 5, Adding Complex Form Interactions*, we'll look at a situation where multiple expectations are beneficial.)

You might hear this guideline and be instantly horrified. You might be imagining an explosion of tiny tests. But if you're ready to write matchers, you can aim for one expectation per test and still keep the number of tests down.

The matcher we're going to build in this section is called `toContainText`. It will replace the following expectation:

```
expect(appointmentTable().textContent).toContain("Ashley");
```

It will replace it with the following form, which is slightly more readable:

```
expect(appointmentTable()).toContainText("Ashley");
```

Here's what the output looks like on the terminal:

```

FAIL  test/AppointmentsDayView.test.js
● Appointment > renders the customer first name

expect(element).not.toContainText("Ashley")

Actual text: "CustomerAshley Phone numberStylistServiceNotes"

32 |     const customer = { firstName: "Ashley" };
33 |     render(<Appointment customer={customer} />);
> 34 |     expect(appointmentTable()).not.toContainText("Ashley");
      ^
35 |   });
36 |
37 |   it("renders another customer first name", () => {

at Object.<anonymous> (test/AppointmentsDayView.test.js:34:36)

PASS  test/matchers/toContainText.matcher.test.js

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 29 passed, 30 total
Snapshots:  0 total
Time:        1.114 s
Ran all test suites.
react-tdd/appointments %

```

Figure 3.1 – The output of the `toContainText` matcher when it fails

Let's get started:

1. Create a new directory named `test/matchers`. This is where both the source code and tests for the matchers will live.
2. Create the new `test/matchers/toContainText.test.js` file.
3. Write the first test, as shown here. This test introduces a couple of new ideas. First, it shows that `matcher` is a function that takes two parameters: the actual element and the data to match on. Second, it shows that the function returns an object with a `pass` property. This property is true if the matcher successfully “matched” – in other words, it passed:

```

import { toContainText } from "./toContainText";

describe("toContainText matcher", () => {
  it("returns pass is true when text is found in the
given DOM element", () => {
    const domElement = {
      textContent: "text to find"
    };
    const result = toContainText(
      domElement,

```

```
        "text to find"
    ) ;
    expect(result.pass).toBe(true);
}) ;
}) ;
```

4. Create another new file called `test/matchers/toContainText.js`. This first test is trivial to make pass:

```
export const toContainText = (
  received,
  expectedText
) => ({
  pass: true
}) ;
```

5. We need to triangulate to get to the real implementation. Write the next test, as shown here:

```
it("return pass is false when the text is not found in
the given DOM element", () => {
  const domElement = { textContent: "" } ;
  const result = toContainText(
    domElement,
    "text to find"
) ;
  expect(result.pass).toBe(false);
}) ;
```

6. Now, continue the implementation for our matcher, as shown here. At this stage, you have a functioning matcher – it just needs to be plugged into Jest:

```
export const toContainText = (
  received,
  expectedText
) => ({
  pass: received.textContent.includes(expectedText)
}) ;
```

7. Before we make use of this, it's good practice to fill in an expected second property of your return value: `message`. So, go ahead and do that. The following test shows that we expect the message to contain the matcher text itself, as a useful reminder to the programmer:

```
it("returns a message that contains the source line if no
match", () => {
  const domElement = { textContent: "" };
  const result =toContainText(
    domElement,
    "text to find"
  );
  expect(
    stripTerminalColor(result.message())
  ).toContain(
    `expect(element).toContainText("text to find")`
  );
}) ;
```

Understanding the `message` function

The requirements for the `message` function are complex. At a basic level, it is a helpful string that is displayed when the expectation fails. However, it's not just a string – it's a function that returns a string. This is a performance feature: the value of `message` does not need to be evaluated unless there is a failure. But even more complicated is the fact that the message should change, depending on whether the expectation was negated or not. If `pass` is `false`, then the `message` function should assume that the matcher was called in the “positive” sense – in other words, without a `.not` qualifier. But if `pass` is `true`, and the `message` function ends up being invoked, then it's safe to assume that it *was* negated. We'll need another test for this negated case, which comes a little later.

8. This function uses a `stripTerminalColor` function that we should now define, above the test suite. Its purpose is to remove any ASCII escape codes that add colors:

```
const stripTerminalColor = (text) =>
  text.replace(/\x1B\[|\d+m/g, "");
```

Testing ASCII escape codes

As you've seen already, when Jest prints out test failures, you'll see a bunch of red and green colorful text. That's achieved by printing ASCII escape codes within the text string.

This is a tricky thing to test. Because of that, we're making a pragmatic choice to not bother testing colors. Instead, the `stripTerminalColor` function strips out these escape codes from the string so that you can test the text output as if it was plain text.

9. Make that test pass by making use of Jest's `matcherHint` and `printExpected` functions, as shown here. It isn't particularly clear how the `matcherHint` function works but, hopefully, you can convince yourself that it does what we expect by running tests and seeing the last one pass! The `printExpected` functions add quotes to our value and colors it green:

```
import {
  matcherHint,
  printExpected,
} from "jest-matcher-utils";

export consttoContainText = (
  received,
  expectedText
) => {
  const pass =
    received.textContent.includes(expectedText);

  const message = () =>
    matcherHint(
      "toContainText",
      "element",
      printExpected(expectedText),
      {}
    );

  return { pass, message };
};
```

Learning about Jest's matcher utilities

At the time of writing, I've found the best way to learn what the Jest matcher utility functions do is to read their source. You could also avoid them entirely if you like – there's no obligation to use them.

10. Now comes the complicated part. Add the following test, which specifies the scenario of a failed expectation when using the negated matcher. The message should reflect that the matcher was negated, as shown here:

```
it("returns a message that contains the source line if
negated match", () => {
  const domElement = { textContent: "text to find" };
  const result = toContainText(
    domElement,
    "text to find"
  );
  expect(
    stripTerminalColor(result.message())
  ).toContain(
    `expect(container).not.toContainText("text to find")`
  );
}) ;
```

11. To make that pass, pass a new option to `matcherHint`:

```
...
matcherHint(
  "toContainText",
  "element",
  printExpected(expectedText),
  { isNot: pass }
);
...
```

12. There's one final test to add. We can print out the actual `textContent` property value of the element, which will help debug test failures when they occur. Add the following test:

```
it("returns a message that contains the actual text", () => {
  const domElement = { textContent: "text to find" };
  const result = toContainText(
    domElement,
    "text to find"
  );
  expect(
    stripTerminalColor(result.message())
  ).toContain(`Actual text: "text to find"`);
});
```

13. Make it pass by adjusting your matcher code, as shown here. Note the use of the new `printReceived` function, which is the same as `printExpected` except it colors the text red instead of green:

```
import {
  matcherHint,
  printExpected,
  printReceived,
} from "jest-matcher-utils";

export const toContainText = (
  received,
  expectedText
) => {
  const pass =
    received.textContent.includes(expectedText);

  const sourceHint = () =>
    matcherHint(
      "toContainText",
      "element",
      printExpected(expectedText),
      { isNot: pass }
    );
}
```

```
) ;

const actualTextHint = () =>
  "Actual text: " +
  printReceived(received.textContent);

const message = () =>
  [sourceHint(), actualTextHint()].join("\n\n");

return { pass, message };
};
```

14. It's time to plug the test into Jest. To do that, create a new file called `test/domMatchers.js` with the following content:

```
import {
 toContainText
} from "./matchers/toContainText";

expect.extend({
 toContainText,
});
```

15. Open `package.json` and update your Jest configuration so that it loads this file before your tests run:

```
"jest": {
  ...
  "setupFilesAfterEnv": ["./test/domMatchers.js"]
}
```

16. Your new matcher is ready to use. Open `test/AppointmentsDayView.test.js` and change all your tests that use the `expect(<element>.textContent).toEqual(<text>)` and `expect(<element>.textContent).toContain(<text>)` forms. They should be replaced with `expect(<element>).toContainText(<text>)`.
17. Run your tests; you should see them all still passing. Take a moment to play around and see how your matcher works. First, change one of the expected text values to something incorrect, and watch the matcher fail. See how the output messages look. Then, change the expected value back to the correct one, but negate the matcher by changing it to `.not.toContainText`. Finally, revert your code to the all-green state.

Why do we test-drive matchers?

You should write tests for any code that isn't just simply calling other functions or setting variables. At the start of this chapter, you extracted functions such as `render` and `click`. These functions didn't need tests because you were just transplanting the same line of code from one file to another. But this matcher does something much more complex – it must return an object that conforms to the pattern that Jest requires. It also makes use of Jest's utility functions to build up a helpful message. That complexity warrants tests.

If you are building matchers for a library, you should be more careful with your matcher's implementation. For example, we didn't bother to check that the received value is an HTML element. That's fine because this matcher exists in our code base only, and we control how it's used. When you package matchers for use in other projects, you should also verify that the function inputs are values you're expecting to see.

You've now successfully test-driven your first matcher. There will be more opportunities for you to practice this skill as this book progresses. For now, we'll move on to the final part of our cleanup: creating some fluent DOM helpers.

Extracting DOM helpers

In this section, we'll pull out a bunch of little functions that will help our tests become more readable. This will be straightforward compared to the matcher we've just built.

The `reactTestExtensions.js` module already contains three functions that you've used: `initializeReactContainer`, `render`, and `click`.

Now, we'll add four more: `element`, `elements`, `typesOf`, and `textOf`. These functions are designed to help your tests read much more like plain English. Let's take a look at an example. Here are the expectations for one of our tests:

```
const listChildren = document.querySelectorAll("li");
expect(listChildren[0].textContent).toEqual("12:00");
expect(listChildren[1].textContent).toEqual("13:00");
```

We can introduce a function, `elements`, that is a shorter version of `document.querySelectorAll`. The shorter name means we can get rid of the extra variable:

```
expect(elements("li")[0].textContent).toEqual("12:00");
expect(elements("li")[1].textContent).toEqual("13:00");
```

This code is now calling `querySelectorAll` twice – so it's doing more work than before – but it's also shorter and more readable. And we can go even further. We can boil this down to one `expect`

call by matching on the `elements` array itself. Since we need `textContent`, we will simply build a mapping function called `textOf` that takes that input array and returns the `textContent` property of each element within it:

```
expect(textOf(elements("li"))).toEqual(["12:00", "13:00"]);
```

The `toEqual` matcher, when applied to arrays, will check that each array has the same number of elements and that each element appears in the same place.

We've reduced our original three lines of code to just one!

Let's go ahead and build these new helpers:

1. Open `test/reactTestExtensions.js` and add the following definitions at the bottom of the file. You'll notice that the `elements` are using `Array.from`. This is so that the resulting array can be mapped over by both `typesOf` and `textOf`:

```
export const element = (selector) =>
  document.querySelector(selector);

export const elements = (selector) =>
  Array.from(document.querySelectorAll(selector));

export const typesOf = (elements) =>
  elements.map((element) => element.type);

export const textOf = (elements) =>
  elements.map((element) => element.textContent);
```

2. Open `test/AppointmentsDayView.test.js` and change the extensions import to include all these new functions:

```
import {
  initializeReactContainer,
  render,
  click,
  element,
  elements,
  textOf,
  typesOf,
} from "./reactTestExtensions";
```

3. Now, do a search and replace for `document.querySelectorAll`, replacing each occurrence with `elements`. Run `npm test` and verify that your tests still pass.
4. Search for and replace `document.querySelector`, replacing each occurrence with `element`. Again, run your tests and check that everything is fine.
5. You will see that the test renders at the time of the appointment. Replace the existing expectations with this one:

```
expect(textOf(elements("li"))).toEqual([
  "12:00", "13:00"
]);
```

6. Find the "has a button element in each li" test and replace the existing expectations with the following single expectation. Observe that the expectation on the length of the array is no longer necessary if your expectation tests the entire array:

```
expect(typesOf(elements("li > *"))).toEqual([
  "button",
  "button",
]);
```

7. The final three tests pull out the second button on the screen using `elements("button")[1]`. Push this definition up, just below the `beforeEach` block, and give it a more descriptive name:

```
const secondButton = () => elements("button")[1];
```

8. Now, you can use this in the three tests. Go ahead and update them now. For example, the middle test can be updated as follows:

```
click(secondButton());
expect(secondButton().className).toContain("toggled");
```

9. As a final touch, inline the `listChild` and `listElement` variables that appear in some of the tests – in other words, remove the use of variables and call the function directly within the expectation. As an example, the "renders an ol element to display appointments" test can have its expectation rewritten, as follows:

```
expect(element("ol")).not.toBeNull();
```

10. Run `npm test` one final time and verify that everything is still green.

Not all helpers need to be extracted

You'll notice that the helpers you have extracted are all very generic – they make no mention of the specific components under test. It's good to keep helpers as generic as possible. On the other hand, sometimes it helps to have very localized helper functions. In your test suite, you already have one called `appointmentsTable` and another called `secondButton`. These should remain in the test suite because they are local to the test suite.

In this section, you've seen our final technique for simplifying your test suites, which is to pull out fluent helper functions that help keep your expectations short and help them read like plain English.

You've also seen the trick of running expectations on an array of items rather than having an expectation for individual items. This isn't always the appropriate course of action. You'll see an example of this in *Chapter 5, Adding Complex Form Interactions*.

Summary

This chapter focused on improving our test suites. Readability is crucially important. Your tests act as specifications for your software. Each component test must clearly state what the expectation of the component is. And when a test fails, you want to be able to understand why it's failed as quickly as possible.

You've seen that these priorities are often in tension with our usual idea of what good code is. For example, in our tests, we are willing to sacrifice performance if it makes the tests more readable.

If you've worked with React tests in the past, think about how long an average test was. In this chapter, you've seen a couple of mechanisms for keeping your test short: building domain-specific matchers and extracting little functions for querying the DOM.

You've also learned how to pull out React initialization code to avoid clutter in our test suites.

In the next chapter, we'll move back to building new functionality into our app: data entry with forms.

Exercises

Using the techniques you've just learned, create a new matcher named `toHaveClass` that replaces the following expectation:

```
expect(secondButton().className).toContain("toggled");
```

With your new matcher in place, it should read as follows:

```
expect(secondButton()).toHaveClass("toggled");
```

There is also the negated form of this matcher:

```
expect(secondButton().className).not.toContain("toggled");
```

Your matcher should work for this form and display an appropriate failure message.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- The following GitHub repository contains useful matchers for testing React components:
<https://github.com/jest-community/jest-extended>
- The following link provides the source of Jest's matcher utilities, which I find useful for figuring out how to write simple matchers: <https://github.com/facebook/jest/tree/main/packages/jest-matcher-utils>

4

Test-Driving Data Input

In this chapter, you'll explore React forms and controlled components.

Forms are an essential part of building web applications, being the primary way that users enter data. If we want to ensure our application works, then invariably, that'll mean we need to write automated tests for our forms. What's more, there's a lot of plumbing required to get forms working in React, making it even more important that they're well-tested.

Automated tests for forms are all about the user's behavior: entering text, clicking buttons, and submitting the form when complete.

We will build out a new component, `CustomerForm`, which we will use when adding or modifying customers. It will have three text fields: first name, last name, and phone number.

In the process of building this form, you'll dig deeper into testing complex DOM element trees. You'll learn how to use parameterized tests to repeat a group of tests without duplicating code.

The following topics will be covered in this chapter:

- Adding a form element
- Accepting text input
- Submitting a form
- Duplicating tests for multiple form fields

By the end of this chapter, you'll have a decent understanding of test-driving HTML forms with React.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter04>.

Adding a form element

An HTML form is a bunch of fields wrapped in a `form` element. Even though we're mostly interested in the fields, we need to start with the `form` element itself. That's what we'll build in this section.

Let's create our first form by following these steps:

1. Create a new file called `test/CustomerForm.test.js` and add the following scaffold. It contains all the usual imports and component test initialization that you've seen in the previous chapters:

```
import React from "react";
import {
  initializeReactContainer,
  render,
  element,
} from "./reactTestExtensions";
import { CustomerForm } from "../src/CustomerForm";

describe("CustomerForm", () => {
  beforeEach(() => {
    initializeReactContainer();
  });
});
```

2. Now you're ready to create your first test. Add the following test to the `describe` block:

```
it("renders a form", () => {
  render(<CustomerForm />);
  expect(element("form")).not.toBeNull();
});
```

3. We have a complete test, so let's run it and see what happens:

```
FAIL test/CustomerForm.test.js
● Test suite failed to run

      Cannot find module '../src/CustomerForm' from
      'CustomerForm.test.js'
```

The failure tells us that it can't find the module. That's because we haven't created it yet.

4. So, create a blank file named `src/CustomerForm.js`. Running your test again should give you the following output:

```
FAIL test/CustomerForm.test.js
● CustomerForm > renders a form

      Element type is invalid: expected a string (for
      built-in components) or a class/function (for composite
      components) but got: undefined. You likely forgot to
      export your component from the file it's defined in, or
      you might have mixed up default and named imports.

          8 |
          9 |   export const render = (component) =>
> 10 |     act(() =>
          11 |       ReactDOM.createRoot(...).render(...)
          |         ^
          |
          12 |   );
          11 |
          12 |   export const click = (element) =>
          13 |     act(() => element.click());
```

Stack traces from test helper code

Jest's stack trace points to a failure within our extensions code, not the test itself. If our code was in an npm module, Jest would have skipped those test lines from its output. Thankfully, the error message is helpful enough.

5. To fix this issue, we need to add an export that matches the import we wrote at the top of our test file. Add the following line to `src/CustomerForm.js`:

```
export const CustomerForm = () => null;
```

6. Running some tests gives the actual expectation failure:

```
● CustomerForm > renders a form
```

```
expect(received).not.toBeNull()
Received: null
```

This can be fixed by making the component return something:

```
import React from "react";

export const CustomerForm = () => <form />;
```

Before moving on, let's pull out a helper for finding the `form` element. As in the previous chapter, this is arguably premature as we have only one test using this code right now. However, we'll appreciate having the helper when we come to write our form submission tests later.

7. Open `test/reactTestExtensions.js` and add the following function:

```
export const form = (id) => element("form");
```

8. Modify your test file by adding the following `import`. You can leave the `element` import in place because we'll use it later in the next section:

```
import {
  initializeReactContainer,
  render,
  element,
  form,
} from "./reactTestExtensions";
```

9. Finally, update your test to use the helper, as shown here. After this, your test should still be passing:

```
it("renders a form", () => {
  render(<CustomerForm />);
  expect(form()).not.toBeNull();
});
```

That's all there is to creating the basic `form` element. With that wrapper in place, we're now ready to add our first field element: a text box.

Accepting text input

In this section, we'll add a text box to allow the customer's first name to be added or edited.

Adding a text field is more complicated than adding the `form` element. First, there's the element itself, which has a `type` attribute that needs to be tested. Then, we need to prime the element with the initial value. Finally, we'll need to add a label so that it's obvious what the field represents.

Let's start by rendering an HTML text input field onto the page:

1. Add the following test to `test/CustomerForm.test.js`. It contains three expectations (there's an exercise at the end of this chapter that you can follow to pull these out as a single matcher):

```
it("renders the first name field as a text box", () => {
  render(<CustomerForm />);
  const field = form().elements.firstName;
  expect(field).not.toBeNull();
  expect(field.tagName).toEqual("INPUT");
  expect(field.type).toEqual("text");
}) ;
```

Relying on the DOM's Form API

This test makes use of the Form API: any form element allows you to access all of its input elements using the `elements` indexer. You give it the element's `name` attribute (in this case, `firstName`) and that element is returned.

This means we must check the returned element's tag. We want to make sure it is an `<input>` element. If we hadn't used the Form API, one alternative would have been to use `elements("input") [0]`, which returns the first input element on the page. This would make the expectation on the element's `tagName` property unnecessary.

2. Let's move a bit faster. We'll make all the expectations pass at once. Update `CustomerForm` so that it includes a single input field, as shown here:

```
export const CustomerForm = () => (
  <form>
    <input type="text" name="firstName" />
  </form>
) ;
```

3. Since this form will be used when modifying existing customers as well as adding new ones, we need to design a way to get the existing customer data into the component. We'll do that by setting an `original` prop that contains the form data. Add the following test:

```
it("includes the existing value for the first name", () => {
  const customer = { firstName: "Ashley" };
  render(<CustomerForm original={customer} />);
```

```
    const field = form().elements.firstName;
    expect(field.value).toEqual("Ashley");
});
```

- To make this test pass, change the component definition to the following. We will use a prop to pass in the previous `firstName` value:

```
export const CustomerForm = ({ original }) => (
  <form>
    <input
      type="text"
      name="firstName"
      value={original.firstName} />
  </form>
);
```

- Upon running the tests again, you'll see that although this test now passes, the first two tests fail because they don't specify the `original` prop. What's more, we have a warning:

Warning: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.

- To fix the initial tests, create a new constant, `blankCustomer`, that will act as our “base” customer. It'll do just fine for tests that don't care about specific field values, such as our first two tests. Add this definition just above the `beforeEach` block:

```
const blankCustomer = {
  firstName: "",
```

What about specifying an empty object for the `original` prop?

In this object definition, we set the `firstName` value to an empty string. You may think that either `undefined` or `null` would be good candidates for the value. That way, we could sidestep having to define an object like this and just pass an empty object, `{ }`. Unfortunately, React will warn you when you attempt to set a controlled component's initial value to `undefined`, which we want to avoid. It's no big deal, and besides that, an empty string is a more realistic default for a text box.

7. Update the first two tests so that they render with the `original` prop set, as shown here. With this change in place, you should have three passing tests, but the warning remains:

```
it("renders a form", () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(form()).not.toBeNull();
});

it("renders the first name field as a text box", () => {
  render(<CustomerForm original={blankCustomer} />);
  const field = form().elements.firstName;
  expect(field).not.toBeNull();
  expect(field.tagName).toEqual("INPUT");
  expect(field.type).toEqual("text");
});
```

8. To get rid of the warning, add the word `readOnly` to the `input` tag. You might be thinking: surely, we don't want a read-only field? You're right, but we need a further test, for modifying the input value, before we can avoid using the `readOnly` keyword. We'll add that test a little further on:

```
<input
  type="text"
  name="firstName"
  value={original.firstName}
  readOnly
/>
```

Tip

Always consider React warnings to be a test failure. Don't proceed without first fixing any warnings.

9. The last two tests include the following line, which reaches inside the form to pull out the `firstName` field:

```
const field = form().elements.firstName;
```

Let's promote this to be a function in `test/reactTestExtensions.js`. Open that file and add the following definition after the definition for `form`:

```
export const field = (fieldName) =>
  form().elements[fieldName];
```

10. Then, import it into `test/CustomerForm.js`:

```
import {
  initializeReactContainer,
  render,
  element,
  form,
  field,
} from "./reactTestExtensions";
```

11. Change the last test you wrote so that it uses the new helper:

```
it("includes the existing value for the first name", () => {
  const customer = { firstName: "Ashley" };
  render(<CustomerForm original={customer} />);
  expect(field("firstName").value).toEqual("Ashley");
});
```

12. Update the first test in the same way:

```
it("renders the first name field as a text box", () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(field("firstName")).not.toBeNull();
  expect(field("firstName")).toEqual("INPUT");
  expect(field("firstName")).toEqual("text");
});
```

13. Next up, we'll add a label to the field. Add the following test, which uses the `element` helper:

```
it("renders a label for the first name field", () => {
  render(<CustomerForm original={blankCustomer} />);
  const label = element("label[for=firstName]");
```

```
    expect(label).not.toBeNull();
});
```

14. Make this pass by inserting the new element into your JSX for `CustomerForm`:

```
<form
  <label htmlFor="firstName" />
  ...
</form>
```

The `htmlFor` attribute

The JSX `htmlFor` attribute sets the HTML `for` attribute. `for` couldn't be used in JSX because it is a reserved JavaScript keyword. The attribute is used to signify that the label matches a form element with the given ID – in this case, `firstName`.

15. Let's add some text content to that label:

```
it("renders 'First name' as the first name label
content", () => {
  render(<CustomerForm original={blankCustomer} />);
  const label = element("label[for=firstName]");
  expect(label).toContainText("First name");
});
```

16. Update the `label` element to make the test pass:

```
<form
  <label htmlFor="firstName">First name</label>
  ...
</form>
```

17. Finally, we need to ensure that our input has an ID that matches it with the label's `htmlFor` value so that they match up. Add the following test:

```
it("assigns an id that matches the label id to the first
name field", () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(field("firstName").id).toEqual("firstName");
});
```

18. Making that pass is as simple as adding the new attribute:

```
<form>
  <label htmlFor="firstName">First name</label>
  <input
    type="text"
    name="firstName"
    id="firstName"
    value={firstName}
    readOnly
  />
</form>
```

We've now created *almost* everything we need for this field: the input field itself, its initial value, and its label. But we don't have any behavior for handling changes to the value – that's why we have the `readOnly` flag.

Change behavior only makes sense in the context of submitting the form with updated data: if you can't submit the form, there's no point in changing the field value. That's what we'll cover in the next section.

Submitting a form

For this chapter, we will define "submit the form" to mean "call the `onSubmit` callback function with the current `customer` object." The `onSubmit` callback function is a prop we'll be passing.

This section will introduce one way of testing form submission. In *Chapter 6, Exploring Test Doubles*, we will update this to a call to `global.fetch` that sends our customer data to our application's backend API.

We'll need a few different tests to specify this behavior, each test building up the functionality we need in a step-by-step fashion. First, we'll have a test that ensures the form has a submit button. Then, we'll write a test that clicks that button without making any changes to the form. We'll need another test to check that submitting the form does not cause page navigation to occur. Finally, we'll end with a test submission after the value of the text box has been updated.

Submitting without any changes

Let's start by creating a button in the form. Clicking it will cause the form to submit:

1. Start by adding a test to check whether a submit button exists on the page:

```
it("renders a submit button", () => {
  render(<CustomerForm original={blankCustomer} />);
```

```
const button = element("input[type=submit]");
expect(button).not.toBeNull();
});
```

2. To make that pass, add the following single line at the bottom of the form's JSX:

```
<form>
  ...
  <input type="submit" value="Add" />
</form>
```

3. The following test introduces a new concept, so we'll break it down into its component parts. To start, create a new test, `starting`, as follows:

```
it("saves existing first name when submitted", () => {
  expect.hasAssertions();
});
```

The `hasAssertions` expectation tells Jest that it should expect at least one assertion to occur. It tells Jest that at least one assertion must run within the scope of the test; otherwise, the test has failed. You'll see why this is important in the next step.

4. Add the following part of the test into the outline, below the `hasAssertions` call:

```
const customer = { firstName: "Ashley" };
render(
  <CustomerForm
    original={customer}
    onSubmit={({ firstName }) =>
      expect(firstName).toEqual("Ashley")
    }
  />
);
```

This function call is a mix of the **Arrange** and **Assert** phases in one. The **Arrange** phase is the `render` call itself, and the **Assert** phase is the `onSubmit` handler. This is the handler that we want React to call on form submission.

5. Finish off the test by adding the following line just below the call to `render`. This is the **Act** phase of our test, which in this test is the last phase of the test:

```
const button = element("input[type=submit]");
click(button);
```

Using hasAssertions to avoid false positives

You can now see why we need `hasAssertions`. The test is written out of order, with the assertions defined within the `onSubmit` handler. If we did not use `hasAssertions`, this test would pass right now because we never call `onSubmit`.

I don't recommend writing tests like this. In *Chapter 6, Exploring Test Doubles*, we'll discover **test doubles**, which allow us to restore the usual *Arrange-Act-Assert* order to help us avoid the need for `hasAssertions`. The method we're using here is a perfectly valid TDD practice; it's just a little messy, so you will want to refactor it eventually.

6. Now, you need to import `click`:

```
import {  
    initializeReactContainer,  
    render,  
    element,  
    form,  
    field,  
    click,  
} from "./reactTestExtensions";
```

7. Making this test pass is straightforward, despite the complicated test setup. Change the component definition so that it reads as follows:

```
export const CustomerForm = ({  
    original,  
    onSubmit  
}) => (  
    <form onSubmit={() => onSubmit(original)}>  
        ...  
    </form>  
);
```

8. Now, run the test with `npm test`. You'll discover that the test passed but we have a new warning, as shown here:

```
console.error  
Error: Not implemented: HTMLFormElement.prototype.submit  
      at module.exports (.../node_modules/jsdom/lib/jsdom/browser/not-implemented.js:9:17)
```

Something is not quite right. This warning is highlighting something very important that we need to take care of. Let's stop here and look at it in detail.

Preventing the default submit action

This `Not implemented` console error is coming from the JSDOM package. HTML forms have a default action when submitted: they navigate to another page, which is specified by the `form` element's `action` attribute. JSDOM does not implement page navigation, which is why we get a `Not implemented` error.

In a typical React application like the one we're building, we don't want the browser to navigate. We want to stay on the same page and allow React to update the page with the result of the submit operation.

The way to do that is to grab the `event` argument from the `onSubmit` prop and call `preventDefault` on it:

```
event.preventDefault();
```

Since that's production code, we need a test that verifies this behavior. We can do this by checking the event's `defaultPrevented` property:

```
expect(event.defaultPrevented).toBe(true);
```

So, now the question becomes, how do we get access to this `Event` in our tests?

We need to create the `event` object ourselves and dispatch it directly using the `dispatchEvent` DOM function on the form element. This event needs to be marked as `cancelable`, which will allow us to call `preventDefault` on it.

Why clicking the submit button won't work

In the last couple of tests, we purposely built a submit button that we could click to submit the form. While that will work for all our other tests, for this specific test, it does *not* work. That's because JSDOM will take a `click` event and internally convert it into a `submit` event. There is no way we can get access to that `submit` event object if JSDOM creates it. Therefore, we need to directly fire the `submit` event.

This isn't a problem. Remember that, in our test suite, we strive to act as a real browser would – by clicking a submit button to submit the form – but having one test work differently isn't the end of the world.

Let's put all of this together and fix the warning:

1. Open `test/reactTestExtensions.js` and add the following, just below the `click` definition. We'll use this in the next test:

```
export const submit = (formElement) => {
  const event = new Event("submit", {
    bubbles: true,
    cancelable: true,
  });
  act(() => formElement.dispatchEvent(event));
  return event;
};
```

Why do we need the `bubbles` property?

If all of this wasn't complicated enough, we also need to make sure the event `bubbles`; otherwise, it won't make it to our event handler.

When JSDOM (or the browser) dispatches an event, it traverses the element hierarchy looking for an event handler to handle the event, starting from the element the event was dispatched on, working upwards via parent links to the root node. This is known as **boiling**.

Why do we need to ensure this event bubbles? Because React has its *own* event handling system that is triggered by events reaching the React root element. The `submit` event must bubble up to our `container` element before React will process it.

2. Import the new helper into `test/CustomerForm.test.js`:

```
import {
  ...
  submit,
} from "./reactTestExtensions";
```

3. Add the following test to the bottom of the `CustomerForm` test suite. It specifies that `preventDefault` should be called when the form is submitted:

```
it("prevents the default action when submitting the
form", () => {
  render(  
  <CustomerForm />
```

```
<CustomerForm
  original={blankCustomer}
  onSubmit={() => {}}
/>
);

const event = submit(form());

expect(event.defaultPrevented).toBe(true);
});
```

4. To make that pass, first, update `CustomerForm` so that it has an explicit return:

```
export const CustomerForm = ({
  original,
  onSubmit
}) => {
  return (
    <form onSubmit={() => onSubmit(original)}>
      ...
    </form>
  );
};
```

5. Just above the return, add a new function, `handleSubmit`, and update the form so that it calls that instead:

```
export const CustomerForm = ({
  original,
  onSubmit
}) => {
  const handleSubmit = (event) => {
    event.preventDefault();
    onSubmit(original);
  };

  return (
```

```
<form onSubmit={handleSubmit}>
</form>
) ;
};
```

6. Run your tests and ensure they are all passing.

Submitting changed values

It's finally the time to introduce some state into our component. We will specify what should happen when the text field is used to update the customer's first name.

The most complicated part of what we're about to do is dispatching the DOM change event. In the browser, this event is dispatched after every keystroke, notifying the JavaScript application that the text field value content has changed. An event handler receiving this event can query the `target` element's `value` property to find out what the current value is.

Crucially, we're responsible for setting the `value` property before we dispatch the `change` event. We do that by calling the `value` property setter.

Somewhat unfortunately for us testers, React has change tracking behavior that is designed for the browser environment, not the Node test environment. In our tests, this change tracking logic suppresses change events like the ones our tests will dispatch. We need to circumvent this logic, which we can do with a helper function called `originalValueProperty`, as shown here:

```
const originalValueProperty = (reactElement) => {
  const prototype =
    Object.getPrototypeOf(reactElement);
  return Object.getOwnPropertyDescriptor(
    prototype,
    "value"
  );
};
```

As you'll see in the next section, we'll use this function to bypass React's change tracking and trick it into processing our event, just like a browser would.

Only simulating the final change

Rather than creating a `change` event for each keystroke, we'll manufacture just the final instance. Since the event handler always has access to the full value of the element, it can ignore all intermediate events and process just the last one that is received.

Let's begin with a little bit of refactoring:

1. We're going to use the submit button to submit the form. We figured out how to access that button in a previous test:

```
const button = element("input [type=submit]");
```

Let's move this definition into `test/reactTestExtensions.js` so that we can use it on our future tests. Open that file now and add this definition to the bottom:

```
export const submitButton = () =>
  element("input [type=submit]");
```

2. Move back to `test/CustomerForm.test.js` and add the new helper to the imports:

```
import {
  ...
  submitButton,
} from "./reactTestExtensions";
```

3. Update the `renders a submit button` test so that it uses that new helper, as shown here:

```
it("renders a submit button", () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(submitButton()).not.toBeNull();
});
```

The helper extraction dance

Why are we doing this dance of writing a variable in a test (such as `const button = ...`) only to then extract it as a function moments later, as we just did with `submitButton`?

Following this approach is a systematic way of building a library of helper functions, meaning you don't have to think too heavily about the "right" design. First, start with a variable. If it turns out that you'll use that variable a second or third time, then extract it into a function. No big deal.

4. It's time to write the next test. This is very similar to the last test, except now, we need to make use of a new `change` helper function. We'll define this in the next step:

```
it("saves new first name when submitted", () => {
  expect.hasAssertions();
  render(
    <CustomerForm
```

```
        original={blankCustomer}
        onSubmit={({ firstName }) =>
          expect(firstName).toEqual("Jamie")
        }
      />
);
change(field("firstName"), "Jamie");
click(submitButton());
});
```

5. This function uses the new `change` helper that was discussed at the beginning of this section. Add the following definitions to `test/reactTestExtensions.js`:

```
const originalValueProperty = (reactElement) => {
  const prototype =
    Object.getPrototypeOf(reactElement);
  return Object.getOwnPropertyDescriptor(
    prototype,
    "value"
  );
};

export const change = (target, value) => {
  originalValueProperty(target).set.call(
    target,
    value
  );
  const event = new Event("change", {
    target,
    bubbles: true,
  });
  act(() => target.dispatchEvent(event));
};
```

Figuring out interactions between React and JSDOM

The implementation of the `change` function shown here is not obvious. As we saw earlier with the `bubbles` property, React does some pretty clever stuff on top of the DOM's usual event system.

It helps to have a high-level awareness of how React works. I also find it helpful to use the Node debugger to step through JSDOM and React source code to figure out where the flow is breaking.

6. To make this pass, move to `src/CustomerForm.js` and import `useState` into the module by modifying the existing React import:

```
import React, { useState } from "react";
```

7. Change the customer constant definition to be assigned via a call to `useState`. The default state is the original value of `customer`:

```
const [ customer, setCustomer ] = useState(original);
```

8. Create a new arrow function that will act as our event handler. You can put this just after the `useState` line that you added in the previous step:

```
const handleChangeFirstName = ({ target }) =>
  setCustomer((customer) => ({
    ...customer,
    firstName: target.value
  }));
```

9. In the returned JSX, modify the `input` element, as shown here. We are replacing the `readOnly` property with an `onChange` property and hooking it up to the handler we just created. Now, the `value` property also needs to be updated so that it can use React's component state rather than the component prop:

```
<input
  type="text"
  name="firstName"
  id="firstName"
  value={customer.firstName}
  onChange={handleChangeFirstName}
/>
```

10. Go ahead and run the test; it should now be passing.

With that, you've learned how to test-drive the `change` DOM event, and how to hook it up with React's component state to save the user's input. Next, it's time to repeat the process for two more text boxes.

Duplicating tests for multiple form fields

So far, we've written a set of tests that fully define the `firstName` text field. Now, we want to add two more fields, which are essentially the same as the `firstName` field but with different `id` values and labels.

Before you reach for copy and paste, stop and think about the duplication you could be about to add to both your tests and your production code. We have six tests that define the first name. This means we would end up with 18 tests to define three fields. That's a lot of tests without any kind of grouping or abstraction.

So, let's do both – that is, group our tests and abstract out a function that generates our tests for us.

Nesting describe blocks

We can nest `describe` blocks to break similar tests up into logical contexts. We can invent a convention for how to name these `describe` blocks. Whereas the top level is named after the form itself, the second-level `describe` blocks are named after the form fields.

Here's how we'd like them to end up:

```
describe("CustomerForm", () => {
  describe("first name field", () => {
    // ... tests ...
  });
  describe("last name field", () => {
    // ... tests ...
  });
  describe("phone number field", () => {
    // ... tests ...
  });
}) ;
```

With this structure in place, you can simplify the `it` descriptive text by removing the name of the field. For example, "renders the first name field as a text box" becomes "renders as a text box" because it has already been scoped by the "first name field" `describe` block. Because of the way Jest displays `describe` block names before test names in the test output, each of these still reads like a plain-English sentence, but without the verbiage. In the example just given, Jest will show us `CustomerForm first name field renders as a text box`.

Let's do that now for the first name field. Wrap the six existing tests in a `describe` block, and then rename the tests, as shown here:

```
describe("first name field", () => {
  it("renders as a text box" ... );

  it("includes the existing value" ... );

  it("renders a label" ... );

  it("assigns an id that matches the label id" ... );

  it("saves existing value when submitted" ... );

  it("saves new value when submitted" ... );
});
```

Be careful not to include the `preventsDefault` test out of this, as it's not field-specific. You may need to adjust the positioning of your tests in your test file.

That covers grouping the tests. Now, let's look at using test generator functions to remove repetition.

Generating parameterized tests

Some programming languages, such as Java and C#, require special framework support to build parameterized tests. But in JavaScript, we can very easily roll our own parameterization because our test definitions are just function calls. We can use this to our advantage by pulling out each of the existing six tests as functions that take parameter values.

This kind of change requires some diligent refactoring. We'll do the first two tests together, and then you can either repeat these steps for the remaining five tests or jump ahead to the next tag in the GitHub repository:

1. Starting with `renders as a text box`, wrap the entirety of the `it` call in an arrow function, and then call that function straight after, as shown here:

```
const itRendersAsATextBox = () =>
  it("renders as a text box", () => {
    render(<CustomerForm original={blankCustomer} />);
    expect(field("firstName")).not.toBeNull();
    expect(field("firstName").tagName).toEqual(
```

```
        "INPUT"
    );
    expect(field("firstName").type).toEqual("text");
});

itRendersAsATextBox();
```

2. Verify that all your tests are passing.
3. Parameterize this function by promoting the `firstName` string to a function parameter. Then, you'll need to pass in the `firstName` string into the function call itself, as shown here:

```
const itRendersAsATextBox = (fieldName) =>
  it("renders as a text box", () => {
    render(<CustomerForm original={blankCustomer} />);
    expect(field(fieldName)).not.toBeNull();
    expect(field(fieldName).tagName).toEqual("INPUT");
    expect(field(fieldName).type).toEqual("text");
  });

itRendersAsATextBox("firstName");
```

4. Again, verify that your tests are passing.
5. Push the `itRendersAsATextBox` function up one level, into the parent `describe` scope. That will allow you to use it in subsequent `describe` blocks.
6. Use the same procedure for the next test, `includes the existing value`:

```
const itIncludesTheExistingValue = (
  fieldName,
  existing
) =>
  it("includes the existing value", () => {
    const customer = { [fieldName]: existing };
    render(<CustomerForm original={customer} />);
    expect(field(fieldName).value).toEqual(existing);
  });

itIncludesTheExistingValue("firstName", "Ashley");
```

7. Verify your tests are passing and then push `itIncludesTheExistingValue` up one level, into the parent `describe` scope.
8. Repeat this process for the label tests, which can be included in one function. The second test can use a parameter within its test definition, as shown here:

```
const itRendersALabel = (fieldName, text) => {
  it("renders a label for the text box", () => {
    render(<CustomerForm original={blankCustomer} />);
    const label = element(`label[for=${fieldName}]`);
    expect(label).not.toBeNull();
  });

  it(`renders '${text}' as the label content`, () => {
    render(<CustomerForm original={blankCustomer} />);
    const label = element(`label[for=${fieldName}]`);
    expect(label).toContainText(text);
  });
};
```

9. Repeat the same process for the three remaining tests:

```
const itAssignsAnIdThatMatchesTheLabelId = (
  fieldName
) =>
  ...
const itSubmitsExistingValue = (fieldName, value) =>
  ...
const itSubmitsNewValue = (fieldName, value) =>
  ...
```

Important note

Check the completed solution for the full listing. This can be found in the `Chapter04/Complete` directory.

10. With all that done, your `describe` block will succinctly describe what the first name field does:

```
describe("first name field", () => {
  itRendersAsATextBox("firstName");
  itIncludesTheExistingValue("firstName", "Ashley");
  itRendersALabel("firstName", "First name");
  itAssignsAnIdThatMatchesTheLabelId("firstName");
  itSubmitsExistingValue("firstName", "Ashley");
  itSubmitsNewValue("firstName", "Jamie");
});
```

Take a step back and look at the new form of the `describe` block. It is now very quick to understand the specification for how this field should work.

Solving a batch of tests

Now, we want to duplicate those six tests for the last name field. But how do we approach this? We do this test by test, just as we did with the first name field. However, this time, we should go much faster as our tests are one-liners, and the production code is a copy and paste job.

So, for example, the first test will be this:

```
describe("last name field", () => {
  itRendersAsATextBox("lastName");
});
```

You'll need to update `blankCustomer` so that it includes the new field:

```
const blankCustomer = {
  firstName: "",
  lastName: "",
};
```

That test can be made to pass by adding the following line to our JSX, just below the `firstName` input field:

```
<input type="text" name="lastName" />
```

This is just the start for the input field; you'll need to complete it as you add the next few tests.

Go ahead and add the remaining five tests, along with their implementation. Then, repeat this process for the phone number field. When adding the submit tests for the phone number, make sure that you provide a string value made up of numbers, such as "012345". Later in this book, we'll add validations to this field that will fail if you don't use the right values now.

Jumping ahead

You might be tempted to try to solve all 12 new tests at once. If you're feeling confident, go for it!

If you want to see a listing of all the tests in a file, you must invoke Jest with a single file. Run the `npm test test/CustomerForm.test.js` command to see what that looks like. Alternatively, you can run `npx jest --verbose` to run all the tests with full test listings:

```
PASS test/CustomerForm.test.js
CustomerForm
  ✓ renders a form (28ms)
    first name field
      ✓ renders as a text box (4ms)
      ✓ includes the existing value (3ms)
      ✓ renders a label (2ms)
      ✓ saves existing value when submitted (4ms)
      ✓ saves new value when submitted (5ms)
    last name field
      ✓ renders as a text box (3ms)
      ✓ includes the existing value (2ms)
      ✓ renders a label (6ms)
      ✓ saves existing value when submitted (2ms)
      ✓ saves new value when submitted (3ms)
    phone number field
      ✓ renders as a text box (2ms)
      ✓ includes the existing value (2ms)
      ✓ renders a label (2ms)
      ✓ saves existing value when submitted (3ms)
      ✓ saves new value when submitted (2ms)
```

Modifying handleChange so that it works with multiple fields

Time for a small refactor. After adding all three fields, you will have ended up with three very similar `onChange` event handlers:

```
const handleChangeFirstName = ({ target }) =>
  setCustomer((customer) => ({
    ...customer,
    firstName: target.value
 )));

const handleChangeLastName = ({ target }) =>
  setCustomer((customer) => ({
    ...customer,
    lastName: target.value
 )));

const handleChangePhoneNumber = ({ target }) =>
  setCustomer((customer) => ({
    ...customer,
    phoneNumber: target.value
 ))};
```

You can simplify these down into one function by making use of the `name` property on `target`, which matches the field ID:

```
const handleChange = ({ target }) =>
  setCustomer(customer => ({
    ...customer,
    [target.name]: target.value
 ))};
```

Testing it out

At this stage, your the `AppointmentsDayView` instance is complete. Now is a good time to try it out for real.

Update your entry point in `src/index.js` so that it renders a new `CustomerForm` instance, rather than `AppointmentsDayView`. By doing so, you should be ready to manually test:

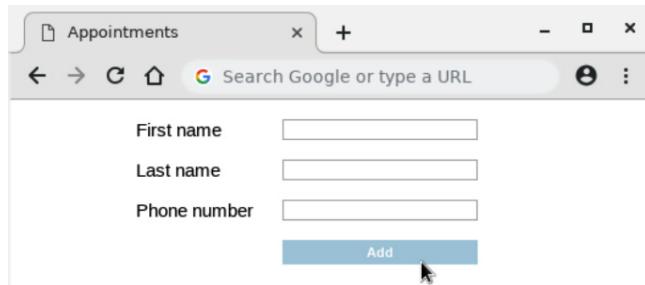


Figure 4.1 – The completed CustomerForm

With that, you have learned one way to quickly duplicate specifications across multiple form fields: since `describe` and `it` are plain old functions, you can treat them just like you would with any other function and build your own structure around them.

Summary

In this chapter, you learned how to create an HTML form with text boxes. You wrote tests for the `form` element, and for `input` elements of types `text` and `submit`.

Although the text box is about the most basic input element there is, we've taken this opportunity to dig much deeper into test-driven React. We've discovered the intricacies of raising `submit` and `change` events via JSDOM, such as ensuring that `event.preventDefault()` is called on the event to avoid a browser page transition.

We've also gone much further with Jest. We extracted common test logic into modules, used nested `describe` blocks, and built assertions using DOM's Form API.

In the next chapter, we'll test-drive a more complicated form example: a form with select boxes and radio buttons.

Exercises

The following are some exercises for you to try out:

1. Extract a `labelFor` helper into `test/reactTestExtensions.js`. It should be used like so:

```
expect(labelFor(fieldName)).not.toBeNull();
```

2. Add a `toBeInputFieldType` matcher that replaces the three expectations in the `itRendersAsATextBox` function. It should be used like so:

```
expect(field(fieldName)).toBeInputFieldType("text");
```


5

Adding Complex Form Interactions

It's time to apply what you've learned to a more complicated HTML setup. In this chapter, we'll test-drive a new component: `AppointmentForm`. It contains a select box, for selecting the service required, and a grid of radio buttons that form a calendar view for selecting the appointment time.

Combining both layout and form input, the code in this chapter shows how TDD gives you a structure for your work that makes even complicated scenarios straightforward: you will use your tests to grow the component into a component hierarchy, splitting out functionality from the main component as it begins to grow.

In this chapter, we will cover the following topics:

- Choosing a value from a select box
- Constructing a calendar view
- Test-driving radio button groups
- Reducing effort when constructing components

By the end of the chapter, you'll have learned how to apply test-driven development to complex user input scenarios. These techniques will be useful for all kinds of form components, not just select boxes and radio buttons.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter05>.

Choosing a value from a select box

Let's start by creating a component for booking new appointments, named `AppointmentForm`.

The first field is a select box for choosing which service the customer requires: cut, color, blow-dry, and so on. Let's create that now:

1. Create a new file, `test/AppointmentForm.test.js`, with the following test and setup:

```
import React from "react";
import {
  initializeReactContainer,
  render,
  field,
  form,
} from "./reactTestExtensions";
import { AppointmentForm } from "../src/AppointmentForm";

describe("AppointmentForm", () => {
  beforeEach(() => {
    initializeReactContainer();
  });

  it("renders a form", () => {
    render(<AppointmentForm />);
    expect(form()).not.toBeNull();
  });
});
```

2. Make this test pass by implementing and creating a new file, `src/AppointmentForm.js`, as shown here:

```
import React from "react";

export const AppointmentForm = () => <form />;
```

3. Create a nested `describe` block for the service field. We'll jump to this right away because we know this form will have multiple fields:

```
describe("service field", () => {
});
```

4. Add the following test to the describe block:

```
it("renders as a select box", () => {
  render(<AppointmentForm />);
  expect(field("service")).not.toBeNull();
  expect(field("service").tagName).toEqual("SELECT");
}) ;
```

5. To make this test pass, modify the `AppointmentForm` component, as follows:

```
export const AppointmentForm = () => (
  <form
    <select name="service" />
  </form>
) ;
```

6. Run the tests and ensure they are all passing.

With that, we've done the basic scaffolding for the new select box field so that it's ready to be populated with `option` elements.

Providing select box options

Our salon provides a whole range of salon services. We should ensure that they are all listed in the app. We could start our test by defining our expectations, like this:

```
it("lists all salon services", () => {
  const selectableServices = [
    "Cut",
    "Blow-dry",
    "Cut & color",
    "Beard trim",
    "Cut & beard trim",
    "Extensions"
  ];
  ...
}) ;
```

If we do this, we'll end up repeating the same array of services in our test code and our production code. We can avoid that repetition by focusing our unit tests on the *behavior* of the select box rather than the static data that populates it: what should the select box *do*?

As it turns out, we can specify the functionality of our select box with just *two* items in our array. There's another good reason for keeping it to just two, which is that keeping the array brief helps us focus the test on what's important: the behavior, not the data.

That leaves the question, how do we use only two items in our test when we need six items for the production code?

We'll do this by introducing a new prop, `selectableServices`, to `AppointmentForm`. Our tests can choose to specify a value if they need to. In our production code, we can specify a value for the component's `defaultProps`.

`defaultProps` is a nifty mechanism that React offers for setting default prop values that will be used when required props are not explicitly provided.

For our tests that *don't* care about the select box values, we can avoid passing the prop and ignore it entirely in the test. For the tests that *do* care, we can provide a short, two-item array for our tests.

How do we verify the real select box values?

Testing static data does happen, just not within our unit tests. One place this can be tested is within acceptance tests, which we'll look at in *Part 4, Behavior-Driven Development with Cucumber*.

We'll start with a test to ensure the first value is a blank entry. This is the value that's initially selected when the user creates a new appointment: no option is selected. Let's write that test now:

1. Add the following test at the end of the `AppointmentForm` test suite. It specifies that the very first item in the select box is blank, meaning the user is not automatically assigned a choice from our list of services:

```
it("has a blank value as the first value", () => {
  render(<AppointmentForm />);
  const firstOption = field("service").childNodes[0];
  expect(firstOption.value).toEqual("");
});
```

2. Make that pass by adding a blank option element to the existing `select` element:

```
export const AppointmentForm = () => (
  <form>
    <select name="service">
      <option />
    </select>
  </form>
);
```

- Back in your tests, add this new helper just after the `beforeEach` block. We'll make use of it in our next test to build an array of all the labels of the select box options:

```
const labelsOfAllOptions = (element) =>
  Array.from(
    element.childNodes,
    (node) => node.textContent
  );
```

- Add the following test. This makes use of a new prop, `selectableServices`, which is simply the array of available options:

```
it("lists all salon services", () => {
  const services = ["Cut", "Blow-dry"];

  render(
    <AppointmentForm selectableServices={services} />
  );

  expect(
    labelsOfAllOptions(field("service"))
  ).toEqual(expect.arrayContaining(services));
});
```

Choosing test data

I've used "real" data for my expected services: Cut and Blow-dry. It's also fine to use non-real names such as Service A and Service B. Often, that can be more descriptive. Both are valid approaches.

- Let's make this pass. Change the component definition, as follows:

```
export const AppointmentForm = ({
  selectableServices
}) => (
  <form>
    <select name="service">
      <option />
      {selectableServices.map(s => (
        <option key={s}>{s}</option>
      ))}
    </select>
  </form>
);
```

```
        })
      </select>
    </form>
  ) ;
```

6. Check that the latest test now passes. However, you will see that our earlier tests break because of the introduction of the new prop.
7. We can make these tests pass again using `defaultProps`. Just below the definition of the `AppointmentForm` function in `src/AppointmentForm.js`, add the following:

```
AppointmentForm.defaultProps = {
  selectableServices: [
    "Cut",
    "Blow-dry",
    "Cut & color",
    "Beard trim",
    "Cut & beard trim",
    "Extensions",
  ]
} ;
```

8. Run your tests and verify they are passing.

That's all there is to it. With that, we've learned how to define the behavior of our component using a short two-item array and saved the real data for `defaultProps`.

Preselecting a value

Let's ensure that our component preselects the value that has already been saved if we're editing an existing appointment:

1. Define a `findOption` arrow function at the top of the `describe` block. This function searches the DOM tree for a particular text node:

```
const findOption = (selectBox, textContent) => {
  const options = Array.from(selectBox.childNodes);
  return options.find(
    option => option.textContent === textContent
  );
} ;
```

2. In our next test, we can find that node and then check that it is selected:

```
it("pre-selects the existing value", () => {
  const services = ["Cut", "Blow-dry"];
  const appointment = { service: "Blow-dry" };
  render(
    <AppointmentForm
      selectableServices={services}
      original={appointment}
    />
  );
  const option = findOption(
    field("service"),
    "Blow-dry"
  );
  expect(option.selected).toBe(true);
}) ;
```

3. To make this pass, set the value property on the root `select` tag:

```
<select
  name="service"
  value={original.service}
  readOnly>
```

Accessible rich internet applications (ARIA) labels

If you have experience with building React applications, you may be expecting to set the `aria-label` property on the `select` element. However, one of this chapter's *Exercises* is to add a `label` element for this `select` box that will ensure an ARIA label is set implicitly by the browser.

4. You'll need to change your component props so that it includes the new `service` prop:

```
export const AppointmentForm = ({
  original,
  selectableServices
}) =>
```

5. Run your tests. Although this test is now passing, you'll find the previous tests are failing because the original prop has not been set. To fix them, first, define a new constant, blankAppointment, just above your `beforeEach` block. We'll use this in each of the failing tests:

```
const blankAppointment = {  
  service: "",  
};
```

6. Update your previous tests so that they use this new constant as the value for the `original` prop. For example, the very first test for `AppointmentForm` will look as follows:

```
it("renders a form", () => {  
  render(  
    <AppointmentForm original={blankAppointment} />  
  );  
  expect(form()).not.toBeNull();  
});
```

7. Run the tests again with `npm test`; all your tests should be passing. (If they aren't, go back and check that you've got an `original` prop value for each test.)
8. Let's finish with a small bit of refactoring. Your last two tests both have the same definition for services. Pull that out of each test, placing it above the definition of `blankAppointment`. Make sure that you delete that line from both tests:

```
describe("AppointmentForm", () => {  
  const blankAppointment = {  
    service: "",  
  };  
  const services = ["Cut", "Blow-dry"];  
  ...  
});
```

That completes this test, but there is still more functionality to add if we want a fully functional select box. Completing those tests is left as one of the *Exercises* at the end of this chapter. They work the same as the tests for the text boxes in `CustomerForm`.

If you compare our select box tests to those of the text box, you will see that it's a similar pattern but with a couple of additional techniques: we used `defaultProps` to separate the definition of production data from test behavior, and we defined a couple of localized helper methods, `labelsOfAllOptions` and `findOption`, to help keep our tests short.

Let's move on to the next item in our form: the time of the appointment.

Constructing a calendar view

In this section, we'll learn how to use our existing helpers, such as `element` and `elements`, mixed with CSS selectors, to select specific elements we're interested in within our HTML layout.

But first, let's start with some planning.

We'd like `AppointmentForm` to display available time slots over the next 7 days as a grid, with columns representing days and rows representing 30-minute time slots, just like a standard calendar view. The user will be able to quickly find a time slot that works for them and then select the right radio button before submitting the form:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
09:00	<input type="radio"/>				<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
09:30	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>
10:00			<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
10:30		<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>		<input type="radio"/>	
11:00							
11:30							
12:00							
12:30							
13:00							
13:30							
14:00							
14:30							
15:00							
15:30							
16:00							
16:30							
17:00							
17:30							
18:00							
18:30							

Figure 5.1 – The visual design of our calendar view

Here's an example of the HTML structure that we're aiming to build. We can use this as a guide as we write out our React component:

```
<table id="time-slots">
  <thead>
    <tr>
      <th></th>
      <th>Oct 11</th>
      <th>Oct 12</th>
```

```
<th>Oct 13</th>
</tr>
</thead>
<tbody>
<tr>
<th>9:00</th>
<td>
<input type="option" name="timeSlot" value="..." />
</td>
</tr>
<!-- ... two more cells ... -->
</tbody>
</table>
```

In the next few sections, we'll test-drive the `table` element itself, then build a header column for times of the day, and then a header for days of the week.

Adding the table

Let's begin by building `table` itself:

1. Create a nested `describe` block with a new test at the bottom of `test/AppointmentForm.test.js`:

```
describe("time slot table", () => {
  it("renders a table for time slots with an id", () => {
    render(
      <AppointmentForm original={blankAppointment} />
    );
    expect(
      element("table#time-slots")
    ).not.toBeNull();
  });
});
```

2. You'll need to pull the `element` helper into your imports:

```
import {
  initializeReactContainer,
  render,
```

```
field,  
form,  
element,  
} from "./reactTestExtensions";
```

3. To make that pass, move to `src/AppointmentForm.js` and define a new `TimeSlotTable` component, above the definition of `AppointmentForm`. We don't need to mark this one as an export as it will only be referenced by `AppointmentForm`:

```
const TimeSlotTable = () => <table id="time-slots" />;
```

Why add an ID?

The ID is important because that's what the application's CSS uses to find the `table` element. Although it's not covered in this book, if you're using CSS and it defines selectors based on element IDs, then you should treat those IDs as a kind of technical specification that your code must satisfy. That's why we write unit tests for them.

4. Add this component to your `AppointmentForm` JSX, right at the bottom, just below the `select` tag:

```
<form>  
  ...  
  <TimeSlotTable />  
</form>;
```

Run the tests and verify that they are all passing.

That's all there is to the `table` element. Now, let's get some data into the first column.

Adding a header column

For the next test, we'll test the left-hand header column that displays a list of times. We'll introduce two new props, `salonOpensAt` and `salonClosesAt`, which inform the component of which time to show each day. Follow these steps:

1. Add the following test:

```
it("renders a time slot for every half an hour between  
open and close times", () => {  
  render(  
    <AppointmentForm  
      original={blankAppointment}
```

```

        salonOpensAt={9}
        salonClosesAt={11}
    />
);
const timesOfDayHeadings = elements("tbody >* th");
expect(timesOfDayHeadings[0]).toContainText(
    "09:00"
);
expect(timesOfDayHeadings[1]).toContainText(
    "09:30"
);
expect(timesOfDayHeadings[3]).toContainText(
    "10:30"
);
}) ;
});

```

Asserting on array patterns

In this example, we are checking `textContent` on *three* array entries, even though there are four entries in the array.

Properties that are the same for all array entries only need to be tested on one entry. Properties that vary per entry, such as `textContent`, need to be tested on two or three entries, depending on how many you need to test a pattern.

For this test, I want to test that it starts and ends at the right time and that each time slot increases by 30 minutes. I can do that with assertions on array entries 0, 1, and 3.

This test “breaks” our rule of one expectation per test. However, in this scenario, I think it’s okay. An alternative approach might be to use the `textOf` helper instead.

2. You’ll need to pull the `elements` helper into your imports:

```

import {
    initializeReactContainer,
    render,
    field,
    form,
    element,
    elements,
} from "./reactTestExtensions";

```

3. To make this pass, add the following functions above the `TimeSlotTable` component. They calculate the list of daily time slots:

```
const timeIncrements = (
  numTimes,
  startTime,
  increment
) =>
  Array(numTimes)
    .fill([startTime])
    .reduce((acc, _, i) =>
      acc.concat([startTime + i * increment]))
);

const dailyTimeSlots = (
  salonOpensAt,
  salonClosesAt
) => {
  const totalSlots =
    (salonClosesAt - salonOpensAt) * 2;
  const startTime = new Date()
    .setHours(salonOpensAt, 0, 0, 0);
  const increment = 30 * 60 * 1000;
  return timeIncrements(
    totalSlots,
    startTime,
    increment
  );
};
```

4. Define the `toTimeValue` function, as follows:

```
const toTimeValue = timestamp =>
  new Date(timestamp).toTimeString().substring(0, 5);
```

5. Now, you can make use of those two functions. Update `TimeSlotTable` so that it reads as follows:

```
const TimeSlotTable = ({
```

```
    salonOpensAt,
    salonClosesAt
}) => {
  const timeSlots = dailyTimeSlots(
    salonOpensAt,
    salonClosesAt);
  return (
    <table id="time-slots">
      <tbody>
        {timeSlots.map(timeSlot => (
          <tr key={timeSlot}>
            <th>{toTimeValue(timeSlot)}</th>
          </tr>
        ))}
      </tbody>
    </table>
  );
},
```

6. In the JSX for AppointmentForm, pass the `salonOpensAt` and `salonClosesAt` props to TimeSlotTable:

```
export const AppointmentForm = ({
  original,
  selectableServices,
  service,
  salonOpensAt,
  salonClosesAt
}) => (
  <form>
    ...
    <TimeSlotTable
      salonOpensAt={salonOpensAt}
      salonClosesAt={salonClosesAt} />
  </form>
);
```

7. Fill in `defaultProps` for both `salonOpensAt` and `salonsCloseAt`:

```
AppointmentForm.defaultProps = {  
  salonOpensAt: 9,  
  salonClosesAt: 19,  
  selectableServices: [ ... ]  
};
```

8. Run the tests and make sure everything is passing.

That's all there is to adding the left-hand side column of headings.

Adding a header row

Now, what about the column headings? In this section, we'll create a new top row that contains these cells, making sure to leave an empty cell in the top-left corner, since the left column contains the time headings and not data. Follow these steps:

1. Add the following test:

```
it("renders an empty cell at the start of the header  
row", () =>  
  render(  
    <AppointmentForm original={blankAppointment} />  
  );  
  const headerRow = element("thead > tr");  
  expect(headerRow.firstChild).toContainText("");  
);
```

2. Modify the table JSX so that it includes a new table row:

```
<table id="time-slots">  
  <thead>  
    <tr>  
      <th />  
    </tr>  
  </thead>  
  <tbody>  
    ...  
  </tbody>  
</table>
```

3. For the rest of the header row, we'll show the 7 days starting from today. `AppointmentForm` will need to take a new prop, `today`, which is the first day to display within the table. The value that's assigned to that prop is stored in a variable named `specificDate`. This name has been chosen to highlight the fact that this chosen date affects the rendered day output, for example, "Sat 01":

```
it("renders a week of available dates", () => {
  const specificDate = new Date(2018, 11, 1);
  render(
    <AppointmentForm
      original={blankAppointment}
      today={specificDate}
    />
  );
  const dates = elements(
    "thead >* th:not(:first-child)"
  );
  expect(dates).toHaveLength(7);
  expect(dates[0]).toContainText("Sat 01");
  expect(dates[1]).toContainText("Sun 02");
  expect(dates[6]).toContainText("Fri 07");
});
```

Why pass a date into the component?

When you're testing a component that deals with dates and times, you almost always want a way to control the time values that the component will see, as we have in this test. You'll rarely want to just use the real-world time because that can cause intermittent failures in the future. For example, your test may assume that a month has at least 30 days in the year, which is only true for 11 out of 12 months. It's better to fix the month to a specific month rather than have an unexpected failure when February comes around.

For an in-depth discussion on this topic, take a look at <https://reacttdd.com/controlling-time>.

4. To make that pass, first, create a function that lists the 7 days we're after, in the same way we did with time slots. You can place this just after the `toTimeValue` function:

```
const weeklyDateValues = (startDate) => {
  const midnight = startDate.setHours(0, 0, 0, 0);
  const increment = 24 * 60 * 60 * 1000;
```

```
    return timeIncrements(7, midnight, increment);
}
```

5. Define the `toShortDate` function, which formats our date as a short string:

```
const toShortDate = (timestamp) => {
  const [day, , dayOfMonth] = new Date(timestamp)
    .toString()
    .split(" ");
  return `${day} ${dayOfMonth}`;
};
```

6. Modify `TimeSlotTable` so that it takes the new `today` prop and uses the two new functions:

```
const TimeSlotTable = ({
  salonOpensAt,
  salonClosesAt,
  today
}) => {
  const dates = weeklyDateValues(today);
  ...
  return (
    <table id="time-slots">
      <thead>
        <tr>
          <th />
          {dates.map(d => (
            <th key={d}>{toShortDate(d)}</th>
          )));
        </tr>
      </thead>
      ...
    </table>
  );
};
```

7. Within `AppointmentForm`, pass the `today` prop from `AppointmentForm` into `TimeSlotTable`:

```
export const AppointmentForm = ({  
  original,  
  selectableServices,  
  service,  
  salonOpensAt,  
  salonClosesAt,  
  today  
}) => {  
  ...  
  return <form>  
    <TimeSlotTable  
      ...  
      salonOpensAt={salonOpensAt}  
      salonClosesAt={salonClosesAt}  
      today={today}  
    />  
  </form>;  
};
```

8. Finally, add a `defaultProp` for `today`. Set it to the current date by calling the `Date` constructor:

```
AppointmentForm.defaultProps = {  
  today: new Date(),  
  ...  
}
```

9. Run the tests. They should be all green.

With that, we're done with our table layout. You've seen how to write tests that specify the table structure itself and fill in both a header column and a header row. In the next section, we'll fill in the table cells with radio buttons.

Test-driving radio button groups

Now that we have our table with headings in place, it's time to add radio buttons to each of the table cells. Not all cells will have radio buttons – only those that represent an available time slot will have a radio button.

This means we'll need to pass in another new prop to `AppointmentForm` that will help us determine which time slots to show. This prop is `availableTimeSlots`, which is an array of objects that list times that are still available. Follow these steps:

1. Add the following test, which establishes a value for the `availableTimeSlots` prop and then checks that radio buttons have been rendered for each of those slots:

```
it("renders radio buttons in the correct table cell positions", () => {
  const oneDayInMs = 24 * 60 * 60 * 1000;
  const today = new Date();
  const tomorrow = new Date(
    today.getTime() + oneDayInMs
  );
  const availableTimeSlots = [
    { startsAt: today.setHours(9, 0, 0, 0) },
    { startsAt: today.setHours(9, 30, 0, 0) },
    { startsAt: tomorrow.setHours(9, 30, 0, 0) },
  ];

  render(
    <AppointmentForm
      original={blankAppointment}
      availableTimeSlots={availableTimeSlots}
      today={today}
    />
  );
  expect(cellsWithRadioButtons()).toEqual([0, 7, 8]);
});
```

2. Notice that this test uses a `cellsWithRadioButtons` helper, which we need to define now. You can place this just above the test; there's no need to move it to the extension's module since it's specific to this one component:

```
const cellsWithRadioButtons = () =>
  elements("input[type=radio]").map((el) =>
    elements("td").indexOf(el.parentNode)
  );
```

3. This test checks that there are radio buttons in the first two time slots for today. These will be in cells 0 and 7 since `elements` returns matching elements in page order. We can make this test pass very simply by adding the following to our `AppointmentForm` render method, just below `th` within each `tr`:

```
{timeSlots.map(timeSlot =>
  <tr key={timeSlot}>
    <th>{toTimeValue(timeSlot)}</th>
    {dates.map(date => (
      <td key={date}>
        <input type="radio" />
      </td>
    )));
  </tr>
)}
```

At this point, your test will be passing.

We didn't need to use `availableTimeSlots` in our production code, even though our tests require it! Instead, we just put a radio button in *every* cell! This is obviously "broken." However, if you think back to our rule of only ever implementing the simplest thing that will make the test pass, then it makes sense. What we need now is another test to prove the opposite – that certain radio buttons do *not* exist, given `availableTimeSlots`.

Hiding input controls

How can we get to the right implementation? We can do this by testing that having no available time slots renders no radio buttons at all:

1. Add the following test:

```
it("does not render radio buttons for unavailable time
slots", () => {
  render(
    <AppointmentForm
      original={blankAppointment}
      availableTimeSlots={[ ]}
    />
  );
  expect(
    elements("input [type=radio]")
  ).toHaveLength(0);
})
```

```
    ).toHaveLength(0) ;  
});
```

2. To make that pass, first, move to `src/AppointmentForm.js` and define the `mergeDateAndTime` function above the `TimeSlotTable` component. This takes the date from a column header, along with a time from a row header, and converts them into a timestamp that we can use to compare against the `startsAt` fields in `availableTimeSlots`:

```
const mergeDateAndTime = (date, timeSlot) => {  
  const time = new Date(timeSlot);  
  return new Date(date).setHours(  
    time.getHours(),  
    time.getMinutes(),  
    time.getSeconds(),  
    time.getMilliseconds()  
  );  
};
```

3. Update `TimeSlotTable` so that it takes the new `availableTimeSlots` prop:

```
const TimeSlotTable = ({  
  salonOpensAt,  
  salonClosesAt,  
  today,  
  availableTimeSlots  
}) => {  
  ...  
};
```

4. Replace the existing radio button element in `TimeSlotTable` with a JSX conditional:

```
{dates.map(date =>  
  <td key={date}>  
    {availableTimeSlots.some(availableTimeSlot =>  
      availableTimeSlot.startsAt ===  
      mergeDateAndTime(date, timeSlot)  
    )  
    ? <input type="radio" />  
    : null  
  }
```

```
        </td>
    )}
```

5. Also, update AppointmentForm so that it takes the new prop, and then pass it through to TimeSlotTable:

```
export const AppointmentForm = ({  
  original,  
  selectableServices,  
  service,  
  salonOpensAt,  
  salonClosesAt,  
  today,  
  availableTimeSlots  
) => {  
  ...  
  return (  
    <form>  
      ...  
      <TimeSlotTable  
        salonOpensAt={salonOpensAt}  
        salonClosesAt={salonClosesAt}  
        today={today}  
        availableTimeSlots={availableTimeSlots} />  
    </form>  
  );  
};
```

6. Although your test will now be passing, the rest will be failing: they need a value for the availableTimeSlots prop. To do that, first, add the following definitions to the top of AppointmentForm:

```
describe("AppointmentForm", () => {  
  const today = new Date();  
  const availableTimeSlots = [  
    { startsAt: today.setHours(9, 0, 0, 0) },  
    { startsAt: today.setHours(9, 30, 0, 0) },  
  ];
```

7. Go through each test and update each call to render to specify an `availableTimeSlots` prop with a value of `availableTimeSlots`. For example, the first test should have the following render call:

```
render(  
  <AppointmentForm  
    original={blankAppointment}  
    availableTimeSlots={availableTimeSlots}  
  />  
) ;
```

Handling sensible defaults for props

Adding a default value for a new prop in every single test is no one's idea of fun. Later in this chapter you'll learn how to avoid prop explosion in your tests by introducing a `testProps` object to group sensible default prop values.

8. Let's continue with the next test. We must ensure each radio button has the correct value. We'll use the `startsAt` value for each radio button's value. Radio button values are strings, but the appointment object property, `startsAt`, is a number. We'll use a standard library function, `parseInt`, to convert the button value back into a usable number:

```
it("sets radio button values to the startsAt value of the  
corresponding appointment", () => {  
  render(  
    <AppointmentForm  
      original={blankAppointment}  
      availableTimeSlots={availableTimeSlots}  
      today={today}  
    />  
  ) ;  
  const allRadioValues = elements(  
    "input[type=radio]"  
  ) .map(({ value }) => parseInt(value));  
  const allSlotTimes = availableTimeSlots.map(  
    ({ startsAt }) => startsAt  
  );  
  expect(allRadioValues) .toEqual(allSlotTimes);  
}) ;
```

Defining constants within tests

Sometimes, it's preferable to keep constants within a test rather than pulling them out as helpers. In this case, these helpers are only used by this one test and are very specific in what they do. Keeping them inline helps you understand what the functions are doing without having to search through the file for the function definitions.

9. In your production code, pull out the ternary that contained the original call to `mergeDateAndTime` into a new component. Take care to add the new name and `value` attributes to the `input` element:

```
const RadioButtonIfAvailable = ({  
  availableTimeSlots,  
  date,  
  timeSlot,  
}) => {  
  const startsAt = mergeDateAndTime(date, timeSlot);  
  
  if (  
    availableTimeSlots.some(  
      (timeSlot) => timeSlot.startsAt === startsAt  
    )  
  ) {  
    return (  
      <input  
        name="startsAt"  
        type="radio"  
        value={startsAt}  
      />  
    );  
  }  
  return null;  
};
```

The `name` property

Radio buttons with the same `name` attribute are part of the same group. Clicking one radio button will check that button and uncheck all others in the group.

10. You can now use this within `TimeSlotTable`, replacing the existing ternary with an instance of this functional component. After this, your tests should be passing:

```
{dates.map(date =>
  <td key={date}>
    <RadioButtonIfAvailable
      availableTimeSlots={availableTimeSlots}
      date={date}
      timeSlot={timeSlot}
    />
  </td>
) }
```

Now that you've got the radio buttons displaying correctly, it's time to give them some behavior.

Selecting a radio button in a group

Let's see how we can use the `checked` property on the input element to ensure we set the right initial value for our radio button.

For this, we'll use a helper called `startsAtField` that takes an index and returns the radio button at that position. To do that, the radio buttons must all be given the same name. This joins the radio button into a group, which means only one can be selected at a time. Follow these steps:

1. Start by adding the `startsAtField` helper at the top of the time slot table's `describe` block:

```
const startsAtField = (index) =>
  elements("input [name=startsAt] ")[index];
```

2. Add the following test. It passes in an existing appointment with a `startsAt` value set to the second item in the `availableTimeSlots` list. Choosing the second item rather than the first isn't strictly necessary (since the default will be for *all* radio buttons to be unchecked), but it can help highlight to future maintainers that a specific value has been chosen and is being checked:

```
it("pre-selects the existing value", () => {
  const appointment = {
    startsAt: availableTimeSlots[1].startsAt,
  };
  render(
    <AppointmentForm
      original={appointment}
      availableTimeSlots={availableTimeSlots}
```

```
        today={today}
      />
    ) ;
  expect(startsAtField(1).checked).toEqual(true);
}) ;
```

3. To make that pass, first, add a new `checkedTimeSlot` prop to `TimeSlotTable` that has the value of the original `startsAt` value:

```
<TimeSlotTable
  salonOpensAt={salonOpensAt}
  salonClosesAt={salonClosesAt}
  today={today}
  availableTimeSlots={availableTimeSlots}
  checkedTimeSlot={appointment.startsAt}
/>
```

4. Update `TimeSlotTable` so that it makes use of this new prop, passing it through to `RadioButtonIfAvailable`:

```
const TimeSlotTable = ({
  ...
  checkedTimeSlot,
}) => {
  ...
  <RadioButtonIfAvailable
    availableTimeSlots={availableTimeSlots}
    date={date}
    timeSlot={timeSlot}
    checkedTimeSlot={checkedTimeSlot}
  />
  ...
};
```

5. Now, you can make use of that in `RadioButtonIfAvailable`, setting the `isChecked` prop on the input element, as shown here. After this change, your test should be passing:

```
const RadioButtonIfAvailable = ({
  ...
  checkedTimeSlot,
```

```
}) => {
  const startsAt = mergeDateAndTime(date, timeSlot);
  if (
    availableTimeSlots.some(
      (a) => a.startsAt === startsAt
    )
  ) {
    const isChecked = startsAt === checkedTimeSlot;
    return (
      <input
        name="startsAt"
        type="radio"
        value={startsAt}
        checked={isChecked}
      />
    );
  }
  return null;
};
```

That's it for setting the initial value. Next, we'll hook up the component with the `onChange` behavior.

Handling field changes through a component hierarchy

Throughout this chapter, we have slowly built up a component hierarchy: `AppointmentForm` renders a `TimeSlotTable` component that renders a bunch of `RadioButtonIfAvailable` components that may (or may not) render the radio button input elements.

The final challenge involves how to take an `onChange` event from the input element and pass it back up to `AppointmentForm`, which will control the appointment object.

The code in this section will make use of the `useCallback` hook. This is a form of performance optimization: we can't write a test to specify that this behavior exists. A good rule of thumb is that if you're passing functions through as props, then you should consider using `useCallback`.

The `useCallback` hook

The `useCallback` hook returns a **memoized** callback. This means you always get the same reference back each time it's called, rather than a new constant with a new reference. Without this, child components that are passed the callback as a prop (such as `TimeSlotTable`) would re-render each time the parent re-renders, because the different reference would cause it to believe that a re-render was required.

Event handlers on `input` elements don't need to use `useCallback` because event handler props are handled centrally; changes to those props do not require re-renders.

The second parameter to `useCallback` is the set of dependencies that will cause `useCallback` to update. In this case, it's `[]`, an empty array, because it isn't dependent on any props or other functions that may change. Parameters to the function such as `target` don't count, and `setAppointment` is a function that is guaranteed to remain constant across re-renders.

See the *Further reading* section at the end of this chapter for a link to more information on `useCallback`.

Since we haven't done any work on submitting `AppointmentForm` yet, we need to start there. Let's add a test for the form's submit button:

1. Add the following test to your `AppointmentForm` test suite, which tests for the presence of a submit button. This can go at the top of the test suite, just underneath the `renders a form` test:

```
it("renders a submit button", () => {
  render(
    <AppointmentForm original={blankAppointment} />
  );
  expect(submitButton()).not.toBeNull();
});
```

2. You'll also need to import the `submitButton` helper into your tests:

```
import {
  initializeReactContainer,
  render,
  field,
  form,
  element,
  elements,
  submitButton,
} from "../reactTestExtensions";
```

3. To make that pass, add the button at the bottom of your `AppointmentForm`:

```
<form>
  ...
<input type="submit" value="Add" />
```

```
</form>
```

4. For the next test, let's submit the form and check that we get the original `startsAt` value submitted back. We'll use the same `expect.hasAssertions` technique that we saw in the previous chapter. The test verifies that the `onSubmit` prop was called with the original, unchanged `startsAt` value:

```
it("saves existing value when submitted", () => {
  expect.hasAssertions();
  const appointment = {
    startsAt: availableTimeSlots[1].startsAt,
  };
  render(
    <AppointmentForm
      original={appointment}
      availableTimeSlots={availableTimeSlots}
      today={today}
      onSubmit={({ startsAt }) =>
        expect(startsAt).toEqual(
          availableTimeSlots[1].startsAt
        )
      }
    />
  );
  click(submitButton());
});
```

5. Since this test uses the `click` helper, you'll need to import it:

```
import {
  initializeReactContainer,
  render,
  field,
  form,
  element,
  elements,
  submitButton,
  click,
} from "./reactTestExtensions";
```

6. For this test, all we need is to get the form's `onSubmit` event handler in place. At this stage, it will simply submit the `original` object without any registered changes. Update the `AppointmentForm` component, as shown here:

```
export const AppointmentForm = ({  
  ...,  
  onsubmit,  
}) => {  
  const handleSubmit = (event) => {  
    event.preventDefault();  
    onsubmit(original);  
  };  
  
  return (  
    <form onSubmit={handleSubmit}>  
      ...  
    </form>  
  );  
};
```

7. With that test passing, let's add the final test. This uses a `click` action rather than `change`, which we used for the text box and select box. We click on the desired radio button just as a user would:

```
it("saves new value when submitted", () => {  
  expect.hasAssertions();  
  const appointment = {  
    startsAt: availableTimeSlots[0].startsAt,  
  };  
  
  render(  
    <AppointmentForm  
      original={appointment}  
      availableTimeSlots={availableTimeSlots}  
      today={today}  
      onSubmit={({ startsAt }) =>  
        expect(startsAt).toEqual(  
          availableTimeSlots[1].startsAt
```

```
        )
    }
/>
);
click(startsAtField(1));
click(submitButton());
}) ;
```

- Now, the fun begins. Let's work from the top down: we'll start by defining a new appointment state object, which we'll then use in a new event handler that modifies the current appointment when a radio button is clicked. Move to `src/AppointmentForm.js` and update your React import so that it reads as follows:

```
import React, { useState, useCallback } from "react";
```

- Introduce a new appointment state object and update your `checkedTimeSlot` prop to use this object, rather than the `original` prop value:

```
export const AppointmentForm = ({
  ...
}) => {
  const [appointment, setAppointment] =
    useState(original);
  ...
  return (
    <form>
      ...
      <TimeSlotTable
        ...
        checkedTimeSlot={appointment.startsAt}
      />
      ...
    </form>
  );
};
```

10. Update the `handleSubmit` function so that it uses `appointment` rather than `original`:

```
const handleSubmit = (event) => {
  event.preventDefault();
  onSubmit(appointment);
};
```

The call to preventDefault

I'm avoiding writing the test for `preventDefault` since we've covered it previously. In a real application, I would almost certainly add that test again.

11. Now, it's time for the new event handler. This is the one that makes use of `useCallback` so that we can safely pass it through to `TimeSlotTable` and beyond. Add the following definition just below the `useState` call you added in the previous step. The handler uses `parseInt` to convert between our radio button's string value and the numeric timestamp value we'll be storing:

```
const handleStartsAtChange = useCallback(
  ({ target: { value } }) =>
    setAppointment((appointment) => ({
      ...appointment,
      startsAt: parseInt(value),
    })),
  []
);
```

12. We've got to weave the event handler through to the `input` element, just like we did with `checkedTimeSlot`. Start by passing it into `TimeSlotTable`:

```
<TimeSlotTable
  salonOpensAt={salonOpensAt}
  salonClosesAt={salonClosesAt}
  today={today}
  availableTimeSlots={availableTimeSlots}
  checkedTimeSlot={appointment.startsAt}
  handleChange={handleStartsAtChange}
/>
```

13. Then, update `TimeSlotTable`, taking that prop and passing it through to `RadioButtonIfAvailable`:

```
const TimeSlotTable = ({  
  ...,  
  handleChange,  
}) => {  
  ...,  
  <RadioButtonIfAvailable  
    availableTimeSlots={availableTimeSlots}  
    date={date}  
    timeSlot={timeSlot}  
    checkedTimeSlot={checkedTimeSlot}  
    handleChange={handleChange}  
  />  
  ...  
};
```

14. Finally, in `RadioButtonIfAvailable`, remove the `readOnly` property on the input field and set `onChange` in its place:

```
const RadioButtonIfAvailable = ({  
  availableTimeSlots,  
  date,  
  timeSlot,  
  checkedTimeSlot,  
  handleChange  
) => {  
  ...  
  return (  
    <input  
      name="startsAt"  
      type="radio"  
      value={startsAt}  
      checked={isChecked}  
      onChange={handleChange}  
    />  
  );  
  ...  
};
```

At this point, your test should pass, and your time slot table should be fully functional.

This section has covered a great deal of code: conditionally rendering `input` elements, as well as details of radio button elements, such as giving a group name and using the `onChecked` prop, and then passing its `onChange` event through a hierarchy of components.

This is a good moment to manually test what you've built. You'll need to update `src/index.js` so that it loads `AppointmentForm`, together with sample data. These changes are included in the `Chapter05/Complete` directory:

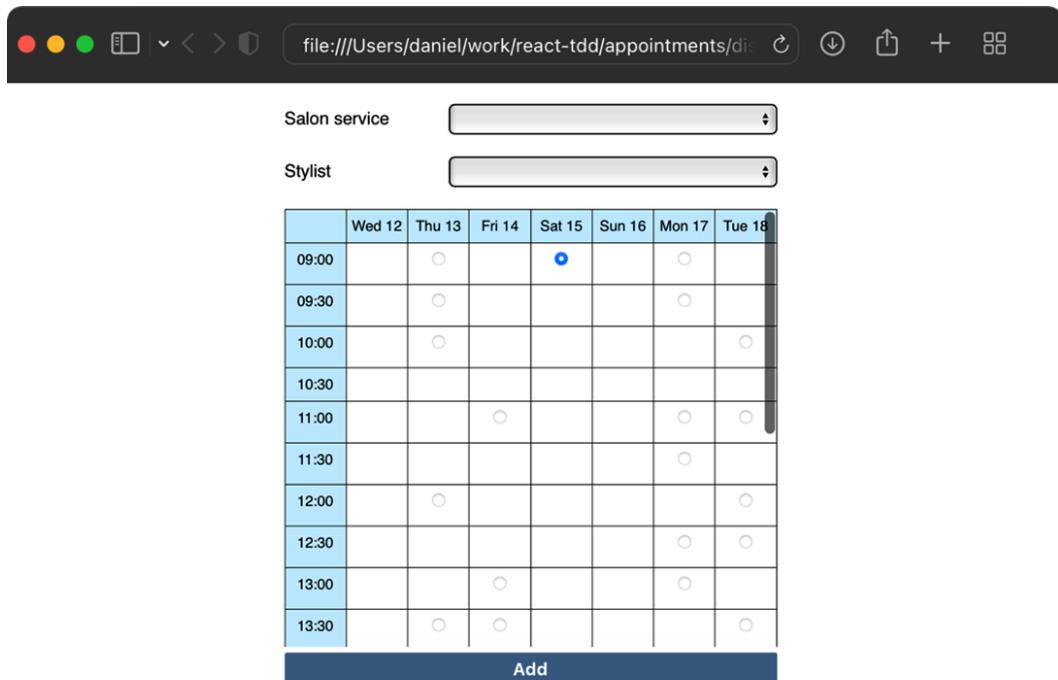


Figure 5.2 – `AppointmentForm` on show

You've now completed the work required to build the radio button table. Now it's time to refactor.

Reducing effort when constructing components

Let's look at a couple of simple ways to reduce the amount of time and code needed for test suites like the one we've just built: first, extracting builder functions, and second, extracting objects to store sensible defaults for our component props.

Extracting test data builders for time and date functions

You've already seen how we can extract reusable functions into namespaces of their own, such as the `render`, `click`, and `element` DOM functions. A special case of this is the **builder** function, which constructs objects that you'll use in the **Arrange** and **Act** phases of your test.

The purpose of these functions is not just to remove duplication but also for simplification and to aid with comprehension.

We already have one candidate in our test suite, which is the following code:

```
const today = new Date();
today.setHours(9, 0, 0, 0);
```

We'll update our test suite so that it uses a builder function called `todayAt`, which will save a bit of typing:

```
todayAt(9);
```

We'll also extract the `today` value as a constant as we'll also make use of that.

Builders for domain objects

Most often, you'll create builder functions for the domain objects in your code base. In our case, that would be `customer` or `appointment` objects, or even the time slot objects with the single `startsAt` field. Our code base hasn't progressed enough to warrant this, so we'll start with builders for the `Date` objects that we're using. We'll write more builders later in this book.

Let's get started:

1. Create a new directory, `test/builders`. This is where all our `builder` functions will live.
2. Create a new file, `test/builders/time.js`. This is where we'll throw everything related to time.
3. Add the following constant to your new file:

```
export const today = new Date();
```

4. Back in `test/AppointmentForm.test.js`, add the following import, just below your other imports:

```
import { today } from "./builders/time";
```

5. Delete the definition of the `today` constant in the test suite.

6. In `test/builders/time.js`, add the following definition of `todayAt`. Notice that this does allow us to specify hours, minutes, seconds, and milliseconds if we so choose, but it has a default value of 0 for each if we don't specify them. We'll make use of this full form in one of the tests. We must also make a copy of the `today` constant by calling the `Date` constructor. This ensures that we don't unexpectedly modify the `today` constant on any callers of this function:

```
export const todayAt = (
  hours,
  minutes = 0,
  seconds = 0,
  milliseconds = 0
) =>
  new Date(today).setHours(
    hours,
    minutes,
    seconds,
    milliseconds
);
```

Immutability of builder functions

If your namespaces use shared constant values, like we're doing with `today` here, make sure your functions don't inadvertently mutate them.

7. Back in `test/AppointmentForm.test.js`, update your import so that it includes the new function:

```
import { today, todayAt } from "./builders/time";
```

8. Time for a search and replace! Find all occurrences of the following:

```
today.setHours(9, 0, 0, 0)
```

Replace it with the following:

```
todayAt(9)
```

9. Find all occurrences of the following:

```
today.setHours(9, 30, 0, 0)
```

Replace it with the following:

```
todayAt(9, 30)
```

10. Ensure your tests are still passing.
11. Move these lines from the test suite into `test/builders/time.js`:

```
const oneDayInMs = 24 * 60 * 60 * 1000;
const tomorrow = new Date(
  today.getTime() + oneDayInMs
);
```

12. Rather than use the `tomorrow` constant directly, let's write a `tomorrowAt` helper for that:

```
export const tomorrowAt = (
  hours,
  minutes = 0,
  seconds = 0,
  milliseconds = 0
) =>
  new Date(tomorrow).setHours(
    hours,
    minutes,
    seconds,
    milliseconds
);
```

13. Update your import so that it includes the new function:

```
import {
  today,
  todayAt,
  tomorrowAt
} from "./builders/time";
```

14. Delete the definitions of `oneDayInMs` and `tomorrow` from the test suite.
15. Find the following expression:

```
tomorrow.setHours(9, 30, 0, 0)
```

Replace it with the following code:

```
tomorrowAt(9, 30)
```

16. Run the tests again; they should be passing.

We'll make use of these helpers again in *Chapter 7, Testing useEffect and Mocking Components*. However, there's one more extraction we can do before we finish with this chapter.

Extracting a test props object

A test props object is an object that sets sensible defaults for props that you can use to reduce the size of your `render` statements. For example, look at the following render call:

```
render(  
  <AppointmentForm  
    original={blankAppointment}  
    availableTimeSlots={availableTimeSlots}  
    today={today}  
  />  
) ;
```

Depending on the test, some (or all) of these props may be irrelevant to the test. The `original` prop is necessary so that our `render` function doesn't blow up when rendering existing field values. But if our test is checking that we show a label on the page, we don't care about that – and that's one reason we created the `blankAppointment` constant. Similarly, `availableTimeSlots` and the `today` prop may not be relevant to a test.

Not only that, but often, our components can end up needing a whole lot of props that are necessary for a test to function. This can end up making your tests extremely verbose.

Too many props?

The technique you're about to see is one way of dealing with many required props. But having a lot of props (say, more than four or five) might be a hint that the design of your components can be improved. Can the props be joined into a complex type? Or should the component be split into two or more components?

This is another example of listening to your tests. If the tests are difficult to write, take a step back and look at your component design.

We can define an object named `testProps` that exists at the top of our `describe` block:

```
const testProps = {  
  original: { ... },  
  availableTimeSlots: [ ... ],  
  today: ...  
}
```

This can then be used in the `render` call, like this:

```
render(<AppointmentForm {...testProps} />);
```

If the test does depend on a prop, such as if its expectation mentions part of the `props` value, then you shouldn't rely on the hidden-away value in the `testProps` object. Those values are sensible defaults. The values in your test should be prominently displayed, as in this example:

```
const appointment = {
  ...blankAppointment,
  service: "Blow-dry"
};
render(
  <AppointmentForm {...testProps} original={appointment} />
);
const option = findOption(field("service"), "Blow-dry");
expect(option.selected).toBe(true);
```

Notice how the `original` prop is still included in the `render` call after `testProps`.

Sometimes, you'll want to explicitly include a prop, even if the value is the same as the `testProps` value. That's to highlight its use within the test. We'll see an example of that in this section.

When to use an explicit prop

As a rule of thumb, if the prop is used in your test assertions, or if the prop's value is crucial for the scenario the test is testing, then the prop should be included explicitly in the `render` call, even if its value is the same as the value defined in `testProps`.

Let's update the `AppointmentForm` test suite so that it uses a `testProps` object:

1. In your test suite, find the definitions for `services`, `availableTimeSlots`, and `blankAppointment`. These should be near the top.
2. Add the following `testProps` definition just after the other definitions:

```
const testProps = {
  today,
  selectableServices: services,
  availableTimeSlots,
  original: blankAppointment,
};
```

3. The first test in the suite looks like this:

```
it("renders a form", () => {
  render(
    <AppointmentForm
      original={blankAppointment}
      availableTimeSlots={availableTimeSlots}
    />
  );
  expect(form()).not.toBeNull();
});
```

This can be updated to look as follows:

```
it("renders a form", () => {
  render(<AppointmentForm {...testProps} />);
  expect(form()).not.toBeNull();
});
```

4. The next two tests, renders a submit button and renders as a select box, can use the same change. Go ahead and do that now.
5. Next up, we have the following test:

```
it("has a blank value as the first value", () => {
  render(
    <AppointmentForm
      original={blankAppointment}
      availableTimeSlots={availableTimeSlots}
    />
  );
  const firstOption = field("service").childNodes[0];
  expect(firstOption.value).toEqual("");
});
```

Since this test depends on having a blank value passed in for the `service` field, let's keep the `original` prop there:

```
it("has a blank value as the first value", () => {
  render(
    <AppointmentForm
      {...testProps}
```

```
        original={blankAppointment}
      />
    ) ;
  const firstOption = field("service").childNodes[0];
  expect(firstOption.value).toEqual("");
}) ;
```

We've effectively hidden the `availableTimeSlots` property, which was noise before.

6. Next, we have a test that makes use of `selectableServices`:

```
it("lists all salon services", () => {
  const services = ["Cut", "Blow-dry"];
  render(
    <AppointmentForm
      original={blankAppointment}
      selectableServices={services}
      availableTimeSlots={availableTimeSlots}
    />
  ) ;

  expect(
    labelsOfAllOptions(field("service"))
  ).toEqual(expect.arrayContaining(services));
}) ;
```

This test uses the `services` constant in its expectation, so this is a sign that we need to keep that as an explicit prop. Change it so that it matches the following:

```
it("lists all salon services", () => {
  const services = ["Cut", "Blow-dry"];

  render(
    <AppointmentForm
      {...testProps}
      selectableServices={services}
    />
  ) ;

  expect(
```

```
    labelsOfAllOptions(field("service"))
  ).toEqual(expect.arrayContaining(services));
});
```

7. In the next test, it's just `availableTimeSlots` that we can get rid of since both `services` and `appointments` are defined in the test itself:

```
it("pre-selects the existing value", () => {
  const services = ["Cut", "Blow-dry"];
  const appointment = { service: "Blow-dry" };
  render(
    <AppointmentForm
      {...testProps}
      original={appointment}
      selectableServices={services}
    />
  );
  const option = findOption(
    field("service"),
    "Blow-dry"
  );
  expect(option.selected).toBe(true);
});
```

The remaining tests in this test suite are in the nested `describe` block for the time slot table. Updating this is left as an exercise for you.

You've now learned yet more ways to clean up your test suites: extracting test data builders and extracting a `testProps` object. Remember that using the `testProps` object isn't always the right thing to do; it may be better to refactor your component so that it takes fewer props.

Summary

In this chapter, you learned how to use two types of HTML form elements: select boxes and radio buttons.

The component we've built has a decent amount of complexity, mainly due to the component hierarchy that's used to display a calendar view, but also because of the date and time functions we've needed to help display that view.

That is about as complex as it gets: writing React component tests shouldn't feel any more difficult than it has in this chapter.

Taking a moment to review our tests, the biggest issue we have is the use of `expect .hasAssertions` and the unusual **Arrange-Assert-Act** order. In *Chapter 6, Exploring Test Doubles*, we'll discover how we can simplify these tests and get them back into **Arrange-Act-Assert** order.

Exercises

The following are some exercises for you to try out:

1. Add a `toBeElementWithTag` matcher that replaces the two expectations in the `renders as a select` box test. It should be used like so:

```
expect(field("service")).toBeElementWithTag("select");
```

2. Complete the remaining tests for the `AppointmentForm` select box:
 - Renders a label
 - Assigns an ID that matches the label ID
 - Saves an existing value when submitted
 - Saves a new value when submitted

These tests are practically the same as they were for `CustomerForm`, including the use of the `change` helper. If you want a challenge, you can try extracting these form test helpers into a module of their own that is shared between `CustomerForm` and `AppointmentForm`.

3. Update the time slot table tests so that they use the `testProps` object.
4. Update the `AppointmentsDayView` component so that it uses the `todayAt` builder, where appropriate.
5. Add the ability to choose a stylist before choosing a time slot. This should be a select box that filters based on the service required, as not all stylists will be qualified to provide all services. You'll need to decide on a suitable data structure to hold this data. Modify `availableTimeSlots` so that it lists which stylists are available at each time, and update the table to reflect which stylist has been chosen and their availability during the week.

Further reading

The `useCallback` hook is useful when you're passing event handlers through a hierarchy of components. Take a look at the React documentation for tips on how to ensure correct usage: <https://reactjs.org/docs/hooks-reference.html#usecallback>.

6

Exploring Test Doubles

In this chapter, we'll look at the most involved piece of the TDD puzzle: test doubles.

Jest has a set of convenience functions for test doubles, such as `jest.spyOn` and `jest.fn`. Unfortunately, using test doubles well is a bit of a dark art. If you don't know what you're doing, you can end up with complicated, brittle tests. Maybe this is why Jest doesn't promote them as a first-class feature of its framework.

Don't be turned off: test doubles are a highly effective and versatile tool. The trick is to restrict your usage to a small set of well-defined patterns, which you'll learn about in the next few chapters.

In this chapter, we will build our own set of hand-crafted test double functions. They work pretty much just how Jest functions do, but with a simpler (and more clunky) interface. The aim is to take the magic out of these functions, showing you how they are built and how they can be used to simplify your tests.

In the test suites you've built so far, some tests didn't use the normal **Arrange-Act-Assert (AAA)** test format. These are the tests that start with `expect.hasAssertions`. In a real code base, I would always avoid using this function and instead use test doubles, which help reorder the test into AAA order. We'll start there: refactoring our existing tests to use our hand-crafted test doubles, and then swapping them out for Jest's own test double functions.

The following topics will be covered in this chapter:

- What is a test double?
- Submitting forms using spies
- Spying on the Fetch API
- Stubbing `fetch` responses
- Migrating to Jest's built-in test double support

By the end of the chapter, you'll have learned how to make effective use of Jest's test double functionality.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter06>

The code samples for this chapter and beyond contain extra commits that add a working backend to the application. This allows you to make requests to fetch data, which you'll start doing in this chapter.

In the companion code repository, from `Chapter06/Start` onward, the `npm run build` command will automatically build the server.

You can then start the application by using the `npm run serve` command and browsing to `http://localhost:3000` or `http://127.0.0.1:3000`.

If you run into problems

Check out the *Troubleshooting* section of the repository's `README.md` file if you're not able to get the application running.

What is a test double?

A *unit* in unit testing refers to a single function or component that we focus on for the duration of that test. The **Act** phase of a test should involve just one action on one unit. But units don't act in isolation: functions call other functions, and components render child components and call callback props passed in from parent components. Your application can be thought of as a web of dependencies, and test doubles help us to design and test those dependencies.

When we're writing tests, we **isolate** the unit under test. Often, that means we avoid exercising any of the **collaborating objects**. Why? Firstly, it helps us work toward our goal of independent, laser-focused tests. Secondly, sometimes those collaborating objects have side effects that would complicate our tests.

To give one example, with React components, we sometimes want to avoid rendering child components because they perform network requests when they are mounted.

A **test double** is an object that acts in place of a collaborating object. In *Chapter 4, Test-Driving Data Input*, you saw an example of a collaborator: the `onSubmit` function, which is a prop passed to both `CustomerForm` and `AppointmentForm`. We can swap that out with a test double in our tests. As we'll see, that helps us define the relationship between the two.

The most important place to use test doubles is at the edges of our system when our code interacts with anything external to the page content: **HyperText Transfer Protocol (HTTP)** requests, filesystem access, sockets, local storage, and so on.

Test doubles are categorized into several different types: spies, stubs, mocks, dummies, and fakes. We normally only use the first two, and that's what we'll concentrate on in this chapter.

Learning to avoid fakes

A **fake** is any test double that has any kind of logic or control structure within it, such as conditional statements or loops. Other types of test objects, such as spies and stubs, are made up entirely of variable assignments and function calls.

One type of fake you'll see is an in-memory repository. You can use this in place of **Structured Query Language (SQL)** data stores, message brokers, and other complex sources of data.

Fakes are useful when testing complex collaborations between two units. We'll often start by using spies and stubs and then refactor to a fake once the code starts to feel unwieldy. A single fake can cover a whole set of tests, which is simpler than maintaining a whole bunch of spies and stubs.

We avoid fakes for these reasons:

- Any logic requires tests, which means we must write tests for fakes, even though they are part of the test code. Spies and stubs don't require tests.
- Often, spies and stubs work in place of fakes. Only a small category of testing is simpler when we use fakes.
- Fakes increase test brittleness because they are shared between tests, unlike other test doubles.

Now that we've covered the theory of test doubles, let's move on to using them in our code.

Submitting forms using spies

In this section, you'll hand-craft a reusable spy function and adjust your tests to get them back into AAA order.

Here's a reminder of how one of those tests looked, from the `CustomerForm` test suite. It's complicated by the fact it's wrapped in a test generator, but you can ignore that bit for now—it's the test content that's important:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    expect.hasAssertions();
    const customer = { [fieldName]: value };
    render(
      <CustomerForm
        original={customer}
```

```
    onSubmit={ (props) =>
      expect(props[fieldName]).toEqual(value)
    }
  />
);

click(submitButton());
});
```

There are a couple of issues with this code, as follows:

- The **Assert** phase of the test—the expectation—appears wrapped within the **Act** phase. That makes the test difficult to read and understand.
- The call to `expect.hasAssertions` is ugly and is only there because our expectation is called as part of the `onSubmit` function, which may or may not be called.

We can fix both issues by building a spy.

What is a spy?

A **spy** is a type of test double that records the arguments it is called with so that those values can be inspected later.

Untangling AAA

To move the expectation under the **Act** phase of the test, we can introduce a variable to store the `firstName` value that's passed into the `onSubmit` function. We then write the expectation against that stored value.

Let's do that now, as follows:

1. Modify the `savesExistingValue` value when submitted test-generator function in `test/CustomerForm.test.js`, like so:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    let submitArg;
    const customer = { [fieldName]: value };

    render(
      <CustomerForm
```

```
    original={customer}
    onSubmit={submittedCustomer => (
      submitArg = submittedCustomer
    )}
  />
);
click(submitButton());

expect(submitArg).toEqual(customer);
});
```

The `submitArg` variable is assigned within our `onSubmit` handler and then asserted in the very last line of the test. This fixes both the issues we had with the first test: our test is back in AAA order and we got rid of the ugly `expect.hasAssertions()` call.

2. If you run your tests now, they should be green. However, any time you refactor tests in this way, you should verify that you're still testing the right thing by unwinding the production code and watching the test fail. To check that our tests still work, locate this line in `src/CustomerForm.js`:

```
<form id="customer" onSubmit={handleSubmit}>
```

Remove the `onSubmit` prop entirely, like so:

```
<form id="customer">
```

3. Run `npm test`. You'll get multiple test failures from various different tests. However, we're only interested in this one test generator, so update its declaration to `it.only` rather than `it`, as follows:

```
it.only("saves existing value when submitted", () => {
```

4. Now, you should have just three failures, one for each of the fields that uses this generator function, as illustrated in the following code snippet. That's a good sign; any fewer and we would have been generating false positives:

```
FAIL test/CustomerForm.test.js
```

```
● CustomerForm > first name field > saves existing
value when submitted
```

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

```
Expected: {"firstName": "existingValue"}
Received: undefined
```

5. We've proved the test works, so you can go ahead and change the `it.only` declaration back to `it`, and reinsert the `onSubmit` prop that you removed from `CustomerForm.js`.

The code you've written in this test shows the essence of the spy function: we set a variable when the spy is called, and then we write an expectation based on that variable value.

But we don't yet have an *actual* spy function. We'll create that next.

Making a reusable spy function

We still have other tests within both `CustomerForm` and `AppointmentForm` that use the `expect.hasAssertions` form. How can we reuse what we've built in this one test across everything else? We can create a generalized spy function that can be used any time we want spy functionality.

Let's start by defining a function that can stand in for any single-argument function, such as the event handlers we would pass to the `onSubmit` form prop, as follows:

1. Define the following function at the top of `test/CustomerForm.test.js`. Notice how the `fn` definition has a similar format to the `onSubmit` handler we used in the previous test:

```
const singleArgumentSpy = () => {
  let receivedArgument;
  return {
    fn: arg => (receivedArgument = arg),
    receivedArgument: () => receivedArgument
  };
};
```

2. Rewrite your test generator to use this function. Although your tests should still pass, remember to watch your tests fail by unwinding the production code. The code is illustrated in the following snippet:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const submitSpy = singleArgumentSpy();
    const customer = { [fieldName]: value };

    render(  
  <CustomerForm  
    onSubmit={submitSpy}  
    customer={customer} />
```

```

<CustomerForm
  original={customer}
  onSubmit={submitSpy.fn}
/>
);
click(submitButton());

expect(submitSpy.receivedArgument()).toEqual(
  customer
);
});

```

3. Make your spy function work for functions with any number of arguments by replacing `singleArgumentSpy` with the following function:

```

const spy = () => {
  let receivedArguments;
  return {
    fn: (...args) => (receivedArguments = args),
    receivedArguments: () => receivedArguments,
    receivedArgument: n => receivedArguments[n]
  };
};

```

This uses parameter destructuring to save an entire array of parameters. We can use `receivedArguments` to return that array or use `receivedArgument (n)` to retrieve the *n*th argument.

4. Update your test code to use this new function, as shown in the following code snippet. You can include an extra expectation that checks `toBeDefined` on `receivedArguments`. This is a way of saying “I expect the function to be called”:

```

const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const submitSpy = spy();
    const customer = { [fieldName]: value };

    render(
      <CustomerForm
        original={customer}
    
```

```

        onSubmit={submitSpy.fn}
      />
    ) ;
click(submitButton()) ;

expect(
  submitSpy.receivedArguments()
).toBeDefined();
expect(submitSpy.receivedArgument(0)).toEqual(
  customer
) ;
}) ;

```

That's really all there is to a spy: it's just there to keep track of when it was called, and the arguments it was called with.

Using a matcher to simplify spy expectations

Let's write a matcher that encapsulates these expectations into one single statement, like this:

```
expect(submitSpy).toBeCalledWith(value);
```

This is more descriptive than using a `.toBeDefined()` argument on the matcher. It also encapsulates the notion that if `receivedArguments` hasn't been set, then it hasn't been called.

Throwaway code

We'll **spike** this code—in other words, we won't write tests. That's because soon, we'll replace this with Jest's own built-in spy functionality. There's no point in going too deep into a “real” implementation since we're not intending to keep it around for long.

We'll start by replacing the functionality of the first expectation, as follows:

- Add the following code at the bottom of `test/domMatchers.js`. It adds the new matcher, ready for our tests:

```

expect.extend({
  toBeCalled(received) {
    if (received.receivedArguments() === undefined) {
      return {
        pass: false,
        message: 'Expected to be called'
      }
    }
  }
})

```

```
        message: () => "Spy was not called.",
    );
}

return {
  pass: true,
  message: () => "Spy was called.",
};
},
)) ;
```

2. Update the test to use the new matcher, replacing the first expectation that used `toBeDefined`, as follows:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const submitSpy = spy();
    const customer = { [fieldName]: value };

    render(
      <CustomerForm
        original={customer}
        onSubmit={submitSpy.fn}
      />
    );
    click(submitButton());

    expect(submitSpy).toBeCalled(customer);
    expect(submitSpy.receivedArgument(0)).toEqual(
      customer
    );
  });
});
```

3. Verify the new matcher works by commenting out the call to `onSubmit` in your production code and watching the test fail. Then, undo the comment and try the negated form in your `.not.toBeCalled` test.

4. Now we can work on the second expectation—the one that checks the function arguments. Add the following code to your new matcher and rename it from `toBeCalled` to `toBeCalledWith`:

```
expect.extend({
  toBeCalledWith(received, ...expectedArguments) {
    if (received.receivedArguments() === undefined) {
      ...
    }

    const notMatch = !this.equals(
      received.receivedArguments(),
      expectedArguments
    );

    if (notMatch) {
      return {
        pass: false,
        message: () =>
          `Spy called with the wrong arguments: ${received.receivedArguments()} +
          ${received.receivedArguments().join(', ')},
        `;
    }
  }

  return ...;
},
});
```

Using `this.equals` in a matcher

The `this.equals` method is a special type of equality function that can be used in matchers. It does deep equality matching, meaning it will recurse through hashes and arrays looking for differences. It also allows the use of `expect.anything()`, `expect.objectContaining()`, and `expect.arrayContaining()` special functions.

If you were test-driving this matcher and had extracted it into its own file, you wouldn't use `this.equals`. Instead, you'd import the `equals` function from the `@jest/expect-utils` package. We'll do this in *Chapter 7, Testing `useEffect` and Mocking Components*.

5. Update your test to merge both expectations into one, as follows:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    ...
    click(submitButton());
    expect(submitSpy).toBeCalledWith(customer);
  });
}
```

6. Make this fail by changing the `onSubmit` call in your `CustomerForm` test suite to send obviously wrong data—for example, `onSubmit(1, 2, 3)`. Then, try the negated form of the matcher too.

This completes our spy implementation, and you've seen how to test callback props. Next, we'll look at spying on a more difficult function: `global.fetch`.

Spying on the fetch API

In this section, we'll use the Fetch API to send customer data to our backend service. We already have an `onSubmit` prop that is called when the form is submitted. We'll morph this `onSubmit` call into a `global.fetch` call, in the process of adjusting our existing tests.

In our updated component, when the **Submit** button is clicked, a POST HTTP request is sent to the `/customers` endpoint via the `fetch` function. The body of the request will be a **JavaScript Object Notation (JSON)** object representation of our customer.

The server implementation that's included in the GitHub repository will return an updated `customer` object with an additional field: the `customer.id` value.

If the `fetch` request is successful, we'll call a new `onSave` callback prop with the `fetch` response. If the request isn't successful, `onSave` won't be called and we'll instead render an error message.

You can think of `fetch` as a more advanced form on `onSubmit`: both are functions that we'll call with our `customer` object. But `fetch` needs a special set of parameters to define the HTTP request being made. It also returns a `Promise` object, so we'll need to account for that, and the request body needs to be a string, rather than a plain object, so we'll need to make sure we translate it in our component and in our test suite.

One final difference: `fetch` is a global function, accessible via `global.fetch`. We don't need to pass that as a prop. In order to spy on it, we replace the original function with our spy.

Understanding the Fetch API

The following code samples show how the `fetch` function expects to be called. If you're unfamiliar with this function, see the *Further reading* section at the end of this chapter.

With all that in mind, we can plan our route forward: we'll start by replacing the global function with our own spy, then we'll add new tests to ensure we call it correctly, and finally, we'll update our `onSubmit` tests to adjust its existing behavior.

Replacing global functions with spies

We've seen how to spy on a callback prop, by simply passing the spy as the callback's prop value. To spy on a global function, we simply overwrite its value before our test runs and reset it back to the original function afterward.

Since `global.fetch` is a required dependency of your component—it won't function without it—it makes sense to set a default spy in the test suite's `beforeEach` block so that the spy is primed in all tests. The `beforeEach` block is also a good place for setting default return values of stubs, which we'll do a little later in the chapter.

Follow these steps to set a default spy on `global.fetch` for your test suite:

1. Add the following declarations at the top of the outer `describe` block in `test/CustomerForm.test.js`:

```
describe("CustomerForm", () => {
  const originalFetch = global.fetch;
  let fetchSpy;

  ...
})
```

The `originalFetch` constant will be used when restoring the spy after our tests are complete. The `fetchSpy` variable will be used to store our `fetch` object so that we can write expectations against it.

2. Change the `beforeEach` block to read as follows. This sets up `global.fetch` as a spy for every test in your test suite:

```
beforeEach(() => {
  initializeReactContainer();
  fetchSpy = spy();
  global.fetch = fetchSpy.fn;
});
```

- Just below the `beforeEach` block, add an `afterEach` block to unset your mock, like so:

```
afterEach(() => {
  global.fetch = originalFetch;
});
```

Resetting global spies with original values

It's important to reset any global variables that you replace with spies. This is a common cause of test interdependence: with a "dirty" spy, one test may break because some other test failed to reset its spies.

In this specific case, the Node.js runtime environment doesn't actually have a `global.fetch` function, so the `originalFetch` constant will end up as `undefined`. You could argue, then, that this is unnecessary: in our `afterEach` block, we could simply delete the `fetch` property from `global` instead.

Later in the chapter, we'll modify our approach to setting global spies when we use Jest's built-in spy functions.

With the global spy in place, you're ready to make use of it in your tests.

Test-driving fetch argument values

It's time to add `global.fetch` to our component. When the `submit` button is clicked, we want to check that `global.fetch` is called with the right arguments. Similar to how we tested `onSubmit`, we'll split this into a test for each field, specifying that each field must be passed along.

It turns out that `global.fetch` needs a whole bunch of parameters passed to it. Rather than test them all in one single unit test, we're going to split up the tests according to their meaning.

We'll start by checking the basics of the request: that it's a `POST` request to the `/customers` endpoint. Follow these steps:

- Add the following new test at the bottom of your `CustomerForm` test suite. Notice how `onSubmit` is given an empty function definition—`() => {}`—rather than a spy since we aren't interested in that prop in this test:

```
it("sends request to POST /customers when submitting the
form", () => {
  render(
    <CustomerForm
      original={blankCustomer}
      onSubmit={() => {}}
    />
```

```
) ;
click(submitButton());
expect(fetchSpy).toBeCalledWith(
  "/customers",
  expect.objectContaining({
    method: "POST",
  })
);
}) ;
```

2. Run tests with `npm test` and verify that you receive an expectation failure with a `Spy was not called` message, as shown in the following code snippet:

● `CustomerForm > sends request to POST /customers when submitting the form`

`Spy was not called.`

```
163 |      );
164 |      click(submitButton());
> 165 |      expect(fetchSpy).toBeCalledWith(
|           ^
166 |      "/customers",
167 |      expect.objectContaining({
168 |        method: "POST",
```

3. To make that pass, modify `CustomerForm`'s `handleSubmit` function by adding a call to `global.fetch` *above* the call to `onSubmit`, as shown in the following code snippet:

```
const handleSubmit = (event) => {
  event.preventDefault();
  global.fetch("/customers", {
    method: "POST",
  });
  onSubmit(customer);
};
```

Side-by-side implementations

This is a side-by-side implementation. We leave the “old” implementation—the call to `onSubmit`—in place so that the other tests continue to pass.

- With that test passing, add the next one. In this test, we test all the plumbing that’s necessary for our request, which we’ll call “configuration,” but you can think of this as batching up all the constant, less relevant information. This test also uses two new functions, `expect.anything` and `expect.objectContaining`, which are shown in the following code snippet:

```
it("calls fetch with the right configuration", () => {
  render(
    <CustomerForm
      original={blankCustomer}
      onSubmit={() => {}}
    />
  ) ;
  click(submitButton());
  expect(fetchSpy).toBeCalledWith(
    expect.anything(),
    expect.objectContaining({
      credentials: "same-origin",
      headers: {
        "Content-Type": "application/json",
      },
    })
  );
}) ;
```

Testing a subset of properties with `expect.anything` and `expect.objectContaining`

The `expect.anything` function is a useful way of saying: “I don’t care about this argument in this test; I’ve tested it somewhere else.” It’s another great way of keeping your tests independent of each other. In this case, our previous test checks that the first parameter is set to `/customers`, so we don’t need to test that again in this test.

The `expect.objectContaining` function is just like `expect.arrayContaining`, and allows us to test just a slice of the full argument value.

5. Run that test and observe the test failure. You can see in the following code snippet that our matcher hasn't done a great job of printing the message: the second actual parameter is printed as [object Object]. Let's ignore that for now since later in the chapter, we'll move to using Jest's built-in matcher:

```
● CustomerForm > calls fetch with the right configuration when submitting the form
```

```
Spy was called with the wrong arguments: /customers, [object Object].
```

6. To make that pass, simply insert the remaining properties into your call to `global.fetch`:

```
const handleSubmit = (event) => {
  event.preventDefault();
  global.fetch("/customers", {
    method: "POST",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" },
  });
  onSubmit(customer);
};
```

That gets the plumbing in place for our `global.fetch` call, with each of the constant arguments defined and in its place. Next, we'll add in the dynamic argument: the request body.

Reworking existing tests with the side-by-side implementation

You've already started to build out the side-by-side implementation by using new tests. Now, it's time to rework the existing tests. We'll remove the old implementation (`onSubmit`, in this case) and replace it with the new implementation (`global.fetch`).

Once we've completed that, all the tests will point to `global.fetch` and so we can update our implementation to remove the `onSubmit` call from the `handleSubmit` function.

We've got two tests to update: the test that checks submitting existing values, and the test that checks submitting new values. They are complicated by the fact that they are wrapped in test-generator functions. That means as we change them, we should expect all the generated tests—one for each field—to fail as a group. It's not ideal, but the process we're following would be the same even if it were just a plain test.

Let's get started with the test you've already worked on in this chapter, for submitting existing values. Follow these steps:

1. Move back to the `itSubmitsExistingValue` test-generator function and update it by inserting a new expectation at the bottom. Leave the existing expectation as it is (for now). Run the test and ensure the generated test fails. The code is illustrated in the following snippet:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const customer = { [fieldName]: value };
    const submitSpy = spy();
    render(
      <CustomerForm
        original={customer}
        onSubmit={submitSpy.fn}
      />
    );
    click(submitButton());
    expect(submitSpy).toBeCalledWith(customer);

    expect(fetchSpy).toBeCalledWith(
      expect.anything(),
      expect.objectContaining({
        body: JSON.stringify(customer),
      })
    );
  });
}
```

2. To make that pass, update the `handleSubmit` function in your `CustomerForm` component, as shown in the following code snippet. After this change, your tests should pass:

```
const handleSubmit = (event) => {
  event.preventDefault();
  global.fetch("/customers", {
    method: "POST",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(original),
  });
}
```

```
    onSubmit(customer);
};
```

3. The final test reference to the `onSubmit` prop is the `itSubmitsnewValue` test generator. This test still uses the old `expect.hasAssertions` style; we'll get round to deleting that later. For now, simply add in a new expectation at the bottom of the test, as shown here:

```
const itSubmitsnewValue = (fieldName, value) =>
  it("saves new value when submitted", () => {
    ...
    expect(fetchSpy).toBeCalledWith(
      expect.anything(),
      expect.objectContaining({
        body: JSON.stringify({
          ...blankCustomer,
          [fieldName]: value,
        }),
      })
    );
  });
});
```

4. Run the test and verify that this test fails with a `Spy was called with the wrong arguments: /customers, [object Object]` failure message.
5. To make that pass, it's a case of changing `original` to `customer` in your `handleSubmit` function, as follows:

```
const handleSubmit = (event) => {
  event.preventDefault();
  global.fetch("/customers", {
    method: "POST",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(customer),
  });
  onSubmit(customer);
};
```

6. Your call to `fetch` is now complete, so you can remove the original implementation. Start by removing the `onSubmit` prop and the `submitSpy` variable from the `itSubmitsExistingValue` test generator. The new version looks like this:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const customer = { [fieldName]: value };
    render(<CustomerForm original={customer} />);
    click(submitButton());
    expect(fetchSpy).toBeCalledWith(
      expect.anything(),
      expect.objectContaining({
        body: JSON.stringify(customer),
      })
    );
  });
}
```

7. Do the same for `itSubmitsnewValue`—you can get rid of the `hasAssertions` call too. The new version looks like this:

```
const itSubmitsnewValue = (fieldName, value) =>
  it("saves new value when submitted", () => {
    render(<CustomerForm original={blankCustomer} />);
    change(field(fieldName), value);
    click(submitButton());

    expect(fetchSpy).toBeCalledWith(
      expect.anything(),
      expect.objectContaining({
        body: JSON.stringify({
          ...blankCustomer,
          [fieldName]: value,
        }),
      })
    );
  });
}
```

8. Remove the call to `onSubmit` in the `handleSubmit` method.

9. Remove the `onSubmit` prop from the `CustomerForm` component definition.
10. Finally, remove the `onSubmit` prop from the `prevents the default action...` test.
11. Verify that all your tests are passing with `npm test`.

You've now seen how you can continue your side-by-side implementation by reworking tests. Once all the tests are reworked, you can delete the original implementation.

Our tests have gotten pretty long-winded again. Let's finish this section with a little cleanup.

Improving spy expectations with helper functions

When we're writing expectations for our spies, we aren't just limited to using the `toBeCalledWith` matcher. We can pull out arguments and give them names, and then use standard Jest matchers on them instead. This way, we can avoid all the ceremony with `expect.anything` and `expect.objectContaining`.

Let's do that now. Proceed as follows:

1. Add a new helper function, `bodyOfLastFetchRequest`, at the top of `CustomerForm`, as follows:

```
const bodyOfLastFetchRequest = () =>
  JSON.parse(fetchSpy.receivedArgument(1).body);
```

2. Update your `itSubmitsExistingValue` test generator to use this new helper to simplify its expectation. Note here the use of `toMatchObject`, which takes the place of `expect.objectContaining` in the previous version of this test:

```
const itSubmitsExistingValue = (fieldName, value) =>
  it("saves existing value when submitted", () => {
    const customer = { [fieldName]: value };
    render(<CustomerForm original={customer} />);
    click(submitButton());
    expect(bodyOfLastFetchRequest()).toMatchObject(
      customer
    );
  });
});
```

3. Since you've modified your test, you should verify that it still tests the right thing: mark it as `it.only` and then delete the `body` property from the `global.fetch` call. Check the test fails, and then undo the change, getting you back to a passing test.

4. Repeat for the `itSubmitsnewValue` test helper, as shown here:

```
const itSubmitsnewValue = (fieldName, value) =>
  it("saves new value when submitted", () => {
    render(<CustomerForm original={blankCustomer} />);
    change(field(fieldName), value);
    click(submitButton());

    expect(bodyOfLastFetchRequest()).toMatchObject({
      [fieldName]: value,
    });
  });
}
```

These tests are now looking very smart!

This has been a complicated change: we've replaced the `onSubmit` prop with a call to `global.fetch`. We did that by introducing a global spy in the `beforeEach` block and writing a side-by-side implementation while we reworked our tests.

In the next part of this chapter, we'll add to our knowledge of spies, turning them into stubs.

Stubbing fetch responses

As with many HTTP requests, our `POST /customers` endpoint returns data: it will return the customer object together with a newly generated identifier that the backend has chosen for us. Our application will make use of this by taking the new ID and sending it back to the parent component (although we won't build this parent component until *Chapter 8, Building an Application Component*).

To do that, we'll create a new `CustomerForm` prop, `onSave`, which will be called with the result of the `fetch` call.

But hold on—didn't we just remove an `onSubmit` prop? Yes, but this isn't the same thing. The original `onSubmit` prop received the form values submitted by the user. This `onSave` prop is going to receive the customer object from the server after a successful save.

To write tests for this new `onSave` prop, we'll need to provide a stub value for `global.fetch`, which essentially says, "This is the return value of calling the `POST /customers` endpoint with `global.fetch`."

What is a stub?

A **stub** is a test double that always returns the same value when it is invoked. You decide what this value is when you construct the stub.

In this section, we'll upgrade our hand-crafted spy function so that it can also stub function return values. Then, we'll use it to test the addition of the new `onSave` prop to `CustomerForm`. Finally, we'll use it to display an error to the user if, for some reason, the server failed to save the new customer object.

Upgrading spies to stubs

A stub is different from a spy because it's not interested in tracking the call history of the function being stubbed—it just cares about returning a single value.

However, it turns out that our existing tests that use spies will also need to stub values. That's because as soon as we use the returned value in our production code, the spy must return something; otherwise, the test will break. So, all spies end up being stubs, too.

Since we already have a `spy` function, we can “upgrade” it so that it has the ability to stub values too. Here's how we can do this:

1. In `test/CustomerForm.test.js`, change the `spy` function to include the following new variable declaration at the top. This variable will store the value, ready to be returned by our function:

```
let returnValue;
```

2. Change the `fn` definition to the one shown here:

```
fn: (...args) => {
  receivedArguments = args;
  return returnValue;
},
```

3. Add this new function to your spy object, which sets the `returnValue` variable:

```
stubReturnValue: value => returnValue = value
```

It's as simple as that: your function is now both a spy and a stub. Let's make use of it in our tests.

Acting on the fetch response

So far, the `handleSubmit` function causes a `fetch` request to be made, but it doesn't do anything with the response. In particular, it doesn't *wait* for a response; the `fetch` API is asynchronous and returns a promise. Once that promise resolves, we can do something with the data that's returned.

The next tests we'll write will specify what our component should do with the resolved data.

The asynchronous form of act

When we're dealing with promises in React callbacks, we need to use the asynchronous form of `act`. It looks like this:

```
await act(async () => performAsyncAction());
```

The `performAsyncAction` function doesn't necessarily need to return a promise; `act` will wait for the browser's `async` task queue to complete before it returns.

The action may be a button click, form submission, or any kind of input field event. It could also be a component render that has a `useEffect` hook that performs some asynchronous side effects, such as loading data.

Adding async tasks to an existing component

Now, we'll use the asynchronous form of `act` to test that the `fetch` promise is awaited. Unfortunately, introducing `async/await` into our `handleSubmit` function will then require us to update all our submission tests to use the asynchronous form of `act`.

As usual, we start with the test. Proceed as follows:

1. Define a test helper function in `test/CustomerForm.test.js` that builds you a type of `Response` object to mimic what would be returned from the `fetch` API. That means it returns a `Promise` object with an `ok` property with a value of `true`, and a `json` function that returns another `Promise` that, when resolved, returns the JSON we pass in. You can define this just under your `spy` function, like so:

```
const fetchResponseOk = (body) =>
  Promise.resolve({
    ok: true,
    json: () => Promise.resolve(body)
 });
```

fetch return values

The `ok` property returns `true` if the HTTP response status code was in the `2xx` range. Any other kind of response, such as `404` or `500`, will cause `ok` to be `false`.

2. Add the following code to `test/reactTestExtensions.js`, just below the definition of `click`:

```
export const clickAndWait = async (element) =>
  act(async () => click(element));
```

3. Now, import the new helper function into `test/CustomerForm.test.js`, as follows:

```
import {  
  ...,  
  clickAndWait,  
} from "./reactTestExtensions";
```

4. Add the next test to the `CustomerForm` test suite, which checks that the `onSave` prop function is called when the user submits the form, and passes back the customer object. The best place for this test is under the `calls fetch with correct configuration` test. The code is illustrated in the following snippet:

```
it("notifies onSave when form is submitted", async () =>  
{  
  const customer = { id: 123 };  
  fetchSpy.stubReturnValue(fetchResponseOk(customer));  
  const saveSpy = spy();  
  
  render(  
    <CustomerForm  
      original={customer}  
      onSave={saveSpy.fn}  
    />  
  );  
  await clickAndWait(submitButton());  
  
  expect(saveSpy).toBeCalledWith(customer);  
});
```

5. To make this pass, start by defining a new `onSave` prop for `CustomerForm`, in `src/CustomerForm.js`, as follows:

```
export const CustomerForm = ({  
  original, onSave  
}) => {  
  ...  
};
```

6. Add the following code at the end of `handleSubmit`. The function is now declared `async` and uses `await` to unwrap the promise returned from `global.fetch`:

```
const handleSubmit = async (event) => {
  event.preventDefault();
  const result = await global.fetch(...);
  const customerWithId = await result.json();
  onSave(customerWithId);
};
```

7. If you run tests, you'll notice that although your latest test passes, your previous test fails and there's a whole bunch of unhandled promise exceptions. In fact, anything that submits the form will fail, because they use the `fetchSpy` variable that's initialized in the `beforeEach` block, and this is not a stub—it's just a plain old spy. Fix that now by giving the spy a return value, within `beforeEach`. In this case, we don't need to give it a customer; an empty object will do, as illustrated in the following code snippet:

```
beforeEach(() => {
  ...
  fetchSpy.stubReturnValue(fetchResponseOk({}));
});
```

Dummy values in `beforeEach` blocks

When stubbing out global functions such as `global.fetch`, always set a default dummy value within your `beforeEach` block and then override it in individual tests that need specific stubbed values.

8. Run tests again. You might see some odd behavior at this point; I see my recent test supposedly run six times with failures! What's happening is that our previous tests are now firing off a whole bunch of promises that continue running even when the tests end. Those asynchronous tasks cause Jest to incorrectly report failures. To solve this, we need to update all our tests to use `await clickAndWait`. In addition, the tests need to be marked as `async`. Do this now for every test that calls `click`. An example is shown here:

```
it("sends HTTP request to POST /customers when submitting data", async () => {
  render(<CustomerForm original={blankCustomer} />);
  await clickAndWait(submitButton());
  ...
});
```

9. Delete the `click` import, leaving `clickAndWait`.
10. There's one more test that has this issue, and that's the test that submits the form: prevents the default action when submitting the form. This test calls our `submit` helper function. We need to wrap that in `act`, too. Let's create a `submitAndWait` helper function in our test extensions file. Add the following function just below `submit` to `test/reactTestExtensions.js`:

```
export const submitAndWait = async (formElement) =>
  act(async () => submit(formElement));
```

11. Add `submitAndWait` into your import statements, just below `clickAndWait`, as follows:

```
import {
  ...
  submitAndWait,
} from "./reactTestExtensions";
```

12. Now, you can update the test to use the new helper function, like so:

```
it("prevents the default action when submitting the
form", async () => {
  render(<CustomerForm original={blankCustomer} />);
  const event = await submitAndWait(form());
  expect(event.defaultPrevented).toBe(true);
});
```

13. If you run tests again, we still have test failures (although thankfully, things look more orderly with the `async` tasks being properly accounted for). You'll see that you now have a bunch of failures that say `onSave` is not a function. To fix that, we need to ensure we specify the `onSave` prop for every test that submits the form. A blank, no-op function will do. An example is shown here. Go ahead and add this prop to every test that submits the form. After this change, your tests should be passing without any warnings:

```
it("calls fetch with correct configuration", async () =>
{
  render(
    <CustomerForm
      original={blankCustomer}
      onSave={() => {}}
    />
  );
});
```

```
...  
});
```

Introducing testProps objects when required props are added

The introduction of this `onSave` no-op function creates noise, which doesn't help with the readability of our test. This would be a perfect opportunity to introduce a `testProps` object, as covered in *Chapter 5, Adding Complex Form Interactions*.

14. Add another test to ensure that we do not call `onSave` when the `fetch` response has an error status (in other words, when the `ok` property is set to `false`). Start by defining another helper, `fetchResponseError`, right under `fetchResponseOk`, as illustrated in the following code snippet. This one doesn't need a body as we aren't interested in it just yet:

```
const fetchResponseError = () =>  
  Promise.resolve({ ok: false });
```

15. Use the new function in the next `CustomerForm` test, as follows:

```
it("does not notify onSave if the POST request returns an  
error", async () => {  
  fetchSpy.stubReturnValue(fetchResponseError());  
  const saveSpy = spy();  
  
  render(  
    <CustomerForm  
      original={blankCustomer}  
      onSave={saveSpy.fn}  
    />  
  );  
  await clickAndWait(submitButton());  
  
  expect(saveSpy).not.toBeCalledWith();  
});
```

Negating `toBeCalledWith`

This expectation is not what we really want: this one would pass if we still called `onSave` but passed the wrong arguments—for example, if we wrote `onSave(null)`. What we really want is `.not.toBeCalled()`, which will fail if `onSave` is called in any form. But we haven't built that matcher. Later in the chapter, we'll fix this expectation by moving to Jest's built-in spy function.

16. To make this pass, move the `onSave` call into a new conditional in `handleSubmit`, as follows:

```
const handleSubmit = async (event) => {
  ...
  const result = ...;
  if (result.ok) {
    const customerWithId = await result.json();
    onSave(customerWithId);
  }
};
```

As you've seen, moving a component from synchronous to asynchronous behavior can really disrupt our test suites. The steps just outlined are fairly typical of the work needed when this happens.

Async component actions can cause misreported Jest test failures

If you're ever surprised to see a test fail and you're at a loss to explain why it's failing, double-check all the tests in the test suite to ensure that you've used the `async` form of `act` when it's needed. Jest won't warn you when a test finishes with `async` tasks still to run, and since your tests are using a shared DOM document, those `async` tasks will affect the results of subsequent tests.

Those are the basics of dealing with `async` behavior in tests. Now, let's add a little detail to our implementation.

Displaying errors to the user

Let's display an error to the user if the `fetch` returns an `ok` value of `false`. This would occur if the HTTP status code returned was in the `4xx` or `5xx` range, although for our tests we won't need to worry about the specific status code. Follow these steps:

1. Add the following test to `test/CustomerForm.test.js`. This checks that an area is shown on the page for errors. It relies on the ARIA role of `alert`, which is a special signifier for screen readers that this area could change to hold important information:

```
it("renders an alert space", async () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(element("[role=alert]")).not.toBeNull();
});
```

2. To make that pass, first, define a new `Error` component, as follows. This can live in `src/CustomerForm.js`, just above the `CustomerForm` component itself:

```
const Error = () => (
  <p role="alert" />
);
```

3. Then, add an instance of that component into the `CustomerForm`'s JSX, just at the top of the `form` element, as follows:

```
<form>
  <Error />
  ...
</form>
```

4. Back in `test/CustomerForm.test.js`, add the next test, which checks the error message in the alert, as follows:

```
it("renders error message when fetch call fails", async () => {
  fetchSpy.mockReturnValue(fetchResponseError());

  render(<CustomerForm original={blankCustomer} />);
  await clickAndWait(submitButton());

  expect(element("[role=alert]")).toContainText(
    "error occurred"
  );
});
```

5. To make that pass, all we need to do is hardcode the string in the `Error` component. We'll use another test to triangulate to get to the real implementation, as follows:

```
const Error = () => (
  <p role="alert">
    An error occurred during save.
  </p>
);
```

6. Add the final test to `test/CustomerForm.test.js`, like so:

```
it("initially has no text in the alert space", async () =>
{
  render(<CustomerForm original={blankCustomer} />);
  expect(element("[role=alert]")).not.toContainText(
    "error occurred"
  );
});
```

7. To make this pass, introduce a new `error` state variable at the top of the `CustomerForm` definition, like so:

```
const [error, setError] = useState(false);
```

8. Change the `handleSubmit` function, as follows:

```
const handleSubmit = async (event) => {
  ...
  if (result.ok) {
    ...
  } else {
    setError(true);
  }
}
```

9. In the component's JSX, update the `Error` instance to include a new `hasError` prop and set it to the `error` state, like so:

```
<form>
  <Error hasError={error} />
  ...
</form>
```

10. All that remains is to complete the `Error` component with the new prop, as follows:

```
const Error = ({ hasError }) => (
  <p role="alert">
    {hasError ? "An error occurred during save." : ""}
  </p>
);
```

That's it for our `CustomerForm` implementation. Time for a little cleanup of our tests.

Grouping stub scenarios in nested describe contexts

A common practice is to use nested `describe` blocks to set up stub values as scenarios for a group of tests. We have just written four tests that deal with the scenario of the `POST /customers` endpoint returning an error. Two of these are good candidates for a nested `describe` context.

We can then pull up the stub value into a `beforeEach` block. Let's start with the `describe` block. Follow these steps:

1. Look at the last four tests you've written. Two of them are about the alert space and are not related to the error case. Leave those two in place, and move the other two into a new `describe` block named `when POST requests return an error`, as shown here:

```
it("renders an alert space", ...)
it("initially has no text in the alert space", ...)

describe("when POST request returns an error", () => {
  it("does not notify onSave if the POST request returns
  an error", ...)
  it("renders error message when fetch call fails", ...)
});
```

2. Notice how the two of the test descriptions repeat themselves, saying the same thing as the `describe` block but in slightly different ways? Remove the `if/when` statements from the two test descriptions, as follows:

```
describe("when POST request returns an error", () => {
  it("does not notify onSave", ...)
  it("renders error message ", ...)
});
```

3. The two tests have identical `global.fetch` stubs. Pull that stub up into a new `beforeEach` block, as shown here:

```
describe("when POST request returns an error", () => {

  beforeEach(() => {
    fetchSpy.stubReturnValue(fetchResponseError());
  });
```

```
    ...  
})
```

4. Finally, delete the stub call from the two tests, leaving just the stub call in the `beforeEach` block.

You've now seen how to use nested `describe` blocks to describe specific test scenarios, and that covers all the basic stubbing techniques. In the next section, we'll continue our cleanup by introducing Jest's own spy and stub functions, which are slightly simpler than the ones we've built ourselves.

Migrating to Jest's built-in test double support

So far in this chapter, you've built your own hand-crafted spy function, with support for stubbing values and with its own matcher. The purpose of that has been to teach you how test doubles work and to show the essential set of spy and stub patterns that you'll use in your component tests.

However, our spy function and the `toBeCalledWith` matcher are far from complete. Rather than investing any more time in our hand-crafted versions, it makes sense to switch to Jest's own functions now. These work in essentially the same way as our spy function but have a couple of subtle differences.

This section starts with a rundown of Jest's test double functionality. Then, we'll migrate the `CustomerForm` test suite away from our hand-crafted spy function. Finally, we'll do a little more cleanup by extracting more test helpers.

Using Jest to spy and stub

Here's a rundown of Jest test double support:

- To create a new spy function, call `jest.fn()`. For example, you might write `const fetchSpy = jest.fn()`.
- To override an existing property, call `jest.spyOn(object, property)`. For example, you might write `jest.spyOn(global, "fetch")`.
- To set a return value, call `spy.mockReturnValue()`. You can also pass this value directly to the `jest.fn()` call.
- You can set multiple return values by chaining calls to `spy.mockReturnValueOnce()`.
- When your function returns promises, you can use `spy.mockResolvedValue()` and `spy.mockRejectedValue()`.
- To check that your spy was called, use `expect(spy).toBeCalled()`.
- To check the arguments passed to your spy, you can use `expect(spy).toBeCalledWith(arguments)`. Or, if your spy is called multiple times and you want to check the last time it was called, you can use `expect(spy).toHaveLastBeenCalledWith(arguments)`.

- Calling `spy.mockReset()` removes all mocked implementations, return values, and existing call history from a spy.
- Calling `spy.mockRestore()` will remove the mock and give you back the original implementation.
- In Jest's configuration section of your `package.json` file, you can set `restoreMocks` to `true` and all spies that were created with `jest.spyOn` will be automatically restored after each test.
- When using `toBeCalledWith`, you can pass an argument value of `expect.anything()` to say that you don't care what the value of that argument is.
- You can use `expect.objectMatching(object)` to check that an argument has all the properties of the object you pass in, rather than being exactly equal to the object.
- When your spy is called multiple times, you can check the parameters passed to specific calls by using `spy.mock.calls[n]`, where `n` is the call number (for example, `calls[0]` will return the arguments for the first time it was called).
- If you need to perform complex matching on a specific argument, you can use `spy.mock.calls[0][n]`, where `n` is in the argument number.
- You can stub out and spy on entire modules using the `jest.mock()` function, which we'll look at in the next chapter.

There's a lot more available with the Jest API, but these are the core features and should cover most of your test-driven use cases.

Migrating the test suite to use Jest's test double support

Let's convert our `CustomerForm` tests away from our hand-crafted spy function. We'll start with the `fetchSpy` variable.

We'll use `jest.spyOn` for this. It essentially creates a spy with `jest.fn()` and then assigns it to the `global.fetch` variable. The `jest.spyOn` function keeps track of every object that has been spied on so that it can auto-restore them without our intervention, using the `restoreMock` configuration property.

It also has a feature that blocks us from spying on any property that isn't already a function. That will affect us because Node.js doesn't have a default implementation of `global.fetch`. We'll see how to solve that issue in the next set of steps.

It's worth pointing out a really great feature of `jest.fn()`. The returned spy object acts as both the function itself and the mock object. It does this by attaching a special `mock` property to the returned function. The upshot of this is that we no longer need a `fetchSpy` variable to store our spy object. We can just refer to `global.fetch` directly, as we're about to see.

Follow these steps:

1. Update the `beforeEach` block to read as follows. This uses `mockResolvedValue` to set a return value wrapped in a promise (as opposed to `mockReturnValue`, which just returns a value with no promise involved):

```
beforeEach(() => {
  initializeReactContainer();
  jest
    .spyOn(global, "fetch")
    .mockResolvedValue(fetchResponseOk({}));
});
```

2. There are two lines in the `CustomerForm` test suite that follow the pattern shown here:

```
fetchSpy.stubResolvedValue(...);
```

Go ahead and replace them with the following code:

```
global.fetch.mockResolvedValue(...);
```

3. There are two expectations that check `fetchSpy`. Go ahead and replace `expect(fetchSpy)` with `expect(global.fetch)`. Removing the `fetchSpy` variable gives you greater readability and understanding of what's happening. Here's one of the expectations:

```
expect(global.fetch).toBeCalledWith(
  "/customers",
  expect.objectContaining({
    method: "POST",
  })
);
```

4. The `bodyOfLastFetchRequest` function needs to be updated to use the `mock` property of the Jest spy object. Update it to read as follows:

```
const bodyOfLastFetchRequest = () => {
  const allCalls = global.fetch.mock.calls;
  const lastCall = allCalls[allCalls.length - 1];
  return JSON.parse(lastCall[1].body);
};
```

5. Open package.json and add the `restoreMocks` property, which ensures the `global.fetch` spy is reset to its original setting after each test. The code is illustrated in the following snippet:

```
"jest": {  
  ...,  
  "restoreMocks": true  
}
```

6. That should be it for your `global.fetch` spy. You can delete the `afterEach` block, the `fetchSpy` variable declaration, and the `originalFetch` constant definition.
7. Let's move on to `saveSpy`. Back in your `CustomerForm` test suite, find the `notifies onSave` when `form` is `submitted` test. Update it as shown in the following code snippet. We're replacing the use of `spy()` with `jest.fn()`. Notice how we no longer need to set the `onSave` prop to `saveSpy.fn` but just `saveSpy` itself:

```
it("notifies onSave when form is submitted", async () =>  
{  
  const customer = { id: 123 };  
  global.fetch.mockResolvedValue(  
    fetchResponseOk(customer)  
  );  
  const saveSpy = jest.fn();  
  render(  
    <CustomerForm  
      original={blankCustomer}  
      onSave={saveSpy}  
    />  
  );  
  await clickAndWait(submitButton());  
  expect(saveSpy).toBeCalledWith(customer);  
});
```

8. Repeat for the `does not notify onSave if the POST request returns an error` test.
9. Delete your `spy` function definition at the top of the test suite.
10. Delete your `toBeCalledWith` matcher in `test/domMatchers.js`.

11. We're now close to a working test suite. Try running your tests—you'll see the following error:

```
Cannot spy the fetch property because it is not a
function; undefined given instead
```

12. To fix this, we need to let Jest think that `global.fetch` is indeed a function. The simplest way to do this is to set a dummy implementation when your test suite launches. Create a `test/globals.js` file and add the following definition to it:

```
global.fetch = () => Promise.resolve({});
```

13. Now, back in `package.json`, add that file to the `setupFilesAfterEnv` property, like so:

```
"setupFilesAfterEnv": [
  "./test/domMatchers.js",
  "./test/globals.js"
],
```

14. Run all tests with `npm test`. They should be passing.

15. There's just one final cleanup to do. Find the following expectation:

```
expect(saveSpy).not.toBeCalledWith();
```

As mentioned earlier in the chapter, this expectation is not correct, and we only used it because our hand-rolled matcher didn't fully support this use case. What we want is for the expectation to fail if `onSave` is called in any form. Now that we're using Jest's own matchers, we can solve this more elegantly. Replace this expectation with the following code:

```
expect(saveSpy).not.toBeCalled();
```

Your `CustomerForm` test suite is now fully migrated. We'll end this chapter by extracting some more helpers.

Extracting fetch test functionality

`CustomerForm` is not the only component that will call `fetch`: one of the exercises is to update `AppointmentForm` to also submit appointments to the server. It makes sense to reuse the common code we've used by pulling it out into its own module. Proceed as follows:

1. Create a file named `test/spyHelpers.js` and add the following function definition, which is the same as the function in your test suite, except this time it's marked as an export:

```
export const bodyOfLastFetchRequest = () => {
  const allCalls = global.fetch.mock.calls;
  const lastCall = allCalls[allCalls.length - 1];
```

```
    return JSON.parse(lastCall[1].body) ;
}
```

2. Create a file named `test/builders/fetch.js` and add the following two functions to it:

```
export const fetchResponseOk = (body) =>
  Promise.resolve({
    ok: true,
    json: () => Promise.resolve(body),
  });

export const fetchResponseError = () =>
  Promise.resolve({ ok: false });
```

3. Delete those definitions from within `test/CustomerForm.test.js` and replace them with an `import` statement, as illustrated in the following code snippet. After this change, run your tests and check they are still passing:

```
import { bodyOfLastFetchRequest } from "./spyHelpers";
import {
  fetchResponseOk,
  fetchResponseError,
} from "./builders/fetch";
```

4. Finally, we can simplify `fetchResponseOk` and `fetchResponseError` by removing the call to `Promise.resolve` shown here. That's because Jest's `mockResolvedValue` function will automatically wrap the value in a promise:

```
export const fetchResponseOk = (body) => ({
  ok: true,
  json: () => Promise.resolve(body),
});

export const fetchResponseError = () => ({
  ok: false,
});
```

5. Ensure you've run all tests and you're on green before continuing.

You're now ready to reuse these functions in the `AppointmentForm` test suite.

Summary

We've just explored test doubles and how they are used to verify interactions with collaborating objects, such as component props (`onSave`) and browser API functions (`global.fetch`). We looked in detail at spies and stubs, the two main types of doubles you'll use. You also saw how to use a side-by-side implementation as a technique to keep your test failures under control while you switch from one implementation to another.

Although this chapter covered the primary patterns you'll use when dealing with test doubles, we have one major one still to cover, and that's how to spy on and stub out React components. That's what we'll look at in the next chapter.

Exercises

Try the following exercises:

1. Add a test to the `CustomerForm` test suite to specify that the error state is cleared when the form is submitted a second time with all validation errors corrected.
2. Update the `AppointmentForm` test suite to use `jest.fn()` and `jest.spyOn()`.
3. Extend `AppointmentForm` so that it submits an appointment using a POST request to `/appointments`. The `/appointments` endpoint returns a `201 Created` response without a body, so you don't need to call `json` on the response object or send back any parameters to `onSave`.

Further reading

For more information, refer to the following sources:

- A cheat sheet showing all the Jest mocking constructs you'll need for testing React code bases
<https://reacttdd.com/mockng-cheatsheet>
- A good introduction to the different kinds of test doubles
<https://martinfowler.com/articles/mocksArentStubs.html>
- An introduction to using the Fetch API
<https://github.github.io/fetch>
- Information on the ARIA alert role:
https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/alert_role

7

Testing `useEffect` and Mocking Components

In the previous chapter, you saw how test doubles can be used to verify network requests that occur upon user actions, such as clicking a submit button. We can also use them to verify side effects when our components mount, like when we're fetching data from the server that the component needs to function. In addition, test doubles can be used to verify the rendering of child components. Both use cases often occur together with **container components**, which are responsible for simply loading data and passing it to another component for display.

In this chapter, we'll build a new component, `AppointmentsDayViewLoader`, that loads the day's appointments from the server and passes them to the `AppointmentsDayView` component that we implemented in *Chapter 2, Rendering Lists and Detail Views*. By doing so, the user can view a list of appointments occurring today.

In this chapter, we will cover the following topics:

- Mocking child components
- Fetching data on mount with `useEffect`
- Variants of the `jest.mock` call

These are likely the most difficult tasks you'll encounter while test-driving React components.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter07>

Mocking child components

In this section, we're going to use the `jest.mock` test helper to replace the child component with a dummy implementation. Then, we'll write expectations that check whether we passed the right props to the child component and that it is correctly rendered on the screen.

But first, let's take a detailed look at how mocked components work.

How to mock components, and why?

The component we're going to build in this chapter has the following shape:

```
export const AppointmentsDayViewLoader = ({ today }) => {
  const [appointments, setAppointments] = useState([]);

  useEffect(() => {
    // fetch data from the server
    const result = await global.fetch(...);

    // populate the appointments array
    setAppointments(await result.json());
  }, [today]);

  return (
    <AppointmentsDayView appointments={appointments} />
  );
}
```

Its purpose is to display all the current appointments for a given day. This information is then passed into the component as the `today` prop. The component's job is to fetch data from the server and then pass it to the `AppointmentsDayView` component, which we built previously and already tested.

Think about the tests we may need. First, we'd want a test to prove that `AppointmentsDayView` loads with no appointments shown initially. Then, we'd want some tests that verify our `global.fetch` call is called successfully, and the returned data is passed into `AppointmentsDayView`.

How do we test that `AppointmentsDayView` is called with the right data? We could repeat some of the tests we have already written in the test suite for `AppointmentsDayView` – for example, by testing that a list of appointments is displayed, and that the relevant appointment data is shown.

However, we'd then be introducing repetition into our test suites. If we modify how `AppointmentsDayView` works, we'll have two places to update tests.

An alternative is to mock the component with a spy object. For this, we can use the `jest.mock` function, in tandem with a spy. This is how it will look:

```
jest.mock("../src/AppointmentsDayView", () => ({
  AppointmentsDayView: jest.fn(() => (
    <div id="AppointmentsDayView" />
  )),
}));
```

The first argument to the function is the file path that is being mocked. It must match the path that's passed to the `import` statement. This function is mocking the entire module:

```
import { MyComponent } from "some/file/path";

jest.mock("/some/file/path", ...);

describe("something that uses MyComponent", () => {
});
```

In the preceding code, Jest hoists this call to the top of the file and hooks into import logic so that when the `import` statement is run, your mock is returned instead.

Any time `AppointmentsDayView` is referenced in either the test suite or the component under test, you'll get this mock value rather than the real component. Instead of rendering our day view, we'll get a single `div` with an `id` value of `AppointmentsDayView`.

The second parameter is the **module factory parameter**. This is a factory method that is invoked when the mock is imported. It should return a set of named exports – in our case, this means a single component, `AppointmentsDayView`.

Because the mock definition is hoisted to the top of the file, you can't reference any variables in this function: they won't have been defined by the time your function is run. However, you can write JSX, as we have done here!

The complexity of component mock setup

This code is super cryptic, I know. Thankfully, you generally just need to write it once. I often find myself copy-pasting mocks when I need to introduce a new one into a test suite. I'll look up a previous one I wrote in some other test suite and copy it across, changing the relevant details.

So, now comes the big question: why would you want to do this?

Firstly, using mocks can improve test organization by encouraging multiple test suites with independent surface areas. If both a parent component and its child component are non-trivial components, then having two separate test suites for those components can help reduce the complexity of your test suites.

The parent component's test suite will contain just a handful of tests to prove that the child component was rendered and passed the expected prop value.

By mocking out the child component in the parent component's test suite, you are effectively saying, *"I want to ignore this child component right now, but I promise I'll test its functionality elsewhere!"*

A further reason is that you may already have tests for the child component. This is the scenario we find ourselves in: we already have tests for `AppointmentsDayView`, so unless we want to repeat ourselves, it makes sense to mock out the component wherever it's used.

An extension of this reason is the use of library components. Because someone else built them, you have reason to trust that they've been tested and do the right thing. And since they're library components, chances are they do something quite complex anyway, so rendering them within your tests may have unintended side effects.

Perhaps you have a library component that builds all sorts of elaborate HTML widgets and you don't want your test code to know that. Instead, you can treat it as a **black box**. In that scenario, it's preferable to verify the prop values that are passed to the component, again trusting that the component works as advertised.

Library components often have complex component APIs that allow the component to be configured in many ways. Mocking the component allows you to write contract tests that ensure you're setting up props correctly. We'll see this later in *Chapter 11, Test-Driving React Router*, when we mock out React Router's `Link` component.

The final reason to mock components is if they have side effects on mount, such as performing network requests to pull in data. By mocking out the component, your test suite does not need to account for those side effects. We'll do this in *Chapter 8, Building an Application Component*.

With all that said, let's start building our new component.

Testing the initial component props

We'll start by building a test suite for the new component:

1. Create a new file, `test/AppointmentsDayViewLoader.js`, and add all the following imports. We're importing not just the component under test (`AppointmentsDayViewLoader`) but also the child component we'll be mocking (`AppointmentsDayView`):

```
import React from "react";
```

```
import {
  initializeReactContainer,
  render,
  element,
} from "./reactTestExtensions";
import {
  AppointmentsDayViewLoader
} from "../src/AppointmentsDayViewLoader";
import {
  AppointmentsDayView
} from "../src/AppointmentsDayView";
```

2. Add the mock setup, just below the imports:

```
jest.mock("../src/AppointmentsDayView", () => ({
  AppointmentsDayView: jest.fn(() => (
    <div id="AppointmentsDayView" />
  )),
}));
```

3. Start with the first test, as shown here. This checks that the component we just mocked out is rendered. The mock renders a `div` element with an `id` value of `AppointmentsDayView`. The test looks up the `id` value using the `element` helper and checks that it isn't null:

```
describe("AppointmentsDayViewLoader", () => {
  beforeEach(() => {
    initializeReactContainer();
  });

  it("renders an AppointmentsDayView", () => {
    await render(<AppointmentsDayViewLoader />);
    expect(
      element("#AppointmentsDayView")
    ).not.toBeNull();
  });
});
```

Use of the ID attribute

If you have experience with **React Testing Library**, you may have come across the use of `data-testid` for identifying components. If you want to use these mocking techniques with React Testing Library, then you can use `data-testid` instead of the `id` attribute, and then find your element using the `queryByTestId` function.

Although it's generally recommended not to rely on `data-testid` for selecting elements within your test suites, that doesn't apply to mock components. You need IDs to be able to tell them apart because you could end up with more than a few mocked components all rendered by the same parent. Giving an ID to each component is the simplest way to find them for these DOM presence tests. Remember that the mocks will never make it outside of your unit testing environment, so there's no harm in using IDs.

For more discussions on mocking strategies with React Testing Library, head over to <https://reacttdd.com/mock-with-react-testing-library>.

4. Let's make that test pass. Create a new file, `src/AppointmentsDayViewLoader.js`, and go ahead and fill in the implementation, as follows. It does nothing but render the component, which is all the test asked for:

```
import React from "react";
import {
  AppointmentsDayView
} from "./AppointmentsDayView";

export const AppointmentsDayViewLoader = () => (
  <AppointmentsDayView />
);
```

5. Time for the next test. We'll check whether the initial value of the props sent to `AppointmentsDayView` is what we expect. We'll do this by using the `toBeCalledWith` matcher, which we've used already. Notice the second parameter value of `expect.anything()`: that's needed because React passes a second parameter to the component function when it's rendered. You'll never need to be concerned with this in your code – it's an internal detail of React's implementation – so we can safely ignore it. We'll use `expect.anything` to assert that we don't care what that parameter is:

```
it("initially passes empty array of appointments to AppointmentsDayView", () => {
  await render(<AppointmentsDayViewLoader />);

  expect(AppointmentsDayView).toBeCalledWith(
    { appointments: [] },
```

```
    expect.anything()
  ) ;
}) ;
```

Verifying props and their presence in the DOM

It's important to test both props that were passed to the mock and that the stubbed value is rendered in the DOM, as we have done in these two tests. In *Chapter 8, Building an Application Component*, we'll see a case where we want to check that a mocked component is unmounted after a user action.

6. Make that pass by updating your component definition, as shown here:

```
export const AppointmentsDayViewLoader = () => (
  <AppointmentsDayView appointments={[]} />
);
```

You've just used your first mocked component! You've seen how to create the mock, and the two types of tests needed to verify its use. Next, we'll add a `useEffect` hook to load data when the component is mounted and pass it through to the `appointments` prop.

Fetching data on mount with `useEffect`

The appointment data we'll load comes from an endpoint that takes start and end dates. These values filter the result to a specific time range:

```
GET /appointments/<from>-<to>
```

Our new component is passed a `today` prop that is a `Date` object with the value of the current time. We will calculate the `from` and `to` dates from the `today` prop and construct a URL to pass to `global.fetch`.

To get there, first, we'll cover a bit of theory on testing the `useEffect` hook. Then, we'll implement a new `renderAndWait` function, which we'll need because we're invoking a promise when the component is mounted. Finally, we'll use that function in our new tests, building out the complete `useEffect` implementation.

Understanding the `useEffect` hook

The `useEffect` hook is React's way of running side effects. The idea is that you provide a function that will run each time any of the hook's dependencies change. That dependency list is specified as the second parameter to the `useEffect` call.

Let's take another look at our example:

```
export const AppointmentsDayViewLoader = ({ today }) => {
  useEffect(() => {
    // ... code runs here
  }, [today]);

  // ... render something
}
```

The hook code will run any time the `today` prop changes. This includes when the component first mounts. When we test-drive this, we'll start with an empty dependency list and then use a specific test to force a refresh when the component is remounted with a new `today` prop value.

The function you pass to `useEffect` should return another function. This function performs teardown: it is called any time the value changes, especially *before* the hook function is invoked again, enabling you to cancel any running tasks.

We'll explore this return function in detail in *Chapter 15, Adding Animation*. However, for now, you should be aware that this affects how we call promises. We can't do this:

```
useEffect(async () => { ... }, []);
```

Defining the outer function as `async` would mean it returns a promise, not a function. We must do this instead:

```
useEffect(() => {
  const fetchAppointments = async () => {
    const result = await global.fetch(...);
    setAppointments(await result.json());
  };

  fetchAppointments();
}, [today]);
```

When running tests, if you were to call `global.fetch` directly from within the `useEffect` hook, you'd receive a warning from React. It would alert you that the `useEffect` hook should not return a promise.

Using setters inside useEffect Hook functions

React guarantees that setters such as `setAppointments` remain static. This means they don't need to appear in the `useEffect` dependency list.

To get started with our implementation, we'll need to ensure our tests are ready for render calls that run promises.

Adding the `renderAndWait` helper

Just as we did with `clickAndWait` and `submitAndWait`, now we need `renderAndWait`. This will render the component and then wait for our `useEffect` hook to run, including any promise tasks.

To be clear, this function is necessary not because of the `useEffect` hook itself – just a normal sync `act` call would ensure that it runs – because of the promise that `useEffect` runs:

1. In `test/reactTestExtensions.js`, add the following function below the definition of `render`:

```
export const renderAndWait = (component) =>
  act(async () => (
    ReactDOM.createRoot(container).render(component)
  )
);
```

2. Update the import in your test suite so that it references this new function:

```
import {
  initializeReactContainer,
  renderAndWait,
  element,
} from "./reactTestExtensions";
```

3. Then, update the first test:

```
it("renders an AppointmentsDayView", async () => {
  await renderAndWait(<AppointmentsDayViewLoader />);
  expect(
    element("#AppointmentsDayView")
  ).not.toBeNull();
});
```

4. Add the second test, which checks that we send an empty array of appointments to `AppointmentsDayView` before the server has returned any data:

```
it("initially passes empty array of appointments to AppointmentsDayView", async () => {
  await renderAndWait(<AppointmentsDayViewLoader />);
```

```
    expect(AppointmentsDayView).toBeCalledWith(
      { appointments: [] },
      expect.anything()
    );
  });
}
```

Make sure to check that these tests are passing before you continue.

Adding the useEffect hook

We're about to introduce a `useEffect` hook with a call to `global.fetch`. We'll start by mocking that call using `jest.spyOn`. Then, we'll continue with the test:

1. Add the following new imports to the top of the test suite:

```
import { todayAt } from "./builders/time";
import { fetchResponseOk } from "./builders/fetch";
```

2. Define a sample set of appointments at the top of the `describe` block:

```
describe("AppointmentsDayViewLoader", () => {
  const appointments = [
    { startsAt: todayAt(9) },
    { startsAt: todayAt(10) },
  ];
  ...
});
```

3. To set up `global.fetch` so that it returns this sample array, modify the test suite's `beforeEach` block, as shown here:

```
beforeEach(() => {
  initializeReactContainer();
  jest
    .spyOn(global, "fetch")
    .mockResolvedValue(fetchResponseOk(appointments));
});
```

4. It's time for our test. We assert that when the component is mounted, we should expect to see a call to `global.fetch` being made with the right parameters. Our test calculates what the right parameter values should be – it should be from midnight today to midnight tomorrow:

```
it("fetches data when component is mounted", async () =>
{
  const from = todayAt(0);
  const to = todayAt(23, 59, 59, 999);

  await renderAndWait(
    <AppointmentsDayViewLoader today={today} />
  );

  expect(global.fetch).toBeCalledWith(
    `/appointments/${from}-${to}`,
    {
      method: "GET",
      credentials: "same-origin",
      headers: { "Content-Type": "application/json" },
    }
  );
}) ;
```

5. To make this test pass, first, we'll need to introduce a `useEffect` hook into the component file:

```
import React, { useEffect } from "react";
```

6. Now, we can update the component to make the call, as follows. Although this is a lot of code already, notice how we aren't making use of the `return` value yet: there's no state being stored and `AppointmentsDayView` still has its `appointments` prop set to an empty array. We'll fill that in later:

```
export const AppointmentsDayViewLoader = (
  { today }
) => {
  useEffect(() => {
    const from = today.setHours(0, 0, 0, 0);
    const to = today.setHours(23, 59, 59, 999);

    const fetchAppointments = async () => {
      await global.fetch(
        `/appointments/${from}-${to}`,
        {
          method: "GET",
          credentials: "same-origin",
          headers: { "Content-Type": "application/json" },
        }
      );
    };
  });
};
```

```
        method: "GET",
        credentials: "same-origin",
        headers: {
          "Content-Type": "application/json"
        },
      }
    );
  };

  fetchAppointments();
}, []);
}

return <AppointmentsDayView appointments={ [] } />;
};
```

7. Before continuing with the next test, let's set a default value for the `today` prop so that any callers don't need to specify this:

```
AppointmentsDayViewLoader.defaultProps = {
  today: new Date(),
};
```

8. The next test will ensure we use the return value of our `global.fetch` call. Notice how we use the `toHaveBeenCalledWith` matcher. This is important because the first render of the component will be an empty array. It's the second call that will contain data:

```
it("passes fetched appointments to AppointmentsDayView once they have loaded", async () => {
  await renderAndWait(<AppointmentsDayViewLoader />);

  expect(
    AppointmentsDayView
  ).toHaveBeenCalledWith(
    { appointments },
    expect.anything()
  );
});
```

9. To make that pass, first, update your component's import to pull in the `useState` function:

```
import React, { useEffect, useState } from "react";
```

10. Now, update your component's definition, as shown here:

```
export const AppointmentsDayViewLoader = (
  { today }
) => {
  const [
    appointments, setAppointments
  ] = useState([]);

  useEffect(() => {
    ...
    const fetchAppointments = async () => {
      const result = await global.fetch(
        ...
      );
      setAppointments(await result.json());
    };

    fetchAppointments();
  }, []);

  return (
    <AppointmentsDayView
      appointments={appointments}
    />
  );
};
```

This completes the basic `useEffect` implementation – our component is now loading data. However, there's a final piece we must address with the `useEffect` implementation.

Testing the useEffect dependency list

The second parameter to the `useEffect` call is a dependency list that defines the variables that should cause the effect to be re-evaluated. In our case, the `today` prop is the important one. If the component is re-rendered with a new value for `today`, then we should pull down new appointments from the server.

We'll write a test that renders a component twice. This kind of test is very important any time you're using the `useEffect` hook. To support that, we'll need to adjust our render functions to ensure they only create one root:

1. In `test/reactTestExtensions.js`, add a new top-level variable called `reactRoot` and update `initializeReactContainer` to set this variable:

```
export let container;

let reactRoot;

export const initializeReactContainer = () => {
  container = document.createElement("div");
  document.body.replaceChildren(container);

  reactRoot = ReactDOM.createRoot(container);
};
```

2. Now, update the definitions of `render` and `renderAndWait` so that they use this `reactRoot` variable. After making this change, you'll be able to re-mount components within a single test:

```
export const render = (component) =>
  act(() => reactRoot.render(component));

export const renderAndWait = (component) =>
  act(async () => reactRoot.render(component));
```

3. Back in your test suite, update `import` so that it includes `today`, `tomorrow`, and `tomorrowAt`. We'll use these in the next test:

```
import {
  today,
  todayAt,
  tomorrow,
  tomorrowAt
} from "./builders/time";
```

- Now, add the test. This renders the component twice, with two separate values for the `today` prop. Then, it checks whether `global.fetch` was called twice:

```
it("re-requests appointment when today prop changes",
  async () => {
    const from = tomorrowAt(0);
    const to = tomorrowAt(23, 59, 59, 999);

    await renderAndWait(
      <AppointmentsDayViewLoader today={today} />
    );
    await renderAndWait(
      <AppointmentsDayViewLoader today={tomorrow} />
    );

    expect(global.fetch).toHaveBeenCalledWith(
      `/appointments/${from}-${to}`,
      expect.anything()
    );
  });
}
```

- If you run the test now, you'll see that `global.fetch` is only being called once:

```
AppointmentsDayViewLoader ' re-requests appointment
when today prop changes

expect(
  jest.fn()
).toHaveBeenCalledWith(...expected)

Expected: "/appointments/1643932800000-1644019199999",
Anything
Received: "/appointments/1643846400000-1643932799999",
{"credentials": "same-origin", "headers": {"Content-
Type": "application/json"}, "method": "GET"}
```

- Making it pass is a one-word change. Find the second parameter of the `useEffect` call and change it from an empty array, as shown here:

```
useEffect(() => {
```

```
    ...
}, [today]);
```

That's it for the implementation of this component. In the next section, we'll clean up our test code with a new matcher.

Building matchers for component mocks

In this section, we'll introduce a new matcher, `toBeRenderedWithProps`, that simplifies the expectations for our mock spy object.

Recall that our expectations look like this:

```
expect(AppointmentsDayView).toBeCalledWith(
  { appointments },
  expect.anything()
);
```

Imagine if you were working on a team that had tests like this. Would a new joiner understand what that second argument, `expect.anything()`, is doing? Will *you* understand what this is doing if you don't go away for a while and forget how component mocks work?

Let's wrap that into a matcher that allows us to hide the second property.

We need *two* matchers to cover the common use cases. The first, `toBeRenderedWithProps`, is the one we'll work through in this chapter. The second, `toBeFirstRenderedWithProps`, is left as an exercise for you.

The matcher, `toBeRenderedWithProps`, will pass if the component is *currently rendered* with the given props. This function will be equivalent to using the `toHaveBeenCalledWith` matcher.

The essential part of this matcher is when it pulls out the last element of the `mock.calls` array:

```
const mockedCall =
  mockedComponent.mock.calls[
    mockedComponent.mock.calls.length - 1
];
```

The `mock.calls` array

Recall that every mock function that's created with `jest.spyOn` or `jest.fn` will have a `mock.calls` property, which is an array of all the calls. This was covered in *Chapter 6, Exploring Test Doubles*.

The second matcher is `toBeFirstRenderedWithProps`. This will be useful for any test that checks the initial value of the child props and before any `useEffect` hooks have run. Rather than picking the last element of the `mock.calls` array, we'll just pick the first:

```
const mockedCall = mockedComponent.mock.calls[0];
```

Let's get started with `toBeRenderedWithProps`:

1. Create a new matcher test file at `test/matchers/toBeRenderedWithProps.test.js`. Add the following imports:

```
import React from "react";
import {
  toBeRenderedWithProps,
} from "./toBeRenderedWithProps";
import {
  initializeReactContainer,
  render,
} from "../reactTestExtensions";
```

2. Add the following test setup. Since our tests will be operating on a spy function, we can set that up in our `beforeEach` block, as shown here:

```
describe("toBeRenderedWithProps", () => {
  let Component;

  beforeEach(() => {
    initializeReactContainer();
    Component = jest.fn(() => <div />);
  });
});
```

3. As usual, our first test is to check that `pass` returns `true`. Notice how we must render the component before we call the matcher:

```
it("returns pass is true when mock has been rendered", () => {
  render(<Component />);
  const result = toBeRenderedWithProps(Component, {});
  expect(result.pass).toBe(true);
});
```

4. To make this pass, create a new file for the matcher, `test/matchers/toBeRenderedWithProps.js`, and add the following implementation:

```
export const toBeRenderedWithProps = (
  mockedComponent,
  expectedProps
) => ({ pass: true });
```

5. It's time to triangulate. For the next test, let's check that `pass` is `false` when we don't render the component before calling it:

```
it("returns pass is false when the mock has not been
rendered", () => {
  const result = toBeRenderedWithProps(Component, {});
  expect(result.pass).toBe(false);
});
```

6. To get the test to green, all we've got to do is check that the mock was called at least once:

```
export const toBeRenderedWithProps = (
  mockedComponent,
  expectedProps
) => ({
  pass: mockedComponent.mock.calls.length > 0,
});
```

7. Next, we'll need to check that `pass` is `false` if the props don't match. We can't write the opposite test – that `pass` is `true` if the props match – because that test would already pass, given our current implementation:

```
it("returns pass is false when the properties do not
match", () => {
  render(<Component a="b" />);
  const result = toBeRenderedWithProps(
    Component, {
      c: "d",
    }
  );
  expect(result.pass).toBe(false);
});
```

8. For the component code, we'll use the `equals` function from inside the `expect-utils` package, which is already installed as part of Jest. This tests for deep equality but also allows you to make use of `expect` helpers such as `expect.anything` and `expect.objectContaining`:

```
import { equals } from "@jest/expect-utils";

export const toBeRenderedWithProps = (
  mockedComponent,
  expectedProps
) => {
  const mockedCall = mockedComponent.mock.calls[0];
  const actualProps = mockedCall ?
    mockedCall[0] : null;
  const pass = equals(actualProps, expectedProps);
  return { pass };
};
```

9. For our final test, we want an example that shows that this matcher works that the expectation will match on the last rendering of the mock:

```
it("returns pass is true when the properties of the last
render match", () => {
  render(<Component a="b" />);
  render(<Component c="d" />);
  const result = toBeRenderedWithProps(
    Component,
    { c: "d" }
  );
  expect(result.pass).toBe(true);
});
```

10. To make that pass, we need to update the implementation so that it chooses the last element of the `mock.calls` array, rather than the first:

```
export const toBeRenderedWithProps = (
  mockedComponent,
  expectedProps
) => {
  const mockedCall =
    mockedComponent.mock.calls[
```

```
    mockedComponent.mock.calls.length - 1
  ];
  ...
};


```

11. We'll leave our implementation here. Completing the tests for the message property is left as an exercise for you, but they follow the same order as the tests shown in *Chapter 3, Refactoring the Test Suite*. For now, move to `test/domMatchers.js` and register your new matcher:

```
import {
  toBeRenderedWithProps,
} from "./matchers/toBeRenderedWithProps";

expect.extend({
  ...,
  toBeRenderedWithProps,
}) ;


```

12. Finally, back in your test suite, update the test that checks the `appointments` prop. It should look as follows; it's much nicer now that the `expect.anything` parameter value has gone:

```
it("passes fetched appointments to AppointmentsDayView once they have loaded", async () => {
  await renderAndWait(<AppointmentsDayViewLoader />);
  expect(AppointmentsDayView).toBeRenderedWithProps({
    appointments,
  });
}) ;


```

With that, you've learned how to build a matcher for component mocks, which reduces the verbiage that we originally had when we used the built-in `toBeCalledWith` matcher.

The other test in this test suite needs a second matcher, `toBeFirstRenderedWithProps`. The implementation of this is left as an exercise for you.

In the next section, we'll look at a variety of ways that component mocks can become more complicated.

Variants of the `jest.mock` call

Before we finish up this chapter, let's take a look at some variations on the `jest.mock` call that you may end up using.

The key thing to remember is to *keep your mocks as simple as possible*. If you start to feel like your mocks need to become more complex, you should treat that as a sign that your components are overloaded and should be broken apart in some way.

That being said, there are cases where you must use different forms of the basic component mock.

Removing the spy function

To begin with, you can simplify your `jest.mock` calls by not using `jest.fn`:

```
jest.mock("../src/AppointmentsDayView", () => ({
  AppointmentsDayView: () => (
    <div id="AppointmentsDayView" />
  ) ,
})) ;
```

With this form, you've set a stub return value, but you won't be able to spy on any props. This is sometimes useful if, for example, you've got multiple files that are testing this same component but only some of them verify the mocked component props. It can also be useful with third-party components.

Rendering the children of mocked components

Sometimes, you'll want to render grandchild components, skipping out the child (their parent). This often happens, for example, when a third-party component renders a complex UI that is difficult to test: perhaps it loads elements via the shadow DOM, for example. In that case, you can pass `children` through your mock:

```
jest.mock("../src/AppointmentsDayView", () => ({
  AppointmentsDayView: jest.fn(({ children }) => (
    <div id="AppointmentsDayView">{children}</div>
  )) ,
})) ;
```

We will see examples of this in *Chapter 11, Test-Driving React Router*.

Checking multiple instances of the rendered component

There are occasions when you'll want to mock a component that has been rendered multiple times into the document. How can you tell them apart? If they have a unique ID prop (such as `key`), you can use that in the `id` field:

```
jest.mock("../src/AppointmentsDayView", () => ({
```

```
AppointmentsDayView: jest.fn(({ key }) => (
  <div id={`AppointmentsDayView${key}`}>
    </div>
)) ;
```

Approach with caution!

One of the biggest issues with mocking components is that mock definitions can get out of control. But mock setup is complicated and can be very confusing. Because of this, you should avoid writing anything but the simplest mocks.

Thankfully, most of the time, the plain form of component mock is all you'll need. These variants are useful occasionally but should be avoided.

We'll see this variation in action in *Chapter 11, Test-Driving React Router*.

Alternatives to module mocks

Mocking out an entire module is fairly heavy-handed. The mock you set up must be used for all the tests in the same test module: you can't mix and match tests, some using the mock and some not. If you wanted to do this with `jest.mock`, you'd have to create two test suites. One would have the mock and the other wouldn't.

You also have the issue that the mock is at the module level. You can't just mock out one part of the module. Jest has functions that allow you to reference the original implementation called `requireActual`. For me, that involves moving into the danger zone of overly complex test doubles, so I refrain from using it – I have encountered a use case that needed it.

However, there are alternatives to using `jest.mock`. One is shallow rendering, which utilizes a special renderer that renders a single parent component, ignoring all child components other than standard HTML elements. In a way, this is even more heavy-handed because *all* your components end up mocked out.

For CommonJS modules, you can also overwrite specific exports inside modules, simply by assigning new values to them! This gives you a much more granular way of setting mocks at the test level. However, this is not supported in **ECMAScript**, so in the interests of maximum capability, you may want to avoid this approach.

For examples of these alternative approaches and a discussion on when you may want to use them, take a look at <https://reactddd.com/alternatives-to-module-mocks>.

Summary

This chapter covered the most complex form of mocking: setting up component mocks with `jest.mock`.

Since mocking is a complex art form, it's best to stick with a small set of established patterns, which I've shown in this chapter. You can also refer to the code in *Chapter 11, Test-Driving React Router*, for examples that show some of the variations that have been described in this chapter.

You also learned how to test-drive a `useEffect` hook before writing another matcher.

You should now feel confident with testing child components by using component mocks, including loading data into those components through `useEffect` actions.

In the next chapter, we'll extend this technique further by pulling out callback props from mock components and invoking them within our tests.

Exercises

The following are some exercises for you to try out:

1. Complete the message property tests on the `toBeRenderedWithProps` matcher.
2. Add the `toBeFirstRenderedWithProps` matcher and update your test suite to use this matcher. Since this matcher is very similar to `toBeRenderedWithProps`, you can add it to the same module file that contains the `toBeRenderedWithProps` matcher. You can also try to factor out any shared code into its own function that both matchers can use.
3. Add a `toBeRendered` matcher that checks if a component was rendered without checking its props.
4. Complete the matchers you've written so that they throw an exception if the passed argument is not a Jest mock.
5. Create a new component, `AppointmentFormLoader`, that calls the `GET /availableTimeSlots` endpoint when mounted. It should render an `AppointmentForm` component with its `appointments` prop set to the data returned from the server.

Further reading

To learn how to mock components without relying on `jest.mock`, please check out <https://reacttdd.com/alternatives-to-module-mocks>.

8

Building an Application Component

The components you've built so far have been built in isolation: they don't fit together, and there's no workflow for the user to follow when they load the application. Up to this point, we've been manually testing our components by swapping them in and out of our index file, `src/index.js`.

In this chapter, we'll tie all those components into a functioning system by creating a root application component, `App`, that displays each of these components in turn.

You have now seen almost all the TDD techniques you'll need for test-driving React applications. This chapter covers one final technique: testing callback props.

In this chapter, we will cover the following topics:

- Formulating a plan
- Using state to control the active view
- Test-driving callback props
- Making use of callback values

By the end of this chapter, you'll have learned how to use mocks to test the root component of your application, and you'll have a working application that ties together all the components you've worked on in *Part 1* of this book.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter08>

Formulating a plan

Before we jump into the code for the App component, let's do a little up-front design so that we know what we're building.

The following diagram shows all the components you've built and how App will connect them:

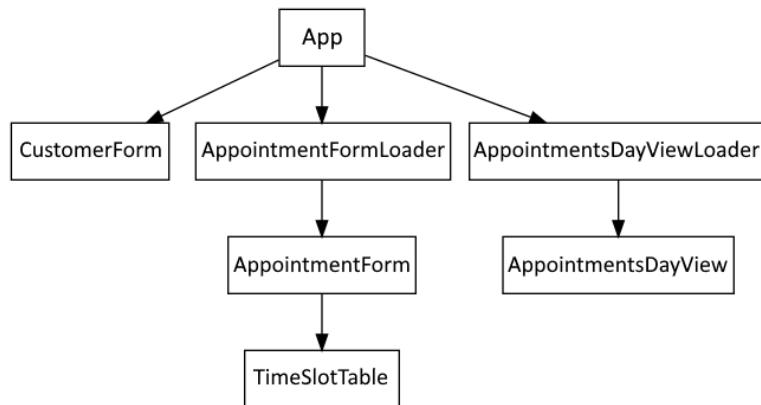


Figure 8.1 – The component hierarchy

Here's how it'll work:

1. When the user first loads the app, they will see a list of today's appointments using the `AppointmentsDayView` component, which will have its appointment data populated by its container `AppointmentsDayViewLoader` component.
2. At the top of the screen, the user will see a button labeled **Add customer and appointment**. Clicking that button makes `AppointmentsDayView` disappear and `CustomerForm` appear.
3. When the form is filled out and the submit button is clicked, the user is shown `AppointmentForm` and can add a new appointment for that customer.
4. Once they've added the appointment, they'll be taken back to `AppointmentsDayView`.

This first step is shown in the following screenshot. Here, you can see the new button in the top-left corner. The `App` component will render this button and then orchestrate this workflow:

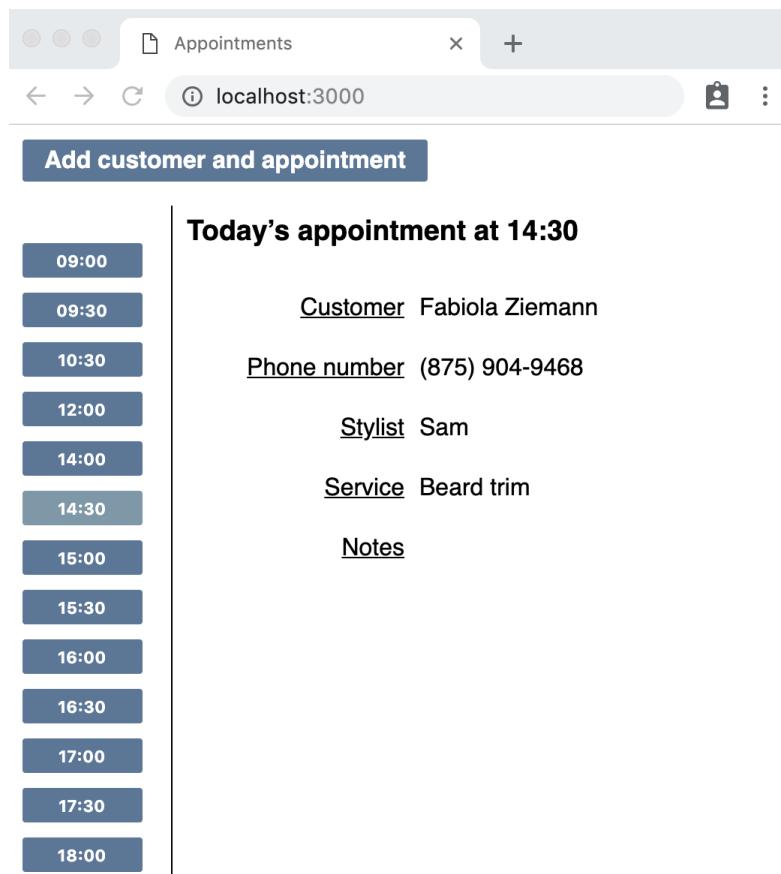


Figure 8.2 – The app showing the new button in the top-left corner

This is a very simple workflow that supports just a single use case: adding a new customer and an appointment at the same time. Later in this book, we'll add support for creating appointments for existing customers.

With that, we're ready to build the new App component.

Using state to control the active view

In this section, we'll start building a new App component, in the usual way. First, we'll display an `AppointmentsDayViewLoader` component. Because this child component makes a network request when mounted, we'll mock it out. Then, we'll add a button inside a menu element, at the top of the page. When this button is clicked, we switch out the `AppointmentsDayViewLoader` component for a `CustomerForm` component.

We will introduce a state variable named `view` that defines which component is currently displayed. Initially, it will be set to `dayView`. When the button is clicked, it will change to `addCustomer`.

The JSX constructs will initially use a ternary to switch between these two views. Later, we'll add a third value called `addAppointment`. When we do that, we'll "upgrade" our ternary expression to a `switch` statement.

To get started, follow these steps:

1. Create a new file, `test/App.test.js`, with the following imports for the new `App` component:

```
import React from "react";
import {
  initializeReactContainer,
  render,
} from "./reactTestExtensions";
import { App } from "../src/App";
```

2. Next, import `AppointmentsDayViewLoader` and mock its implementation:

```
import {
  AppointmentsDayViewLoader
} from "../src/AppointmentsDayViewLoader";

jest.mock("../src/AppointmentsDayViewLoader", () => ({
  AppointmentsDayViewLoader: jest.fn(() => (
    <div id="AppointmentsDayViewLoader" />
  )),
}));
```

3. Now, let's add our first test, which checks that `AppointmentsDayViewLoader` has been rendered:

```
describe("App", () => {
  beforeEach(() => {
    initializeReactContainer();
  });

  it("initially shows the AppointmentDayViewLoader", () => {
    render(<App />);
```

```
    expect(AppointmentsDayViewLoader).toBeRendered();
  });
});
```

4. Make that pass by adding the following code to a new file, `src/App.js`:

```
import React from "react";
import ReactDOM from "react-dom";
import {
  AppointmentsDayViewLoader
} from "./AppointmentsDayViewLoader";

export const App = () => (
  <AppointmentsDayViewLoader />
);
```

5. For the second test, we'll add a menu to the top of the page. For this, we'll need the element matcher, so add that to your test suite imports:

```
import {
  initializeReactContainer,
  render,
  element,
} from "./reactTestExtensions";
```

6. Add the second test:

```
it("has a menu bar", () => {
  render(<App />);
  expect(element("menu")).not.toBeNull();
});
```

7. To make that pass, change the `App` component so that it includes a `menu` element above the loader component:

```
export const App = () => (
  <>
    <menu />
    <AppointmentsDayViewLoader />
```

```
    </>
)
```

8. Next, we want to display a button in the menu that, when clicked, will switch to `CustomerForm`. Add the following test, which asserts that the button appears on the page, using a CSS selector to find the rendered button element. This uses the `:first-of-type` pseudoclass to ensure we find the first button (later in this book, we'll add a second button to the menu):

```
it("has a button to initiate add customer and appointment action", () => {
  render(<App />);
  const firstButton = element(
    "menu > li > button:first-of-type"
  );
  expect(firstButton).toContainText(
    "Add customer and appointment"
  );
}) ;
```

9. To make that pass, change the menu in the `App` component to the following:

```
<menu>
  <li>
    <button type="button">
      Add customer and appointment
    </button>
  </li>
</menu>
```

10. For the next test, we must check that clicking the button renders `CustomerForm`. We must also mock this component out. To do that, we'll need the component that's been imported into the test suite. Add the following line to `test/App.test.js`:

```
import { CustomerForm } from "../src/CustomerForm";
```

11. Just below that, add the following mock definition, which is our standard mock definition:

```
jest.mock("../src/CustomerForm", () => ({
  CustomerForm: jest.fn(() => (
    <div id="CustomerForm" />
  )),
})) ;
```

Why mock a component that has no effects on mount?

This component already has a test suite so that we can use a test double and verify the right props to avoid re-testing functionality we've tested elsewhere. For example, the `CustomerForm` test suite has a test to check that the submit button calls the `onSave` prop with the saved customer object. So, rather than extending the test surface area of `App` so that it includes that submit functionality, we can mock out the component and call `onSave` directly instead. We'll do that in the next section.

12. To click the button, we'll need our click helper. Bring that in now:

```
import {
  initializeReactContainer,
  render,
  element,
  click,
} from "./reactTestExtensions";
```

13. Now, add the test. This introduces a helper function, `beginAddingCustomerAndAppointment`, which finds the button and clicks it. We'll pull that out now because we'll be using it in most of the remaining tests:

```
const beginAddingCustomerAndAppointment = () =>
  click(element("menu > li > button:first-of-type"));

it("displays the CustomerForm when button is clicked",
  async () => {
  render(<App />);
  beginAddingCustomerAndAppointment();
  expect(element("#CustomerForm")).not.toBeNull();
});
```

14. Making this pass involves adding a component state to track that we've clicked the button. In `src/App.js`, import the two hooks we'll need, `useState` and `useCallback`, and import `CustomerForm` too:

```
import React, { useState, useCallback } from "react";
import { CustomerForm } from "./CustomerForm";
```

15. In the `App` component, define the new view state variable and initialize it to the `dayView` string, which we'll use to represent `AppointmentsDayViewLoader`:

```
const [view, setView] = useState("dayView");
```

16. Just below that, add a new callback named `transitionToAddCustomer`, which we'll attach to the button's `onClick` handler in the next step. This callback updates the view state variable so that it points to the second page, which we'll call `addCustomer`:

```
const transitionToAddCustomer = useCallback(
  () => setView("addCustomer"),
  []
);
```

17. Plug that into the `onClick` prop of the button:

```
<button
  type="button"
  onClick={transitionToAddCustomer}>
  Add customer and appointment
</button>
```

18. Now, all that's left is to modify our JSX to ensure the `CustomerForm` component is rendered when the `view` state variable is set to `addCustomer`. Notice how the test doesn't force us to hide `AppointmentsDayViewLoader`. That will come in a later test. For now, we just need the simplest code that will make our test pass. Update your JSX, as shown here:

```
return (
  <>
  <menu>
    ...
  </menu>
  {view === "addCustomer" ? <CustomerForm /> : null}
</>
);
```

Testing for the presence of a new component

Strictly speaking, this *isn't* the simplest way to make the test pass. We could make it pass by *always* rendering a `CustomerForm` component, regardless of the value of `view`. Then, we'd need to triangulate with a second test that proves the component is not initially rendered. I'm skipping this step for brevity, but feel free to add it in if you prefer.

19. We need to ensure that we pass an `original` prop to `CustomerForm`. In this workflow, we're creating a new customer so that we can give it a blank customer object, just like the

one we used in the `CustomerForm` test suite. Add the following test below it. We'll define `blankCustomer` in the next step:

```
it("passes a blank original customer object to CustomerForm", async () => {
  render(<App />);
  beginAddingCustomerAndAppointment();
  expect(CustomerForm).toBeRenderedWithProps(
    expect.objectContaining({
      original: blankCustomer
    })
  );
});
```

20. Create a new file, `test/builders/customer.js`, and add a definition for `blankCustomer`:

```
export const blankCustomer = {
  firstName: "",
  lastName: "",
  phoneNumber: "",
};
```

21. Import that new definition into your App test suite:

```
import { blankCustomer } from "./builders/customer";
```

Value builders versus function builders

We've defined `blankCustomer` as a constant value, rather than a function. We can do this because all the code we've written treats variables as immutable objects. If that wasn't the case, we may prefer to use a function, `blankCustomer()`, that generates new values each time it is called. That way, we can be sure that one test doesn't accidentally modify the setup for any subsequent tests.

22. Let's make that test pass. First, define `blankCustomer` at the top of `src/App.js`:

```
const blankCustomer = {
  firstName: "",
  lastName: "",
  phoneNumber: "",
};
```

Using builder functions in both production and test code

You now have the same `blankCustomer` definition in both your production and test code. This kind of duplication is usually okay, especially since the object is so simple. But for non-trivial builder functions, you should consider test-driving the implementation and then making good use of it within your test suite.

23. Then, simply reference that value by setting it as an `original` prop of `CustomerForm`. After making this change, your test should be passing:

```
{view === "addCustomer" ? (
  <CustomerForm original={blankCustomer} />
) : null}
```

24. Next, add the following test to hide `AppointmentsDayViewLoader` when a customer is being added:

```
it("hides the AppointmentsDayViewLoader when button is clicked", async () => {
  render(<App />);
  beginAddingCustomerAndAppointment();
  expect(
    element("#AppointmentsDayViewLoader")
  ).toBeNull();
});
```

25. To make that pass, we need to move `AppointmentsDayViewLoader` into the ternary expression, in place of the null:

```
{ view === "addCustomer" ? (
  <CustomerForm original={blankCustomer} />
) : (
  <AppointmentsDayViewLoader />
)}
```

26. Let's hide the button bar, too:

```
it("hides the button bar when CustomerForm is being displayed", async () => {
  render(<App />);
  beginAddingCustomerAndAppointment();
```

```
    expect(element("menu")).toBeNull();
});
```

27. To solve this, we need to lift the ternary out of the JSX entirely, as shown in the following code. This is messy, but we'll improve its implementation in the next section:

```
return view === "addCustomer" ? (
  <CustomerForm original={blankCustomer} />
) : (
  <>
  <menu>
  ...
  </menu>
  <AppointmentsDayViewLoader />
</>
);
```

With that, you have implemented the initial step in the workflow – that is changing the screen from an `AppointmentsDayViewLoader` component to a `CustomerForm` component. You did this by changing the `view` state variable from `dayView` to `addCustomer`. For the next step, we'll use the `onSave` prop of `CustomerForm` to alert us when it's time to update `view` to `addAppointment`.

Test-driving callback props

In this section, we'll introduce a new extension function, `propsOf`, that reaches into a mocked child component and returns the props that were passed to it. We'll use this to get hold of the `onSave` callback prop value and invoke it from our test, mimicking what would happen if the real `CustomerForm` had been submitted.

It's worth revisiting why this is something we'd like to do. Reaching into a component and calling the prop directly seems complicated. However, the alternative is *more* complicated and *more* brittle.

The test we want to write next is the one that asserts that the `AppointmentFormLoader` component is shown after `CustomerForm` has been submitted and a new customer has been saved:

```
it("displays the AppointmentFormLoader after the CustomerForm is submitted", async () => {
  // ...
});
```

Now, imagine that we wanted to test this without a mocked `CustomerForm`. We would need to fill in the real `CustomerForm` form fields and hit the submit button. That may seem reasonable, but we'd be increasing the surface area of our App test suite to include the `CustomerForm` component. Any changes to the `CustomerForm` component would require not only the `CustomerForm` tests to be updated but also now the App tests. This is the exact scenario we'll see in *Chapter 9, Form Validation*, when we update `CustomerForm` so that it includes field validation.

By mocking the child component, we can reduce the surface area and reduce the likelihood of breaking tests when child components change.

Mocked components require care

Even with mocked components, our parent component test suite can still be affected by child component changes. This can happen if the meaning of the props changes. For example, if we updated the `onSave` prop on `CustomerForm` to return a different value, we'd need to update the App tests to reflect that.

Here's what we've got to do. First, we must define a `propsOf` function in our extensions module. Then, we must write tests that mimic the submission of a `CustomerForm` component and transfer the user to an `AppointmentFormLoader` component. We'll do that by introducing a new `addAppointment` value for the view state variable. Follow these steps:

1. In `test/reactTestExtensions.js`, add the following definition of `propsOf`. It looks up the last call to the mocked component and returns its props:

```
export const propsOf = (mockComponent) => {
  const lastCall = mockComponent.mock.calls[
    mockComponent.mock.calls.length - 1
  ];
  return lastCall[0];
};
```

2. Back in `test/App.test.js`, update the extensions import so that it includes `propsOf`:

```
import {
  initializeReactContainer,
  render,
  element,
  click,
  propsOf,
} from "./reactTestExtensions";
```

3. You also need to import the `act` function from React's test utils. Our test will wrap its invocation of the callback prop to ensure that any setters are run before the call returns:

```
import { act } from "react-dom/test-utils";
```

4. There's one more import to add – the import for `AppointmentFormLoader`:

```
import {  
  AppointmentFormLoader  
} from "../src/AppointmentFormLoader";
```

5. Just below that, define its mock using the standard component mock definition:

```
jest.mock("../src/AppointmentFormLoader", () => ({  
  AppointmentFormLoader: jest.fn(() => (  
    <div id="AppointmentFormLoader" />  
  )),  
})) ;
```

6. We're almost ready for our test. First, though, let's define a helper function, `saveCustomer`. This is the key part of the code that invokes the prop. Note that this sets a default customer object of `exampleCustomer`. We'll use this default value to avoid having to specify the customer in each test where the value doesn't matter:

```
const exampleCustomer = { id: 123 };  
  
const saveCustomer = (customer = exampleCustomer) =>  
  act(() => propsOf(CustomerForm).onSave(customer));
```

Using `act` within the test suite

This is the first occasion that we've willingly left a reference to `act` within our test suite. In every other use case, we managed to hide calls to `act` within our extensions module. Unfortunately, that's just not possible here – at least, it's not possible with the way we wrote `propsOf`. An alternative approach would be to write an extension function named `invokeProp` that takes the name of a prop and invokes it for us:

```
invokeProp(CustomerForm, "onSave", customer);
```

The downside of this approach is that you've now downgraded `onSave` from an object property to a string. So, we'll ignore this approach for now and just live with `act` usage in our test suite.

7. Let's write our test. We want to assert that `AppointmentsFormLoader` is displayed once `CustomerForm` has been submitted:

```
it("displays the AppointmentFormLoader after the CustomerForm is submitted", async () => {
    render(<App />);
    beginAddingCustomerAndAppointment();
    saveCustomer();

    expect(
        element("#AppointmentFormLoader")
    ).not.toBeNull();
}) ;
```

8. Making this pass will involve adding a new value to the view state variable, `addAppointment`. With this third value, the ternary expression is no longer fit for purpose because it can only handle two possible values of view. So, before we continue making this pass, let's refactor that ternary so that it uses a `switch` statement. Skip the test you just wrote using `it.skip`.

9. Replace the return statement of your component with the following:

```
switch (view) {
  case "addCustomer":
    return (
      <CustomerForm original={blankCustomer} />
    );
  default:
    return (
      <>
        <menu>
          <li>
            <button
              type="button"
              onClick={transitionToAddCustomer}>
              Add customer and appointment
            </button>
          </li>
        </menu>
      <AppointmentsDayViewLoader />
    );
}
```

```
    </>
  ) ;
}
```

10. Once you've verified that your tests still pass, un-skip your latest test by changing `it.skip` back to `it`.
11. The component should update the view to `addAppointment` whenever the `onSave` prop of `CustomerForm` is called. Let's do that with a new callback handler. Add the following code just below the definition of `transitionToAddCustomer`:

```
const transitionToAddAppointment = useCallback (
  () => {
    setView("addAppointment")
  }, []);
```

12. Modify the `CustomerForm` render expression so that it takes this as a prop:

```
<CustomerForm
  original={blankCustomer}
  onSave={transitionToAddAppointment}
/>
```

13. Hook up the new `addAppointment` value by adding the following `case` statement to the switch. After making this change, your test should be passing:

```
case "addAppointment":
  return (
    <AppointmentFormLoader />
  );
```

14. For the next test, we need to pass a value for the `original` prop, this time to `AppointmentFormLoader`. Note the double use of `expect.objectContaining`. This is necessary because our appointment is not going to be a simple blank appointment object. This time, the appointment will have a customer ID passed to it. That customer ID is the ID of the customer we've just added – we'll write a test for that next:

```
it("passes a blank original appointment object to
CustomerForm", async () => {
  render(<App />);
  beginAddingCustomerAndAppointment();
  saveCustomer();
```

```
expect(AppointmentFormLoader).toBeRenderedWithProps(  
  expect.objectContaining({  
    original:  
      expect.objectContaining(blankAppointment),  
    })  
  );  
) ;
```

15. We need a builder function, just like with `blankCustomer`. Create a new file, `test/builders/appointment.js`, and add the following definition:

```
export const blankAppointment = {  
  service: "",  
  stylist: "",  
  startsAt: null,  
};
```

16. Update the test code to import that:

```
import {  
  blankAppointment  
} from "./builders/appointment";
```

17. Then, create the same thing in `src/App.js`:

```
const blankAppointment = {  
  service: "",  
  stylist: "",  
  startsAt: null,  
};
```

18. Finally, you can make the test pass by setting the `original` prop, as shown here:

```
<AppointmentFormLoader original={blankAppointment} />
```

We're almost done with the display of `AppointmentFormLoader`, but not quite: we still need to take the customer ID we receive from the `onSave` callback and pass it into `AppointmentFormLoader`, by way of the `original` prop value, so that `AppointmentForm` knows which customer we're creating an appointment for.

Making use of callback values

In this section, we'll introduce a new state variable, `customer`, that will be set when `CustomerForm` receives the `onSave` callback. After that, we'll do the final transition in our workflow, from `addAppointment` back to `dayView`.

Follow these steps:

1. This time, we'll check that the new customer ID is passed to `AppointmentFormLoader`. Remember in the previous section how we gave `saveCustomer` a `customer` parameter? We'll make use of that in this test:

```
it("passes the customer to the AppointmentForm", async () => {
  const customer = { id: 123 };

  render(<App />);
  beginAddingCustomerAndAppointment();
  saveCustomer(customer);

  expect(AppointmentFormLoader).toBeRenderedWithProps(
    expect.objectContaining({
      original: expect.objectContaining({
        customer: customer.id,
      }),
    })
  );
});
```

2. For this to work, we'll need to add a state variable for the customer. Add the following at the top of the `App` component:

```
const [customer, setCustomer] = useState();
```

3. When we built the `onSave` prop of `CustomerForm` back in *Chapter 6, Exploring Test Doubles*, we passed it the updated customer object. Update the `transitionToAddAppointment` handler so that it takes this parameter value and saves it using the `setCustomer` setter:

```
const transitionToAddAppointment = useCallback(
  (customer) => {
    setCustomer(customer);
});
```

```

        setView("addAppointment")
    }, [] );

```

4. Pass that through to `AppointmentFormLoader` by creating a new `original` object value that merges the customer ID into `blankAppointment`:

```

case "addAppointment":
    return (
        <AppointmentFormLoader
            original={{
                ...blankAppointment,
                customer: customer.id,
            }}
        />
    );

```

5. It's time for the final test for this component. We complete the user workflow by asserting that once the appointment is saved, the view updates back to `dayView`:

```

const saveAppointment = () =>
    act(() => propsOf(AppointmentFormLoader).onSave());

it("renders AppointmentDayViewLoader after
AppointmentForm is submitted", async () => {
    render(<App />);
    beginAddingCustomerAndAppointment();
    saveCustomer();
    saveAppointment();
    expect(AppointmentsDayViewLoader).toBeRendered();
});

```

6. Define a new function to set the state back to `dayView`:

```

const transitionToDayView = useCallback(
    () => setView("dayView"),
    []
);

```

7. Pass this function to `AppointmentsFormLoader` to ensure it's called when the appointment is saved. After this, your tests should be complete and passing:

```
case "addAppointment":  
  return (  
    <AppointmentFormLoader  
      original={{  
        ...blankAppointment,  
        customer: customer.id,  
      }}  
      onSave={transitionToDayView}  
    />  
  ) ;
```

We're done!

Now, all that's left is to update `src/index.js` to render the `App` component. Then, you can manually test this to check out your handiwork:

```
import React from "react";  
import ReactDOM from "react-dom";  
import { App } from "./App";  
  
ReactDOM  
  .createRoot(document.getElementById("root"))  
  .render(<App />);
```

To run the application, use the `npm run serve` command. For more information see the *Technical requirements* section in *Chapter 6, Exploring Test Doubles*, or consult the `README.md` file in the repository.

Summary

This chapter covered the final TDD technique for you to learn – mocked component callback props. You learned how to get a reference to a component callback using the `propsOf` extension, as well as how to use a state variable to manage the transitions between different parts of a workflow.

You will have noticed how *all* the child components in `App` were mocked out. This is often the case with top-level components, where each child component is a relatively complex, self-contained unit.

In the next part of this book, we'll apply everything we've learned to more complex scenarios. We'll start by introducing field validation into our `CustomerForm` component.

Exercises

The following are some exercises for you to try out:

1. Update your `CustomerForm` and `AppointmentForm` tests to use the new builders you've created.
2. Add a test to `AppointmentForm` that ensures that the customer ID is submitted when the form is submitted.

Part 2 – Building Application Features

This part builds on the basic techniques you've learned in *Part 1* by applying them to real-world problems that you'll encounter in your work, and introduces libraries that many React developers use: React Router, Redux, and Relay (GraphQL). The goal is to show you how the TDD workflow can be used even for large applications.

This part includes the following chapters:

- *Chapter 9, Form Validation*
- *Chapter 10, Filtering and Searching Data*
- *Chapter 11, Test-Driving React Router*
- *Chapter 12, Test-Driving Redux*
- *Chapter 13, Test-Driving GraphQL*

9

Form Validation

For many programmers, TDD makes sense when it involves *toy* programs that they learn in a training environment. But they find it hard to join the dots when they are faced with the complexity of *real-world* programs. The purpose of this part of this book is for you to apply the techniques you've learned to real-world applications.

This chapter takes a somewhat self-indulgent journey into form validation. Normally, with React, you'd reach for a ready-made form library that handles validation for you. But in this chapter, we'll hand-craft our own validation logic, as an example of how real-world complexity can be conquered with TDD.

You will uncover an important architectural principle when dealing with frameworks such as React: take every opportunity to move logic out of framework-controlled components and into plain JavaScript objects.

In this chapter, we will cover the following topics:

- Performing client-side validation
- Handling server errors
- Indicating form submission status

By the end of the chapter, you'll have seen how tests can be used to introduce validation into your React forms.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter09>

Performing client-side validation

In this section, we'll update the `CustomerForm` and `AppointmentForm` components so that they alert the user to any issues with the text they've entered. For example, if they enter non-digit characters into the phone number field, the application will display an error.

We'll listen for the DOM's `blur` event on each field to take the current field value and run our validation rules on it.

Any validation errors will be stored as strings, such as `First name is required`, within a `validationErrors` state variable. Each field has a key in this object. An undefined value (or absence of a value) represents no validation error, and a string value represents an error. Here's an example:

```
{  
  firstName: "First name is required",  
  lastName: undefined,  
  phoneNumber: "Phone number must contain only numbers, spaces,  
  and any of the following: + - ( ) ."  
}
```

This error is rendered in the browser like this:

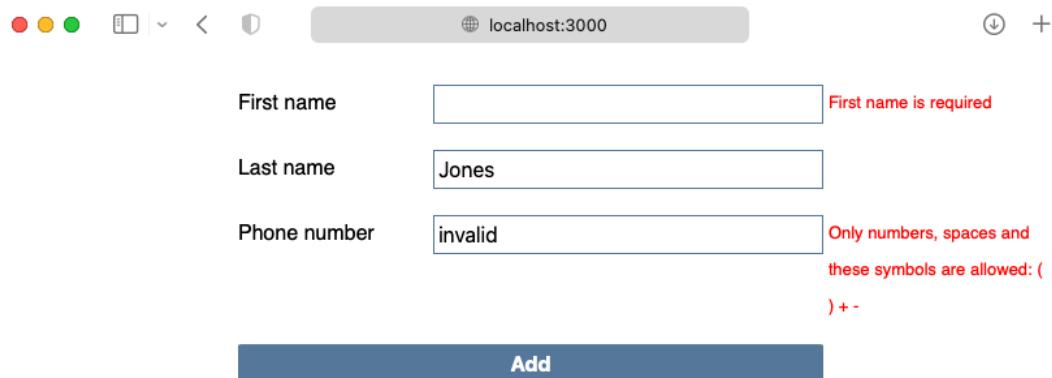


Figure 9.1 – Validation errors displayed to the user

To support tests that manipulate the keyboard focus, we need a new function that simulates the `focus` and `blur` events being raised when the user completes a field value. We'll call this function `withFocus`. It wraps a test-supplied action (such as changing the field value) with the `focus`/ `blur` events.

This section will start by checking that the `CustomerForm` first name field is supplied. Then, we'll generalize that validation so that it works for all three fields in the form. After that, we'll ensure validation also runs when the submit button is pressed. Finally, we'll extract all the logic we've built into a separate module.

Validating a required field

Each of the three fields on our page – `firstName`, `lastName`, and `phoneNumber` – are required fields. If a value hasn't been provided for any of the fields, the user should see a message telling them that. To do that, each of the fields will have an alert message area, implemented as a `span` with an ARIA role of `alert`.

Let's begin by adding that alert for the `firstName` field, and then making it operational by validating the field when the user removes focus:

1. Add the following new test to the bottom of the `CustomerForm` test suite. It should be within a new nested `describe` block named `validation`. This test checks that an alert space has been rendered. Notice the CSS selector: it's a bit of a hack. We're primarily interested in finding an element that matches `[role=alert]`. However, we are also qualifying it with the `firstNameError` ID since we'll eventually have multiple alert spaces – one for each field:

```
describe("validation", () => {
  it("renders an alert space for first name validation errors", () => {
    render(<CustomerForm original={blankCustomer} />);
    expect(
      element("#firstNameError [role=alert]")
    ).not.toBeNull();
  });
});
```

2. To make that pass, move to `src/CustomerForm.js` and add the following `span` definition, just below the `firstName` input field:

```
<input
  type="text"
  name="firstName"
  id="firstName"
  value={customer.firstName}
  onChange={handleChange}
/>
<span id="firstNameError" role="alert" />
```

3. Next, we want to check that the field has an `aria-describedby` field that points to the error alert. This helps screen readers understand the content of the page. Add the following new test at the bottom of the test suite:

```
it("sets alert as the accessible description for the first name field", async () => {
  render(<CustomerForm original={blankCustomer} />);
  expect(
    field(
      "firstName"
    ).getAttribute("aria-describedby")
  ).toEqual("firstNameError");
});
```

4. To make that pass, add the `aria-describedby` attribute to the `firstName` field definition:

```
<input
  type="text"
  name="firstName"
  id="firstName"
  value={customer.firstName}
  onChange={handleChange}
  aria-describedby="firstNameError"
/>
```

5. The next test we'll write will use the `blur` DOM event to trigger validation. For this test, we'll start by building a new test extension, `withFocus`, which calls the `focus` event to ensure the target element has focus, then runs an action – such as entering text into the focused field – and ends by invoking the `blur` event. In `test/reactTestExtensions.js`, add the following definition for the `withFocus` function:

```
export const withFocus = (target, fn) =>
  act(() => {
    target.focus();
    fn();
    target.blur();
  });
```

The focus and blur sequence

The initial call to `focus` is needed because if the element isn't focused, JSDOM will think that `blur` has nothing to do.

6. In `test/CustomerForm.test.js`, import the new `withFocus` function:

```
import {  
  ...,  
  withFocus,  
} from "./reactTestExtensions";
```

7. Add the following new test at the bottom of the test suite (still inside the `validation` nested `describe` block). It checks that if the user enters a blank name value, they'll see a message telling them that a value is required:

```
it("displays error after blur when first name field is  
blank", () => {  
  render(<CustomerForm original={blankCustomer} />);  
  
  withFocus(field("firstName"), () =>  
    change(field("firstName"), " ");  
  )  
  
  expect(  
    element("#firstNameError[role=alert]")  
  ).toContainText("First name is required");  
});
```

8. To make this pass, we need to hardcode the message:

```
<span id="firstNameError" role="alert">  
  First name is required  
</span>
```

9. Let's triangulate to replace the hardcoded. The following test asserts that the alert message is initially blank. Notice the use of `toEqual` rather than `not.toContainText`: this is forward planning. When we come to generalize this function in the next section, the alert text could be anything:

```
it("initially has no text in the first name field alert  
space", async () => {
```

```

    render(<CustomerForm original={blankCustomer} />);
    expect(
      element("#firstNameError[role=alert]").textContent
    ).toEqual("");
  });
}

```

A matcher for empty text content

Although not covered in this book, this would be a good opportunity to build a new matcher such as `toHaveNoText`, or maybe not `.toContainAnyText`.

10. To make this test pass, we'll add support for running validation rules within `CustomerForm`. Start by adding the following inline function definition at the top of `src/CustomerForm.js`, just below the imports but above the `CustomerForm` component definition. This is our first validation rule, `required`, which returns an error string if the supplied value is empty, and `undefined` otherwise:

```

const required = value =>
  !value || value.trim() === ""
    ? "First name is required"
    : undefined;

```

11. Within the `CustomerForm` component, define a `validationErrors` state variable, initially set to an empty object:

```

const [
  validationErrors, setValidationErrors
] = useState({});

```

12. Create a handler function inside `CustomerForm` that can be used when the user switches focus away from the first name field. It runs the `required` validation we defined in the first step, and then saves the response in the `validationErrors` state object:

```

const handleBlur = ({ target }) => {
  const result = required(target.value);
  setValidationErrors({
    ...validationErrors,
    firstName: result
  });
}

```

13. Next, define a function that the JSX will use to choose which message to display, named `hasFirstNameError`:

```
const hasFirstNameError = () =>
  validationErrors.firstName !== undefined;
```

14. All that's left is to modify our JSX so that it invokes the validation logic, and then displays the validation error. Use the following code to set the `onBlur` handler on the existing input field for `firstName` and to render the error text just after it. After this change, your test should be passing:

```
<input
  type="text"
  name="firstName"
  ...
  onBlur={handleBlur}
/>
<span id="firstNameError" role="alert">
  {hasFirstNameError()
    ? validationErrors["firstName"]
    : ""}
</span>
```

You now have a completed, working system for validating the first name field.

Generalizing validation for multiple fields

Next, we'll add the required validation to the last name and phone number fields.

Since we're on green, we can refactor our existing code *before* we write the next test. We will update the JSX and the `hasFirstNameError` and `handleBlur` functions so that they work for all the fields on the form.

This will be an exercise in systematic refactoring: breaking the refactoring down into small steps. After each step, we're aiming for our tests to still be green:

1. First, we'll extract a function containing a JSX snippet for rendering errors. Just above the JSX return value in `CustomerForm`, add a new function named `renderFirstNameError` with the following content:

```
const renderFirstNameError = () => (
  <span id="firstNameError" role="alert">
```

```

    {hasFirstNameError()
      ? validationErrors["firstName"]
      : ""}
    <span>
  ) ;

```

- Now, you can use that in the JSX to replace the `span` alert. Your tests should still be passing at each step:

```

<input
  type="text"
  name="firstName"
  ...
/>
{renderFirstNameError() }

```

- Next, we'll introduce a parameter into this function that will reference the ID of the field we're showing the error from. Adjust the line you just added to introduce that new parameter:

```

<input
  type="text"
  name="firstName"
  ...
/>
{renderFirstNameError("firstName") }

```

Always having green tests – JavaScript versus TypeScript

This section is written in a way that your tests should still be passing at every step. In the preceding step, we passed a parameter to `renderFirstNameError` that the function can't accept yet. In JavaScript, this is perfectly fine. In TypeScript, you'll get a type error when attempting to build your source.

- Introduce that parameter into the `renderFirstNameError` function as follows, replacing occurrences of the `firstName` string with the `fieldName` variable. Your tests should still be passing after this change:

```

const renderFirstNameError = (fieldName) => (
  <span id={`${fieldName}Error`} role="alert">
    {hasFirstNameError()
      ? validationErrors[fieldName]

```

```
: ""
<span>
) ;
```

5. Repeat the same process for the `hasFirstNameError` function by adding a parameter value:

```
const renderFirstNameError = (fieldName) => (
  <span id={`$${fieldName}Error`} role="alert">
    {hasFirstNameError(fieldName)
      ? validationErrors[fieldName]
      : ""}
    <span>
  ) ;
```

6. Add the `fieldName` parameter to `hasFirstNameError` and modify the function body so that it uses the parameter in place of the `firstName` error property:

```
const hasFirstNameError = fieldName =>
  validationErrors[fieldName] !== undefined;
```

7. Now, rename `renderFirstNameError` so that it becomes `renderError` and `hasFirstNameError` so that it becomes `hasError`.

Refactoring support in your IDE

Your IDE may have renaming support built in. If it does, you should use it. Automated refactoring tools lessen the risk of human error.

8. Let's tackle `handleBlur`. We're already passing the `target` parameter, and we can use `target.name` to key into a map that then tells us which validator to run for each field:

```
const handleBlur = ({ target }) => {
  const validators = {
    firstName: required
  };
  const result =
    validators[target.name](target.value);
  setValidationErrors({
    ...validationErrors,
    [target.name]: result
  });
}
```

```
    }) ;
}
```

As you can see, the first half of the function (the definition of `validators`) is now static data that defines how the validation should happen for `firstName`. This object will be extended later, with the `lastName` and `phoneNumber` fields. The second half is generic and will work for any input field that's passed in, so long as a validator exists for that field.

9. The required validator is hardcoded with the first name description. Let's pull out the entire message as a variable. We can create a higher-order function that returns a validation function that uses this message. Modify `required` so that it looks as follows:

```
const required = description => value =>
  !value || value.trim() === ""
    ? description
    : undefined;
```

10. Finally, update the validator so that it calls this new required function:

```
const validators = {
  firstName: required("First name is required")
};
```

At this point, your tests should be passing and you should have a fully generalized solution. Now, let's generalize the tests too, by converting our four validation tests into test generator functions:

1. Define a new `errorFor` helper at the top of the validations nested `describe` block. This will be used in the test generators:

```
const errorFor = (fieldName) =>
  element(`#${fieldName}Error[role=alert]`);
```

2. Find the first test you've written in this section (`renders an alert space...`). Modify it, as shown here, by wrapping it in a function definition that takes a `fieldName` parameter. Use that parameter in the test description and the expectation, replacing the use of `firstName`, and making use of the new `errorFor` helper to find the appropriate field:

```
const itRendersAlertForFieldValidation = (fieldName) => {
  it(`renders an alert space for ${fieldName} validation errors`, async () => {
    render(<CustomerForm original={blankCustomer} />);
    expect(errorFor(fieldName)).not.toBeNull();
```

```
    }) ;  
};
```

3. Since you've now lost the test for the first name, add that back in with a call to the new test generator, just below it:

```
itRendersAlertForFieldValidation("firstName") ;
```

4. Repeat the same process for the second test: wrap it in a function definition, introduce a `fieldName` parameter, and replace `firstName` with `fieldName` within the test description and expectation:

```
const itSetsAlertAsAccessibleDescriptionForField = (  
  fieldName  
) => {  
  it(`sets alert as the accessible description for the  
  ${fieldName} field`, async () => {  
    render(<CustomerForm original={blankCustomer} />);  
    expect(  
      field(fieldName).getAttribute(  
        "aria-describedby"  
      )  
      .toEqual(`>${fieldName}Error`);  
    );  
  });  
};
```

5. Then, re-introduce the test case for the `firstName` field:

```
itSetsAlertAsAccessibleDescriptionForField(  
  "firstName"  
) ;
```

6. Next, it's time to tackle the chunkiest test – the `displays error after blur...` test. The previous two test generators used just one parameter, `fieldName`. This one needs two more, `value` and `description`, that are used in the **Act** phase and the **Assert** phase, respectively:

```
const itInvalidatesFieldWithValue = (  
  fieldName,  
  value,  
  description  
) => {
```

```

    it(`displays error after blur when ${fieldName} field
is '${value}'`, () =>
  render(<CustomerForm original={blankCustomer} />);

  withFocus(field(fieldName), () =>
    change(field(fieldName), value)
  );

  expect(
    errorFor(fieldName)
  ).toContainText(description);
}) ;
}

```

7. Just below that test generator definition, re-introduce the test case for the `first_name` field:

```

itInvalidatesFieldWithValue(
  "firstName",
  " ",
  "First name is required"
);

```

8. Finally, repeat the same process for the fourth test:

```

const itInitiallyHasNoTextInTheAlertSpace = (fieldName)
=> {
  it(`initially has no text in the ${fieldName} field
alert space`, async () => {
    render(<CustomerForm original={blankCustomer} />);
    expect(
      errorFor(fieldName).textContent
    ).toEqual("");
  });
};

```

9. Then, re-introduce the `firstName` test case:

```
itInitiallyHasNoTextInTheAlertSpace("firstName");
```

- After all that effort, it's time to use the new test generators to build out the validation for the `lastName` field. Add the following single line at the bottom of your test suite:

```
itRendersAlertForFieldValidation("lastName");
```

- To make that pass, simply add the code to the `CustomerForm` JSX by rendering another alert just below the `lastName` field:

```
<label htmlFor="lastName">Last name</label>
<input
  type="text"
  name="lastName"
  id="lastName"
  value={customer.lastName}
  onChange={handleChange}
/>
{renderError("lastName")}
```

- Next, we must create the test for the `aria-describedby` attribute:

```
itSetsAlertAsAccessibleDescriptionForField(
  "lastName"
);
```

- To make it pass, add that attribute to the `lastName` input element:

```
<input
  type="text"
  name="lastName"
  ...
  aria-describedby="lastNameError"
/>
```

- Next, add the test for the required validation rule:

```
itInvalidatesFieldWithValue(
  "lastName",
  " ",
  "Last name is required"
);
```

15. Given all the hard work we've done already, making this test pass is now super simple. Add a `lastName` entry to the `validators` object, as shown here:

```
const validators = {
  firstName: required("First name is required"),
  lastName: required("Last name is required"),
};
```

16. For completeness, we need to add the fourth and final test for the `lastName` field. This test passes already since we're relying on the mechanism we've just generalized. However, given that it's a one-liner, it's worth specifying, even if it's not necessary:

```
itInitiallyHasNoTextInTheAlertSpace("lastName");
```

17. Repeat *Steps 10 to 16* for the `phoneNumber` field.

Who needs test generator functions?

Test generator functions can look complex. You may prefer to keep duplication in your tests or find some other way to extract common functionality from your tests.

There is a downside to the test generator approach: you won't be able to use `it.only` or `it.skip` on individual tests.

With that, we've covered the required field validation. Now, let's add a different type of validation for the `phoneNumber` field. We want to ensure the phone number only contains numbers and a few special characters: brackets, dashes, spaces, and pluses.

To do that, we'll introduce a `match` validator that can perform the phone number matching we need, and a `list` validator that composes validations.

Let's add that second validation:

1. Add the following new test:

```
itInvalidatesFieldWithValue(
  "phoneNumber",
  "invalid",
  "Only numbers, spaces and these symbols are allowed: ( "
) + -"
);
```

2. Add the following definition at the top of `src/CustomerForm.js`. This expects a regular expression, `re`, which can then be matched against:

```
const match = (re, description) => value =>
  !value.match(re) ? description : undefined;
```

Learning regular expressions

Regular expressions are a flexible mechanism for matching string formats. If you're interested in learning more about them, and how to test-drive them, take a look at <https://reacttdd.com/testing-regular-expressions>.

3. Now, let's go for the `list` validator function. This is quite a dense piece of code that returns a short-circuiting validator. It runs each validator that it's given until it finds one that returns a string, and then returns that string. Add this just below the definition for `match`:

```
const list = (...validators) => value =>
  validators.reduce(
    (result, validator) => result || validator(value),
    undefined
  );
```

4. Replace the existing `phoneNumber` validation in the `handleBlur` function with the following validation, which uses all three validator functions:

```
const validators = {
  ...
  phoneNumber: list(
    required("Phone number is required"),
    match(
      /^[0-9+() \-]*$/,
      "Only numbers, spaces and these symbols are allowed: ( ) + -"
    )
  );
};
```

5. Your test should now be passing. However, if you look back at the test we just wrote, it says nothing about the allowed set of characters: it just says that `invalid` is not a valid phone number. To prove the use of the *real* regular expression, we need an inverse test to check that any combination of characters works. You can add this in; it should already pass:

```
it("accepts standard phone number characters when validating", () => {
    render(<CustomerForm original={blankCustomer} />);

    withFocus(field("phoneNumber"), () =>
        change(field("phoneNumber"), "0123456789+() - ")
    );

    expect(errorFor("phoneNumber")).not.toContainText(
        "Only numbers"
    );
});
```

Is this a valid test?

This test passes without any required changes. That breaks our rule of only writing tests that fail.

We got into this situation because we did too much in our previous test: all we needed to do was prove that the `invalid` string wasn't a valid phone number. But instead, we jumped ahead and implemented the full regular expression.

If we had triangulated “properly,” with a dummy regular expression to start, we would have ended up in the same place we are now, except we'd have done a bunch of extra intermediate work that ends up being deleted.

In some scenarios, such as when dealing with regular expressions, I find it's okay to short-circuit the process as it saves me some work.

With that, you've learned how to generalize validation using TDD.

Submitting the form

What should happen when we submit the form? For our application, if the user clicks the submit button before the form is complete, the submission process should be canceled and all the fields should display their validation errors at once.

We can do this with two tests: one to check that the form isn't submitted while there are errors, and another to check that all the fields are showing errors.

Before we do that, we'll need to update our existing tests that submit the form, as they all assume that the form has been filled in correctly. First, we need to ensure that we pass valid customer data that can be overridden in each test.

Let's get to work on the `CustomerForm` test suite:

1. We need a new builder to help represent a `validCustomer` record. We'll update many of our existing tests to use this new value. In `test/builders/customer.js`, define the following object:

```
export const validCustomer = {
  firstName: "first",
  lastName: "last",
  phoneNumber: "123456789"
};
```

2. In `test/CustomerForm.test.js`, update the import that contains `blankCustomer`, pulling in the new `validCustomer` too:

```
import {
  blankCustomer,
  validCustomer,
} from "./builders/customer";
```

3. Starting at the top, modify each test that simulates a submit event. Each should be mounted with this new `validCustomer` object. After making these changes, run your tests and make sure they are still passing before continuing:

```
render(<CustomerForm original={validCustomer} />);
```

4. Add a new test for submitting the form. This can go alongside the other submit tests, rather than in the validation block:

```
it("does not submit the form when there are validation errors", async () => {
  render(<CustomerForm original={blankCustomer} />);

  await clickAndWait(submitButton());
  expect(global.fetch).not.toBeCalled();
});
```

5. To make this pass, first, define the following `validateMany` function inside the `CustomerForm` component. Its job is to validate many fields at once. It takes a single parameter, `fields`, which is an object of the field values we care about:

```
const validateMany = fields =>
  Object.entries(fields).reduce(
    (result, [name, value]) => ({
      ...result,
      [name]: validators[name](value)
    }) ,
    {}
  );

```

6. The `validateMany` function references the `validators` constant, but that constant is currently defined in the `handleBlur` function. Pull that definition up so that it exists at the top of the component scope and is now accessible by both `handleBlur` and `validateMany`.
7. We need a new function to check for errors across all fields. That's `anyErrors`; add that now, as shown here. It returns `true` if we had any errors at all, and `false` otherwise:

```
const anyErrors = errors =>
  Object.values(errors).some(error => (
    error !== undefined
  )
);

```

8. Now, we can use `validateMany` and `anyErrors` in our `handleSubmit` function, as shown here. We're going to wrap most of the existing functions in a conditional. Your test should pass after adding this code:

```
const handleSubmit = async e =>
  e.preventDefault();
  const validationResult = validateMany(customer);
  if (!anyErrors(validationResult)) {
    ... existing code ...
  }
}
```

9. Let's move on to the next test. We need a couple of new imports, `textOf` and `elements`, so that we can write an expectation across all three of the alert spaces. Add these now:

```
import {
  ...
}
```

```
    textOf,  
    elements,  
} from "./reactTestExtensions";
```

10. Next, add the following test at the bottom of the test suite. We want to check whether any errors appear on the screen:

```
it("renders validation errors after submission fails",  
async () => {  
  render(<CustomerForm original={blankCustomer} />);  
  await clickAndWait(submitButton());  
  expect(  
    textOf(elements("[role=alert]"))  
  ).not.toEqual("");  
});
```

Using the alert role on multiple elements

This chapter uses multiple alert spaces, one for each form field. However, screen readers do not behave well when multiple alert roles show alerts at the same time – for example, if clicking the submit button causes a validation error to appear on all three of our fields.

An alternative approach would be to rework the UI so that it has an additional element that takes on the alert role when any errors are detected; after that, it should remove the alert role from the individual field error descriptions.

11. This one is easy to pass; we simply need to call `setValidationErrors` with `validationResult` when `anyErrors` returns `false`:

```
if (!anyErrors(validationResult)) {  
  ...  
} else {  
  setValidationErrors(validationResult);  
}
```

You've now seen how to run all field validations when the form is submitted.

Extracting non-React functionality into a new module

One useful design guideline is to get out of “framework land” as soon as possible. You want to be dealing with plain JavaScript objects. This is especially true for React components: extract as much logic as possible out into standalone modules.

There are a few different reasons for this. First, testing components is harder than testing plain objects. Second, the React framework changes more often than the JavaScript language itself. Keeping our code bases up to date with the latest React trends is a large-scale task if our code base is, first and foremost, a React code base. If we keep React at bay, our lives will be simpler in the longer term. So, we always prefer to write plain JavaScript when it's an option.

Our validation code is a great example of this. We have several functions that do not care about React at all:

- The validators: `required`, `match`, and `list`
- `hasError` and `anyErrors`
- `validateMany`
- Some of the code in `handleBlur`, which is like a single-entry equivalent of `validateMany`

Let's pull all of these out into a separate namespace called `formValidation`:

1. Create a new file called `src/formValidation.js`.
2. Move across the function definitions for `required`, `match`, and `list` from the top of `CustomerForm`. Make sure you delete the old definitions!
3. Add the word `export` to the front of each definition in the new module.
4. Add the following import to the top of `CustomerForm`, and then check that your tests are still passing:

```
import {  
  required,  
  match,  
  list,  
} from './formValidation';
```

5. In `src/CustomerForm.js`, change `renderError` so that it passes the errors from `state` into `hasError`:

```
const renderError = fieldName => {  
  if (hasError(validationErrors, fieldName)) {  
    ...  
  }  
}
```

6. Update `hasError` so that it includes the new `validationErrors` argument, and uses that rather than `state`:

```
const hasError = (validationErrors, fieldName) =>
  validationErrors[fieldName] !== undefined;
```

7. Update `validateMany` so that it passes in the list of validators as its first argument, rather than using `state`:

```
const validateMany = (validators, fields) =>
  Object.entries(fields).reduce(
    (result, [name, value]) => ({
      ...result,
      [name]: validators[name](value)
    }) ,
    {}
  );
```

8. Update `handleBlur` so that it uses `validateMany`:

```
const handleBlur = ({ target }) => {
  const result = validateMany(validators, {
    [target.name] : target.value
  });
  setValidationErrors({
    ...validationErrors,
    ...result
  });
}
```

9. Update `handleSubmit` so that it passes `validators` to `validateMany`:

```
const validationResult = validateMany(
  validators,
  customer
);
```

10. Move `hasError`, `validateMany`, and `anyErrors` into `src/formValidation.js`, ensuring you delete the functions from the `CustomerForm` component.

11. Add the word `export` in front of each of these definitions.

12. Update the import so that it pulls in these functions:

```
import {  
  required,  
  match,  
  list,  
  hasError,  
  validateMany,  
  anyErrors,  
} from './formValidation';
```

Although this is enough to extract the code out of React-land, we've only just made a start. There is plenty of room for improvement with this API. There are a couple of different approaches that you could take here. The exercises for this chapter contain some suggestions on how to do that.

Using test doubles for validation functions

You may be thinking, do these functions now need their own unit tests? And should I update the tests in `CustomerForm` so that test doubles are used in place of these functions?

In this case, I would probably write a few tests for `formValidation`, just to make it clear how each of the functions should be used. This isn't test-driving since you already have the code, but you can still mimic the experience by writing tests as you normally would.

When extracting functionality from components like this, it often makes sense to update the original components to simplify and perhaps move across tests. In this instance, I wouldn't bother. The tests are high-level enough that they make sense, regardless of how the code is organized internally.

This section covered how to write validation logic for forms. You should now have a good awareness of how TDD can be used to implement complex requirements such as field validations. Next, we'll integrate server-side errors into the same flow.

Handling server errors

The `/customers` endpoint may return a `422 Unprocessable Entity` error if the customer data failed the validation process. This could happen if, for example, the phone number already exists within the system. If this happens, we want to withhold calling the `onSave` callback and instead display the errors to the user and give them the chance to correct them.

The body of the response will contain error data very similar to the data we've built for the validation framework. Here's an example of the JSON that would be received:

```
{  
  "errors": {  
    "phoneNumber": "Phone number already exists in the system"  
  }  
}
```

We'll update our code to display these errors in the same way our client errors appeared. Since we already handle errors for `CustomerForm`, we'll need to adjust our tests in addition to the existing `CustomerForm` code.

Our code to date has made use of the `ok` property that's returned from `global.fetch`. This property returns `true` if the HTTP status code is `200`, and `false` otherwise. Now, we need to be more specific. For a status code of `422`, we want to display new errors, and for anything else (such as a `500` error), we want to fall back to the existing behavior.

Let's add support for those additional status codes:

1. Update the `fetchResponseError` method in `test/builders/fetch.js`, as shown here:

```
const fetchResponseError = (  
  status = 500,  
  body = {}  
) => ({  
  ok: false,  
  status,  
  json: () => Promise.resolve(body),  
}) ;
```

2. Write a test for `422` errors in `test/CustomerForm.test.js`. I've placed this toward the top of the file, next to the other tests that manipulate the HTTP response:

```
it("renders field validation errors from server", async  
() => {  
  const errors = {  
    phoneNumber: "Phone number already exists in the  
    system"  
  };  
  global.fetch.mockResolvedValue(
```

```

        fetchResponseError(422, { errors })
    );
render(<CustomerForm original={validCustomer} />);
await clickAndWait(submitButton());
expect(errorFor("phoneNumber")).toContainText(
    errors.phoneNumber
);
})
;
}
);

```

3. To make that pass, add a new branch to the nested conditional statement in `handleSubmit`, which handles the response of the fetch request:

```

if (result.ok) {
    setError(false);
    const customerWithId = await result.json();
    onSave(customerWithId);
} else if (result.status === 422) {
    const response = await result.json();
    setValidationErrors(response.errors);
} else {
    setError(true);
}

```

Your tests should now be passing.

This section has shown you how to integrate server-side errors into the same client-side validation logic that you already have. To finish up, we'll add some frills.

Indicating form submission status

It'd be great if we could indicate to the user that their form data is being sent to our application servers. The GitHub repository for this book contains a spinner graphic and some CSS that we can use. All that our React component needs to do is display a `span` element with a class name of `submittingIndicator`.

Before we write out the tests, let's look at how the production code will work. We will introduce a new `submitting` boolean state variable that is used to toggle between states. It will be toggled to `true` just before we perform the fetch request and toggled to `false` once the request completes. Here's how we'll modify `handleSubmit`:

```

...
if (!anyErrors(validationResult)) {

```

```
setSubmitting(true);  
const result = await global.fetch(...);  
setSubmitting(false);  
...  
}  
...
```

If submitting is set to `true`, then we will render the spinner graphic. Otherwise, we will render nothing.

Testing state before promise completion

One of the trickiest aspects of testing React components is testing what happens *during* a task. That's what we need to do now: we want to check that the submitting indicator is shown while the form is being submitted. However, the indicator disappears as soon as the promise completes, meaning that we can't use the standard `clickAndWait` function we've used up until now because it will return at the point *after* the indicator has disappeared!

Recall that `clickAndWait` uses the asynchronous form of the `act` test helper. That's the core of the issue. To get around this, a *synchronous* form of our function, `click`, will be needed to return *before* the task queue completes – in other words, before the `global.fetch` call returns any results.

However, to stop React's warning sirens from going off, we still need to include the asynchronous `act` form *somewhere* in our test. React knows the submit handler returns a promise and it expects us to wait for its execution via a call to `act`. We need to do that after we've checked the toggle value of `submitting`, not before.

Let's build that test now:

1. Add `act` as an import to `test/CustomerForm.test.js`:

```
import { act } from "react-dom/test-utils";
```

2. Re-add the `click` function import:

```
import {  
  ...,  
  click,  
  clickAndWait,  
} from "./reactTestExtensions";
```

3. Create a new nested describe block at the bottom of the CustomerForm test suite, just below the existing form submission tests. This submits the call itself within a synchronous click, as explained previously. Then, we must wrap the expectation in an async act call that suppresses any warnings or errors from React:

```
describe("submitting indicator", () => {
  it("displays when form is submitting", async () => {
    render(
      <CustomerForm
        original={validCustomer}
        onSave={() => {}}
      />
    );
    click(submitButton());
    await act(async () => {
      expect(
        element("span.submittingIndicator")
      ).not.toBeNull();
    });
  });
});
```

4. To make this pass, we just need to show that span within the JSX. Place that just after the submit button, as follows:

```
return (
  <form id="customer" onSubmit={handleSubmit}>
    ...
    <input type="submit" value="Add" />
    <span className="submittingIndicator" />
  </form>
);
```

5. Now, we need to triangulate, to ensure the indicator only shows when the form has been submitted and not before:

```
it("initially does not display the submitting indicator",
() => {
  render(<CustomerForm original={validCustomer} />);
```

```
    expect(element(".submittingIndicator")).toBeNull();
});
```

6. We can make this pass by using a flag called `submitting`. It should be set to `false` when the indicator is disabled, and `true` when it's enabled. Add the following state variable to the top of the `CustomerForm` component:

```
const [submitting, setSubmitting] = useState(false);
```

7. Change the submitting span indicator so that it reads as follows:

```
{submitting ? (
  <span className="submittingIndicator" />
) : null}
```

8. The new test will now be passing, but the original test will be failing. We had to switch `submittingIndicator` to `true` just before we called `fetch`. In `handleSubmit`, add this line just above the call to `fetch`. After adding this code, your test should be passing:

```
if (!anyErrors(validationResult)) {
  setSubmitting(true);
  const result = await global.fetch(/* ... */);
  ...
}
```

9. Add this final test, which checks that the indicator disappears once the response has been received. This test is very similar to our first test for the submitting indicator:

```
it("hides after submission", async () => {
  render(
    <CustomerForm
      original={validCustomer}
      onSave={() => {}}
    />
  );
  await clickAndWait(submitButton());
  expect(element(".submittingIndicator")).toBeNull();
});
```

10. This time, we need to add a `setSubmitting` call *after* the fetch:

```
if (!anyErrors(validationResult)) {  
    setSubmitting(true);  
    const result = await global.fetch(/* ... */);  
    setSubmitting(false);  
    ...  
}
```

That's everything; your tests should all be passing.

Refactoring long methods

After this, our `handleSubmit` function is long – I have counted 23 lines in my implementation. That is too long for my liking!

Refactoring `handleSubmit` into smaller methods is an exercise left for you; see the *Exercises* section for more details. But here are a couple of hints for how you can go about that systematically:

- Extract blocks into methods; in this case, that means the contents of `if` statements. For example, if there are no validation errors, you could call out to a `doSave` method, which does the submission.
- Look for **temporal coupling** and see if there are other ways to format that code. In this case, we have the `submitting` state variable, which is set to `true`, before the `fetch` call, and then `false` after. This could be done differently.

Now, let's summarize this chapter.

Summary

This chapter has shown you how TDD can be applied beyond just toy examples. Although you may not ever want to implement form validation yourself, you can see how complex code can be test-driven using the same methods that you learned in the first part of this book.

First, you learned how to validate field values at an appropriate moment: when fields lose focus and when forms are submitted. You also saw how server-side errors can be integrated into that, and how to display an indicator to show the user that data is in the process of being saved.

This chapter also covered how to move logic from your React components into their own modules.

In the next chapter, we'll add a new feature to our system: a snazzy search interface.

Exercises

The following are some exercises for you to complete:

1. Add a feature that clears any validation errors when the user corrects them. Use the `onChange` handler for this rather than `onBlur`, since we want to let the user know as soon as they've corrected the error.
2. Add a feature that disables the submit button once the form has been submitted.
3. Write tests for each of the functions within the `formValidation` module.
4. The `handleSubmit` function is long. Extract a `doSave` function that pulls out the main body of the `if` statement.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- A guide to regular expressions, explained by examples
<https://reacttdd.com/testing-regular-expressions>
- More information on ARIA annotations such as `aria-describedby`
<https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Annotations>

10

Filtering and Searching Data

In this chapter, we'll continue applying the techniques we've already learned to another, more complex use case.

As we work through the chapter, we'll learn how to adjust a component's design using tests to show us where the design is lacking. Test-driven development really helps highlight design issues when the tests get knarly. Luckily, the tests we've already written give us the confidence to change course and completely reinvent our design. With each change, we simply run `npm test` and have our new implementation verified in a matter of seconds.

In the current workflow, users start by adding a new customer and then immediately book an appointment for that customer. Now, we'll expand on that by allowing them to choose an existing customer before adding an appointment.

We want users to be able to quickly search through customers. There could be hundreds, maybe thousands, of customers registered with this salon. So, we'll build a `CustomerSearch` search component that will allow our users to search for customers by name and to page through the returned results.

In this chapter, you'll learn about the following topics:

- Displaying tabular data fetched from an endpoint
- Paging through a large dataset
- Filtering data
- Performing actions with render props

The following screenshot shows how the new component will look:

First name	Last name	Phone number	Actions
Baron	Dach	(178) 475-7047	<button>Create appointment</button>
Camille	Daniel	(835) 232-8112	<button>Create appointment</button>
Clyde	Daugherty	(511) 507-5445	<button>Create appointment</button>
Dallas	Howe	(843) 386-2265	<button>Create appointment</button>
Dallin	Witting	(572) 818-3195	<button>Create appointment</button>
Dalton	Keebler	(019) 670-4711	<button>Create appointment</button>
Dalton	Nitzsche	(343) 797-9932	<button>Create appointment</button>
Damaris	Feil	(080) 816-6033	<button>Create appointment</button>
Damian	Kuphal	(155) 691-8189	<button>Create appointment</button>
Damian	Gulgowski	(379) 350-6818	<button>Create appointment</button>

Figure 10.1 – The new CustomerSearch component

By the end of the chapter, you'll have built a relatively complex component using all the techniques you've learned so far.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter10>

Displaying tabular data fetched from an endpoint

In this section, we'll get the basic form of the table in place, with an initial set of data retrieved from the server when the component is mounted.

The server **application programming interface (API)** supports GET requests to `/customers`. There is a `searchTerm` parameter that takes the string the user is searching for. There is also an `after` parameter that is used to retrieve the next page of results. The response is an array of customers, as shown here:

```
[{ id: 123, firstName: "Ashley" }, ... ]
```

Sending a request to `/customers` with no parameters will return the first 10 of our customers, in alphabetical order by first name.

This gives us a good place to start. When the component mounts, we'll perform this basic search and display the results in a table.

Skipping the starting point

If you're following along using the GitHub repository, be aware that this chapter starts with a barebones `CustomerSearch` component already implemented, and it has already been hooked up to the `App` component. The component is displayed by clicking on the **Search appointments** button in the top menu.

Let's start with our first test for the new `CustomerSearch` component. Follow these steps:

1. Open `test/CustomerSearch.test.js` and add the first test. It checks that a table has been rendered with the four headings that we want to see. The code is illustrated in the following snippet:

```
it("renders a table with four headings", async () => {
  await renderAndWait(<CustomerSearch />) ;
  const headings = elements("table th") ;
  expect(textOf(headings)).toEqual([
    "First name",
    "Last name",
    "Phone number",
    "Actions",
  ]) ;
}) ;
```

2. That test should be simple to pass with the following definition for `CustomerSearch` in `src/CustomerSearch.js`:

```
export const CustomerSearch = () => (
  <table>
    <thead>
      <tr>
        <th>First name</th>
        <th>Last name</th>
        <th>Phone number</th>
        <th>Actions</th>
      </tr>
    </thead>
    </table>
);
```

3. In order to display data, the component will need to make a GET request. Write out this next test, which specifies that behavior:

```
it("fetches all customer data when component mounts",
  async () => {
  await renderAndWait(<CustomerSearch />);
  expect(global.fetch).toBeCalledWith("/customers", {
    method: "GET",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" },
  });
});
```

4. To make that pass, add a `useEffect` hook to the component that performs the search. We need to use the same `useEffect` ceremony that we've seen before, using an inline function to ensure we don't return a value and passing an empty array to the dependency list, which ensures the effect only runs when the component is first mounted. The code is illustrated in the following screenshot:

```
export const CustomerSearch = () => {

  useEffect(() => {
    const fetchData = async () =>
      await global.fetch("/customers", {
```

```
        method: "GET",
        credentials: "same-origin",
        headers: {
          "Content-Type": "application/json"
        },
      ) ;

      fetchData();
    }, []);
}

return (
  ...
)
};
```

- Now, it's time to code what happens depending on the data returned. We'll start by figuring out the display of a single row of data. Add a definition of `oneCustomer` at the top of the file, above the `describe` block, as follows:

```
const oneCustomer = [
  {
    id: 1,
    firstName: "A",
    lastName: "B",
    phoneNumber: "1"
  },
];
```

- Make use of that definition in the next test, shown in the following code snippet, which verifies that the component displays all the customer data for a single customer row:

```
it("renders all customer data in a table row", async () => {
  global.fetch.mockResolvedValue(
    fetchResponseOk(oneCustomer)
  );
  await renderAndWait(<CustomerSearch />);
  const columns = elements("table > tbody > tr > td");
  expect(columns[0]).toContainText("A");
```

```
    expect(columns[1]).toContainText("B");
    expect(columns[2]).toContainText("1");
});
```

7. To make this pass, we'll need to use component state to pass data back from the `useEffect` hook into the next render cycle. Create a new state variable, `customers`, which has an initial value of the empty array (`[]`), as follows:

```
const [customers, setCustomers] = useState([]);
```

8. Save the results of the search into `customers` by modifying the definition of `useEffect`, as shown here:

```
const fetchData = async () => {
  const result = await global.fetch(...);
  setCustomers(await result.json());
};
```

9. We're ready to display the data. We'll do that with a new `CustomerRow` component that is responsible for displaying a single row of customer information. Add its implementation above the definition of `CustomerSearch`. Notice here how the final column is blank; it will hold action buttons that perform various operations on the specific customer record. We'll use a separate test later to fill out that functionality:

```
const CustomerRow = ({ customer }) => (
  <tr>
    <td>{customer.firstName}</td>
    <td>{customer.lastName}</td>
    <td>{customer.phoneNumber}</td>
    <td />
  </tr>
);
```

10. All that's left is to make use of this new component in `CustomerSearch`. Add the following `tbody` element, which renders `CustomerRow` for the first customer, if it exists. After adding this code, your test should now be passing:

```
return (
  <table>
    <thead>
      ...
    </thead>
```

```
</thead>
<tbody>
{customers[0] ? (
  <CustomerRow customer={customers[0]} />
) : null}
</tbody>
</table>
);
```

11. For the final test in this section, let's add a test to show that this works for multiple customers. For that, we need a new result set: `twoCustomers`. This can be placed at the top of the file, after `oneCustomer`, as shown in the following code snippet:

```
const twoCustomers = [
{
  id: 1,
  firstName: "A",
  lastName: "B",
  phoneNumber: "1"
},
{
  id: 2,
  firstName: "C",
  lastName: "D",
  phoneNumber: "2"
}
];
```

12. Then, add a test that makes use of this and checks that two rows are rendered, as follows:

```
it("renders multiple customer rows", async () => {
  global.fetch.mockResolvedValue(
    fetchResponseOk(twoCustomers)
  );
  await renderAndWait(<CustomerSearch />);
  const rows = elements("table tbody tr");
```

```
    expect(rows[1].childNodes[0]).toContainText("C");
});
```

13. Making this pass is a one-liner; change the JSX to map over each customer, instead of pulling out just the first customer:

```
<tbody>
  {customers.map(customer => (
    <CustomerRow
      customer={customer}
      key={customer.id}
    />
  ))
}
</tbody>
```

This gives us a great base to build on for the remaining functionality we'll build in this chapter.

In the next section, we'll introduce the ability to move between multiple pages of search results.

Paging through a large dataset

By default, our endpoint returns 10 records. To get the next 10 records, we can page through the result set by using the `after` parameter, which represents the last customer identifier seen. The server will skip through results until it finds that ID and returns results from the next customer onward.

We'll add **Next** and **Previous** buttons that will help us move between search results. Clicking **Next** will take the ID of the last customer record currently shown on the page and send it as the `after` parameter to the next search request.

To support **Previous**, we'll need to maintain a stack of `after` IDs that we can pop each time the user clicks **Previous**.

Adding a button to move to the next page

Let's start with the **Next** button, which the user can click to bring them to the next page of results. Since we're going to be dealing with multiple buttons on the screens, we'll build a new `buttonWithLabel` helper that will match a button with that label. Follow these steps:

1. In `test/reactTestExtensions.js`, add the following new helper function at the bottom of the file:

```
export const buttonWithLabel = (label) =>
  elements("button").find(
```

```
({ textContent }) => textContent === label  
);
```

2. Back in `test/CustomerSearch.test.js`, update the import statement to include this new helper function, like so:

```
import {  
  ...,  
  buttonWithLabel,  
} from "./reactTestExtensions";
```

3. Write the following test, which will let us get a **Next** button onto the page:

```
it("has a next button", async () => {  
  await renderAndWait(<CustomerSearch />);  
  expect(buttonWithLabel("Next")).not.toBeNull();  
});
```

4. Create a `SearchButtons` component that renders the **Next** button in a menu element, just as we did in `App`. We'll be expanding this menu bar with more buttons in subsequent tests. The code is illustrated here:

```
const SearchButtons = () => (  
  <menu>  
    <li>  
      <button>Next</button>  
    </li>  
  </menu>  
) ;
```

5. Now, render that in `CustomerSearch`, above the table, as follows:

```
return (  
  <>  
    <SearchButtons />  
    <table>  
      ...  
    </table>  
  </>  
) ;
```

6. When the button is clicked, we want to take the last customer ID already displayed and send that back to the server. To make that choice obvious in our tests, we'll use a new return value named `tenCustomers`, which mimics the default number of records coming back from the server API. Place this definition of `tenCustomers` at the top of the file, next to your other customer definitions, like so:

```
const tenCustomers =
  Array.from("0123456789", id => ({ id })
);
```

Making good use of `Array.from`

This definition uses a “clever” version of the `Array.from` function that takes each character of the string and creates an object using that character as input. We end up with 10 objects, each with an `id` property ranging from 0 to 9.

7. The next test checks that when the **Next** button is clicked, the component makes a new GET request with the last seen customer ID. Given our previous definition of `tenCustomers`, that is the customer with ID 9. Notice in the following code snippet how `toHaveBeenCalledWith` is needed since this will be the second call to `global.fetch`:

```
it("requests next page of data when next button is
  clicked", async () => {
  global.fetch.mockResolvedValue(
    fetchResponseOk(tenCustomers)
  );
  await renderAndWait(<CustomerSearch />);
  await clickAndWait(buttonWithLabel("Next"));
  expect(global.fetch).toHaveBeenCalledWith(
    "/customers?after=9",
    expect.anything()
  );
});
```

Avoiding unnecessary fields to highlight important implications

The `tenCustomers` value is only a partial definition for each customer: only the `id` property is included. That's not lazy: it's intentional. Because the logic of taking the last ID is non-obvious, it's important to highlight the `id` property as the key feature of this flow. We won't worry about the other fields because our previous tests check their correct usage.

8. To make this pass, define a handler for the **Next** button that performs the `fetch` request. It calculates the `after` request parameter by taking the last customer in the `customers` state variable, as illustrated in the following code snippet:

```
const handleNext = useCallback(() => {
  const after = customers[customers.length - 1].id;
  const url = `/customers?after=${after}`;
  global.fetch(url, {
    method: "GET",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" }
  });
}, [customers]);
```

9. Give `SearchButtons` a `handleNext` prop and set that as the `onClick` handler on the button, like so:

```
const SearchButtons = ({ handleNext }) => (
  <menu>
    <li>
      <button onClick={handleNext}>Next</button>
    </li>
  </menu>
);
```

10. Hook the handler up to `SearchButtons`, as follows. After this change, your test should be passing:

```
<SearchButtons handleNext={handleNext} />
```

11. Continue by adding the following test. It sets up two `fetch` responses using a sequence of `mockResolvedValueOnce` followed by `mockResolvedValue`. The second response only contains one record. The test asserts that this record is displayed after pressing the **Next** button:

```
it("displays next page of data when next button is clicked", async () => {
  const nextCustomer = [{ id: "next", firstName: "Next" }];
  global.fetch
    .mockResolvedValueOnce(
```

```

        fetchResponseOk(tenCustomers)
    )
    .mockResolvedValue(fetchResponseOk(nextCustomer));
await renderAndWait(<CustomerSearch />);
await clickAndWait(buttonWithLabel("Next"));
expect(elements("tbody tr")).toHaveLength(1);
expect(elements("td")[0]).toContainText("Next");
});

```

12. To make this pass, modify `handleNext` to save its response into the `customers` state variable, as follows:

```

const handleNext = useCallback(async () => {
    ...
    const result = await global.fetch(...);
    setCustomers(await result.json());
}, [customers]);

```

That's it for our **Next** button. Before we move on to the **Previous** button, we need to correct a design issue.

Adjusting the design

Look here at the similarities between the `handleNext` and `fetchData` functions. They are almost identical; the only place they differ is in the first parameter to the `fetch` call. The `handleNext` function has an `after` parameter; `fetchData` has no parameters:

```

const handleNext = useCallback(async () => {
    const after = customers[customers.length - 1].id;
    const url = `/customers?after=${after}`;
    const result = await global.fetch(url, ...);
    setCustomers(await result.json());
}, [customers]);
const fetchData = async () => {
    const result = await global.fetch(`/customers`, ...);
    setCustomers(await result.json());
};

```

We will be adding a **Previous** button, which would result in further duplication if we carried on with this same design. But there's an alternative. We can take advantage of the `useEffect` hook's ability to rerun when the state changes.

We will introduce a new state variable, `queryString`, which `handleNext` will update and `useEffect` will listen for.

Let's do that now. Proceed as follows:

1. Add that new variable now at the top of the `CustomerSearch` component, as shown in the following code snippet. Its initial value is the empty string, which is important:

```
const [queryString, setQueryString] = useState("");
```

2. Replace `handleNext` with the following function:

```
const handleNext = useCallback(() => {
  const after = customers[customers.length - 1].id;
  const newQueryString = `?after=${after}`;
  setQueryString(newQueryString);
}, [customers]);
```

3. Update `useEffect` with the following definition, appending `queryString` to the **Uniform Resource Locator (URL)**. Your tests should still be passing at this point:

```
useEffect(() => {
  const fetchData = async () => {
    const result = await global.fetch(
      `/customers${queryString}`,
      ...
    );
    setCustomers(await result.json());
  };

  fetchData();
}, [queryString]);
```

That's it for the **Next** button: you've seen how to write elegant tests for a complex piece of API orchestration logic, and we've refactored our production code to be elegant, too.

Adding a button to move to the previous page

Let's move on to the **Previous** button:

1. Write out the following test:

```
it("has a previous button", async () => {
  await renderAndWait(<CustomerSearch />);
  expect(buttonWithLabel("Previous")).not.toBeNull();
});
```

2. Make that pass by modifying `SearchButtons` to include the following button, just before the **Next** button:

```
<menu>
  <li>
    <button>Previous</button>
  </li>
  ...
</menu>
```

3. The next test mounts the component, clicks **Next**, and then clicks **Previous**. It expects another call to the endpoint to have been made, but this time identical to the initial page—in other words, with no query string. The code is illustrated here:

```
it("moves back to first page when previous button is clicked", async () => {
  global.fetch.mockResolvedValue(
    fetchResponseOk(tenCustomers)
  );
  await renderAndWait(<CustomerSearch />);
  await clickAndWait(buttonWithLabel("Next"));
  await clickAndWait(buttonWithLabel("Previous"));
  expect(global.fetch).toHaveBeenCalledWith(
    "/customers",
    expect.anything()
  );
});
```

- To make this pass, start by defining a `handlePrevious` function, as follows:

```
const handlePrevious = useCallback(
  () => setQueryString("") ,
  []
);
```

- Modify `SearchButtons` to take a new `handlePrevious` prop, and set that as the `onClick` handler on the new button, like so:

```
const SearchButtons = (
  { handleNext, handlePrevious }
) => (
  <menu>
    <li>
      <button
        onClick={handlePrevious}
      >
        Previous
      </button>
    </li>
    ...
  </menu>
);
```

- Hook up the handler to `SearchButtons`, like so. After this, your test should be passing:

```
<SearchButtons
  handleNext={handleNext}
  handlePrevious={handlePrevious}
/>
```

- The next test is one that'll require us to do some thinking. It simulates clicking **Next** twice, then clicking **Previous** once. For the second **Next** click, we need another set of customers. Add anotherTenCustomers just after the definition of `tenCustomers`, as follows:

```
const anotherTenCustomers =
  Array.from("ABCDEFGHIJ", id => ({ id }));
```

8. Now, add the next test, which checks that the **Previous** button still works after navigating to two more pages:

```
it("moves back one page when clicking previous after multiple clicks of the next button", async () => {
  global.fetch
    .mockResolvedValueOnce(
      fetchResponseOk(tenCustomers)
    )
    .mockResolvedValue(
      fetchResponseOk(anotherTenCustomers)
    );
  await renderAndWait(<CustomerSearch />);
  await clickAndWait(buttonWithLabel("Next"));
  await clickAndWait(buttonWithLabel("Next"));
  await clickAndWait(buttonWithLabel("Previous"));
  expect(global.fetch).toHaveBeenCalledWith(
    "/customers?after=9",
    expect.anything()
  );
});
```

9. We'll make this pass by maintaining a record of the query strings that were passed to the endpoint. For this specific test, we only need to know what the *previous* query string was. Add a new state variable to record that, as follows:

```
const [
  previousQueryString, setPreviousQueryString
] = useState("");
```

Forcing design issues

You may recognize this as an overly complicated design. Let's just go with it for now: we will simplify this again with another test.

10. Change `handleNext` to save the previous query string, making sure that this happens before the call to `setQueryString`. Include `queryString` in the array passed to the second parameter of `useCallback` so that this callback is regenerated each time the value of `queryString` changes. The code is illustrated in the following snippet:

```
const handleNext = useCallback(queryString => {
  ...
});
```

```
    setPreviousQueryString(queryString) ;  
    setQueryString(newQueryString) ;  
}, [customers, queryString]) ;
```

11. Now, handlePrevious can use this value as the query string to pass to fetchData, as illustrated here. Your test should be passing at this point:

```
const handlePrevious = useCallback(async () =>  
  setQueryString(previousQueryString)  
, [previousQueryString]);
```

That's it for a basic **Previous** button implementation. However, what happens when we want to go back two or more pages? Our current design only has a "depth" of two additional pages. What if we want to support any number of pages?

Forcing design changes using tests

We can use a test to force the design issue. The process of TDD helps us to ensure that we always take time to think about the simplest solution that solves all tests. So, if we add one more test that highlights the limits of the current design, that test becomes a trigger for us to stop, think, and reimplement.

In this case, we can use a stack of previous query strings to remember the history of pages. We'll replace our two state variables, queryString and previousQueryString, with a single state variable, queryStrings, which is a stack of all previous query strings.

Let's get started with the test. Follow these steps:

1. Add this test, which asserts that the **Previous** button works for multiple presses:

```
it("moves back multiple pages", async () => {  
  global.fetch  
    .mockResolvedValue(fetchResponseOk(tenCustomers)) ;  
  await renderAndWait(<CustomerSearch />) ;  
  await clickAndWait(buttonWithLabel("Next")) ;  
  await clickAndWait(buttonWithLabel("Next")) ;  
  await clickAndWait(buttonWithLabel("Previous")) ;  
  await clickAndWait(buttonWithLabel("Previous")) ;  
  expect(global.fetch).toHaveBeenCalledWith  
    "/customers",  
    expect.anything()  
  );  
});
```

2. For this to pass, start by adding a new `queryStrings` state variable, deleting `queryString` and `previousQueryStrings`, as follows:

```
const [queryStrings, setQueryStrings] = useState([]);
```

3. Change `fetchData` as follows. If there are entries in the `queryStrings` array, it sets `queryString` to the last entry, and that value is then passed to the `fetch` call. If there's nothing in the array, then `queryString` will be an empty string:

```
useEffect(() => {
  const fetchData = async () => {
    const queryString =
      queryStrings[queryStrings.length - 1] || "";

    const result = await global.fetch(
      `/customers${queryString}`,
      ...
    );
    setCustomers(await result.json());
  };

  fetchData();
}, [queryStrings]);
```

4. Change `handleNext` as follows. It now *appends* the current query string to the previous query strings:

```
const handleNext = useCallback(() => {
  const after = customers[customers.length - 1].id;
  const newQueryString = `?after=${after}`;
  setQueryStrings([...queryStrings, newQueryString]);
}, [customers, queryStrings]);
```

5. Change `handlePrevious` as follows. The last value is *popped off* the query string stack:

```
const handlePrevious = useCallback(() => {
  setQueryStrings(queryStrings.slice(0, -1));
}, [queryStrings]);
```

You now have a relatively complete implementation for the **Next** and **Previous** buttons. You've also seen how tests can help you alter your design as you encounter issues with it.

Next, we'll continue building out our integration with the `searchTerm` parameter of the `/customers` HTTP endpoint.

Filtering data

In this section, we'll add a textbox that the user can use to filter names. Each character that the user types into the search field will cause a new `fetch` request to be made to the server. That request will contain the new search term as provided by the search box.

The `/customers` endpoint supports a parameter named `searchTerm` that filters search results using those terms, as shown in the following code snippet:

```
GET /customers?searchTerm=Dan

[
  {
    firstName: "Daniel",
    ...
  }
  ...
]
```

Let's start by adding a text field into which the user can input a search term, as follows:

1. Add the following test to the `CustomerSearch` test suite, just below the last test. It simply checks for a new field:

```
it("renders a text field for a search term", async () =>
{
  await renderAndWait(<CustomerSearch />);
  expect(element("input")).not.toBeNull();
});
```

2. In `CustomerSearch`, update your JSX to add that input element at the top of the component, as follows:

```
return (
  <>
  <input />
  ...
</>
);
```

3. Next, we want to check that the `placeholder` attribute for that field is set. We can do this by running the following code:

```
it("sets the placeholder text on the search term field",
  async () => {
  await renderAndWait(<CustomerSearch />);
  expect(
    element("input").getAttribute("placeholder")
  ).toEqual("Enter filter text");
});
```

4. To make that pass, add the placeholder to the input element in your JSX, like so:

```
<input placeholder="Enter filter text" />
```

5. We want to hook this up to the DOM change event: we'll make an `async` fetch request every time the value changes. For that, we'll need a new helper. In `test/reactTestExtensions.js`, add the following definition for `changeAndWait`, just below the definition of `change`. This allows us to run effects when the DOM change event occurs:

```
export const changeAndWait = async (target, value) =>
  act(async () => change(target, value));
```

6. Import the new helper at the top of `test/CustomerSearch.test.js`, like so:

```
import {
  ...
  changeAndWait,
} from "./reactTestExtensions";
```

7. Each time a new character is entered into the search box, we should perform a new search with whatever text is entered in the textbox. Add the following test:

```
it("performs search when search term is changed", async
() => {
  await renderAndWait(<CustomerSearch />);
  await changeAndWait(element("input"), "name");
  expect(global.fetch).toHaveBeenCalledWith(
    "/customers?searchTerm=name",
    expect.anything()
  );
});
```

8. Define a new `searchTerm` variable, as follows:

```
const [searchTerm, setSearchTerm] = useState("");
```

9. Add a new handler, `handleSearchTextChanged`, as follows. It stores the search term in the state because we'll need to pull it back when moving between pages:

```
const handleSearchTextChanged = (
  { target: { value } }
) => setSearchTerm(value);
```

10. Hook it up to the input element, like so:

```
<input
  value={searchTerm}
  onChange={handleSearchTextChanged}
  placeholder="Enter filter text"
/>
```

11. Now, we can use the `searchTerm` variable in `fetchData` to fetch the updated set of customers from the server, as follows:

```
const fetchData = async () => {
  let queryString = "";
  if (searchTerm !== "") {
    queryString = `?searchTerm=${searchTerm}`;
  } else if (queryStrings.length > 0) {
    queryString =
      queryStrings[queryStrings.length - 1];
  }
  ...
};
```

12. Finally, we need to modify `useEffect` by adding `searchTerm` to the dependency list, as follows. After this, the test should be passing:

```
useEffect(() => {
  ...
}, [queryStrings, searchTerm]);
```

13. We need to ensure that hitting the **Next** button will maintain our search term. Right now, it won't. We can use the following test to fix that:

```
it("includes search term when moving to next page", async () => {
  global.fetch.mockResolvedValue(
    fetchResponseOk(tenCustomers)
  );
  await renderAndWait(<CustomerSearch />);
  await changeAndWait(element("input"), "name");
  await clickAndWait(buttonWithLabel("Next"));
  expect(global.fetch).toHaveBeenCalledWith(
    "/customers?after=9&searchTerm=name",
    expect.anything()
  );
});
```

14. To make this pass, let's force the behavior into `fetchData` with an addition to the `if` statement, as follows:

```
const fetchData = async () => {
  let queryString;
  if (queryStrings.length > 0 && searchTerm !== "") {
    queryString =
      queryStrings[queryStrings.length - 1]
      + `&searchTerm=${searchTerm}`;
  } else if (searchTerm !== '') {
    queryString = `?searchTerm=${searchTerm}`;
  } else if (queryStrings.length > 0) {
    queryString =
      queryStrings[queryStrings.length - 1];
  }
  ...
};
```

We've made this test pass... but this is a mess! Any `if` statement with so many moving parts (variables, operators, conditions, and so on) is a signal that the design isn't as good as it can be. Let's fix it.

Refactoring to simplify the component design

The issue is the `queryString` data structure and its historical counterpart, the `queryStrings` state variable. The construction is complex.

How about we just store the *original data* instead—the ID of the customer in the last table row? Then, we can construct the `queryString` data structure immediately before fetching, since in reality, `queryString` is an input to the `fetch` request only. Keeping the raw data seems like it will be simpler.

Let's plan out our refactor. At each of the following stages, our tests should still be passing, giving us confidence that we're still on the right path:

1. First, move the query string building logic from `handleNext` into `fetchData`, changing the values that are stored in `queryStrings` from query strings to customer IDs in the process.
2. Then, change the names of those variables, using your editor's search and replace facility.
3. Finally, simplify the logic in `fetchData`.

Doesn't sound so hard, does it? Let's begin, as follows:

1. At the top of the component, replace the `queryStrings` variable with this new one:

```
const [lastRowIds, setLastRowIds] = useState([]);
```

2. Use your editor's search and replace facility to change all occurrences of `queryStrings` to `lastRowIds`.
3. Likewise, change the call to `setQueryStrings` to a call to `setLastRowIds`. Your tests should still be passing at this point.
4. Delete the following line from `handleNext`:

```
const newQueryString = `?after=${after}`;
```

5. On the line below that, change the call to `fetchData` to pass in `after` instead of the now deleted `newQueryString`, as follows:

```
const handleNext = useCallback(() => {
  const after = customers[customers.length - 1].id;
  setLastRowIds([...lastRowIds, after]);
}, [customers, lastRowIds]);
```

6. In the same function, rename `after` `currentLastRowId`. Your tests should still be passing at this point.

7. It's time to simplify the logic within `fetchData`. Create a `searchParams` function that will generate the search parameters for us, given values for `after` and `searchTerm`. This can be defined outside of your component. The code is illustrated here:

```
const searchParams = (after, searchTerm) => {
  let pairs = [];
  if (after) {
    pairs.push(`after=${after}`);
  }
  if (searchTerm) {
    pairs.push(`searchTerm=${searchTerm}`);
  }
  if (pairs.length > 0) {
    return `?${pairs.join("&")}`;
  }
  return "";
};
```

8. Finally, update `fetchData` to use this new function in place of the existing query string logic, as shown here. At this point, your tests should be passing, with a vastly simpler and easier-to-understand implementation:

```
const fetchData = async () => {
  const after = lastRowIds[lastRowIds.length - 1];
  const queryString = searchParams(after, searchTerm);
  const response = await global.fetch(...);
};
```

You've now built a functional search component. You introduced a new helper, `changeAndWait`, and extracted out a `searchParams` function that could be reused in other places.

Next, we'll add a final mechanism to the `CustomerSearch` component.

Performing actions with render props

Each row of the table will hold a **Create appointment** action button. When the user has found the customer that they are searching for, they can press this button to navigate to the `AppointmentForm` component, creating an appointment for that customer.

We'll display these actions by using a **render prop** that is passed to `CustomerSearch`. The parent component—in our case, `App`—uses this to insert its own rendering logic into the child component. `App` will pass a function that displays a button that causes a view transition in `App` itself.

Render props are useful if the child component should be unaware of the context it's operating in, such as the workflow that `App` provides.

Unnecessarily complex code alert!

The implementation you're about to see could be considered more complex than it needs to be. There are other approaches to solving this problem: you could simply have `CustomerSearch` render `AppointmentFormLoader` directly, or you could allow `CustomerSearch` to render the button and then invoke a callback such as `onSelect(customer)`.

Render props are probably more useful to library authors than to any application authors since library components can't account for the context they run within.

The testing techniques we need for render props are much more complex than anything we've seen so far, which you can take as another sign that there are "better" solutions.

To begin with, we'll add the `renderCustomerActions` prop to `CustomerSearch` and render it in a new table cell. Follow these steps:

1. In `test/CustomerSearch.test.js`, write the following test:

```
it("displays provided action buttons for each customer",
  async () => {
  const actionSpy = jest.fn(() => "actions");
  global.fetch.mockResolvedValue(
    fetchResponseOk(oneCustomer)
  );
  await renderAndWait(
    <CustomerSearch
      renderCustomerActions={actionSpy}
    />
  );
  const rows = elements("table tbody td");
  expect(rows[rows.length - 1])
    .toContainText("actions");
}) ;
```

2. Set a default `renderCustomerActions` prop so that our existing tests won't start failing when we begin using the new prop, as follows. This goes at the bottom of `src/CustomerSearch.js`:

```
CustomerSearch.defaultProps = {
  renderCustomerActions: () => {}
};
```

3. Destructure that prop in the top line of the `CustomerSearch` component, like so:

```
export const CustomerSearch = (
  { renderCustomerActions }
) => {
  ...
};
```

4. Pass it through to `CustomerRow`, like so:

```
<CustomerRow
  customer={customer}
  key={customer.id}
  renderCustomerActions={renderCustomerActions}
/>
```

5. In `CustomerRow`, update the fourth `td` cell to call this new prop, as follows:

```
const CustomerRow = (
  { customer, renderCustomerActions }
) => (
  <tr>
    <td>{customer.firstName}</td>
    <td>{customer.lastName}</td>
    <td>{customer.phoneNumber}</td>
    <td>{renderCustomerActions()}</td>
  </tr>
);
```

6. For the next test, we want to check that this render prop receives the specific customer record that applies to that row. Here's how we can do this:

```
it("passes customer to the renderCustomerActions prop",
  async () => {
```

```
const actionSpy = jest.fn(() => "actions");
global.fetch.mockResolvedValue(
  fetchResponseOk(oneCustomer)
);
await renderAndWait(
  <CustomerSearch
    renderCustomerActions={actionSpy}
  />
);
expect(actionSpy).toBeCalledWith(oneCustomer[0]);
}) ;
```

7. To make this pass, all you have to do is update the JSX call that you just wrote to include the customer as a parameter, as follows:

```
<td>{ renderCustomerActions(customer) }</td>
```

That's all there is to invoking the render prop inside the `CustomerSearch` component. Where it gets difficult is test-driving the implementation of the render prop itself, in the `App` component.

Testing render props in additional render contexts

Recall that the `App` component has a `view` state variable that determines which component the user is currently viewing on the screen. If they are searching for customers, then `view` will be set to `searchCustomers`.

Pressing the **Create appointment** button on the `CustomerSearch` component should have the effect of setting `view` to `addAppointment`, causing the user's screen to hide the `CustomerSearch` component and show the `AppointmentForm` component.

We also need to set the `App` component's `customer` state variable to the customer that the user just selected in the `CustomerSearch` component.

All of this will be done in the render prop that `App` passes to `customer`.

The big question is: how do we test-drive the implementation of this render prop?

There are a few different ways we could do it:

- You could render an actual `CustomerSearch` component within your `App` components, navigate to a customer, and click the **Create appointment** button. While this is simple, it also introduces a dependency in your test suite, increasing its surface area. And since your current `App` tests have a module-level mock for `CustomerSearch`, you'd need to create a new test suite for those tests, which increases maintenance overhead.

- You could modify the `CustomerSearch` mock to have some mechanism to trigger a `render` prop. This involves making the mock definition more complex than the standard form. That is an immediate red flag for me, for the reasons stated in *Chapter 7, Testing useEffect and Mocking Components*. This solution moves to the back of the pile.
- You could pull out the `render` prop from your `CustomerSearch` component, render it, then find the **Create appointment** button and click it. This is the approach we'll continue with.

If we use our `render` and `renderAndWait` functions to render this additional prop, it will replace the rendered `App` component. We would then click the button and we'd observe nothing happening because `App` has gone.

What we need is a second React root that can be used to just render that *additional* piece of the DOM. Our test can simply pretend that *it* is the `CustomerSearch` component.

To do this, we need a new render component that we'll call `renderAdditional`. Let's add that now and then write our test, as follows:

1. In `test/reactTestExtensions.js`, add the following function definition, just below the definition of `renderAndWait`:

```
export const renderAdditional = (component) => {
  const container = document.createElement("div");
  act(() =>
    ReactDOM.createRoot(container).render(component)
  );
  return container;
};
```

2. In `test/App.test.js`, update the `import` statement to pull in this new extension, like so:

```
import {
  ...
  renderAdditional,
} from "./reactTestExtensions";
```

3. Locate the `search customers` nested `describe` block and add a `searchFor` helper function that calls the `render` prop for the supplied customer, as follows:

```
const searchFor = (customer) =>
  propsOf(CustomerSearch)
    .renderCustomerActions(customer);
```

- Now, add the test. This renders the prop and checks that a button has been rendered, as illustrated in the following code snippet:

```
it("passes a button to the CustomerSearch named Create appointment", async () => {
  render(<App />);
  navigateToSearchCustomers();
  const buttonContainer =
    renderAdditional(searchFor());
  expect(
    buttonContainer.firstChild
  ).toBeElementWithTag("button");
  expect(
    buttonContainer.firstChild
  ).toContainText("Create appointment");
});
```

- In `src/App.js`, add the following function just above the returned JSX:

```
const searchActions = () => (
  <button>Create appointment</button>
);
```

- Set the prop on `CustomerSearch`, as follows. Your test should pass after this change:

```
case "searchCustomers":
  return (
    <CustomerSearch
      renderCustomerActions={searchActions}
    />
  );
};
```

- Back in `test/CustomerSearch.test.js`, add the next test, as follows. This uses the same helper function, but this time clicks the button and verifies that `AppointmentFormLoader` was shown with the correct customer ID:

```
it("clicking appointment button shows the appointment form for that customer", async () => {
  const customer = { id: 123 };
  render(<App />);
  navigateToSearchCustomers();
```

```
const buttonContainer = renderAdditional(
  searchFor(customer)
);
click(buttonContainer.firstChild);

expect(
  element("#AppointmentFormLoader")
).not.toBeNull();
expect(
  propsOf(AppointmentFormLoader).original
).toMatchObject({ customer: 123 });
});
```

8. To make that pass, update `searchActions` in `src/App.js` to use the `customer` parameter that will be passed to it by `CustomerSearch`, as follows:

```
const searchActions = (customer) => (
  <button
    onClick={
      () => transitionToAddAppointment(customer)
    }>
    Create appointment
  </button>
);
```

That's all there is to it: you've now used `renderAdditional` to trigger your render props and check that it works as expected.

This technique can be very handy when working with third-party libraries that expect you to pass render props.

That completes this feature; go ahead and manually test if you'd like to see it all in action.

Summary

This chapter has explored building out a component with some complex user interactions between the user interface and an API. You've created a new table component and integrated it into the existing application workflow.

You have seen how to make large changes to your component's implementation, using your tests as a safety mechanism.

You have also seen how to test render props using an additional render root—a technique that I hope you don't have to use too often!

In the next chapter, we'll use tests to integrate React Router into our application. We'll continue with the `CustomerSearch` component by adding the ability to use the browser location bar to specify search criteria. That will set us up nicely for introducing Redux and GraphQL later on.

Exercises

1. Disable the **Previous** button if the user is on the first page and disable the **Next** button if the current listing has fewer than 10 records on display.
2. Extract the `searchParams` function into a separate module that handles any number of parameters and uses the `encodeURIComponent` JavaScript function to ensure the values are encoded correctly.
3. The `/customers` endpoint supports a `limit` parameter that allows you to specify the maximum number of records that are returned. Provide a mechanism for the user to change the limit on each page.

11

Test-Driving React Router

React Router is a popular library of components that integrate with the browser's own navigation system. It manipulates the browser's address bar so that changes in your UI appear as page transitions. To the user, it seems like they are navigating between separate pages. In reality, they remain on the same page and avoid an expensive page reload.

In this chapter, we'll refactor our example appointments system to make use of React Router. Unlike the rest of the book, this chapter is not a walkthrough. That's because the refactoring process is quite long and laborious. Instead, we'll look at each of the main changes in turn.

This chapter covers the following:

- Designing React Router applications from a test-first perspective
- Testing components within a router
- Testing router links
- Testing programmatic navigation

By the end of the chapter, you'll have learned all the necessary techniques for test-driving React Router integrations.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter11>

Designing React Router applications from a test-first perspective

This section is a run-down of all the major pieces of the React Router ecosystem, just in case you’re not familiar with it. It also contains guidance on how to test a system that relies on React Router.

A list of all the React Router pieces

Here’s what you’ll be working with from the React Router library:

- A `Router` component. You’ll generally have one of these, and there are a bunch of different types. The basic one is `BrowserRouter` but you’ll undoubtedly upgrade to `HistoryRouter` if you need to manipulate history outside of the router, which, since you’re writing tests, you will. In *Chapter 12, Test-Driving Redux*, you’ll also see how this is necessary if you’re causing page transitions to occur within Redux actions.
- A `Routes` component. This is analogous to the `switch` statement in our existing `App` component. It has a list of `Route` children and will choose just one of those children to display at one time.
- A set of `Route` components with the `Routes` parent. Each `Route` has a `path` property, for example, `/addCustomer`, that the `Router` component uses to compare with the window’s current location. The route that matches is the one that is displayed.
- One or more `Link` components. These display like normal HTML hyperlinks, but they don’t act like them; React Router stops the browser from receiving these navigation events and instead sends them back to the `Routes` component, meaning a page transition doesn’t occur.
- The `useNavigate` hook. This is used to perform a page transition as part of a React side effect or event handler.
- The `useLocation` and `useSearchParams` hooks. These are used to get parts of the current window location within your components.

Splitting tests when the window location changes

You can see from this list that React Router’s core function is to manipulate the window location and modify your application’s behavior based on that location.

One way to think about this is that we will utilize the window location as a form of application state that is accessible to all our components. Importantly, this state persists across web requests, because a user can save or bookmark links for use later.

A consequence of this is that we must now split apart some of our unit tests. Take, for example, the **Create appointment** button that was previously used to switch out the main component on display on the page. With React Router in place, this button will become a link. Previously, we had a single unit test named as follows:

```
displays the AppointmentFormLoader after the CustomerForm is submitted
```

But now, we'll split that into two tests:

```
navigates to /addAppointment after the CustomerForm is submitted  
renders AppointmentFormRoute at /addAppointment
```

You can see that the first test stops at the moment the window location changes. The second test begins at the moment the browser navigates to the same location.

It's important to make this change because React Router isn't just refactoring, it's adding a new feature: the URL is now accessible as an entry point into your application.

That is, in essence, the most important thing you need to know before introducing React Router into your projects.

Up-front design for our new routes

Before launching into this refactor, let's take a look at the routes we'll be introducing:

- The default route, `/`, will remain as our `AppointmentsDayViewLoader` together with navigation buttons. This is extracted out as a new component named `MainScreen`.
- A route to add a new customer, at `/addCustomer`.
- A route to add a new appointment for a given customer, at `/addAppointment?customer=<id>`.
- A route to search for customers at `/searchCustomers`. This can receive a set of query string values: `searchTerm`, `limit`, and `previousRowIds`. For example, the query string might look as follows:

```
?searchTerm=An&limit=20&previousRowIds=123,456
```

Next, we'll look at test-driving a `Router` component along with its `Route` children.

Testing components within a router

In this section, we'll look at how to use the primary `Router`, `Routes`, and `Route` components.

No walkthrough in this chapter

As mentioned in the chapter introduction, this chapter does not follow the usual walkthrough approach. The examples shown here are taken from the completed refactoring of our Appointments code base, which you'll find in the `Chapter11/Complete` directory of the GitHub repository.

The `Router` component and its test equivalent

This is a top-level component that hooks into your browser's location mechanics. We do not generally test drive this because JSDOM doesn't deal with page transitions, or have full support for the `window.location` API.

Instead, we put it in the `src/index.js` file:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import { App } from "./App";

ReactDOM.createRoot(
  document.getElementById("root")
).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

This is necessary because if you try to use any of the other React Router components outside of a child of a `Router` component, it will blow up. The same is true for our tests: our components need to be rendered inside of a router. So, we introduce a new render helper called `renderWithRouter`.

This definition is within `test/reactTestExtensions.js`:

```
import { createMemoryHistory } from "history";
import {
  unstable_HistoryRouter as HistoryRouter
} from "react-router-dom";
```

```
export let history;

export const renderWithRouter = (
  component,
  { location } = { location: "" }
) => {
  history = createMemoryHistory({
    initialEntries: [location]
  });

  act(() =>
    reactRoot.render(
      <HistoryRouter history={history}>
        {component}
      </HistoryRouter>
    )
  );
};
```

MemoryRouter versus HistoryRouter

The React Router documentation will suggest you use `MemoryRouter`, which is often good enough. Using `HistoryRouter` allows you to control the history instance that is passed in, meaning you can manipulate it from within your tests.

For more information, take a look at <https://reacttdd.com/memory-router-vs-history-router>.

It's important to export the `history` variable itself if you want to manipulate the window location from within your own tests. A special case of this is if you want to set the window location before mounting the component; in this situation, you can simply pass a `location` property to the `renderWithRouter` function. You'll see how this works next.

Using the `Routes` component to replace a `switch` statement

Now let's look at using the `Routes` component to switch components depending on the window location. This component is generally at the top of the application component hierarchy, and in our case, it is indeed the first component within `App`.

The `Routes` component is analogous to the `switch` statement that existed in the original app. The `switch` statement was using a state variable to determine which component should be shown. The `Routes` component relies on the parent `Router` to feed it the window location as context.

Here's what the original `switch` statement looked like in the `App` component:

```
const [view, setView] = useState("dayView");

...

switch (view) {
  case "addCustomer":
    return (
      <CustomerForm ... />
    );
  case "searchCustomers":
    return (
      <CustomerSearch ... />
    );
  case "addAppointment":
    return (
      <AppointmentFormLoader ... />
    );
  default:
    return ...
}
```

Its `Router` replacement looks like this:

```
<Routes>
  <Route
    path="/addCustomer"
    element={<CustomerForm ... />}
  />
  <Route
    path="/addAppointment"
    element={<AppointmentFormRoute ... />}
  />
```

```
<Route
  path="/searchCustomers"
  element={<CustomerSearchRoute ... />}
/>
<Route path="/" element={<MainScreen />} />
</Routes>
```

The view state variable is no longer needed. Notice how we have a couple of new components with a Route suffix. These components are small wrappers that pull out the customer ID and other parameters from the window location before passing it to the original components. We'll look at those soon.

But first, how do the tests look for these new routes?

For the default route, the tests are simple, and are updates to the tests that were there before:

```
it("initially shows the AppointmentDayViewLoader", () => {
  renderWithRouter(<App />);
  expect(AppointmentsDayViewLoader).toBeRendered();
});

it("has a menu bar", () => {
  renderWithRouter(<App />);
  expect(element("menu")).not.toBeNull();
});
```

The only difference is that we use the `renderWithRouter` helper, not `render`.

The other routes are similar, except that they use the `location` property to set the initial window location, and their assertions are based on mocked components:

```
it("renders CustomerForm at the /addCustomer endpoint", () => {
  renderWithRouter(<App />, {
    location: "/addCustomer"
  });
  expect(CustomerForm).toBeRendered();
});

it("renders AppointmentFormRoute at /addAppointment", () => {
  renderWithRouter(<App />, {
    location: "/addAppointment?customer=123",
  });
});
```

```
    });
    expect(AppointmentFormRoute).toBeRendered();
});

it("renders CustomerSearchRoute at /searchCustomers", () => {
  renderWithRouter(<App />, {
    location: "/searchCustomers"
  });
  expect(CustomerSearchRoute).toBeRendered();
});
```

Using intermediate components to translate URL state

Let's take a closer look at `AppointmentFormRoute` and `CustomerSearchRoute`. What are these components doing?

Here's the definition of `AppointmentFormRoute`:

```
import React from "react";
import { useSearchParams } from "react-router-dom";
import {
  AppointmentFormLoader
} from "./AppointmentFormLoader";

const blankAppointment = {
  service: "",
  stylist: "",
  startsAt: null,
};

export const AppointmentFormRoute = (props) => {
  const [params, _] = useSearchParams();

  return (
    <AppointmentFormLoader
      {...props}
      original={{
        ...blankAppointment,
```

```
        customer: params.get("customer") ,
    } }
/>
);
};
```

This component is an intermediate component that sits between the `Route` component instance for `/addAppointment` and the `AppointmentFormLoader` component instance.

It would have been possible to simply reference the `useSearchParams` function from within `AppointmentFormLoader` itself, but by using this intermediate class, we can avoid modifying that component and keep the two responsibilities separate.

Having a single responsibility per component helps with comprehension. It also means that should we ever wish to rip out React Router at a later date, `AppointmentFormLoader` doesn't need to be touched.

There are a couple of interesting tests for this component. The first is the check for parsing the `customer` search parameter:

```
it("adds the customer id into the original appointment object",
() => {
  renderWithRouter(<AppointmentFormRoute />, {
    location: "?customer=123",
  });
  expect(AppointmentFormLoader).toBeRenderedWithProps({
    original: expect.objectContaining({
      customer: "123",
    }),
  });
});
```

The `location` property sent to `renderWithRouter` is just a standard query string: `?customer=123`. We could have entered a full URL here, but the test is clearer by focusing purely on the query string portion of the URL.

The second test is for the remainder of the props:

```
it("passes all other props through to AppointmentForm", () => {
  const props = { a: "123", b: "456" };
  renderWithRouter(<AppointmentFormRoute {...props} />);
```

```
expect(AppointmentFormLoader).toBeRenderedWithProps(  
  expect.objectContaining({  
    a: "123",  
    b: "456",  
  })  
);  
) ;
```

The test is important because the `Route` element passes through an `onSave` property that is for `AppointmentFormLoader`:

```
<Route  
  path="/addAppointment"  
  element={  
    <AppointmentFormRoute onSave={transitionToDayView} />  
  }  
/>
```

We'll look at what the `transitionToDayView` function does in the *Testing navigation* section a little further on.

Now let's see `CustomerSearchRoute`. This is a little more complicated because it parses some of the query string parameters, using a function called `convertParams`:

```
const convertParams = () => {  
  const [params] = useSearchParams();  
  const obj = {};  
  if (params.has("searchTerm")) {  
    obj.searchTerm = params.get("searchTerm");  
  }  
  if (params.has("limit")) {  
    obj.limit = parseInt(params.get("limit"), 10);  
  }  
  if (params.has("lastRowIds")) {  
    obj.lastRowIds = params  
      .get("lastRowIds")  
      .split(",")  
      .filter((id) => id !== "");  
  }  
}
```

```
    return obj;
};
```

This function replaces the three state variables that were used in the existing `CustomerSearch` component. Since all query string parameters are strings, each value needs to be parsed into the right format. These values are then passed into `CustomerSearch` as props:

```
import React from "react";
import {
  useNavigate,
  useSearchParams,
} from "react-router-dom";
import {
  CustomerSearch
} from "./CustomerSearch/CustomerSearch";

const convertParams = ...; // as above

export const CustomerSearchRoute = (props) => (
  <CustomerSearch
    {...props}
    navigate={useNavigate()}
    {...convertParams()}
  />
);
```

This parameter parsing functionality could have been put into `CustomerSearch`, but keeping that logic in a separate component helps with readability.

This example also shows the use of `useNavigate`, which is passed through to `CustomerSearch`. Passing this hook function return value as a prop means we can test `CustomerSearch` with a standard Jest spy function for the value of `navigate`, avoiding the need to render the test component within a router.

The tests for this component are straightforward. Let's take a look at one example:

```
it("parses lastRowIds from query string", () => {
  const location =
    "?lastRowIds=" + encodeURIComponent("1,2,3");
  renderWithRouter(<CustomerSearchRoute />, { location });
```

```
    expect(CustomerSearch).toBeRenderedWithProps(
      expect.objectContaining({
        lastRowIds: ["1", "2", "3"],
      })
    );
  });
}
```

You've now learned all there is to working with the three components: `Router`, `Routes`, and `Route`. Next up is the `Link` component.

Testing router links

In this section, you'll learn how to use and test the `Link` component. This component is React Router's version of the humble HTML anchor (or `a`) tag.

There are two forms of the `Link` component that we use. The first uses the `to` prop as a string, for example, `/addCustomer`:

```
<Link to="/addCustomer" role="button">
  Add customer and appointment
</Link>
```

The second sets the `to` prop to an object with a `search` property:

```
<Link
  to={{{
    search: objectToQueryString(queryParams),
  }}}
>
  {children}
</Link>
```

This object form also takes a `pathname` property, but we can avoid setting that since the path remains the same for our use case.

We'll look at two different ways of testing links: the standard way (by checking for hyperlinks), and the slightly more painful way of using mocks.

Checking the page for hyperlinks

Here's the `MainScreen` component in `src/App.js`, which shows the navigation links and the appointments day view:

```
export const MainScreen = () => (
  <>
    <menu>
      <li>
        <Link to="/addCustomer" role="button">
          Add customer and appointment
        </Link>
      </li>
      <li>
        <Link to="/searchCustomers" role="button">
          Search customers
        </Link>
      </li>
    </menu>
    <AppointmentsDayViewLoader />
  </>
) ;
```

Extracted component

The `MainScreen` component has been extracted out of `App`. The same code previously lived in the `switch` statement as the default case.

The `Link` component generates a standard HTML anchor tag. This means we create a helper to find a specific link by looking for an anchor tag with a matching `href` attribute. This is in `test/reactTestExtensions.js`:

```
export const linkFor = (href) =>
  elements("a").find(
    (el) => el.getAttribute("href") === href
  ) ;
```

That can be then used to test for the presence of a link and its caption:

```
it("renders a link to the /addCustomer route", async () => {
  renderWithRouter(<App />);
  expect(linkFor("/addCustomer")).toBeDefined();
});

it("captures the /addCustomer link as 'Add customer and
appointment'", async () => {
  renderWithRouter(<App />);
  expect(linkFor("/addCustomer")).toContainText(
    "Add customer and appointment"
);
});
```

Another way to test this would be to click the link and check that it works, as shown in the following test. However, as mentioned at the beginning of this chapter, this test isn't necessary because you've already tested the two "halves" of this test: that the link is displayed, and that navigating to the URL renders the right component:

```
it("displays the CustomerSearch when link is clicked", async () => {
  renderWithRouter(<App />);
  click(linkFor("/searchCustomers"));
  expect(CustomerSearchRoute).toBeRendered();
});
```

That covers the main way to test `Link` components. Another way to test links is to mock the `Link` component, which we'll cover next.

Mocking the Link component

This method is slightly more complicated than simply testing for HTML hyperlinks. However, it does mean you can avoid rendering your component under test within a `Router` component.

The `src/CustomerSearch/RouterButton.js` file contains this component:

```
import React from "react";
import {
  objectToQueryString
} from "../objectToQueryString";
```

```
import { Link } from "react-router-dom";

export const RouterButton = ({  
  queryParams,  
  children,  
  disabled,  
) => (  
  <Link  
    className={disabled ? "disabled" : ""}  
    role="button"  
    to={{  
      search: objectToQueryString(queryParams),  
    }}  
  >  
  {children}  
  </Link>  
) ;
```

To test this using plain `render`, instead of `renderWithRouter`, we'll need to mock out the `Link` component. Here's how that looks in `test/CustomerSearch/RouterButton.test.js`:

```
import { Link } from "react-router-dom";  
import {  
  RouterButton  
} from "../../src/CustomerSearch/RouterButton";  
  
jest.mock("react-router-dom", () => ({  
  Link: jest.fn(({ children }) => (  
    <div id="Link">{children}</div>  
  )),  
})) ;
```

Now, you can happily use that mock in your test:

```
it("renders a Link", () => {  
  render(<RouterButton queryParams={queryParams} />);  
  expect(Link).toBeRenderedWithProps({  
    className: "",  
  });  
});
```

```
        role: "button",
        to: {
          search: "?a=123&b=234",
        },
      });
}) ;
```

There's one final piece to think about. Sometimes, you have a single mocked component that has multiple rendered instances on the same page, and this happens frequently with `Link` instances.

In our case, this is the `SearchButtons` component, which contains a list of `RouterButton` and `ToggleRouterButton` components:

```
<menu>
  ...
<li>
  <RouterButton
    id="previous-page"
    queryParams={previousPageParams()}
    disabled={!hasPrevious}>
    Previous
  </RouterButton>
</li>
<li>
  <RouterButton
    id="next-page"
    queryParams={nextPageParams()}
    disabled={!hasNext}>
    Next
  </RouterButton>
</li>
</menu>
```

When it comes to testing these links, the simplest approach is to use `renderWithRouter` to render the `SearchButtons` components and then check the rendered HTML hyperlinks.

However, if you've decided to mock, then you need a way to easily find the element you've rendered. First, you'd specify the mock to include an `id` property:

```
jest.mock("../src/CustomerSearch/RouterButton", () => ({
  RouterButton: jest.fn(({ id, children }) => (
    <div id={id}>{children}</div>
  )),
}));
```

Then, you can use a new test extension called `propsMatching` to find the specific instance. Here's the definition from `test/reactTestExtensions.js`:

```
export const propsMatching = (mockComponent, matching) => {
  const [k, v] = Object.entries(matching)[0];
  const call = mockComponent.mock.calls.find(
    ([props]) => props[k] === v
  );
  return call?.[0];
};
```

You can then write your test to make use of that, as shown in the following code. Remember though, it's probably going to be easier not to mock this component and simply use `renderWithRouter`, and then check the HTML hyperlinks directly:

```
const previousPageButtonProps = () =>
  propsMatching(RouterButton, { id: "previous-page" });

it("renders", () => {
  render(<SearchButtons {...testProps} />);
  expect(previousPageButtonProps()).toMatchObject({
    disabled: false,
  });
  expect(element("#previous-page")).toContainText(
    "Previous"
  );
});
```

That's everything there is to testing the `Link` component. In the next section, we'll look at the final aspect of testing React Router: navigating programmatically.

Testing programmatic navigation

Sometimes, you'll want to trigger a location change programmatically—in other words, without waiting for a user to click a link.

There are two ways to do this: one using the `useNavigate` hook, and the second using a `history` instance that you pass into your top-level router.

Navigation inside and outside of components

In this chapter, we'll look at just the first method, using the hook. Later, in *Chapter 12, Test-Driving Redux*, we'll use the second method to change the location within a Redux action.

The `useNavigate` hook is the appropriate method when you're able to navigate from within a React component.

In the Appointments application, this happens in two places. The first is after a customer has been added and we want to move the user on to the `/addAppointment` route. The second is after that form has been completed and the appointment has been created—then we want to move them back to the default route.

Since these are very similar, we'll look at just the first.

Here's how the `/addCustomer` route definition looks in `src/App.js`:

```
<Route
  path="/addCustomer"
  element={
    <CustomerForm
      original={blankCustomer}
      onSave={transitionToAddAppointment}
    />
  }
/>
```

Notice the `onSave` prop; this is the callback that gets called once the customer form submission is completed. Here's that callback definition, together with the bits relevant for the `useNavigate` hook:

```
import {
  ...
  useNavigate,
} from "react-router-dom";
```

```
export const App = () => {
  const navigate = useNavigate();
  const transitionToAddAppointment = (customer) =>
    navigate(`/addAppointment?customer=${customer.id}`);
  ...
};
```

When it comes to testing this, clearly, we can't simply rely on the presence of a `Link` component, because there isn't one. Instead, we must call the `onSave` callback:

```
import {
  ...,
  history,
} from "./reactTestExtensions";

...

it("navigates to /addAppointment after the CustomerForm is submitted", () => {
  renderWithRouter(<App />);
  click(linkFor("/addCustomer"));
  const onSave = propsOf(CustomerForm).onSave;
  act(() => onSave(customer));
  expect(history.location.pathname).toEqual(
    "/addAppointment"
  );
});
```

The expectation is to test that the history is updated correctly. This history is the exported constant from `test/reactTestExtensions.js` that is set in the `renderWithRouter` function that we defined in the *Testing components within a router* section.

There is a variation of this. Instead of using the `history` import, you could also simply use the `window.location` instance:

```
expect(
  window.location.pathname
).toEqual("/addAppointment");
```

You've now learned how to test programmatic React Router navigation.

In the next chapter, *Test-Driving Redux*, we'll see how we can use this same history instance from a Redux saga.

Summary

This chapter has shown you how to use React Router in a testable fashion. You have learned how to test-drive the Router, Routes, Route, and Link components. You have seen how to use the React Router useSearchParams and useNavigate hooks.

Most importantly, you've seen that because routes give an extra level of entry into your application, you must split your existing navigation tests into two parts: one to test that a link exists (or is followed), and one to check that if you visit that URL, the right component is displayed.

Now that we've successfully integrated one library, the next one shouldn't be too tricky, right? In the next chapter, we'll apply all the skills we've learned in this chapter to the integration of another library: Redux.

Exercise

In this chapter, there was no walkthrough because the refactoring process is quite involved and would have taken up a decent chunk of time and space.

Use this opportunity to try refactoring yourself. Use a *systematic refactoring* approach to break down the change to React Router into many small steps. At each step, you should still have working software.

You can find a guide on how to approach this type of refactoring at <https://reacttdd.com/refactoring-to-react-router>.

Further reading

The official React Router documentation can be found at the following link:

<https://reacttraining.com/react-router/>

12

Test-Driving Redux

Redux is a **predictable state container**. To the uninitiated, these words mean very little. Thankfully, TDD can help us understand how to think about and implement our Redux application architecture. The tests in the chapter will help you see how Redux can be integrated into any application.

The headline benefit of Redux is the ability to share state between components in a way that provides data consistency when operating in an asynchronous browser environment. The big drawback is that you must introduce a whole bunch of plumbing and complexity into your application.

Here be dragons

For many applications, the complexity of Redux outweighs the benefits. Just because this chapter exists in this book does not mean you should be rushing out to use Redux. In fact, I hope that the code samples contained herein serve as warning enough for the complexity you will be introducing.

In this chapter, we'll build a reducer and a saga to manage the submission of our `CustomerForm` component.

We'll use a testing library named `expect-redux` to test Redux interactions. This library allows us to write tests that are not tied to the `redux-saga` library. Being independent of libraries is a great way of ensuring that your tests are not brittle and are resilient to change: you could replace `redux-saga` with `redux-thunk` and your tests would still work.

This chapter covers the following topics:

- Up-front design for a reducer and a saga
- Test-driving a reducer
- Test-driving a saga
- Switching component state for Redux state

By the end of the chapter, you'll have seen all the techniques you need for testing Redux.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter12>

Up-front design for a reducer and a saga

In this section, we'll do the usual thing of mapping out a rough plan of what we're going to build.

Let's start by looking at what the actual technical change is going to be and discuss why we're going to do it.

We're going to move the logic for submitting a customer—the `doSave` function in `CustomerForm`—out of the React component and into Redux. We'll use a Redux reducer to manage the status of the operation: whether it's currently submitting, finished, or had a validation error. We'll use a Redux saga to perform the asynchronous operation.

Why Redux?

Given the current feature set of the application, there's really no reason to use Redux. However, imagine that in the future, we'd like to support these features:

- After adding a new customer, the `AppointmentForm` component shows the customer information just before submitting it, without having to re-fetch the data from the server
- After finding a customer from the `CustomerSearch` component and choosing to create an appointment, the same customer information is shown on the appointment screen, without having to re-fetch the data

In this future scenario, it *might* make sense to have some shared Redux state for the customer data.

I say “*might*” because there are other, potentially simpler solutions: component context, or perhaps some kind of HTTP response caching. Who knows what the solution would look like? It's too hard to say without a concrete requirement.

To sum up: in this chapter, we'll use Redux to store customer data. It has no real benefit over our current approach, and in fact, has the drawback of all the additional plumbing. However, let's press on, given that the purpose of this book is educational.

Designing the store state and actions

A Redux store is simply an object of data with some restrictions on how it is accessed. Here's how we want ours to look. The object encodes all the information that `CustomerForm` already uses about a `fetch` request to save customer data:

```
{
  customer: {
    status: SUBMITTING | SUCCESSFUL | FAILED | ...

    // only present if the customer was saved successfully
    customer: { id: 123, firstName: "Ashley" ... },

    // only present if there are validation errors
    validationErrors: { phoneNumber: "..." },

    // only present if there was another type of error
    error: true | false
  }
}
```

Redux changes this state by means of named actions. We will have the following actions:

- `ADD_CUSTOMER_REQUEST`, called when the user presses the button to submit a customer. This triggers the saga, which then fires off the remaining actions
- `ADD_CUSTOMER_SUBMITTING`, when the saga begins its work
- `ADD_CUSTOMER_SUCCESSFUL`, when the server saves the customer and returns a new customer ID. With this action, we'll also save the new customer information in the reducer, ready for later use
- `ADD_CUSTOMER_VALIDATION_FAILED`, if the provided customer data is invalid
- `ADD_CUSTOMER_FAILED`, if there is some other reason the server fails to save data

For reference, here's the existing code that we'll be extracting from `CustomerForm`. It's all helpfully in one function, `doSave`, even though it is quite long:

```
const doSave = async () => {
  setSubmitting(true);
  const result = await global.fetch("/customers", {
    method: "POST",
    body: JSON.stringify({
      id: 123,
      firstName: "Ashley",
      lastName: "Harrison",
      email: "ashley@example.com",
      phone: "+1 234 567 8901"
    })
  });
  if (result.ok) {
    const data = await result.json();
    setCustomer(data);
  } else {
    const errors = await result.json();
    setValidationErrors(errors);
  }
}
```

```
        credentials: "same-origin",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(customer),
    );
    setSubmitting(false);
    if (result.ok) {
        setError(false);
        const customerWithId = await result.json();
        onSave(customerWithId);
    } else if (result.status === 422) {
        const response = await result.json();
        setValidationErrors(response.errors);
    } else {
        setError(true);
    }
};
```

We'll replace all this code with a combination of a saga and reducer. We'll start with the reducer, in the next section.

Test-driving a reducer

In this section, we'll test-drive a new reducer function, and then pull out some repeated code.

A reducer is a simple function that takes an action and the current store state as input and returns a new state object as output. Let's build that now, as follows:

1. Create a new file (in a new directory) named `test/reducers/customer.test.js`. Add the following first test, which checks that if the reducer is invoked with an unknown action, our reducer should return a default state for our object. This is standard behavior for Redux reducers, so you should always start with a test like this:

```
import { reducer } from "../../src/reducers/customer";

describe("customer reducer", () => {
    it("returns a default state for an undefined existing state", () => {
        expect(reducer(undefined, {})).toEqual({
            customer: {},
            status: undefined,
```

```
        validationErrors: {},
        error: false
    });
});
});
```

2. Create a `src/reducers/customer.js` file, as follows, and let's make that test pass:

```
const defaultState = {
  customer: {},
  status: undefined,
  validationErrors: {},
  error: false
};

export const reducer = (state = defaultState, action) =>
{
  return state;
};
```

3. For the next test, add in support for the `ADD_CUSTOMER_SUBMITTING` action, as follows. This test checks that when this action is received, the `status` value is updated to `SUBMITTING`:

```
describe("ADD_CUSTOMER_SUBMITTING action", () => {
  const action = { type: "ADD_CUSTOMER_SUBMITTING" };

  it("sets status to SUBMITTING", () => {
    expect(reducer(undefined, action)).toMatchObject({
      status: "SUBMITTING"
    });
  });
});
```

4. Make that pass by replacing the body of the reducer with the following code. We can jump directly to using a `switch` statement here (rather than using an `if` statement) because we know for certain that we'll be filling out other action types:

```
switch(action.type) {
  case "ADD_CUSTOMER_SUBMITTING":
    return { status: "SUBMITTING" };
}
```

```
    default:
        return state;
    }
```

5. Add a second test to the `ADD_CUSTOMER_SUBMITTING` describe block, as follows. This test specifies behavior that's expected for reducer actions: any state that we don't care about (which is `status` in this case) is maintained:

```
it("maintains existing state", () => {
    expect(reducer({ a: 123 }, action)).toMatchObject({
        a: 123
    });
});
```

6. Make that pass by modifying the reducers, as follows:

```
export const reducer = (state = defaultState, action) =>
{
    switch (action.type) {
        case "ADD_CUSTOMER_SUBMITTING":
            return { ...state, status: "SUBMITTING" };
        default:
            return state;
    }
};
```

7. We need to handle the `ADD_CUSTOMER_SUCCESSFUL` action. Start with the two tests shown next. I'm cheating by writing two tests at once, but that's fine because I know they are a close replica of the `ADD_CUSTOMER_SUBMITTING` tests:

```
describe("ADD_CUSTOMER_SUCCESSFUL action", () => {
    const customer = { id: 123 };
    const action = {
        type: "ADD_CUSTOMER_SUCCESSFUL",
        customer
    };

    it("sets status to SUCCESSFUL", () => {
        expect(reducer(undefined, action)).toMatchObject({
            status: "SUCCESSFUL"
    });
});
```

```
    }) ;

    });

    it("maintains existing state", () => {
      expect(
        reducer({ a: 123 }, action)
      ).toMatchObject({ a: 123 });
    });
  });
}
```

8. To make that pass, add a final `case` statement to your reducer, like so:

```
case "ADD_CUSTOMER_SUCCESSFUL":
  return { ...state, status: "SUCCESSFUL" };
```

9. Add a third test, shown next. The action provides a new `customer` object with its assigned ID, which we should save in the reducer for later use:

```
it("sets customer to provided customer", () => {
  expect(reducer(undefined, action)).toMatchObject({
    customer
  });
});
```

10. Make that pass by adding in the `customer` property, as follows:

```
case "ADD_CUSTOMER_SUCCESSFUL":
  return {
    ...state,
    status: "SUCCESSFUL",
    customer: action.customer
  };
};
```

11. Add the next `describe` block, for `ADD_CUSTOMER_FAILED`, as follows:

```
describe("ADD_CUSTOMER_FAILED action", () => {
  const action = { type: "ADD_CUSTOMER_FAILED" };

  it("sets status to FAILED", () => {
    expect(reducer(undefined, action)).toMatchObject({
```

```
        status: "FAILED"
    );
});

it("maintains existing state", () => {
    expect(
        reducer({ a: 123 }, action)
    ).toMatchObject({ a: 123 });
});
});
```

12. Make those both pass by adding a new `case` statement to the `switch` reducer, like so:

```
case "ADD_CUSTOMER_FAILED":
    return { ...state, status: "FAILED" };
```

13. We aren't quite done with `ADD_CUSTOMER_FAILED`. In this case, we also want to set `error` to `true`. Recall that we used an `error` state variable in the `CustomerForm` component to mark when an unexplained error had occurred. We need to replicate that here. Add this third test to the `describe` block, as follows:

```
it("sets error to true", () => {
    expect(reducer(undefined, action)).toMatchObject({
        error: true
    });
});
```

14. Make that pass by modifying the `case` statement, as shown here:

```
case "ADD_CUSTOMER_FAILED":
    return { ...state, status: "FAILED", error: true };
```

15. Add tests for the `ADD_CUSTOMER_VALIDATION_FAILED` action, which occurs if field validation failed. The code is illustrated here:

```
describe("ADD_CUSTOMER_VALIDATION_FAILED action", () => {
    const validationErrors = { field: "error text" };
    const action = {
        type: "ADD_CUSTOMER_VALIDATION_FAILED",
        validationErrors
    };
});
```

```
};

it("sets status to VALIDATION_FAILED", () => {
  expect(reducer(undefined, action)).toMatchObject({
    status: "VALIDATION_FAILED"
  });
});

it("maintains existing state", () => {
  expect(
    reducer({ a: 123 }, action)
  ).toMatchObject({ a: 123 });
});
});
```

16. Make these tests pass with another `case` statement in the reducer, as follows:

```
case "ADD_CUSTOMER_VALIDATION_FAILED":
  return { ...state, status: "VALIDATION_FAILED" };
```

17. This action also needs a third test. This time, the action can include error information on what the validation errors were, as shown in the following code snippet:

```
it("sets validation errors to provided errors", () => {
  expect(reducer(undefined, action)).toMatchObject({
    validationErrors
  });
});
```

18. Make that pass with the change shown here:

```
case "ADD_CUSTOMER_VALIDATION_FAILED":
  return {
    ...state,
    status: "VALIDATION_FAILED",
    validationErrors: action.validationErrors
  };
};
```

That completes the reducer, but before we use it from within a saga, how about we dry these tests up a little?

Pulling out generator functions for reducer actions

Most reducers will follow the same pattern: each action will set some new data to ensure that the existing state is not lost.

Let's write a couple of test-generator functions to do that for us, to help us dry up our tests. Proceed as follows:

1. Create a new file, `test/reducerGenerators.js`, and add the following function to it:

```
export const itMaintainsExistingState = (reducer, action) => {
  it("maintains existing state", () => {
    const existing = { a: 123 };
    expect(
      reducer(existing, action)
    ).toEqual(existing);
  });
};
```

2. Add the following `import` statement to the top of `src/reducers/customer.test.js`:

```
import {
  itMaintainsExistingState
} from "../reducerGenerators";
```

3. Modify your tests to use this function, deleting the test in each `describe` block and replacing it with the following single line:

```
itMaintainsExistingState(reducer, action);
```

4. Back in `test/reducerGenerators.js`, define the following function:

```
export const itSetsStatus = (reducer, action, value) => {
  it(`sets status to ${value}`, () => {
    expect(reducer(undefined, action)).toEqual({
      status: value
    });
  });
};
```

5. Modify the existing import statement to pull in the new function, like so:

```
import {  
  itMaintainsExistingState,  
  itSetsStatus  
} from "../reducerGenerators";
```

6. Modify your tests to use this function, just as you did before. Make sure you run your tests to prove they work! Your tests should now be much shorter. Here's an example of the describe block for ADD_CUSTOMER_SUCCESSFUL:

```
describe("ADD_CUSTOMER_SUBMITTING action", () => {  
  const action = { type: "ADD_CUSTOMER_SUBMITTING" };  
  
  itMaintainsExistingState(reducer, action);  
  itSetsStatus(reducer, action, "SUBMITTING");  
});
```

That concludes the reducer. Before we move on to the saga, let's tie it into the application. We won't make use of it at all, but it's good to get the plumbing in now.

Setting up a store and an entry point

In addition to the reducer we've written, we need to define a function named `configureStore` that we'll then call when our application starts. Proceed as follows:

1. Create a new file named `src/store.js` with the following content. There's no need to test this just yet, as it's a bit like `src/index.js`: plumbing that connects everything together. However, we will utilize it in the next section when we test the saga:

```
import { createStore, combineReducers } from "redux";  
import {  
  reducer as customerReducer  
} from "./reducers/customer";  
  
export const configureStore = (storeEnhancers = []) =>  
  createStore(  
    combineReducers({ customer: customerReducer }),  
    storeEnhancers  
  );
```

2. In `src/index.js`, add the following two `import` statements to the top of the file:

```
import { Provider } from "react-redux";
import { configureStore } from "./store";
```

3. Then, wrap the existing JSX in a `Provider` component, as shown here. This is how all our components will gain access to the Redux store:

```
ReactDOM.createRoot(
  document.getElementById("root")
).render(
  <Provider store={configureStore()}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
);
```

With that in place, we're ready to write the tricky part: the saga.

Test-driving a saga

A saga is a special bit of code that uses JavaScript generator functions to manage asynchronous operations to the Redux store. Because it's super complex, we won't actually test the saga itself; instead, we'll dispatch an action to the store and observe the results.

Before we get started on the saga tests, we need a new test helper function named `renderWithStore`.

Adding the `renderWithStore` test extension

Proceed as follows:

1. At the top of `test/reactTestExtensions.js`, add the following new `import` statements:

```
import { Provider } from "react-redux";
import { storeSpy } from "expect-redux";
import { configureStore } from "../src/store";
```

The `expect-redux` package

For that, we'll use the `expect-redux` package from NPM, which has already been included in the `package.json` file for you—make sure to run `npm install` before you begin.

2. Add a new variable, `store`, and initialize it in `initializeReactContainer`, as illustrated in the following code snippet. This makes use of `storeSpy` from `expect-redux`, which we'll use in our tests to check calls to the store:

```
export let store;

export const initializeReactContainer = () => {
  store = configureStore([storeSpy]);
  container = document.createElement("div");
  document.body.replaceChildren(container);
  reactRoot = ReactDOM.createRoot(container);
};
```

3. Add your new render function below the `renderWithRouter` function, as illustrated in the following code snippet:

```
export const renderWithStore = (component) =>
  act(() =>
    reactRoot.render(
      <Provider store={store}>{component}</Provider>
    )
  );
};
```

4. Finally, add `dispatchStore`, which we'll need when we start dispatching actions in our component, as follows:

```
export const dispatchToStore = (action) =>
  act(() => store.dispatch(action));
```

You've now got all the helpers you need to begin testing both sagas and components that are connected to a Redux store. With all that in place, let's get started on the saga tests.

Using `expect-redux` to write expectations

The saga we're writing will respond to an `ADD_CUSTOMER_REQUEST` action that's dispatched from the `CustomerForm` component when the user submits the form. The functionality of the saga is just the same as the `doSave` function listed in the *Designing the store state and actions* section at the beginning of this chapter. The difference is we'll need to use the saga's function calls of `put`, `call`, and so forth.

Let's begin by writing a generator function named `addCustomer`. Proceed as follows:

1. Create a new file (in a new directory) named `test/sagas/customer.test.js` and add the following code to set up our `describe` block. We initialize a `store` variable that both our sagas and our test expectations will make use of. This is a repeat of the code we had previously in our `initializeReactContainer` test helper—which we can't use here because we're not writing a component:

```
import { storeSpy, expectRedux } from "expect-redux";
import { configureStore } from "../../src/store";

describe("addCustomer", () => {
  let store;

  beforeEach(() => {
    store = configureStore([ storeSpy ]);
  });
}) ;
```

2. Just below the `beforeEach` block, add the following helper function, which gives us a slightly more elegant way of constructing the action—you'll see that in the first test, coming up next:

```
const addCustomerRequest = (customer) => ({
  type: "ADD_CUSTOMER_REQUEST",
  customer,
}) ;
```

3. Now for the first test. What is the first thing our saga should do? It must update our store state to reflect that the form is submitting. That way, the `CustomerForm` component can immediately show a submitting indicator to the user. We use an expectation from `expect-redux` to ensure that we dispatch the right action, as shown here:

```
it("sets current status to submitting", () => {
  store.dispatch(addCustomerRequest());

  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "ADD_CUSTOMER_SUBMITTING" });
}) ;
```

Returning promises from tests

This test returns a promise. This is a shortcut we can use instead of marking our test function as `async` and the expectation with `await`. Jest knows to wait if the test function returns a promise.

4. Let's start with the saga implementation. Create a new file named `src/sagas/customer.js` with the following content. Notice the `function*` syntax, which signifies a generator function, and the use of `put` to fire off another action to the store:

```
import { put } from "redux-saga/effects";

export function* addCustomer() {
  yield put({ type: "ADD_CUSTOMER_SUBMITTING" });
}
```

Generator-function syntax

The arrow-function syntax that we've been using throughout the book does not work for generator functions, so we need to fall back to using the `function` keyword.

5. Before that test will pass, we need to update the store with a **root saga**. That root saga then registers our `addCustomer` saga. Starting with the imports statements, update `src/store.js` to read as follows:

```
import {
  createStore,
  applyMiddleware,
  compose,
  combineReducers
} from "redux";
import createSagaMiddleware from "redux-saga";
import { takeLatest } from "redux-saga/effects";
import { addCustomer } from "./sagas/customer";
import {
  reducer as customerReducer
} from "./sagas/customer";
```

6. Just below those imports, add this definition of `rootSaga`:

```
function* rootSaga() {
  yield takeLatest(
    "ADD_CUSTOMER_REQUEST",
    addCustomer
  );
}
```

7. Now, update `configureStore` to include the saga middleware and “run” `rootSaga`, like so. After this change, your test should pass:

```
export const configureStore = (storeEnhancers = []) => {
  const sagaMiddleware = createSagaMiddleware();

  const store = createStore(
    combineReducers({ customer: customerReducer }),
    compose(
      applyMiddleware(sagaMiddleware),
      ...storeEnhancers
    )
  );
  sagaMiddleware.run(rootSaga);

  return store;
};
```

That completes the first test for the saga, and gets all the necessary plumbing into place. You’ve also seen how to use `put`. Next up, let’s introduce `call`.

Making asynchronous requests with sagas

Within a saga, `call` allows us to perform an asynchronous request. Let’s introduce that now. Follow these steps:

1. Add the following test, to check the call to `fetch`:

```
it("sends HTTP request to POST /customers", async () => {
  const inputCustomer = { firstName: "Ashley" };
  store.dispatch(addCustomerRequest(inputCustomer));
```

```
expect(global.fetch).toBeCalledWith(
  "/customers",
  expect.objectContaining({
    method: "POST",
  })
);
});
```

2. We'll need to define a spy on `global.fetch` for this to work. Change the `beforeEach` block as follows, including the new `customer` constant:

```
beforeEach(() => {
  jest.spyOn(global, "fetch");
  store = configureStore([ storeSpy ]);
});
```

3. In `src/sagas/customer.js`, update the saga import to include the `call` function, like so:

```
import { put, call } from "redux-saga/effects";
```

4. Now, create a `fetch` function and invoke it in the saga with `call`, as follows. After this, your test should be passing:

```
const fetch = (url, data) =>
  global.fetch(url, {
    method: "POST",
  });

export function* addCustomer({ customer }) {
  yield put({ type: "ADD_CUSTOMER_SUBMITTING" });
  yield call(fetch, "/customers", customer);
}
```

5. Alright—now, let's add a test to add in the configuration for our `fetch` request, as follows:

```
it("calls fetch with correct configuration", async () =>
{
  const inputCustomer = { firstName: "Ashley" };
  store.dispatch(addCustomerRequest(inputCustomer));
```

```
    expect(global.fetch).toBeCalledWith(
      expect.anything(),
      expect.objectContaining({
        credentials: "same-origin",
        headers: { "Content-Type": "application/json" },
      })
    );
});
```

6. To make that pass, add the following lines to the `fetch` definition:

```
const fetch = (url, data) =>
  global.fetch(url, {
    method: "POST",
    credentials: "same-origin",
    headers: { "Content-Type": "application/json" }
});
```

7. Now, let's test that we're sending the right customer data across. Here's how we can do that:

```
it("calls fetch with customer as request body", async () => {
  const inputCustomer = { firstName: "Ashley" };
  store.dispatch(addCustomerRequest(inputCustomer));

  expect(global.fetch).toBeCalledWith(
    expect.anything(),
    expect.objectContaining({
      body: JSON.stringify(inputCustomer),
    })
  );
});
```

8. To make that pass, complete the `fetch` definition, as shown here:

```
const fetch = (url, data) =>
  global.fetch(url, {
    body: JSON.stringify(data),
    method: "POST",
```

```
        credentials: "same-origin",
        headers: { "Content-Type": "application/json" }
    };
```

9. For the next test, we want to dispatch an ADD_CUSTOMER_SUCCESSFUL event when the `fetch` call returns successfully. It uses a constant named `customer` that we'll define in the next step. Here's the code we need to execute:

```
it("dispatches ADD_CUSTOMER_SUCCESSFUL on success", () =>
{
    store.dispatch(addCustomerRequest());

    return expectRedux(store)
        .toDispatchAnAction()
        .matching({
            type: "ADD_CUSTOMER_SUCCESSFUL",
            customer
        });
});
```

10. When we set up our `fetch` spy before, we didn't set a return value. So, now, create a `customer` constant and set up the `fetch` spy to return it, like so:

```
const customer = { id: 123 };

beforeEach(() => {
    jest
        .spyOn(global, "fetch")
        .mockReturnValue(fetchResponseOk(customer));
    store = configureStore([ storeSpy ]);
});
```

11. Import `fetchResponseOk`, like so. After this, you'll be able to run your test:

```
import { fetchResponseOk } from "../builders/fetch";
```

12. Make the test pass by processing the result from the `call` function, like so:

```
export function* addCustomer({ customer }) {
    yield put({ type: "ADD_CUSTOMER_SUBMITTING" });
```

```
    const result = yield call(fetch, "/customers",
customer);
    const customerWithId = yield call([result, "json"]);
    yield put({
        type: "ADD_CUSTOMER_SUCCESSFUL",
        customer: customerWithId
    });
}
```

13. What about if the `fetch` call isn't successful, perhaps because of a network failure? Add a test for that, as follows:

```
it("dispatches ADD_CUSTOMER_FAILED on non-specific
error", () => {
    global.fetch.mockReturnValue(fetchResponseError());
    store.dispatch(addCustomerRequest());

    return expectRedux(store)
        .toDispatchAnAction()
        .matching({ type: "ADD_CUSTOMER_FAILED" });
});
```

14. That test makes use of `fetchResponseError`; import it now, like so:

```
import {
    fetchResponseOk,
    fetchResponseError
} from "../builders/fetch";
```

15. Make the test pass by wrapping the existing code in an `if` statement with an `else` clause, as follows:

```
export function* addCustomer({ customer }) {
    yield put({ type: "ADD_CUSTOMER_SUBMITTING" });
    const result = yield call(
        fetch,
        "/customers",
        customer
    );
    if(result.ok) {
```

```
const customerWithId = yield call(
  [result, "json"]
);
yield put({
  type: "ADD_CUSTOMER_SUCCESSFUL",
  customer: customerWithId
});
} else {
  yield put({ type: "ADD_CUSTOMER_FAILED" });
}
}
```

16. Finally, add a test for a more specific type of failure—a validation failure, as follows:

```
it("dispatches ADD_CUSTOMER_VALIDATION_FAILED if
validation errors were returned", () => {
  const errors = {
    field: "field",
    description: "error text"
  };
  global.fetch.mockReturnValue(
    fetchResponseError(422, { errors })
  );

  store.dispatch(addCustomerRequest());

  return expectRedux(store)
    .toDispatchAnAction()
    .matching({
      type: "ADD_CUSTOMER_VALIDATION_FAILED",
      validationErrors: errors
    });
});
```

17. Make that pass with the following code:

```
export function* addCustomer({ customer }) {
  yield put({ type: "ADD_CUSTOMER_SUBMITTING" });
```

```

        const result = yield call(fetch, "/customers",
customer);
        if(result.ok) {
            const customerWithId = yield call(
                [result, "json"]
            );
            yield put({
                type: "ADD_CUSTOMER_SUCCESSFUL",
                customer: customerWithId
            });
        } else if (result.status === 422) {
            const response = yield call([result, "json"]);
            yield put({
                type: "ADD_CUSTOMER_VALIDATION_FAILED",
                validationErrors: response.errors
            });
        } else {
            yield put({ type: "ADD_CUSTOMER_FAILED" });
        }
    }
}

```

The saga is now complete. Compare this function to the function in `CustomerForm` that we're replacing: `doSave`. The structure is identical. That's a good indicator that we're ready to work on removing `doSave` from `CustomerForm`.

In the next section, we'll update `CustomerForm` to make use of our new Redux store.

Switching component state for Redux state

The saga and reducer are now complete and ready to be used in the `CustomerForm` React component. In this section, we'll replace the use of `doSave`, and then as a final flourish, we'll push our React Router navigation into the saga, removing the `onSave` callback from `App`.

Submitting a React form by dispatching a Redux action

At the start of the chapter, we looked at how the purpose of this change was essentially a transplant of `CustomerForm`'s `doSave` function into a Redux action.

With our new Redux setup, we used component state to display a submitting indicator and show any validation errors. That information is now stored within the Redux store, not component state. So, in addition to dispatching an action to replace `doSave`, the component also needs to read state from the store. The component state variables can be deleted.

This has a knock-on effect on our tests. Since the saga tests the failure modes, our component tests for `CustomerForm` simply need to handle various states of the Redux store, which we'll manipulate using our `dispatchToStore` extension.

We'll start by making our component Redux-aware, as follows:

1. Add the following `import` statement to the top of `test/CustomerForm.test.js`:

```
import { expectRedux } from "expect-redux";
```

2. Update the test extensions `import` statement, replacing `render` with `renderWithStore`, and adding the two new imports, as follows:

```
import {
  initializeReactContainer,
  renderWithStore,
  dispatchToStore,
  store,
  ...
} from "./reactTestExtensions";
```

3. Replace all calls to `render` with `renderWithStore`. Be careful if you're doing a search and replace operation: the word `render` appears in some of the test descriptions, and you should keep those as they are.
4. Let's rework a single test: the one with the sends HTTP request to POST /customers when submitting data description. Change that test to the following:

```
it("dispatches ADD_CUSTOMER_REQUEST when submitting data", async () => {
  renderWithStore(
    <CustomerForm {...validCustomer} />
  );
  await clickAndWait(submitButton());
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({
      type: 'ADD_CUSTOMER_REQUEST',
```

```

    customer: validCustomer
  });
}
);

```

- To make this pass, we'll use a side-by-side implementation to ensure our other tests continue to pass. In `handleSubmit`, add the line highlighted in the following code snippet. This calls a new `addCustomerRequest` prop that we'll define soon:

```

const handleSubmit = async (event) => {
  event.preventDefault();
  const validationResult = validateMany(
    validators, customer
  );
  if (!anyErrors(validationResult)) {
    await doSave();
    dispatch(addCustomerRequest(customer));
  } else {
    setValidationErrors(validationResult);
  }
};

```

- That makes use of the `useDispatch` hook. Import that now, as follows:

```
import { useDispatch } from "react-redux";
```

- Then, add this line to the top of the `CustomerForm` component:

```
const dispatch = useDispatch();
```

- To make the test pass, all that's left is the definition of `addCustomerRequest`, which you can add just below the `import` statements and above the `CustomerForm` component definition, like so:

```

const addCustomerRequest = (customer) => ({
  type: "ADD_CUSTOMER_REQUEST",
  customer,
});

```

At this point, your component is now Redux-aware, and it's dispatching the right action to Redux. The remaining work is to modify the component to deal with validation errors coming from Redux rather than the component state.

Making use of store state within a component

Now, it's time to introduce the `useSelector` hook to pull out state from the store. We'll kick things off with the `ADD_CUSTOMER_FAILED` generic error action. Recall that when the reducer receives this, it updates the `error` store state value to `true`. Follow these steps:

1. Find the test named `renders error message when fetch call fails`. Replace it with the implementation shown here. It simulates an `ADD_CUSTOMER_FAILED` action so that we make sure all the Redux wiring is correct. Don't forget to remove the `async` keyword from the test function:

```
it("renders error message when error prop is true", () =>
{
  renderWithStore(
    <CustomerForm {...validCustomer} />
  );
  dispatchToStore({ type: "ADD_CUSTOMER_FAILED" });
  expect(element("[role=alert]").toContainText(
    "error occurred"
  ));
});
```

2. Add an import statement for the `useSelector` hook at the top of `src/CustomerForm.js`, as follows:

```
import {
  useDispatch,
  useSelector
} from "react-redux";
```

3. Call the `useSelector` hook at the top of the `CustomerForm` component, as shown in the following code snippet. It pulls out the `error` state value from the `customer` section of the Redux store:

```
const {
  error,
} = useSelector(({ customer }) => customer);
```

4. Delete any line where `setError` is called. There are two occurrences, both in `doSave`.
5. Now, you can delete the `error`/`setError` pair of variables that are defined with the `useState` hook at the top of `CustomerForm`. Your tests won't run until you do this, due to `error` being declared twice. Your tests should be passing at this stage.

6. The next test, `clears error message when fetch call succeeds`, can be deleted. The reducer, as it stands, doesn't actually do this; completing it is one of the exercises in the *Exercise* section.
7. Find the `does not submit the form when there are validation errors` test and update it as follows. It should pass already:

```
it("does not submit the form when there are validation errors", async () => {
  renderWithStore(
    <CustomerForm original={blankCustomer} />
  );
  await clickAndWait(submitButton());
  return expectRedux(store)
    .toNotDispatchAnAction(100)
    .ofType("ADD_CUSTOMER_REQUEST");
});
```

The `toNotDispatchAnAction` matcher

This matcher should always be used with a timeout, such as 100 milliseconds in this case. That's because, in an asynchronous environment, events may just be slow to occur, rather than not occurring at all.

8. Find the next test, `renders field validation errors from server`. Replace it with the following code, remembering to remove the `async` keyword from the function definition:

```
it("renders field validation errors from server", () => {
  const errors = {
    phoneNumber: "Phone number already exists in the system"
  };
  renderWithStore(
    <CustomerForm {...validCustomer} />
  );
  dispatchToStore({
    type: "ADD_CUSTOMER_VALIDATION_FAILED",
    validationErrors: errors
  });
});
```

```
expect(
  errorFor(phoneNumber)
).toContainText(errors.phoneNumber);
});
```

9. To make this pass, we need to pull out `validationErrors` from the Redux customer store. There's a bit of complexity here: the component already has a `validationErrors` state variable that covers *both* server and client validation errors. We can't replace that entirely, because it handles client errors in addition to server errors.

So, let's *rename* the prop we get back from the server, like so:

```
const {
  error,
  validationErrors: serverValidationErrors,
} = useSelector(({ customer }) => customer);
```

A design issue

This highlights a design issue in our original code. The `validationErrors` state variable had *two* uses, which were mixed up. Our change here will separate those uses.

10. We're not done with this test just yet. Update the `renderError` function to render errors for both `validationErrors` (client-side validation) and `serverValidationErrors` (server-side validation), as follows:

```
const renderError = fieldName => {
  const allValidationErrors = {
    ...validationErrors,
    ...serverValidationErrors
  };
  return (
    <span id={`$ {fieldName}error`} role="alert">
      {hasError(allValidationErrors, fieldName)
        ? allValidationErrors[fieldName]
        : ""}
    </span>
  );
};
```

11. The next tests we need to look at are for the submitting indicator. We'll update these tests to respond to store actions rather than a form submission. Here's the first one:

```
it("displays indicator when form is submitting", () => {
  renderWithStore(
    <CustomerForm {...validCustomer} />
  );
  dispatchToStore({
    type: "ADD_CUSTOMER_SUBMITTING"
  });
  expect(
    element(".submittingIndicator")
  ).not.toBeNull();
});
```

12. To make this pass, add `status` to the `useSelector` call, like so:

```
const {
  error,
  status,
  validationErrors: serverValidationErrors,
} = useSelector(({ customer }) => customer);
```

13. Delete anywhere that `setSubmitting` is called within this component.
 14. Delete the state variable for `submitting`, and replace it with the following line of code. The test should now pass:

```
const submitting = status === "SUBMITTING";
```

15. Then, update the test named `hides indicator when form has submitted`, as follows. This test won't need any change to the production code:

```
it("hides indicator when form has submitted", () => {
  renderWithStore(
    <CustomerForm {...validCustomer} />
  );
  dispatchToStore({
    type: "ADD_CUSTOMER_SUCCESSFUL"
  });
});
```

```
    expect(element(".submittingIndicator")).toBeNull();
});
```

16. Finally, find the disable the submit button when submitting test and modify it in the same way as *Step 12*.

That's it for test changes, and `doSave` is almost fully redundant. However, the call to `onSave` still needs to be migrated across into the Redux saga, which we'll do in the next section.

Navigating router history in a Redux saga

Recall that it is the `App` component that renders `CustomerForm`, and `App` passes a function to the `CustomerForm`'s `onSave` prop that causes page navigation. When the customer information has been submitted, the user is moved onto the `/addAppointment` route.

But now that the form submission happens within a Redux saga, how do we call the `onSave` prop? The answer is that we can't. Instead, we can move page navigation into the saga itself and delete the `onSave` prop entirely.

To do this, we must update `src/index.js` to use `HistoryRouter` rather than `BrowserRouter`. That allows you to pass in your own history singleton object, which you can then explicitly construct yourself and then access via the saga. Proceed as follows:

1. Create a new file named `src/history.js` and add the following content to it. This is very similar to what we already did in `test/reactTestExtensions.js`:

```
import { createBrowserHistory } from "history";

export const appHistory = createBrowserHistory();
```

2. Update `src/index.js`, as shown here:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import {
  unstable_HistoryRouter as HistoryRouter
} from "react-router-dom";
import { appHistory } from "./history";
import { configureStore } from "./store";
import { App } from "./App";

ReactDOM.createRoot(
```

```

    document.getElementById("root")
).render(
  <Provider store={configureStore()}>
    <HistoryRouter history={appHistory}>
      <App />
    </HistoryRouter>
  </Provider>
);

```

3. Now, we can use `appHistory` in our saga. Open `test/sagas/customer.js` and add the following `import` statement to the top of the file:

```
import { appHistory } from "../../src/history";
```

4. Then, add the following two tests to define how the navigation should occur:

```

it("navigates to /addAppointment on success", () => {
  store.dispatch(addCustomerRequest());
  expect(appHistory.location.pathname).toEqual(
    "/addAppointment"
  );
});

it("includes the customer id in the query string when
navigating to /addAppointment", () => {
  store.dispatch(addCustomerRequest());
  expect(
    appHistory.location.search
  ).toEqual("?customer=123");
});

```

5. To make these pass, start by opening `src/sagas/customer.js` and adding the following `import` statement:

```
import { appHistory } from "../history";
```

6. Then, update the `addCustomer` generator function to navigate after a customer has been added successfully, like so:

```

export function* addCustomer({ customer }) {
  ...

```

```
yield put({
  type: "ADD_CUSTOMER_SUCCESSFUL",
  customer: customerWithId,
}) ;
appHistory.push(
  `/addAppointment?customer=${customerWithId.id}`
) ;
}
```

7. Now, all that's left is to delete the existing `onSave` plumbing from `App` and `CustomerForm`. Open `test/App.test.js` and delete the following three tests:

- calls `fetch` with correct configuration
 - navigates to `/addAppointment` after the `CustomerForm` is submitted
 - passes saved customer to `AppointmentFormLoader` after the `CustomerForm` is submitted
8. You can also delete the `beforeEach` block that sets up `global.fetch` in the nested `describe` block labeled when `POST` request returns an error.
 9. In `src/App.js`, delete the definition of `transitionToAddAppointment` and change the `/addCustomer` route to have no `onSave` prop, as shown in the following code snippet. Your `App` tests should be passing at this point:

```
<Route
  path="/addCustomer"
  element={<CustomerForm original={blankCustomer} />}
/>
```

10. Now, we can delete the `onSave` prop from `CustomerForm`. Start by deleting the following tests from the `CustomerForm` test suite that are no longer necessary:
 - notifies `onSave` when form is submitted
 - does not notify `onSave` if the `POST` request returns an error
11. Delete the `onSave` prop from the `CustomerForm` component.
12. Finally, remove the invocation of `doSave` from `handleSubmit`. This function no longer awaits anything, so you can safely remove `async` from the function definition. At this point, all your tests should be passing.

You've now seen how you can integrate a Redux store into your React components, and how you can control React Router navigation from within a Redux saga.

All being well, your application should now be running with Redux managing the workflow.

Summary

This has been a whirlwind tour of Redux and how to refactor your application to it, using TDD.

As warned in the introduction of this chapter, Redux is a complex library that introduces a lot of extra plumbing into your application. Thankfully, the testing approach is straightforward.

In the next chapter, we'll add yet another library: Relay, the GraphQL client.

Exercise

- Modify the customer reducer to ensure that `error` is reset to `false` when the `ADD_CUSTOMER_SUCCESSFUL` action occurs.

Further reading

For more information, have a look at the following sources:

- MDN documentation on generator functions:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function*
- Home page for the `expect-redux` package:
<https://github.com/rradczewski/expect-redux>

13

Test-Driving GraphQL

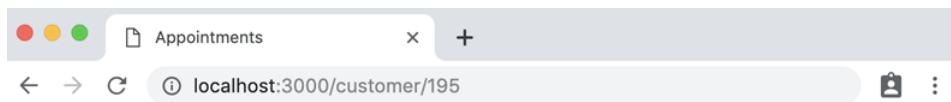
GraphQL offers an alternative to HTTP requests for fetching data. It offers a whole bunch of additional features that can be added to data requests.

As with Redux, GraphQL systems can seem complicated, but TDD helps to provide an approach to understanding and learning.

In this chapter, we'll use the **Relay** library to connect to our backend. We're going to build a new `CustomerHistory` component that displays details of a single customer and their appointment history.

This is a bare-bones GraphQL implementation that shows the fundamentals of test-driving the technology. If you're using other GraphQL libraries instead of Relay, the techniques we'll explore in this chapter will also apply.

Here's what the new `CustomerHistory` component looks like:



The screenshot shows a web browser window with a light gray header bar. On the left are three colored window control buttons (red, yellow, green). In the center is the title bar with the text "Appointments". To the right of the title bar are a close button ("x") and a plus sign ("+"). Below the title bar is a navigation bar with icons for back, forward, and refresh, followed by the URL "localhost:3000/customer/195". To the right of the URL are a magnifying glass icon and a three-dot menu icon. The main content area of the browser displays a user profile for "Danielle Emard". The profile includes her name in bold, a phone number "(381) 477-5075", and a section titled "Booked appointments". Below this, there is a table with four columns: "When", "Stylist", "Service", and "Notes". Two rows of data are shown in the table.

When	Stylist	Service	Notes
Wed Feb 20 2019 16:00	Ashley	Cut	
Wed May 08 2019 10:00	Jo	Blow-dry	

Figure 13.1 – The new `CustomerHistory` component

This chapter covers the following topics:

- Compiling the schema before you begin
- Test-driving the Relay environment
- Fetching GraphQL data from within a component

By the end of the chapter, you'll have explored the test-driven approach to GraphQL.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter13>

Compiling the schema before you begin

The code samples for this chapter already contain some additions:

- The `react-relay`, `relay-compiler`, and `babel-plugin-relay` packages.
- Babel configuration to ensure your build understands the new GraphQL syntax.
- Relay configuration in the `relay.config.json` file. The primary piece of configuration is the location of the schema.
- A GraphQL schema in the file `src/schema.graphql`.
- A server endpoint at `POST/graphq1`, which services incoming GraphQL requests.

It's beyond the scope of this book to go into each of these, but you will need to compile the schema before you begin, which can be done by typing the following command:

```
npx relay-compiler
```

The `npm run build` command has also been modified to run this command for you, just in case you forget. Once everything is compiled, you're ready to write some tests.

Testing the Relay environment

There are a few different ways to approach the integration of Relay into a React application. The method we'll use in this book is the `fetchQuery` function, which is analogous to the `global.fetch` function we've already used for standard HTTP requests.

However, Relay's `fetchQuery` function has a much more complicated setup than `global.fetch`.

One of the parameters of the `fetchQuery` function is the *environment*, and in this section, we'll see what that is and how to construct it.

Why Do We Need to Construct an Environment?

The Relay environment is an extension point where all manner of functionality can be added. Data caching is one example. If you're interested in how to do that, check out the *Further reading* section at the end of this chapter.

We will build a function named `buildEnvironment`, and then another named `getEnvironment` that provides a singleton instance of this environment so that the initialization only needs to be done once. Both functions return an object of type `Environment`.

One of the arguments that the `Environment` constructor requires is a function named `performFetch`. This function, unsurprisingly, is the bit that actually fetches data – in our case, from the `POST /graphql` server endpoint.

In a separate test, we'll check whether `performFetch` is passed to the new `Environment` object. We need to treat `performFetch` as its own unit because we're not going to be testing the behavior of the resulting environment, only its construction.

Building a `performFetch` function

Let's begin by creating our own `performFetch` function:

1. Create a new file, `test/relayEnvironment.test.js`, and add the following setup. This sets up our `global.fetch` spy in the same way as usual. There are two new constants here, `text` and `variables`, which we'll use soon:

```
import {
  fetchResponseOk,
  fetchResponseError
} from "./builders/fetch";
import {
  performFetch
} from "../src/relayEnvironment";

describe("performFetch", () => {
  let response = { data: { id: 123 } };
  const text = "test";
```

```

const variables = { a: 123 };

beforeEach(() => {
  jest
    .spyOn(global, "fetch")
    .mockResolvedValue(fetchResponseOk(response));
});

});

```

2. Then, add the first test, checking that we make the appropriate HTTP request. The call to `performFetch` takes two parameters that contain `text` (wrapped in an object) and `variables`. This mimics how the Relay environment will call the `performFetch` function for each request:

```

it("sends HTTP request to POST /graphql", () => {
  performFetch({ text }, variables);
  expect(global.fetch).toBeCalledWith(
    "/graphql",
    expect.objectContaining({
      method: "POST",
    })
  );
});

```

3. Create a new file, `src/relayEnvironment.js`, and make the test pass with the following code:

```

export const performFetch = (operation, variables) =>
  global
    .fetch("/graphql", {
      method: "POST",
    });

```

4. Add the second of our tests for the HTTP request dance, which ensures we pass the correct request configuration:

```

it("calls fetch with the correct configuration", () => {
  performFetch({ text }, variables);
  expect(global.fetch).toBeCalledWith(
    "/graphql",

```

```
expect.objectContaining({
  credentials: "same-origin",
  headers: { "Content-Type": "application/json" },
})
);
}) ;
```

5. Make that pass by adding the two lines highlighted here:

```
export const performFetch = (operation, variables) =>
  global
    .fetch("/graphql", {
      method: "POST",
      credentials: "same-origin",
      headers: { "Content-Type": "application/json" },
    });
}
```

6. Then, add the third and final test of our HTTP request dance. This one checks that we pass the right request data – the required `text` query and the `variables` argument included within it:

```
it("calls fetch with query and variables as request body", async () => {
  performFetch({ text }, variables);
  expect(global.fetch).toBeCalledWith(
    "/graphql",
    expect.objectContaining({
      body: JSON.stringify({
        query: text,
        variables,
      }),
    })
  );
});
```

7. Make that pass by defining the `body` property for the `fetch` request, as shown here:

```
export const performFetch = (operation, variables) =>
  global
    .fetch("/graphql", {
```

```

        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          query: operation.text,
          variables
        })
      ) ;
    }
  ) ;
}

```

Understanding Operation, Text, and Variables

The `text` property of the `operation` argument is a static piece of data that defines the query, and the `variables` argument will be the piece that is relevant to this specific request.

The tests we're writing in this chapter do not go as far as checking the *behavior* of this Relay plumbing code. When writing this type of unit test, which doesn't exercise behavior, it's important to note that some kind of **end-to-end test** will be necessary. That will ensure your unit tests have the right specification.

8. The next test checks that we return the right data from the function. Relay expects our `performFetch` function to return a promise that will either resolve or reject. In this case, we're resolving it to the `fetch` response:

```

it("returns the request data", async () => {
  const result = await performFetch(
    { text }, variables
  );
  expect(result).toEqual(response);
});

```

9. Make that pass:

```

export const performFetch = (operation, variables) =>
  global
    .fetch("/graphql", ...)
    .then(result => result.json());

```

10. Now, we need to handle the error cases. We need the promise to reject if an HTTP error occurred. We use a new form of the `expect` function that we haven't seen before; it takes a promise and expects it to reject:

```

it("rejects when the request fails", () => {

```

```
global.fetch.mockResolvedValue(
  fetchResponseError(500)
);
return expect(
  performFetch({ text }, variables)
).rejects.toEqual(new Error(500));
});
```

11. In our production code, we'll test whether the fetch response's `ok` property is `false`, and reject the promise if it is. Add the following function:

```
const verifyStatusOk = result => {
  if (!result.ok) {
    return Promise.reject(new Error(500));
  } else {
    return result;
  }
};
```

12. Call that function within your promise chain. After this, our `performFetch` function is complete:

```
export const performFetch = (operation, variables) =>
  global
    .fetch("/graphql", ...)
    .then(verifyStatusOk)
    .then(result => result.json());
```

You've now learned how to specify and test the `performFetch` function required for the Environment constructor. Now, we're ready to do that construction.

Test-driving the Environment object construction

We're going to build a function named `buildEnvironment`, that takes all the various pieces we need to build an `Environment` object. The reason there are so many pieces is that they are all extension points that enable the configuration of the Relay connection.

These pieces are our `performFetch` function and a bunch of other Relay types that come directly from the `relay-runtime` package. We'll use `jest.mock` to mock all these out in one fell swoop.

Let's get started:

1. In the same test file, `test/relayEnvironment.test.js`, update your import to include the new function:

```
import {
  performFetch,
  buildEnvironment
} from "../src/relayEnvironment";
```

2. Now, it's time to import all the relevant pieces we need from the `relay-runtime` package and mock them out. Add the following to the top of the file:

```
import {
  Environment,
  Network,
  Store,
  RecordSource
} from "relay-runtime";
jest.mock("relay-runtime");
```

3. For our first test, we need to test that the `Environment` constructor was called:

```
describe("buildEnvironment", () => {
  const environment = { a: 123 };

  beforeEach(() => {
    Environment.mockImplementation(() => environment);
  });

  it("returns environment", () => {
    expect(buildEnvironment()).toEqual(environment);
  });
});
```

4. Start by adding all the imports in the production code in `src/relayEnvironment.js`:

```
import {
  Environment,
  Network,
```

```
    RecordSource,  
    Store  
} from "relay-runtime";
```

5. Make the test pass by adding this code at the bottom of the file:

```
export const buildEnvironment = () =>  
  new Environment();
```

6. The second test makes sure we pass the right arguments to `Environment`. Its first argument is the result of calling `Network.create`, and the second argument is the result of constructing a `Store` object. The tests need to mock those out and then check the return values:

```
describe("buildEnvironment", () => {  
  const environment = { a: 123 };  
  const network = { b: 234 };  
  const store = { c: 345 };  
  
  beforeEach(() => {  
    Environment.mockImplementation(() => environment);  
    Network.create.mockReturnValue(network);  
    Store.mockImplementation(() => store);  
  });  
  
  it("returns environment", () => {  
    expect(buildEnvironment()).toEqual(environment);  
  });  
  
  it("calls Environment with network and store", () => {  
    expect(Environment).toBeCalledWith({  
      network,  
      store  
    });  
  });  
});
```

Mocking Constructors

Note the difference in how we mock out constructors and function calls. To mock out a new `Store` and a new `Environment`, we need to use `mockImplementation(fn)`. To mock out `Network.create`, we need to use `mockReturnValue(returnValue)`.

7. Make the test pass by updating the function to pass those arguments to the `Environment` constructor:

```
export const buildEnvironment = () =>
  new Environment({
    network: Network.create(),
    store: new Store()
 });
```

8. Next up, we need to ensure that `Network.create` gets a reference to our `performFetch` function:

```
it("calls Network.create with performFetch", () => {
  expect(Network.create).toBeCalledWith(performFetch);
});
```

9. Make that pass by passing `performFetch` to the `Network.create` function:

```
export const buildEnvironment = () =>
  new Environment({
    network: Network.create(performFetch),
    store: new Store()
 });
```

10. The `Store` constructor needs a `RecordSource` object. Add a new mock implementation for `RecordSource` in your test setup:

```
describe("buildEnvironment", () => {
  ...
  const recordSource = { d: 456 };

  beforeEach(() => {
    ...
    RecordSource.mockImplementation(
      () => recordSource
    );
  });
});
```

```
) ;  
} ) ;  
...  
} ) ;
```

11. Add the following test to specify the behavior we want:

```
it("calls Store with RecordSource", () => {  
  expect(Store).toBeCalledWith(recordSource);  
});
```

12. Make that pass by constructing a new RecordSource object:

```
export const buildEnvironment = () =>  
  new Environment({  
    network: Network.create(performFetch),  
    store: new Store(new RecordSource())  
  });
```

And that, would you believe, is it for `buildEnvironment`! At this stage, you will have a valid `Environment` object.

Test-driving a singleton instance of Environment

Because creating `Environment` takes a substantial amount of plumbing, it's common to construct it once and then use that value for the rest of the application.

An Alternative Approach Using `RelayEnvironmentProvider`

There is an alternative approach to using the singleton instance shown here, which is to use React Context. The `RelayEnvironmentProvider` component provided by Relay can help you with that. For more information, see the *Further reading* section at the end of the chapter.

Let's build the `getEnvironment` function:

1. Import the new function at the top of `test/relayEnvironment.test.js`:

```
import {  
  performFetch,  
  buildEnvironment,  
  getEnvironment  
} from "../src/relayEnvironment";
```

2. At the bottom of the file, add a third describe block with the one and only one test for this function:

```
describe("getEnvironment", () => {
  it("constructs the object only once", () => {
    getEnvironment();
    getEnvironment();
    expect(Environment.mock.calls.length).toEqual(1);
  });
});
```

3. In `src/relayEnvironment.js`, make that pass by introducing a top-level variable that stores the result of `getEnvironment` if it hasn't yet been called:

```
let environment = null;
export const getEnvironment = () =>
  environment || (environment = buildEnvironment());
```

That's all for the environment boilerplate. We now have a shiny `getEnvironment` function that we can use within our React components.

In the next section, we'll start on the `CustomerHistory` component.

Fetching GraphQL data from within a component

Now that we have a Relay environment, we can begin to build out our feature. Recall from the introduction that we're building a new `CustomerHistory` component that displays customer details and a list of the customer's appointments. A GraphQL query to return this information already exists in our server, so we just need to call it in the right way. The query looks like this:

```
customer(id: $id) {
  id
  firstName
  lastName
  phoneNumber
  appointments {
    startsAt
    stylist
    service
    notes
```

```
    }  
}
```

This says we get a customer record for a given customer ID (specified by the `$id` parameter), together with a list of their appointments.

Our component will perform this query when it's mounted. We'll jump right in with that functionality, by testing the call to `fetchQuery`:

1. Create a new file, `test/CustomerHistory.test.js`, and add the following setup. We're going to break this setup into parts, as it's long! First up is our import, and the call to mock `relay-runtime` again, so that we can stub `fetchQuery`:

```
import React from "react";  
import { act } from "react-dom/test-utils";  
import {  
  initializeReactContainer,  
  render,  
  renderAndWait,  
  container,  
  element,  
  elements,  
  textOf,  
} from "./reactTestExtensions";  
import { fetchQuery } from "relay-runtime";  
import {  
  CustomerHistory,  
  query  
} from "../src/CustomerHistory";  
import {  
  getEnvironment  
} from "../src/relayEnvironment";  
jest.mock("relay-runtime");  
jest.mock("../src/relayEnvironment");
```

2. Now, let's define some sample data:

```
const date = new Date("February 16, 2019");  
  
const appointments = [
```

```

    },
    startsAt: date.setHours(9, 0, 0, 0),
    stylist: "Jo",
    service: "Cut",
    notes: "Note one"
},
{
  startsAt: date.setHours(10, 0, 0, 0),
  stylist: "Stevie",
  service: "Cut & color",
  notes: "Note two"
}
];
}

const customer = {
  firstName: "Ashley",
  lastName: "Jones",
  phoneNumber: "123",
  appointments
};

```

3. Next, let's get `beforeEach` in place. This stubs out `fetchQuery` with a special `sendCustomer` fake, mimicking the return value of a `fetchQuery` request:

```

describe("CustomerHistory", () => {
  let unsubscribeSpy = jest.fn();

  const sendCustomer = ({ next }) => {
    act(() => next({ customer }));
    return { unsubscribe: unsubscribeSpy };
  };

  beforeEach(() => {
    initializeReactContainer();
    fetchQuery.mockReturnValue(
      { subscribe: sendCustomer }
    );
  };

```

```
    }) ;  
}) ;
```

The Return Value of `fetchQuery`

This function has a relatively complex usage pattern. A call to `fetchQuery` returns an object with `subscribe` and `unsubscribe` function properties. We call `subscribe` with an object with a `next` callback property. That callback is called by Relay's `fetchQuery` each time the query returns a result set. We can use that callback to set our component state. Finally, the `unsubscribe` function is returned from the `useEffect` block so that it's called when the component is unmounted or the relevant props change.

- Finally, add the test, which checks that we call `fetchQuery` in the expected way:

```
it("calls fetchQuery", async () => {  
  await renderAndWait(<CustomerHistory id={123} />);  
  expect(fetchQuery).toBeCalledWith(  
    getEnvironment(), query, { id: 123 }  
  );  
});
```

- Let's make that pass. Create a new file, `src/CustomerHistory.js`, and start it off with the imports and the exported `query` definition:

```
import React, { useEffect } from "react";  
import { fetchQuery, graphql } from "relay-runtime";  
import { getEnvironment } from "./relayEnvironment";  
  
export const query = graphql`  
  query CustomerHistoryQuery($id: ID!) {  
    customer(id: $id) {  
      id  
      firstName  
      lastName  
      phoneNumber  
      appointments {  
        startsAt  
        stylist  
        service  
      }  
    }  
  }  
};
```

```

        notes
    }
}
}
`;

```

- Add the component, together with a `useEffect` Hook:

```

export const CustomerHistory = ({ id }) => {
  useEffect(() => {
    fetchQuery(getEnvironment(), query, { id });
  }, [id]);

  return null;
};

```

- If you run tests now, you might see an error, as shown here:

```

Cannot find module './__generated__/
CustomerHistoryQuery.graphql' from 'src/CustomerHistory.
js'

```

To fix this, run the following command to compile your GraphQL query:

```
npx relay-compiler
```

- Next, we can add a test to show what happens when we pull out some data:

```

it("unsubscribes when id changes", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  await renderAndWait(<CustomerHistory id={234} />);
  expect(unsubscribeSpy).toBeCalled();
});

```

- To make that pass, update the `useEffect` block to return the `unsubscribe` function property:

```

useEffect(() => {
  const subscription = fetchQuery(
    getEnvironment(), query, { id }
  );

```

```
    return subscription.unsubscribe;
}, [id]);
```

10. Then, update your component to render that data, pulling in the customer data:

```
it("renders the first name and last name together in a h2", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  await new Promise(setTimeout);
  expect(element("h2")).toContainText("Ashley Jones");
});
```

11. Then, update your component to include a new state variable, `customer`. This is set by calling `setCustomer` in our definition of the next callback:

```
export const CustomerHistory = ({ id }) => {
  const [customer, setCustomer] = useState(null);

  useEffect(() => {
    const subscription = fetchQuery(
      getEnvironment(), query, { id }
    ).subscribe({
      next: ({ customer }) => setCustomer(customer),
    });
    return subscription.unsubscribe;
  }, [id]);
```

12. Make the test pass by extending your JSX to render the customer data:

```
const { firstName, lastName } = customer;
return (
  <>
  <h2>
    {firstName} {lastName}
  </h2>
  </>
);
```

13. Now, add a test to also render the customer's phone number:

```
it("renders the phone number", async () => {
    await renderAndWait(<CustomerHistory id={123} />);
    expect(document.body).toContainText("123");
});
```

14. Make that pass with the change shown here:

```
const { firstName, lastName, phoneNumber } = customer;
return (
  <>
  <h2>
    {firstName} {lastName}
  </h2>
  <p>{phoneNumber}</p>
</>
);
```

15. Now, let's get started on rendering the appointments:

```
it("renders a Booked appointments heading", async () => {
    await renderAndWait(<CustomerHistory id={123} />);
    expect(element("h3")).not.toBeNull();
    expect(element("h3")).toContainText(
      "Booked appointments"
    );
});
```

16. That's a quick one to fix; add in the h3 element, as shown here:

```
const { firstName, lastName, phoneNumber } = customer;
return (
  <>
  <h2>
    {firstName} {lastName}
  </h2>
  <p>{phoneNumber}</p>
  <h3>Booked appointments</h3>
```

```
</>
);
```

17. Next, we'll render a table for each of the appointments available:

```
it("renders a table with four column headings", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  const headings = elements(
    "table > thead > tr > th"
  );
  expect(textOf(headings)).toEqual([
    "When",
    "Stylist",
    "Service",
    "Notes",
  ]);
});
```

18. Add that table:

```
const { firstName, lastName, phoneNumber } = customer;
return (
  <>
  <h2>
    {firstName} {lastName}
  </h2>
  <p>{phoneNumber}</p>
  <h3>Booked appointments</h3>
  <table>
    <thead>
      <tr>
        <th>When</th>
        <th>Stylist</th>
        <th>Service</th>
        <th>Notes</th>
      </tr>
    </thead>
```

```

    </table>
  </>
);

```

19. For the next set of tests, we'll use a `columnValues` helper, which will find a rendered table element and pull out an array of all the values in a column. We can use this to test that our code displays data for a list of appointments, rather than just one:

```

const columnValues = (columnNumber) =>
  elements("tbody > tr").map(
    (tr) => tr.childNodes[columnNumber]
  );

it("renders the start time of each appointment in the
correct format", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  expect(textOf(columnValues(0))).toEqual([
    "Sat Feb 16 2019 09:00",
    "Sat Feb 16 2019 10:00",
  ]);
});

```

20. Add a new `tbody` element here, just below `thead`. This makes a reference to a new `AppointmentRow` component, which we haven't built yet, but we will do so in the next step:

```





```

21. Now, let's define `AppointmentRow`. Add this above the `CustomerHistory` definition. After this, your test should pass:

```
const toTimeString = (startsAt) =>
  new Date(Number(startsAt))
    .toString()
    .substring(0, 21);

const AppointmentRow = ({ appointment }) => (
  <tr>
    <td>{toTimeString(appointment.startsAt)}</td>
  </tr>
);
```

22. Let's add in the other columns, starting with the stylist:

```
it("renders the stylist", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  expect(textOf(columnValues(1))).toEqual([
    "Jo",
    "Stevie"
  ]);
});
```

23. Add that as the next column in `AppointmentRow`:

```
const AppointmentRow = ({ appointment }) => (
  <tr>
    <td>{toTimeString(appointment.startsAt)}</td>
    <td>{appointment.stylist}</td>
  </tr>
);
```

24. Next is the service field:

```
it("renders the service", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  expect(textOf(columnValues(2))).toEqual([
    "Cut",
    "Cut & color",
  ]);
```

```
    ] );
}) ;
```

25. Again, that involves simply adding a further `td` element to `AppointmentRow`:

```
const AppointmentRow = ({ appointment }) => (
  <tr>
    <td>{toTimeString(appointment.startsAt)}</td>
    <td>{appointment.stylist}</td>
    <td>{appointment.service}</td>
  </tr>
);
```

26. Finally, for rendering information, we'll show the `notes` field too.

```
it("renders notes", async () => {
  await renderAndWait(<CustomerHistory id={123} />);
  expect(textOf(columnValues(3))).toEqual([
    "Note one",
    "Note two",
  ]);
});
```

27. Complete the `AppointmentRow` component, as shown here:

```
const AppointmentRow = ({ appointment }) => (
  <tr>
    <td>{toTimeString(appointment.startsAt)}</td>
    <td>{appointment.stylist}</td>
    <td>{appointment.service}</td>
    <td>{appointment.notes}</td>
  </tr>
);
```

28. We're almost done. Let's display a **Loading** message when data is being submitted to the server. This test should be a new nested `describe` block, just below the test that we've just completed. It uses a `noSend` fake that does nothing; there's no call to `next`. This can be used to mimic the scenario when data is still loading:

```
describe("submitting", () => {
  const noSend = () => unsubscribeSpy;
```

```
beforeEach(() => {
  fetchQuery.mockReturnValue({ subscribe: noSend });
});

it("displays a loading message", async () => {
  await renderAndWait(<CustomerHistory id={123} />);

  expect(element("[role=alert]")).toContainText(
    "Loading"
  );
});
});
});
```

29. To make that pass, introduce a conditional just before the JSX:

```
export const CustomerHistory = ({ id }) => {
  const [customer, setCustomer] = useState(null);

  useEffect(() => {
    ...
  }, [id]);

  if (!customer) {
    return <p role="alert">Loading</p>;
  }
  ...
};
```

30. Finally, let's handle the case when there's an error fetching data. This uses another fake, `errorSend`, that invokes the error callback. It's like the `next` callback and can be used to set state, which we'll see in the next step:

```
describe("when there is an error fetching data", () => {
  const errorSend = ({ error }) => {
    act(() => error());
    return { unsubscribe: unsubscribeSpy };
  };
});
```

```

    beforeEach(() => {
      fetchQuery.mockReturnValue(
        { subscribe: errorSend }
      );
    });

    it("displays an error message", async () => {
      await renderAndWait(<CustomerHistory />);
      expect(element("[role=alert]")).toContainText(
        "Sorry, an error occurred while pulling data from
the server."
      );
    });
  );
}

```

31. To make that pass, you'll need to introduce a new `status` state variable. Initially, this has the `loading` value. When successful, it changes to `loaded`, and when an error occurs, it changes to `failed`. For the `failed` state, we render the specified error message:

```

const [customer, setCustomer] = useState(null);
const [status, setStatus] = useState("loading");

useEffect(() => {
  const subscription = fetchQuery(
    getEnvironment(), query, { id }
  ).subscribe({
    next: ({ customer }) => {
      setCustomer(customer);
      setStatus("loaded");
    },
    error: (_) => setStatus("failed"),
  })
}

return subscription.unsubscribe;
}, [id]);

if (status === "loading") {
  return <p role="alert">Loading</p>;
}

```

```
}

if (status === "failed") {
  return (
    <p role="alert">
      Sorry, an error occurred while pulling data from
      the server.
    </p>
  );
}

const { firstName, lastName, phoneNumber } = customer;
...
```

That completes the new `CustomerHistory` component. You have now learned how to test-drive the use of Relay's `fetchQuery` function in your application, and this component is now ready to integrate with `App`. This is left as an exercise.

Summary

This chapter has explored how to test-drive the integration of a GraphQL endpoint using Relay. You have seen how to test-drive the building of the Relay environment, and how to build a component that uses the `fetchQuery` API.

In *Part 3, Interactivity*, we'll begin work in a new code base that will allow us to explore more complex use cases involving undo/redo, animation, and WebSocket manipulation.

In *Chapter 14, Building a Logo Interpreter*, we'll begin by writing new Redux middleware to handle undo/redo behavior.

Exercises

Integrate the `CustomerHistory` component into the rest of your application by taking the following steps:

1. Add a new route at `/viewHistory?customer=<customer id>` that displays the `CustomerHistory` component, using a new intermediate `CustomerHistoryRoute` component.
2. Add a new `Link` to the search actions on the `CustomerSearch` screen, titled **View history**, that, when pressed, navigates to the new route.

Further reading

The RelayEnvironmentProvider component:

<https://relay.dev/docs/api-reference/relay-environment-provider/>

Part 3 – Interactivity

This part introduces a new code base that allows us to explore more complex scenarios where TDD can be applied. You'll take a deep dive into Redux middleware, animation, and WebSockets. The goal is to show how complex tasks are approached using the TDD workflow.

This part includes the following chapters:

- *Chapter 14, Building a Logo Interpreter*
- *Chapter 15, Adding Animation*
- *Chapter 16, Working with WebSockets*

14

Building a Logo Interpreter

Logo is a programming environment created in the 1960s. It was, for many decades, a popular way to teach children how to code—I have fond memories of writing Logo programs back in high school. At its core, it is a method for building graphics via imperative instructions.

In this part of the book, we'll build an application called **Spec Logo**. The starting point is an already-functioning interpreter and a barebones UI. In the following chapters, we'll bolt on additional features to this codebase.

This chapter provides a second opportunity to test-drive Redux. It covers the following topics:

- Studying the **Spec Logo** user interface
- Undoing and redoing user actions in Redux
- Saving to local storage via Redux middleware
- Changing keyboard focus

By the end of the chapter, you'll have learned how to test-drive complex Redux reducers and middleware.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter14>

Studying the Spec Logo user interface

The interface has two panes: the left pane is the drawing pane, which is where the output from the Logo script appears. On the right side is a prompt where the user can edit instructions:

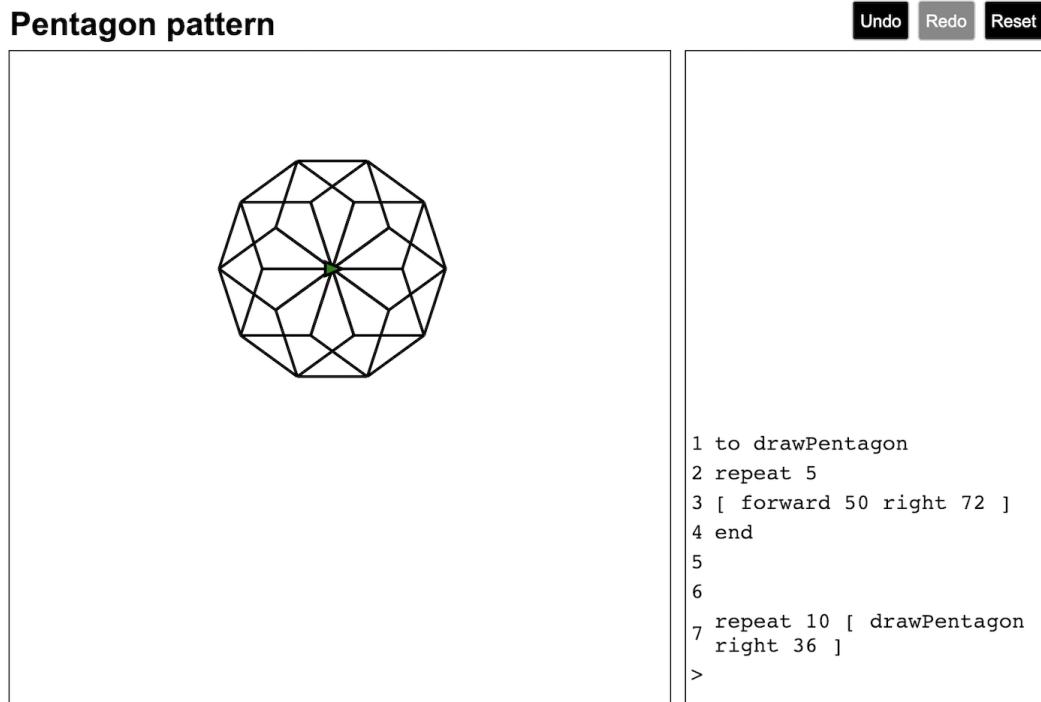


Figure 14.1: The Spec Logo interface

Look at the screenshot. You can see the following:

- The **script name** in the top-left corner. This is a text field that the user can click on to change the name of the current script.
- The **display**, which shows script output on the left-hand side of the page. You can see a shape has been drawn here, which is the result of the Logo statements entered at the prompt.
- The **turtle**, shown in the middle of the screen. This is a little green triangle that marks where drawing commands originate. The turtle has an *x* and *y* position, starting at *0,0*, which is the middle of the screen. The viewable drawing is 600x600 in size, and the turtle can move throughout this area. The turtle also has an angle, initially zero, which is pointing directly right.

- The **prompt** in the bottom right-hand corner, marked with a > symbol. This is where you enter your statements, which can be multiline. Hitting *Enter* will send the current prompt text to the interpreter. If it makes a complete statement, it will be executed, and the prompt cleared ready for your next statement.
- The **statement history** above the prompt. It lists all the previously executed statements. Each is given a number so you can refer back to the statement.
- A **menu bar** in the top-right corner, containing **Undo**, **Redo**, and **Reset** buttons. It is this menu bar that we'll be building out in this chapter.

Although we won't be writing any Logo code in this chapter, it's worth spending some time playing around and making your own drawings with the interpreter. Here's a list of instructions that you can use:

Statement	Alias	Description
forward d	fd	Move the turtle forward a distance of d pixels
backward d	bd	Move the turtle backward a distance of d pixels
right a	rt	Rotate the turtle a degrees clockwise
left a	lt	Rotate the turtle a degrees anti-clockwise
to fn <args> <instructions> end		Defines a function named fn with the arguments (args) and instructions (instructions)
repeat n [<instructions>]		Repeats the provided instructions n times

It's also worth looking through the codebase. The `src/parser.js` file and the `src/language` directory contain the Logo interpreter. There are also corresponding test files in the `test` directory. We won't be modifying these files, but you may be interested in seeing how this functionality has been tested.

There is a single Redux reducer in `src/reducers/script.js`. Its `defaultState` definition neatly encapsulates everything needed to represent the execution of a Logo program. Almost all the app's React components use this state in some way.

In this chapter, we'll be adding two more reducers into this directory: one for undo/redo and one for prompt focus. We'll be making modifications to three React components: `MenuButtons`, `Prompt`, and `ScriptName`.

Let's start by building a new reducer, named `withUndoRedo`.

Undoing and redoing user actions in Redux

In this section, we'll add undo and redo buttons at the top of the page, which allow the user to undo and redo statements that they've previously run. They'll work like this:

1. Initially, both buttons will be disabled.
2. Once the user executes a statement, the **Undo** button will become enabled.
3. When the user clicks the **Undo** button, the last statement will be undone.
4. At that point, the **Redo** button becomes available, and the user can choose to redo the last statement.
5. Multiple actions can be undone and then redone, in sequence.
6. If the user performs a new action while **Redo** is available, the redo sequence is cleared, and the **Redo** button becomes unavailable again.

Aside from adding button elements, the work involved here is building a new reducer, `withUndoRedo`, which will *decorate* the script reducer. This reducer will return the same state as the script reducer, but with two additional properties: `canUndo` and `canRedo`. In addition, the reducer stores `past` and `future` arrays within it that record the past and future states. These will never be returned to the user, just stored, and will replace the current state should the user choose to undo or redo.

Building the reducer

The reducer will be a higher-order function that, when called with an existing reducer, returns a new reducer that returns the state we're expecting. In our production code, we'll replace this store code:

```
combineReducers ({  
  script: scriptReducer  
)
```

We'll replace it with this decorated reducer, which takes exactly the same reducer and wraps it in the `withUndoRedo` reducer that we'll build in this section:

```
combineReducers ({  
  script: withUndoRedo(scriptReducer)  
)
```

To test this, we'll need to use a spy to act in place of the script reducer, which we'll call `decoratedReducerSpy`.

Setting the initial state

Let's make a start by building the reducer itself, before adding buttons to exercise the new functionality:

1. Create a new file named `test/reducers/withUndoRedo.test.js` and add the following setup and test, which specifies what should happen when we pass an undefined state to the reducer. This is equivalent to how we began testing our other reducers, but in this case, we pass the call through to the decorated reducer. The test passes an undefined state to the reducer, which is the required mechanism for initializing a reducer:

```
import {
  withUndoRedo
} from "../../src/reducers/withUndoRedo";

describe("withUndoRedo", () => {
  let decoratedReducerSpy;
  let reducer;

  beforeEach(() => {
    decoratedReducerSpy = jest.fn();
    reducer = withUndoRedo(decoratedReducerSpy);
  });

  describe("when initializing state", () => {
    it("calls the decorated reducer with undefined state and an action", () => {
      const action = { type: "UNKNOWN" };
      reducer(undefined, action);
      expect(decoratedReducerSpy).toBeCalledWith(
        undefined,
        action);
    });
  });
});
```

2. Create a new `src/reducers/withUndoRedo.js` file and make the test pass with the following code:

```
export const withUndoRedo = (reducer) => {
  return (state, action) => {
```

```

        reducer(state, action);
    };
};

```

3. Add the next test to the `describe` block, as shown. This uses the `toMatchObject` matcher, which we first encountered in *Chapter 6, Exploring Test Doubles*:

```

it("returns a value of what the inner reducer returns",
() => {
  decoratedReducerSpy.mockReturnValue({ a: 123 });
  expect(reducer(undefined)).toMatchObject(
    { a : 123 }
  );
});

```

4. Make that pass by adding the `return` keyword:

```

export const withUndoRedo = (reducer) => {
  return (state, action) => {
    return reducer(state, action);
  };
}

```

5. Initially, both `canUndo` and `canRedo` should be `false`, as there are no previous or future states that we can move to. Let's add those two tests as a pair, still in the same `describe` block:

```

it("cannot undo", () => {
  expect(reducer(undefined)).toMatchObject({
    canUndo: false
  });
});

it("cannot redo", () => {
  expect(reducer(undefined)).toMatchObject({
    canRedo: false
  });
});

```

6. To make these tests pass, we need to create a new object with those properties added:

```

export const withUndoRedo = (reducer) => {
  return (state, action) => {

```

```

        return {
          canUndo: false,
          canRedo: false,
          ...reducer(state, action)
        };
      };
    }
  }
}

```

- Let's move on to the meat of the reducer. After performing an action, we want to be able to perform an **Undo** action to revert to the previous state. We'll use the `present` and `future` constants to denote those states:

```

describe("performing an action", () => {
  const innerAction = { type: "INNER" };
  const present = { a: 123 };
  const future = { b: 234 };

  beforeEach(() => {
    decoratedReducerSpy.mockReturnValue(future);
  });

  it("can undo after a new present has been provided", () => {
    const result = reducer(
      { canUndo: false, present },
      innerAction
    );
    expect(result.canUndo).toBeTruthy();
  });
});

```

- Make that pass with the following code. Since we're no longer dealing with an undefined state, this is the moment that we need to wrap our existing code in a conditional block:

```

export const withUndoRedo = (reducer) => {
  return (state, action) => {
    if (state === undefined)
      return {
        canUndo: false,

```

```
        canRedo: false,
        ...reducer(state, action)
    };

    return {
        canUndo: true
    };
};

};
```

9. Next, we make sure we call the reducer again since, for this new block, it won't happen. Write the following test:

```
it("forwards action to the inner reducer", () => {
    reducer(present, innerAction);
    expect(decoratedReducerSpy).toBeCalledWith(
        present,
        innerAction
    );
});
```

10. To make that pass, simply call the reducer before the `return` value:

```
if (state === undefined)
    ...

reducer(state, action);
return {
    canUndo: true
};
```

11. The next test shows that this object also needs to return the new state:

```
it("returns the result of the inner reducer", () => {
    const result = reducer(present, innerAction);
    expect(result).toMatchObject(future);
});
```

-
12. Make that pass by saving the reducer value in a variable named `newPresent` and returning it as part of the returned object:

```
const newPresent = reducer(state, action);
return {
  ...newPresent,
  canUndo: true
};
```

13. The script reducer holds a special value named `nextInstructionId`. We can use this to determine whether the script instruction was processed or whether an error occurred. When a statement is valid, it is executed and `nextInstructionId` is incremented. But when a statement can't be processed, `nextInstructionId` remains the same. We can use that fact to avoid saving history if a statement contains an error. To do that, modify the `present` and `future` constants to include this parameter, and add the new test, as shown next:

```
const present = { a: 123, nextInstructionId: 0 };
const future = { b: 234, nextInstructionId: 1 };

...

it("returns the previous state if nextInstructionId does not increment", () => {
  decoratedReducerSpy.mockReturnValue({
    nextInstructionId: 0
  });
  const result = reducer(present, innerAction);
  expect(result).toBe(present);
});
```

14. Make that pass by wrapping our new `return` block in a conditional, and returning the old state if the condition doesn't pass:

```
const newPresent = reducer(state, action);
if (
  newPresent.nextInstructionId != state.nextInstructionId
) {
  return {
    ...newPresent,
    canUndo: true
}
```

```

    } ;
}
return state;

```

This covers all the functionality for performing any actions *other than Undo and Redo*. The next section covers **Undo**.

Handling the undo action

We'll create a new Redux action, of type `UNDO`, which causes us to push the current state into a new array called `past`:

1. For this test, we can reuse the `present` and `innerAction` properties, so push those up into the outer `describe` block now. Also, define a new `undoAction` Redux action. We'll use it within our first test:

```

describe("withUndoRedo", () => {
  const undoAction = { type: "UNDO" };
  const innerAction = { type: "INNER" };
  const present = { a: 123, nextInstructionId: 0 };
  const future = { b: 234, nextInstructionId: 1 };

  ...
});

```

2. Add a new nested `describe` block with the following test and setup. The `beforeEach` block sets up a scenario where we've already performed an action that will have stored a previous state. We're then ready to undo it within the test:

```

describe("undo", () => {
  let newState;

  beforeEach(() => {
    decoratedReducerSpy.mockReturnValue(future);
    newState = reducer(present, innerAction);
  });

  it("sets present to the latest past entry", () => {
    const updated = reducer(newState, undoAction);
    expect(updated).toMatchObject(present);
  });
}

```

```
    }) ;  
}) ;
```

Performing an action within a `beforeEach` block

Notice the call to the `reducer` function in the `beforeEach` setup. This function is the function under test, so it could be considered part of the **Act** phase that we usually keep within the test itself. However, in this case, the first call to `reducer` is part of the test setup, since all these tests rely on having performed at least one action that can then be undone. In this way, we can consider this `reducer` call to be part of the **Assert** phase.

3. Make that pass by modifying the function as follows. We use a `past` variable to store the previous state. If we receive an UNDO action, we return that value. We also use a `switch` statement since we'll be adding a case for REDO later:

```
export const withUndoRedo = (reducer) => {  
  let past;  
  
  return (state, action) => {  
    if (state === undefined)  
      ...  
  
    switch(action.type) {  
      case "UNDO":  
        return past;  
      default:  
        const newPresent = reducer(state, action);  
        if (  
          newPresent.nextInstructionId !=  
          state.nextInstructionId  
        ) {  
          past = state;  
  
          return {  
            ...newPresent,  
            canUndo: true  
          };  
        }  
    }  
    return state;
```

```
    }
};

};
```

4. Next, let's adjust this so that we can undo any number of levels deep. Add the next test:

```
it("can undo multiple levels", () => {
  const futureFuture = {
    c: 345, nextInstructionId: 3
  };

  decoratedReducerSpy.mockReturnValue(futureFuture);
  newState = reducer(newState, innerAction);

  const updated = reducer(
    reducer(newState, undoAction),
    undoAction
  );

  expect(updated).toMatchObject(present);
});
```

5. For this, we'll need to upgrade `past` to an array:

```
export const withUndoRedo = (reducer) => {
  let past = [];

  return (state, action) => {
    if (state === undefined)
      ...

    switch(action.type) {
      case "UNDO":
        const lastEntry = past[past.length - 1];
        past = past.slice(0, -1);
        return lastEntry;
      default:
```

```
const newPresent = reducer(state, action);

if (
  newPresent.nextInstructionId != state.nextInstructionId
) {
  past = [ ...past, state ];

  return {
    ...newPresent,
    canUndo: true
  };
}

return state;
};

};

};
```

6. There's one final test we need to do. We need to check that after undoing, we can also redo:

```
it("sets canRedo to true after undoing", () => {
  const updated = reducer(newState, undoAction);
  expect(updated.canRedo).toBeTruthy();
});
```

- To make that pass, return a new object comprised of `lastEntry` and the new `canRedo` property:

```
case "UNDO":  
    const lastEntry = past[past.length - 1];  
    past = past.slice(0, -1);  
    return {  
        ...lastEntry,  
        canRedo: true  
    };
```

That's all there is to the UNDO action. Next, let's add the REDO action.

Handling the redo action

Redo is very similar to undo, just reversed:

1. First, add a new definition for the Redux action of type REDO, in the top-level describe block:

```
describe("withUndoRedo", () => {
  const undoAction = { type: "UNDO" };
  const redoAction = { type: "REDO" };

  ...
});
```

2. Underneath the undo describe block, add the following redo describe block with the first test. Be careful with the setup for the spy; the call is mockReturnValueOnce here, not mockReturnValue. The test needs to ensure it takes its value from the stored redo state:

```
describe("redo", () => {
  let newState;

  beforeEach(() => {
    decoratedReducerSpy.mockReturnValueOnce(future);
    newState = reducer(present, innerAction);
    newState = reducer(newState, undoAction);
  });

  it("sets the present to the latest future entry", () =>
{
  const updated = reducer(newState, redoAction);
  expect(updated).toMatchObject(future);
}) ;
});
```

3. To make this pass, in your production code, declare a `future` variable, next to the declaration for `past`:

```
let past = [], future;
```

4. Set this within the UNDO action:

```
case "UNDO":
  const lastEntry = past[past.length - 1];
```

```
past = past.slice(0, -1);
future = state;
```

- Now that it's saved, we can handle the REDO action. Add the following case statement, between the UNDO clause and the default clause:

```
case "UNDO":
  ...
case "REDO":
  return future;
default:
  ...
```

- The next test is for multiple levels of redo. This is slightly more complicated than the same case in the undo block—we'll have to modify the beforeEach block to take us back *twice*. First, pull out the futureFuture value from the undo test and bring it into the outer scope, next to the other values, just below future:

```
const future = { b: 234, nextInstructionId: 1 };
const futureFuture = { c: 345, nextInstructionId: 3 };
```

- Now, update beforeEach to take two steps forward and then two back:

```
beforeEach(() => {
  decoratedReducerSpy.mockReturnValueOnce(future);
  decoratedReducerSpy.mockReturnValueOnce(
    futureFuture
  );
  newState = reducer(present, innerAction);
  newState = reducer(newState, innerAction);
  newState = reducer(newState, undoAction);
  newState = reducer(newState, undoAction);
}) ;
```

- Finally, add the test:

```
it("can redo multiple levels", () => {
  const updated = reducer(
    reducer(newState, redoAction),
    redoAction
) ;
```

```
    expect(updated).toEqual(futureFuture);
});
```

9. To make this pass, start by initializing the `future` variable to be an empty array:

```
let past = [], future = [];
```

10. Update the UNDO clause to push the current value to it:

```
case "UNDO":  
  const lastEntry = past[past.length - 1];  
  past = past.slice(0, -1);  
  future = [...future, state];
```

11. Update the REDO clause to pull out that value we just pushed. After this change, the test should be passing:

```
case "REDO":  
  const nextEntry = future[future.length - 1];  
  future = future.slice(0, -1);  
  return nextEntry;
```

12. There's one final test we need to write for our barebones implementation, which checks that a redo followed by an undo brings us back to the original state:

```
it("returns to previous state when followed by an undo",  
() => {  
  const updated = reducer(  
    reducer(newState, redoAction),  
    undoAction  
  );  
  expect(updated).toEqual(present);  
});
```

13. Make that pass by setting the `past` property in the REDO case:

```
case "REDO":  
  const nextEntry = future[future.length - 1];  
  past = [...past, state];  
  future = future.slice(0, -1);  
  return nextEntry;
```

14. This completes our reducer. However, our implementation has a memory leak! We never clear out the `future` array when we generate new states. If the user repeatedly hit **Undo** and then performed new actions, all their old actions would remain in `future` but become inaccessible, due to `canRedo` being `false` in the latest state.

To test for this scenario, you can simulate the sequence and check that you expect to return `undefined`. This test isn't *great* in that we really shouldn't be sending a REDO action when `canRedo` returns `false`, but that's what our test ends up doing:

```
it("return undefined when attempting a do, undo, do, redo sequence", () => {
  decoratedReducerSpy.mockReturnValue(future);
  let newState = reducer(present, innerAction);
  newState = reducer(newState, undoAction);
  newState = reducer(newState, innerAction);
  newState = reducer(newState, redoAction);
  expect(newState).not.toBeDefined();
});
```

15. To make that pass, simply clear `future` when setting a new state, as shown:

```
if (
  newPresent.nextInstructionId != state.nextInstructionId
) {
  past = [ ...past, state ];
  future = [];
  return {
    ...newPresent,
    canUndo: true
  };
}
```

16. We are now done with the reducer. To finish this off, hook it into our Redux store. Open `src/store.js` and make the following changes:

```
import {
  withUndoRedo
} from "./reducers/withUndoRedo";
```

```

export const configureStore = (
  storeEnhancers = [],
  initialState = {}
) => {
  return createStore(
    combineReducers({
      script: withUndoRedo(scriptReducer)
    }),
    initialState,
    compose(...storeEnhancers)
  );
};

```

Your tests should all be passing and the app should still run.

However, the undo and redo functionality is still not accessible. For that, we need to add some buttons to the menu bar.

Building buttons

The final piece to this puzzle is adding buttons to trigger the new behavior by adding **Undo** and **Redo** buttons to the menu bar:

1. Open `test/MenuButtons.test.js` and add the following `describe` block at the bottom of the file, nested inside the `MenuButtons` `describe` block. It uses a couple of helper functions that have already been defined with the `renderWithStore` file and button:

```

describe("undo button", () => {
  it("renders", () => {
    renderWithStore(<MenuButtons />);
    expect(buttonWithLabel("Undo")).not.toBeNull();
  });
});

```

2. Make that pass by modifying the implementation for `MenuButtons` as shown, in the `src/MenuButtons.js` file:

```

export const MenuButtons = () => {
  ...
  return (
    <>

```

```
<button>Undo</button>
<button
  onClick={() => dispatch(reset())}
  disabled={!canReset}
>
  Reset
</button>
</>
) ;
} ;
```

3. Add the next test, which checks that the button is initially disabled:

```
it("is disabled if there is no history", () => {
  renderWithStore(<MenuButtons />);
  expect(
    buttonWithLabel("Undo").hasAttribute("disabled")
  ).toBeTruthy();
});
```

4. Make that pass by adding a hardcoded `disabled` attribute, as shown:

```
<button disabled={true}>Undo</button>
```

5. Now, we add in the code that will require us to connect with Redux:

```
it("is enabled if an action occurs", () => {
  renderWithStore(<MenuButtons />);
  dispatchToStore({
    type: "SUBMIT_EDIT_LINE",
    text: "forward 10\n"
  });
  expect(
    buttonWithLabel("Undo").hasAttribute("disabled")
  ).toBeFalsy();
});
```

6. Modify `MenuButtons` to pull out `canUndo` from the store. It already uses the `script` state for the `Reset` button behavior, so in this case, we just need to destructure it further:

```
export const MenuButtons = () => {
  const {
    canUndo, nextInstructionId
  } = useSelector(({ script }) => script);
  ...

  const canReset = nextInstructionId !== 0;

  return (
    <>
      <button disabled={!canUndo}>Undo</button>
      <button
        onClick={() => dispatch(reset())}
        disabled={!canReset}
      >
        Reset
      </button>
    </>
  ) ;
}
```

7. The final test for the `Undo` button is to check that it dispatches an `UNDO` action when it is clicked:

```
it("dispatches an action of UNDO when clicked", () => {
  renderWithStore(<MenuButtons />);
  dispatchToStore({
    type: "SUBMIT_EDIT_LINE",
    text: "forward 10\n"
  });
  click(buttonWithLabel("Undo"));
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "UNDO" });
}) ;
```

8. Make that pass by adding the lines highlighted next. We add the new `undo` action helper and then use that to call `dispatch`:

```
const reset = () => ({ type: "RESET" }) ;
const undo = () => ({ type: "UNDO" }) ;

export const MenuButtons = () => {
    ...

    return (
        <>
            <button
                onClick={() => dispatch(undo())}
                disabled={!canUndo}
            >
                Undo
            </button>
            ...
        </>
    ) ;
}
```

9. Repeat from *Step 2* to *Step 8* for the **Redo** button. This time, you'll need to pull out the `canRedo` property from the script state.

That's the last change needed. The undo and redo functionality is now complete.

Next up, we'll move from building a Redux reducer to building Redux middleware.

Saving to local storage via Redux middleware

In this section, we'll update our app to save the current state to *local storage*, a persistent data store managed by the user's web browser. We'll do that by way of Redux middleware.

Each time a statement is executed in the **Spec Logo** environment, the entire set of parsed tokens will be saved via the browser's `LocalStorage` API. When the user next opens the app, the tokens will be read and replayed through the parser.

The parseTokens function

As a reminder, the parser (in `src/parser.js`) has a `parseTokens` function. This is the function we'll call from within our middleware, and in this section, we'll build tests to assert that we've called this function.

We'll write a new piece of Redux middleware for the task. The middleware will pull out two pieces of the script state: `name` and `parsedTokens`.

Before we begin, let's review the browser `LocalStorage` API:

- `window.localStorage.getItem(key)` returns the value of an item in local storage. The value stored is a string, so if it's a serialized object, then we need to call `JSON.parse` to deserialize it. The function returns `null` if no value exists for the given key.
- `window.localStorage.setItem(key, value)` sets the value of an item. The value is serialized as a string, so we need to make sure to call `JSON.stringify` on any objects before we pass them in here.

Building middleware

Let's test-drive our middleware:

1. Create the `src/middleware` and `test/middleware` directories, and then open the `test/middleware/localStorage.test.js` file. To make a start, define two spies, `getItemSpy` and `setItemSpy`, which will make up the new object. We have to use `Object.defineProperty` to set these spies because the `window.localStorage` property is write protected:

```
import {
  save
} from "../../src/middleware/localStorage";

describe("localStorage", () => {
  const data = { a: 123 };
  let getItemSpy = jest.fn();
  let setItemSpy = jest.fn();

  beforeEach(() => {
    Object.defineProperty(window, "localStorage", {
      value: {
        getItem: getItemSpy,
        setItem: setItemSpy
      }
    });
  });

  it("should save data to localStorage", () => {
    save(data);
    expect(getItemSpy).toHaveBeenCalledWith("data");
    expect(localStorage.getItem("data")).toEqual(data);
  });
});
```

```
        setItem: setItemSpy
    }});
});
});
```

2. Let's write our first test for the middleware. This test simply asserts that the middleware does what all middleware should, which is to call `next (action)`. Redux middleware functions have complicated semantics, being functions that return functions that return functions, but our tests will make short work of that:

```
describe("save middleware", () => {
  const name = "script name";
  const parsedTokens = ["forward 10"];
  const state = { script: { name, parsedTokens } };
  const action = { type: "ANYTHING" };
  const store = { getState: () => state };
  let next;

  beforeEach(() => {
    next = jest.fn();
  });

  const callMiddleware = () =>
    save(store)(next)(action);

  it("calls next with the action", () => {
    callMiddleware();
    expect(next).toBeCalledWith(action);
  });
});
```

3. To make that pass, create the `src/middleware/localStorage.js` file and add the following definition:

```
export const save = store => next => action => {
  next(action);
};
```

4. The next test checks that we return that value:

```
it("returns the result of next action", () => {
    next.mockReturnValue({ a : 123 });
    expect(callMiddleware()).toEqual({ a: 123 });
});
```

5. Update the save function to return that value:

```
export const save = store => next => action => {
    return next(action);
};
```

6. Now, check that we add the stringified value to local storage:

```
it("saves the current state of the store in
localStorage", () => {
    callMiddleware();
    expect(setItemSpy).toBeCalledWith("name", name);
    expect(setItemSpy).toBeCalledWith(
        "parsedTokens",
        JSON.stringify(parsedTokens)
    );
});
```

7. To make that pass, complete the implementation of the save middleware:

```
export const save = store => next => action => {
    const result = next(action);
    const {
        script: { name, parsedTokens }
    } = store.getState();
    localStorage.setItem("name", name);
    localStorage.setItem(
        "parsedTokens",
        JSON.stringify(parsedTokens)
    );
    return result;
};
```

8. Let's move on to the `load` function, which isn't middleware but there's no harm in placing it in the same file. Create a new `describe` block with the following test, ensuring `import` is updated as well:

```
import {
  load, save
} from "../../src/middleware/localStorage";

...

describe("load", () => {
  describe("with saved data", () => {
    beforeEach(() => {
      getItemSpy.mockReturnValueOnce("script name");
      getItemSpy.mockReturnValueOnce(
        JSON.stringify([ { a: 123 } ])
      );
    });
  });

  it("retrieves state from localStorage", () => {
    load();
    expect(getItemSpy).toBeCalledWith("name");
    expect(getItemSpy).toHave BeenLastCalledWith(
      "parsedTokens"
    );
  });
});
});
```

9. Make that pass by defining a new function in the production code, by adding `load`, just below the definition of `save`:

```
export const load = () => {
  localStorage.getItem("name");
  localStorage.getItem("parsedTokens");
};
```

10. Now to send this data to the parser. For this, we'll need a `parserSpy` spy function that we use to spy on the parser's `parseTokens` function:

```
describe("load", () => {
  let parserSpy;

  describe("with saved data", () => {
    beforeEach(() => {
      parserSpy = jest.fn();
      parser.parseTokens = parserSpy;
      ...
    });
  });

  it("calls to parsedTokens to retrieve data", () => {
    load();
    expect(parserSpy).toBeCalledWith(
      [ { a: 123 } ],
      parser.emptyState
    );
  });
});
});
```

11. Add the following production code to make that pass:

```
import * as parser from "../parser";

export const load = () => {
  localStorage.getItem("name");
  const parsedTokens = JSON.parse(
    localStorage.getItem("parsedTokens")
  );
  parser.parseTokens(parsedTokens, parser.emptyState);
};
```

12. The next test makes sure the data is returned in the right format:

```
it("returns re-parsed draw commands", () => {
  parserSpy.mockReturnValue({ drawCommands: [] });
});
```

```
expect(
  load().script
).toHaveProperty("drawCommands", []);
});
```

13. Make that pass by returning an object with the parsed response:

```
export const load = () => {
  localStorage.getItem("name");
  const parsedTokens = JSON.parse(
    localStorage.getItem("parsedTokens")
  );
  return {
    script: parser.parseTokens(
      parsedTokens, parser.emptyState
    )
  };
};
```

14. Next, let's add the name to that data structure:

```
it("returns name", () => {
  expect(load().script).toHaveProperty(
    "name",
    "script name"
  );
});
```

15. To make that pass, first, we need to save the name that's returned from local storage, and then we need to insert it into the present object:

```
export const load = () => {
  const name = localStorage.getItem("name");
  const parsedTokens = JSON.parse(
    localStorage.getItem("parsedTokens")
  );
  return {
    script: {
      ...parser.parseTokens(
```

```

        parsedTokens, parser.initialState
    ) ,
    name
}
};

}
;

```

16. Finally, we need to deal with the case where no state has been saved yet. The `LocalStorage` API gives us `null` back in that case, but we'd like to return `undefined`, which will trigger Redux to use the default state. Add this test to the outer `describe` block, so that it won't pick up the extra `getItemSpy` mock values:

```

it("returns undefined if there is no state saved", () =>
{
  getItemSpy.mockReturnValue(null);
  expect(load()).not.toBeDefined();
});

```

17. Make that pass by wrapping the `return` statement in an `if` statement:

```

if (parsedTokens && parsedTokens !== null) {
  return {
    ...
  };
}

```

18. Open `src/store.js` and modify it to include the new middleware. I'm defining a new function, `configureStoreWithLocalStorage`, so that our tests can continue using `configureStore` without interacting with local storage:

```

...
import {
  save, load
} from "./middleware/localStorage";

export const configureStore = (
  storeEnhancers = [],
  initialState = {}
) => {
  return createStore(

```

```
combineReducers({
  script: withUndoRedo(scriptReducer)
}),
initialState,
compose(
  ...
  [
    applyMiddleware(save),
    ...storeEnhancers
  ]
)
);
};

export const configureStoreWithLocalStorage = () =>
  configureStore(undefined, load());
```

19. Open `src/index.js` and replace the call to `configureStore` with a call to `configureStoreWithLocalStorage`. You'll also need to update `import` for this new function:

```
import {
  configureStoreWithLocalStorage
} from "./store";

ReactDOM.createRoot(
  document.getElementById("root")
).render(
  <Provider store={configureStoreWithLocalStorage()}>
    <App />
  </Provider>
);
```

That's it. If you like, this is a great time to run the app for a manual test and try it. Open the browser window, type a few commands, and try it out!

If you're stuck for commands to run a manual test, you can use these:

```
forward 100
right 90
```

```
to drawSquare
  repeat 4 [ forward 100 right 90 ]
end
drawSquare
```

These commands exercise most of the functionality within the interpreter and display. They'll come in handy in *Chapter 15, Adding Animation*, when you'll want to be manually testing as you make changes.

You've learned how to test-drive Redux middleware. For the final part of the chapter, we will write another reducer, this time one that helps us manipulate the browser's keyboard focus.

Changing keyboard focus

The user of our application will, most of the time, be typing in the prompt at the bottom right of the screen. To help them out, we'll move the keyboard focus to the prompt when the app is launched. We should also do this when another element—such as the name text field or the menu buttons—has been used but has finished its job. Then, the focus should revert back to the prompt, ready for another instruction.

React doesn't support setting focus, so we need to use a **React ref** on our components and then drop it into the DOM API.

We'll do this via a Redux reducer. It will have two actions: `PROMPT_FOCUS_REQUEST` and `PROMPT_HAS_FOCUSED`. Any of the React components in our application will be able to dispatch the first action. The `Prompt` component will listen for it and then dispatch the second, once it has focused.

Writing the reducer

We'll start, as ever, with the reducer:

1. Create a new file named `test/reducers/environment.test.js` and add the following `describe` block. This covers the basic case of the reducer needing to return the default state when `undefined` is passed to it:

```
import {
  environmentReducer as reducer
} from "../../src/reducers/environment";

describe("environmentReducer", () => {
  it("returns default state when existing state is
  undefined", () => {
```

```
expect(reducer(undefined, {})).toEqual({
  promptFocusRequest: false
});
});
});
});
```

2. Make the test pass with the following code, in a file named `src/reducers/environment.js`. Since we've built reducers before, we know where we're going with this one:

```
const defaultState = {
  promptFocusRequest: false
};

export const environmentReducer = (
  state = defaultState,
  action) => {
  return state;
};
```

3. Add the next test, which checks that we set the `promptFocusRequest` value:

```
it("sets promptFocusRequest to true when receiving a PROMPT_FOCUS_REQUEST action", () => {
  expect(
    reducer(
      { promptFocusRequest: false},
      { type: "PROMPT_FOCUS_REQUEST" }
    )
  ).toEqual({
    promptFocusRequest: true
  });
});
```

4. Make that pass by adding in a `switch` statement, as shown:

```
export const environmentReducer = (
  state = defaultState,
  action
) => {
  switch (action.type) {
```

```

        case "PROMPT_FOCUS_REQUEST":
            return { promptFocusRequest: true };
    }
    return state;
};

```

5. Add the final test for this reducer:

```

it("sets promptFocusRequest to false when receiving a
PROMPT_HAS_FOCUSED action", () => {
    expect(
        reducer(
            { promptFocusRequest: true },
            { type: "PROMPT_HAS_FOCUSED" }
        )
    ).toEqual({
        promptFocusRequest: false
    });
});

```

6. Finally, make that pass by adding another `case` statement:

```

export const environmentReducer = (... ) => {
    switch (action.type) {
        ...
        case "PROMPT_HAS_FOCUSED":
            return { promptFocusRequest: false };
    }
    ...
}

```

7. Before we can use the new reducer in our tests, we'll need to add it to the store. Open up `src/store.js` and modify it as follows:

```

...
import {
    environmentReducer
} from "./reducers/environment";

```

```
export const configureStore = (
  storeEnhancers = [] ,
  initialState = {}
) => {
  return createStore(
    combineReducers({
      script: withUndoRedo(logoReducer),
      environment: environmentReducer
    }) ,
    ...
  );
};
```

That gives us a new reducer that's hooked into the Redux store. Now, let's make use of that.

Focusing the prompt

Let's move on to the most difficult part of this: focusing the actual prompt. For this, we'll need to introduce a React ref:

1. Open `test/Prompt.test.js` and add the following `describe` block at the bottom, nested within the `Prompt` `describe` block. The test uses the `document.activeElement` property, which is the element that currently has focus. It's also using the `renderInTableWithStore` function, which is the same as the `renderWithStore` helper you've seen already, except that the component is first wrapped in a table:

```
describe("prompt focus", () => {
  it("sets focus when component first renders", () => {
    renderInTableWithStore(<Prompt />);
    expect(
      document.activeElement
    ).toEqual(textArea());
  });
});
```

2. Let's make that pass. We define a new ref using the `useRef` hook and add a `useEffect` hook to focus when the component mounts. Make sure to pull out the new constants from the React constant, which is at the top of the file:

```
import
  React, { useEffect, useRef, useState }
```

```

from "react";

export const Prompt = () => {
  ...
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, [inputRef]);

  return (
    ...
    <textarea
      ref={inputRef}
    />
    ...
  );
};

```

3. For the next test, we'll dispatch an action to the Redux store. Since this test suite hasn't yet got a test that dispatches actions, we'll need to add all the plumbing. Start by importing the `dispatchToStore` function into the test suite:

```

import {
  ...
  dispatchToStore,
} from "./reactTestExtensions";

```

4. Now, we need a new helper function that will clear focus. Because focus will be set as soon as the component mounts, we need to unset it again so we can verify the behavior of our focus request. Once we have that helper, we can add the next test:

```

const jsdomClearFocus = () => {
  const node = document.createElement("input");
  document.body.appendChild(node);
  node.focus();
  node.remove();
}

```

```
it("calls focus on the underlying DOM element if promptFocusRequest is true", async () => {
  renderInTableWithStore(<Prompt />);
  jsdomClearFocus();
  dispatchToStore({ type: "PROMPT_FOCUS_REQUEST" });
  expect(document.activeElement).toEqual(textArea());
});
```

5. To make that pass, first, create a new call to `useSelector` to pull out the `promptFocusRequest` value from the store:

```
export const Prompt = () => {
  const nextInstructionId = ...
  const promptFocusRequest = useSelector(
    ({ environment: { promptFocusRequest } }) =>
    promptFocusRequest
  );
  ...
};
```

6. Then, add a new effect that will run when `promptFocusRequest` changes. This uses the ref to call the DOM's `focus` method on the HTML element:

```
useEffect(() => {
  inputRef.current.focus();
}, [promptFocusRequest]);
```

7. For the next test, dispatch an action when the focus has occurred:

```
it("dispatches an action notifying that the prompt has focused", () => {
  renderWithStore(<Prompt />);
  dispatchToStore({ type: "PROMPT_FOCUS_REQUEST" });
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "PROMPT_HAS_FOCUSED" });
});
```

8. To make that pass, start by adding a new action helper function that we can call within the `Prompt` component:

```
const submitEditLine = ...  
const promptHasFocused = () => (  
  { type: "PROMPT_HAS_FOCUSED" }  
) ;
```

9. Finally, call `promptHasFocused` within the `useEffect` hook:

```
useEffect(() => {  
  inputRef.current.focus();  
  dispatch(promptHasFocused());  
}, [promptFocusRequest]);
```

There is a slight issue with this last code snippet. The dispatched `PROMPT_HAS_FOCUSED` action will set `promptFocusRequest` back to `false`. That then causes the `useEffect` hook to run a second time, with the component re-rendering. This is clearly not intended, nor is it necessary. However, since it has no discernable effect on the user, we can skip fixing it at this time.

This completes the `Prompt` component, which now steals focus anytime the `promptFocusRequest` variable changes value.

Requesting focus in other components

All that's left is to call the request action when required. We'll do this for `ScriptName`, but you could also do it for the buttons in the menu bar:

1. Open `test/ScriptName.test.js`, find the `describe` block named `when the user hits Enter:`, and add the following test:

```
it("dispatches a prompt focus request", () => {  
  return expectRedux(store)  
    .toDispatchAnAction()  
    .matching({ type: "PROMPT_FOCUS_REQUEST" });  
});
```

2. In `src/ScriptName.js`, modify the component to define an action helper named `promptFocusRequest`:

```
const submitScriptName = ...  
  
const promptFocusRequest = () => ({
```

```
    type: "PROMPT_FOCUS_REQUEST",
});
```

3. Call that from within the edit completion handler:

```
const completeEditingScriptName = () => {
  if (editingScriptName) {
    toggleEditingScriptName();
    dispatch(submitScriptName(updatedScriptName));
    dispatch(promptFocusRequest());
  }
};
```

That's it! If you build and run now, you'll see how focus is automatically given to the `prompt` textbox, and if you edit the script name (by clicking on it, typing something, and then hitting *Enter*), you'll see that focus returns to the prompt.

Summary

You should now have a good understanding of test-driving complex Redux reducers and middleware.

First, we added support undo/redo with a Redux decorator reducer. Then, we built Redux middleware to save and load existing states via the browser's `LocalStorage` API. And finally, we looked at how to test-drive changing the browser's focus.

In the next chapter, we'll look at how to test-drive something much more intricate: animation.

Further reading

Wikipedia entry on the Logo programming language:

[https://en.wikipedia.org/wiki/Logo_\(programming_language\)](https://en.wikipedia.org/wiki/Logo_(programming_language))

15

Adding Animation

Animation lends itself to test-driven development just as much as any other feature. In this chapter, we'll animate the Logo turtle movement as the user inputs commands.

There are two types of animation in Spec Logo:

- First, when the turtle moves forward. For example, when the user enters `forward 100` as an instruction, the turtle should move 100 units along, at a fixed speed. As it moves, it will draw a line behind it.
- Second, when the turtle rotates. For example, if the user types `rotate 90`, then the turtle should rotate slowly until it has made a quarter turn.

Much of this chapter is about test-driving the `window.requestAnimationFrame` function. This is the browser API that allows us to animate visual elements on the screen, such as the position of the turtle or the length of a line. The mechanics of this function are explained in the third section of this chapter, *Animating with requestAnimationFrame*.

The importance of manual testing

When writing animation code, it's natural to want to visually check what we're building. Automated tests aren't enough. Manually testing is also important because animation is not something that most programmers do every day. When something is new, it's often better to do lots of manual tests to verify behavior in addition to your automated tests.

In fact, while preparing for this chapter, I did a *lot* of manual testing. The walk-through presented here experiments with several different approaches. There were many, many times that I opened my browser to type `forward 100` or `right 90` to visually verify what was happening.

This chapter covers the following topics:

- Designing animation
- Building an animated line component

- Animating with `requestAnimationFrame`
- Canceling animations with `cancelAnimationFrame`
- Varying animation behavior

The code we'll write is relatively complicated compared to the code in the rest of the book, so we need to do some upfront design first.

By the end of the chapter, you'll have gained a deep understanding of how to test-drive one of the more complicated browser APIs.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter15>

Designing animation

As you read through this section, you may wish to open `src/Drawing.js` and read the existing code to understand what it's doing.

The current `Drawing` component shows a static snapshot of how the drawing looks at this point. It renders a set of **Scalable Vector Graphics (SVG)** lines to represent the path the turtle has taken to this point, and a triangle to represent the turtle.

The component makes use of two child components:

- The `Turtle` component is displayed once and draws an SVG triangle at the given location
- The `StaticLines` component is a set of SVG lines that are drawn onscreen to represent the drawn commands

We will add a new `AnimatedLine` component that represents the current line being animated. As lines complete their animation, they will move into the `StaticLines` collection.

We'll need to do some work to convert this from a static view to an animated representation.

As it stands, the component takes a `turtle` prop and a `drawCommands` prop. The `turtle` prop is the current position of the turtle, given that all the draw commands have already been drawn.

In our new animated drawing, we will still treat `drawCommands` as a list of commands to execute. But rather than relying on a `turtle` prop to tell us where the turtle is, we'll store the *current* position of the turtle as a component state. We will work our way through the `drawCommands` array, one instruction at a time, and update the turtle component state as it animates. Once all instructions

are completed, the turtle component state will match what would have originally been set for the `turtle` prop.

The turtle always starts at the `0, 0` coordinate with an angle of `0`.

We will need to keep track of which commands have already been animated. We'll create another component state variable, `animatingCommandIndex`, to denote the index of the array item that is currently being animated.

We start animating at the `0` index. Once that command has been animated, we increment the index by `1`, moving along to the next command, and animate that. The process is repeated until we reach the end of the array.

This design means that the user can enter new `drawCommands` at the prompt even if animations are currently running. The component will take care to redraw with animations at the same point it left off at.

Finally, are two types of draw commands: `drawLine` and `rotate`. Here are a couple of examples of commands that will appear in the `drawCommands` array:

```
{
  drawCommand: "drawLine",
  id: 123,
  x1: 100,
  y1: 100,
  x2: 200,
  y2: 100
}
{
  drawCommand: "rotate",
  id: 234,
  previousAngle: 0,
  newAngle: 90
}
```

Each type of animation will need to be handled differently. So, for example, the `AnimatedLine` component will be hidden when the turtle is rotating.

That about covers it. We'll follow this approach:

- Start with building the `AnimatedLine` component
- Create a `useEffect` hook in `Drawing` that calls the `window.requestAnimationFrame` function to animate `drawLine` commands

- Cancel the animation of `drawLine` commands when new instructions are added
- Add the animation of turtle rotations

Let's get started with the `AnimatedLine` component.

Building an animated line component

In this section, we'll create a new `AnimatedLine` component.

This component contains no animation logic itself but, instead, draws a line from the start of the line being animated to the current turtle position. Therefore, it needs two props: `commandToAnimate`, which would be one of the `drawLine` command structures shown previously, and the `turtle` prop, containing the position.

Let's begin:

1. Create a new file, `test/AnimatedLine.test.js`, and prime it with the following imports and describe block setup. Notice the inclusion of the sample instruction definition for `horizontalLine`:

```
import React from "react";
import ReactDOM from "react-dom";
import {
  initializeReactContainer,
  render,
  element,
} from "./reactTestExtensions";
import { AnimatedLine } from "../src/AnimatedLine";
import { horizontalLine } from "./sampleInstructions";

const turtle = { x: 10, y: 10, angle: 10 };

describe("AnimatedLine", () => {
  beforeEach(() => {
    initializeReactContainer();
  });

  const renderSvg = (component) =>
    render(<svg>{component}</svg>);
```

```
const line = () => element("line");
});
```

2. Now add the first test, which checks the starting position of the line:

```
it("draws a line starting at the x1,y1 co-ordinate of the
command being drawn", () => {
  renderSvg(
    <AnimatedLine
      commandToAnimate={horizontalLine}
      turtle={turtle}
    />
  );
  expect(line()).not.toBeNull();
  expect(line().getAttribute("x1")).toEqual(
    horizontalLine.x1.toString()
  );
  expect(line().getAttribute("y1")).toEqual(
    horizontalLine.y1.toString()
  );
});
```

3. Create a new file, `src/AnimatedLine.js`, and make your test pass by using the following implementation:

```
import React from "react";

export const AnimatedLine = ({
  commandToAnimate: { x1, y1 }
}) => (
  <line x1={x1} y1={y1} />
);
```

4. On to the next test. In this one, we explicitly set the turtle values so that it's clear to see where the expected values come from:

```
it("draws a line ending at the current position of the
turtle", () => {
  renderSvg(
    <AnimatedLine
```

```

        commandToAnimate={horizontalLine}
        turtle={{ x: 10, y: 20 }}
      />
    );
expect(line().getAttribute("x2")).toEqual("10");
expect(line().getAttribute("y2")).toEqual("20");
}) ;

```

5. To make that pass, we just need to set the `x2` and `y2` props on the `line` element, pulling that in from the `turtle`:

```

export const AnimatedLine = ({
  commandToAnimate: { x1, y1 },
  turtle: { x, y }
}) => (
  <line x1={x1} y1={y1} x2={x} y2={y} />
) ;

```

6. Then we need two tests to set the `strokeWidth` and `stroke` props:

```

it("sets a stroke width of 2", () => {
  renderSvg(
    <AnimatedLine
      commandToAnimate={horizontalLine}
      turtle={turtle}
    />
  );
  expect(
    line().getAttribute("stroke-width")
  ).toEqual("2");
}) ;

it("sets a stroke color of black", () => {
  renderSvg(
    <AnimatedLine
      commandToAnimate={horizontalLine}
      turtle={turtle}
    />
  );
  expect(
    line().getAttribute("stroke")
  ).toEqual("black");
}) ;

```

```
) ;  
expect(  
  line().getAttribute("stroke")  
).toEqual("black");  
) ;
```

7. Finish off the component by adding in those two props:

```
export const AnimatedLine = ({  
  commandToAnimate: { x1, y1 },  
  turtle: { x, y }  
) => (  
  <line  
    x1={x1}  
    y1={y1}  
    x2={x}  
    y2={y}  
    strokeWidth="2"  
    stroke="black"  
  />  
) ;
```

That completes the `AnimatedLine` component.

Next, it's time to add it into `Drawing`, by setting the `commandToAnimate` prop to the current line that's animating and using `requestAnimationFrame` to vary the position of the `turtle` prop.

Animating with requestAnimationFrame

In this section, you will use the `useEffect` hook in combination with `window.requestAnimationFrame` to adjust the positioning of `AnimatedLine` and `Turtle`.

The `window.requestAnimationFrame` function is used to animate visual properties. For example, you can use it to increase the length of a line from 0 units to 200 units over a given time period, such as 2 seconds.

To make this work, you provide it with a callback that will be run at the next repaint interval. This callback is provided with the current animation time when it's called:

```
const myCallback = time => {  
  // animating code here
```

```

};

window.requestAnimationFrame(myCallback);

```

If you know the start time of your animation, you can work out the elapsed animation time and use that to calculate the current value of your animated property.

The browser can invoke your callback at a very high refresh rate, such as 60 times per second. Because of these very small intervals of time, your changes appear as a smooth animation.

Note that the browser only invokes your callback once for every requested frame. That means it's your responsibility to repeatedly call the `requestAnimationFrame` function until the animation time reaches your defined end time, as in the following example. The browser takes care of only invoking your callback when the screen is due to be repainted:

```

let startTime;
let endTimeMs = 2000;
const myCallback = time => {
  if (startTime === undefined) startTime = time;
  const elapsed = time - startTime;

  // ... modify visual state here ...

  if (elapsed < endTimeMs) {
    window.requestAnimationFrame(myCallback);
  }
};

// kick off the first animation frame
window.requestAnimationFrame(myCallback);

```

As we progress through this section, you'll see how you can use this to modify the component state (such as the position of `AnimatedLine`), which then causes your component to rerender.

Let's begin by getting rid of the existing turtle value from the Redux store—we're no longer going to use this, and instead, rely on the calculated turtle position from the `drawCommands` array:

1. Open `test/Drawing.test.js` and find the test with the name `passes the turtle x, y and angle as props to Turtle`. Replace it with the following:

```

it("initially places the turtle at 0,0 with angle 0", () => {
  renderWithStore(<Drawing />);

```

```
expect(Turtle).toBeRenderedWithProps({
  x: 0,
  y: 0,
  angle: 0
}) ;
```

2. Now, in `src/Drawing.js`, you can remove the `turtle` value that was extracted from the Redux store, by replacing the `useSelector` call with this one:

```
const { drawCommands } = useSelector(
  ({ script }) => script
) ;
```

3. We'll replace the existing `turtle` value with a new state variable. This will come in useful later when we start moving the position of the turtle. Start by importing `useState` into `src/Drawing.js`:

```
import React, { useState } from "react";
```

4. Then, just below the call to `useSelector`, add another call to `useState`. After this change, your test should be passing:

```
const [turtle, setTurtle] = useState({
  x: 0,
  y: 0,
  angle: 0
}) ;
```

5. Back in `test/Drawing.test.js`, stub out the `requestAnimationFrame` function in the `describe` block's `beforeEach`:

```
beforeEach(() => {
  ...
  jest
    .spyOn(window, "requestAnimationFrame");
}) ;
```

6. Add the following new `describe` block and test to the bottom of the existing `describe` block, inside the existing `describe` block (so it's nested). It defines an initial state of `horizontalLineDrawn` that has a single line—this line is defined in the `sampleInstructions` file. The test states that we expect `requestAnimationFrame` to be invoked when the component mounts:

```
describe("movement animation", () => {
  const horizontalLineDrawn = {
    script: {
      drawCommands: [horizontalLine],
      turtle: { x: 0, y: 0, angle: 0 },
    },
  };

  it("invokes requestAnimationFrame when the timeout fires", () => {
    renderWithStore(<Drawing />, horizontalLineDrawn);
    expect(window.requestAnimationFrame).toBeCalled();
  });
});
```

7. To make this pass, open `src/Drawing.js` and start by importing the `useEffect` hook:

```
import React, { useState, useEffect } from "react";
```

8. Then, add the new `useEffect` hook into the `Drawing` component. Add the following three lines, just above the `return` statement JSX:

```
export const Drawing = () => {
  ...

  useEffect(() => {
    requestAnimationFrame();
  }, []);

  return ...
};
```

9. Since we're now in the realms of `useEffect`, any actions that cause updates to the component state must occur within an `act` block. That includes any triggered animation frames, and we're about to trigger some. So, back in `test/Drawing.test.js`, add the `act` import now:

```
import { act } from "react-dom/test-utils";
```

10. We also need an import for `AnimatedLine` because, in the next test, we'll assert that we render it. Add the following import, together with its spy setup, as shown:

```
import { AnimatedLine } from "../src/AnimatedLine";
jest.mock("../src/AnimatedLine", () => ({
  AnimatedLine: jest.fn(
    () => <div id="AnimatedLine" />
  ),
}));
```

11. The call to `requestAnimationFrame` requires a `handler` function as an argument. The browser will then call this function during the next animation frame. For the next test, we'll check that the turtle is at the start of the first line when the timer first fires. We need to define a new helper to do this, which is `triggerRequestAnimationFrame`. In a browser environment, this call would happen automatically, but in our test, we play the role of the browser and trigger it in code. It's this call that must be wrapped in an `act` function call since our handler will cause the component state to change:

```
const triggerRequestAnimationFrame = time => {
  act(() => {
    const mock = window.requestAnimationFrame.mock
    const lastCallFirstArg =
      mock.calls[mock.calls.length - 1][0]
    lastCallFirstArg(time);
  });
};
```

12. Now, we're ready to write tests for the animation cycle. The first one is a simple one: at time zero, the turtle position is placed at the *start* of the line. If you check the definition in `test/sampleInstructions.js`, you'll see that `horizontalLine` starts at position 100, 100:

```
it("renders an AnimatedLine with turtle at the start position when the animation has run for 0s", () => {
  renderWithStore(<Drawing />, horizontalLineDrawn);
  triggerRequestAnimationFrame(0);
  expect(AnimatedLine).toBeRenderedWithProps({
    commandToAnimate: horizontalLine,
    turtle: { x: 100, y: 100, angle: 0 }
  });
});
```

Using the turtle position for animation

Remember that the `AnimatedLine` component draws a line from the start position of the `drawLine` instruction to the current turtle position. That turtle position is then animated, which has the effect of the `AnimatedLine` instance growing in length until it finds the end position of the `drawLine` instruction.

13. Making this test pass will be a bit of a *big bang*. To start, extend `useEffect` as shown.

We define two variables, `commandToAnimate` and `isDrawingLine`, which we use to determine whether we should animate at all. The `isDrawingLine` test is necessary because some of the existing tests send no draw commands at all to the component, in which case `commandToAnimate` will be `null`. Yet another test passes a command of an unknown type into the component, which would also blow up if we tried to pull out `x1` and `y1` from it. That explains the call to `isDrawLineCommand`—a function that is defined already at the top of the file:

```
const commandToAnimate = drawCommands[0];
const isDrawingLine =
  commandToAnimate &&
  isDrawLineCommand(commandToAnimate);

useEffect(() => {
  const handleDrawLineFrame = time => {
    setTurtle(turtle => ({
      ...turtle,
      x: commandToAnimate.x1,
      y: commandToAnimate.y1,
    }));
  };
  if (isDrawingLine) {
    requestAnimationFrame(handleDrawLineFrame);
  }
}, [commandToAnimate, isDrawingLine]);
```

Using the functional update setter

This code uses the *functional update* variant of `setTurtle` that takes a function rather than a value. This is used when the new state value depends on the old value. Using this form of setter means that the turtle doesn't need to be in the dependency list of `useEffect` and won't cause the `useEffect` hook to reset itself.

14. At this point, we still aren't rendering `AnimatedLine`, which is what our test expects. Let's fix that now. Start by adding the import:

```
import { AnimatedLine } from "./AnimatedLine";
```

15. Insert this just below the JSX for `StaticLines`. At this point, your test should be passing:

```
<AnimatedLine
  commandToAnimate={commandToAnimate}
  turtle={turtle}
/>
```

16. We need a further test to check that we don't render `AnimatedLine` if no lines are being animated. Add the next test as shown, but don't add it in the movement animation block; instead, place it into the parent context:

```
it("does not render AnimatedLine when not moving", () =>
{
  renderWithStore(<Drawing />, {
    script: { drawCommands: [] }
  });
  expect(AnimatedLine).not.toBeRendered();
});
```

17. Make that pass by wrapping the `AnimatedLine` component with a ternary. We simply return `null` if `isDrawingLine` is false:

```
{isDrawingLine ? (
  <AnimatedLine
    commandToAnimate={commandToAnimate}
    turtle={turtle}
  /> : null}
```

18. We've handled what the *first* animation frame should do; now let's code up the *next* animation frame. In the following test, there are *two* calls to `triggerRequestAnimationFrame`. The first one is used to signify that animation is started; the second one allows us to move. We need the first call (with a time index of 0) to be able to mark the time at which the animation started:

```
it("renders an AnimatedLine with turtle at a position
based on a speed of 5px per ms", () => {
  renderWithStore(<Drawing />, horizontalLineDrawn);
```

```

    triggerRequestAnimationFrame(0);
    triggerRequestAnimationFrame(250);
    expect(AnimatedLine).toBeRenderedWithProps({
      commandToAnimate: horizontalLine,
      turtle: { x: 150, y: 100, angle: 0 }
    });
  });
}

```

Using animation duration to calculate the distance moved

The `handleDrawLineFrame` function, when called by the browser, will be passed a time parameter. This is the current duration of the animation. The turtle travels at a constant velocity, so knowing the duration allows us to calculate where the turtle is.

19. To make this pass, first, we need to define a couple of functions. Scroll up `src/Drawing.js` until you see the definition for `isDrawLineCommand` and add these two new definitions there. The `distance` and `movementSpeed` functions are used to calculate the duration of the animation:

```

const distance = ({ x1, y1, x2, y2 }) =>
  Math.sqrt(
    (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)
  );
const movementSpeed = 5;

```

20. Now we can calculate the duration of the animation; modify `useEffect` as shown:

```

useEffect(() => {
  let duration;
  const handleDrawLineFrame = time => {
    setTurtle(...);
  };
  if (isDrawingLine) {
    duration =
      movementSpeed * distance(commandToAnimate);
    requestAnimationFrame(handleDrawLineFrame);
  }
}, [commandToAnimate, isDrawingLine]);

```

21. By declaring duration as the very first line in the `useEffect` block, the variable is in scope for the `requestAnimationFrame` handler to read it to calculate distance. To do that, we take the elapsed time and divide it by the total duration:

```
useEffect(() => {
  let duration;
  const handleDrawLineFrame = time => {
    const { x1, x2, y1, y2 } = commandToAnimate;
    setTurtle(turtle => ({
      ...turtle,
      x: x1 + ((x2 - x1) * (time / duration)),
      y: y1 + ((y2 - y1) * (time / duration)),
    }));
  };
  if (isDrawingLine) {
    ...
  }
}, [commandToAnimate, isDrawingLine]);
```

22. We're making great progress! In the previous test, we assumed that the starting time is 0, but actually, the browser could give us any time as the start time (the time it gives us is known as the **time origin**). So, let's make sure our calculations work for a non-zero start time. Add the following test:

```
it("calculates move distance with a non-zero animation
start time", () => {
  const startTime = 12345;
  renderWithStore(<Drawing />, horizontalLineDrawn);
  triggerRequestAnimationFrame(startTime);
  triggerRequestAnimationFrame(startTime + 250);
  expect(AnimatedLine).toBeRenderedWithProps({
    commandToAnimate: horizontalLine,
    turtle: { x: 150, y: 100, angle: 0 }
  });
});
```

23. Make that pass by introducing the `start` and `elapsed` times, as shown:

```
useEffect(() => {
  let start, duration;
```

```

const handleDrawLineFrame = time => {
  if (start === undefined) start = time;
  const elapsed = time - start;
  const { x1, x2, y1, y2 } = commandToAnimate;
  setTurtle(turtle => ({
    ...turtle,
    x: x1 + ((x2 - x1) * (elapsed / duration)),
    y: y1 + ((y2 - y1) * (elapsed / duration)),
  }));
};

if (isDrawingLine) {
  ...
}

}, [commandToAnimate, isDrawingLine]);

```

24. Our components need to call `requestAnimationFrame` repeatedly until the duration is reached. At that point, the line should have been fully drawn. In this test, we trigger three animation frames, and we expect `requestAnimationFrame` to have been called three times:

```

it("invokes requestAnimationFrame repeatedly until the duration is reached", () => {
  renderWithStore(<Drawing />, horizontalLineDrawn);
  triggerRequestAnimationFrame(0);
  triggerRequestAnimationFrame(250);
  triggerRequestAnimationFrame(500);
  expect(
    window.requestAnimationFrame.mock.calls
  ).toHaveLength(3);
});

```

25. To make that pass, we need to ensure that `handleDrawLineFrame` triggers another `requestAnimationFrame` when it's run. However, we should only do that until the time that the duration has been reached. Make that pass happen by wrapping the `setTurtle` and `requestAnimationFrame` calls with the following conditional:

```

const handleDrawLineFrame = (time) => {
  if (start === undefined) start = time;
  if (time < start + duration) {
    const elapsed = time - start;

```

```

    const { x1, x2, y1, y2 } = commandToAnimate;
    setTurtle(...);
    requestAnimationFrame(handleDrawLineFrame);
}
};

```

26. For the next test, we will check that when a line has “finished” being drawn, we move on to the next one, if there is one (otherwise, we stop). Add a new `describe` block below the `describe` block we’ve just implemented, with the following test. The second time stamp, 500, is after the duration that is required for `horizontalLine` to be drawn and therefore, `AnimatedLine` should show `verticalLine` instead:

```

describe("after animation", () => {
  it("animates the next command", () => {
    renderWithStore(<Drawing />, {
      script: {
        drawCommands: [horizontalLine, verticalLine]
      }
    });
    triggerRequestAnimationFrame(0);
    triggerRequestAnimationFrame(500);
    expect(AnimatedLine).toBeRenderedWithProps(
      expect.objectContaining({
        commandToAnimate: verticalLine,
      })
    );
  });
});

```

27. To make that pass, we need to introduce a pointer to the command that is currently being animated. This will start at the 0 index, and we’ll increment it each time the animation finishes. Add the following new state variable at the top of the component:

```

const [
  animatingCommandIndex,
  setAnimatingCommandIndex
] = useState(0);

```

28. Update the `commandToAnimate` constant to use this new variable:

```
const commandToAnimate =
  drawCommands [animatingCommandIndex] ;
```

29. Add an `else` clause to the conditional in `handleDrawLineFrame` that increments the value:

```
if (time < start + duration) {
  ...
} else {
  setAnimatingCommandIndex(
    animatingCommandIndex => animatingCommandIndex + 1
  );
}
```

30. For the final test, we want to make sure that only previously animated commands are sent to `StaticLines`. The currently animating line will be rendered by `AnimatedLine`, and lines that haven't been animated yet shouldn't be rendered at all:

```
it("places line in StaticLines", () => {
  renderWithStore(<Drawing />, {
    script: {
      drawCommands: [horizontalLine, verticalLine]
    }
  });
  triggerRequestAnimationFrame(0);
  triggerRequestAnimationFrame(500);
  expect(StaticLines).toBeRenderedWithProps({
    lineCommands: [horizontalLine]
  });
});
```

31. To make that pass, update `lineCommands` to take only the portion of `drawCommands` up until the current `animatingCommandIndex` value:

```
const lineCommands = drawCommands
  .slice(0, animatingCommandIndex)
  .filter(isDrawLineCommand);
```

32. Although the latest test will now pass, the existing test, `sends only line commands to StaticLines`, will now break. Since our latest test covers essentially the same functionality, you can safely delete that test now.

If you run the app, you'll now be able to see lines being animated as they are placed on the screen.

In the next section, we'll ensure the animations behave nicely when multiple commands are entered by the user at the same time.

Cancelling animations with cancelAnimationFrame

The `useEffect` hook we've written has `commandToAnimate` and `isDrawingLine` in its dependency list. That means that when either of these values updates, the `useEffect` hook is torn down and will be restarted. But there are other occasions when we want to cancel the animation. One time this happens is when the user resets their screen.

If a command is currently animating when the user clicks the **Reset** button, we don't want the current animation frame to continue. We want to clean that up.

Let's add a test for that now:

1. Add the following test at the bottom of `test/Drawing.test.js`:

```
it("calls cancelAnimationFrame on reset", () => {
  renderWithStore(<Drawing />, {
    script: { drawCommands: [horizontalLine] }
  });
  renderWithStore(<Drawing />, {
    script: { drawCommands: [] }
  });
  expect(window.cancelAnimationFrame).toBeCalledWith(
    cancelToken
  );
}) ;
```

2. You'll also need to change the `beforeEach` block, making the `requestAnimationFrame` stub return a dummy cancel token, and adding in a new stub for the `cancelAnimationFrame` function:

```
describe("Drawing", () => {
  const cancelToken = "cancelToken";

  beforeEach(() => {
    ...
    jest
      .spyOn(window, "requestAnimationFrame")
```

```
        .mockReturnValue(cancelToken);
        jest.spyOn(window, "cancelAnimationFrame");
    });
}) ;
```

3. To make the test pass, update the `useEffect` hook to store the `cancelToken` value that the `requestAnimationFrame` function returns when it's called. Then return a cleanup function from the `useEffect` hook, which uses that token to cancel the next requested frame. This function will be called by React when it tears down the hook:

```
useEffect(() => {
  let start, duration, cancelToken;
  const handleDrawLineFrame = time => {
    if (start === undefined) start = time;
    if (time < start + duration) {
      ...
      cancelToken = requestAnimationFrame(
        handleDrawLineFrame
      );
    } else {
      ...
    }
  };
  if (isDrawingLine) {
    duration =
      movementSpeed * distance(commandToAnimate);
    cancelToken = requestAnimationFrame(
      handleDrawLineFrame
    );
  }
}

return () => {
  cancelAnimationFrame(cancelToken);
}
});
```

4. Finally, we don't want to run this cleanup if `cancelToken` hasn't been set. The token won't have been set if we aren't currently rendering a line. We can prove that with the following test, which you should add now:

```
it("does not call cancelAnimationFrame if no line
animating", () => {
  jest.spyOn(window, "cancelAnimationFrame");
  renderWithStore(<Drawing />, {
    script: { drawCommands: [] }
  });
  renderWithStore(<React.Fragment />);
  expect(
    window.cancelAnimationFrame
  ).not.toHaveBeenCalled();
});
```

Unmounting a component

This test shows how you can mimic an *unmount* of a component in React, which is simply by rendering `<React.Fragment />` in place of the component under test. React will unmount your component when this occurs.

5. To make that pass, simply wrap the returned cleanup function in a conditional:

```
return () => {
  if (cancelToken) {
    cancelAnimationFrame(cancelToken);
  }
};
```

That's all we need to do for animating the `drawLine` commands. Next up is rotating the turtle.

Varying animation behavior

Our lines and turtle are now animating nicely. However, we still need to handle the second type of draw command: rotations. The turtle will move at a constant speed when rotating to a new angle. A full rotation should take 1 second to complete, and we can use this to calculate the duration of the rotation. For example, a quarter rotation will take 0.25 seconds to complete.

In the last section, we started with a test to check that we were calling `requestAnimationFrame`. This time, that test isn't essential because we've already proved the same design with drawing lines. We can jump right into the more complex tests, using the same `triggerRequestAnimationFrame` helper as before.

Let's update `Drawing` to animate the turtle's coordinates:

1. Add the following test to the bottom of the `Drawing` `describe` block. Create it in another nested `describe` block, just below the last test you wrote. The test follows the same principle as our tests for drawing lines: we trigger two animation frames, one at time 0 ms and one at time 500 ms, and then expect the rotation to have occurred. Both the `x` and `y` coordinates are tested in addition to the `angle`; that's to make sure we continue to pass those through:

```
describe("rotation animation", () => {
  const rotationPerformed = {
    script: { drawCommands: [rotate90] },
  };

  it("rotates the turtle", () => {
    renderWithStore(<Drawing />, rotationPerformed);
    triggerRequestAnimationFrame(0);
    triggerRequestAnimationFrame(500);
    expect(Turtle).toBeRenderedWithProps({
      x: 0,
      y: 0,
      angle: 90
    });
  });
});
```

2. Moving to `src/Drawing.js`, start by adding a definition of `isRotateCommand`, just below the definition of `isDrawLineCommand`:

```
const isRotateCommand = command =>
  command.drawCommand === "rotate";
```

3. In the `Drawing` component, add a new constant, `isRotating`, just below the definition of `isDrawingLine`:

```
const isRotating =
  commandToAnimate &&
  isRotateCommand(commandToAnimate);
```

4. In the `useEffect` hook, define a new handler for rotations, `handleRotationFrame`, just below the definition of `handleDrawLineFrame`. For the purposes of this test, it doesn't need to do much other than set the angle to the new value:

```
const handleRotationFrame = time => {
  setTurtle(turtle => ({
    ...turtle,
    angle: commandToAnimate.newAngle
  }));
};
```

5. We can make use of that to call `requestAnimationFrame` when a rotation command is being animated. Modify the last section of the `useEffect` hook to look as follows, ensuring that you add `isRotating` to the dependency list. The test should pass after this change:

```
useEffect(() => {
  ...

  if (isDrawingLine) {
    ...
  } else if (isRotating) {
    requestAnimationFrame(handleRotationFrame);
  }
}, [commandToAnimate, isDrawingLine, isRotating]);
```

6. Let's add a test to get the duration in and use it within our calculation. This is essentially the same as the last test, but with a different duration and, therefore, a different expected rotation:

```
it("rotates part-way at a speed of 1s per 180 degrees",
() => {
  renderWithStore(<Drawing />, rotationPerformed);
  triggerRequestAnimationFrame(0);
  triggerRequestAnimationFrame(250);
  expect(Turtle).toBeRenderedWithProps({
    x: 0,
    y: 0,
    angle: 45
  });
});
```

7. To make this pass, first, we need to define `rotateSpeed`. You can add this definition just below the definition for `movementSpeed`:

```
const rotateSpeed = 1000 / 180;
```

8. Next, update the conditional at the bottom of the `useEffect` handler to calculate the duration for the `rotate` command:

```
} else if (isRotating) {
  duration =
    rotateSpeed *
    Math.abs(
      commandToAnimate.newAngle -
      commandToAnimate.previousAngle
    );
  requestAnimationFrame(handleRotationFrame);
}
```

9. Update `handleRotationFrame` to use the duration to calculate a proportionate angle to move by:

```
const handleRotationFrame = (time) => {
  const {
    previousAngle, newAngle
  } = commandToAnimate;
  setTurtle(turtle => ({
    ...turtle,
    angle:
      previousAngle +
      (newAngle - previousAngle) * (time / duration)
  }));
};
```

10. Just as with `handleDrawLineFrame`, we need to ensure that we can handle start times other than 0. Add the following test:

```
it("calculates rotation with a non-zero animation start time", () => {
  const startTime = 12345;
  renderWithStore(<Drawing />, rotationPerformed);
  triggerRequestAnimationFrame(startTime);
```

```
triggerRequestAnimationFrame(startTime + 250);
expect(Turtle).toBeRenderedWithProps({
  x: 0,
  y: 0,
  angle: 45
});
});
```

11. Make that pass by adding the `start` and `elapsed` variables. After this, the test should be passing. You'll notice the similarity between `handleDrawLineFrame` and `handleRotationFrame`:

```
const handleRotationFrame = (time) => {
  if (start === undefined) start = time;
  const elapsed = time - start;
  const {
    previousAngle, newAngle
  } = commandToAnimate;
  setTurtle(turtle => ({
    ...turtle,
    angle:
      previousAngle +
      (newAngle - previousAngle) *
      (elapsed / duration)
  }));
};
```

12. Add a test to make sure we're calling `requestAnimationFrame` repeatedly. This is the same test that we used for the `drawLine` handler, except now we're passing in the `rotate90` command. Remember to make sure the test belongs in the nested context, so you can be sure that there's no name clash:

```
it("invokes requestAnimationFrame repeatedly until the
duration is reached", () => {
  renderWithStore(<Drawing />, rotationPerformed);
  triggerRequestAnimationFrame(0);
  triggerRequestAnimationFrame(250);
  triggerRequestAnimationFrame(500);
  expect(
```

```

        window.requestAnimationFrame.mock.calls
    ).toHaveLength(3);
});

```

13. To make this pass, we need to do a couple of things. First, we need to modify `handleRotationFrame` in the same way we did with `handleDrawLineFrame`, by adding a conditional that stops animating after the duration has been reached. Second, we also need to fill in the second part of the conditional to set the turtle location when the animation is finished:

```

const handleRotationFrame = (time) => {
  if (start === undefined) start = time;
  if (time < start + duration) {
    ...
  } else {
    setTurtle(turtle => ({
      ...turtle,
      angle: commandToAnimate.newAngle
    }));
  }
};

```

Handling the end animation state

This `else` clause wasn't necessary with the `drawLine` handler because, as soon as a line finishes animating, it will be passed to `StaticLines`, which renders all lines with their full length. This isn't the case with the rotation angle: it remains fixed until the next rotation. Therefore, we need to ensure it's at its correct final value.

14. We've got one final test. We need to increment the current animation command once the animation is done. As with the same test in the previous section, this test should live *outside* the `describe` block we've just used since it has a different test setup:

```

it("animates the next command once rotation is complete",
  async () => {
    renderWithStore(<Drawing />, {
      script: {
        drawCommands: [rotate90, horizontalLine]
      }
    });
  }
);

```

```

triggerRequestAnimationFrame(0);
triggerRequestAnimationFrame(500);
triggerRequestAnimationFrame(0);
triggerRequestAnimationFrame(250);
expect(Turtle).toBeRenderedWithProps({
  x: 150,
  y: 100,
  angle: 90
});
});
}
);

```

15. To make that pass, add the call to `setNextCommandToAnimate` into the `else` condition:

```

} else {
  setTurtle(turtle => ({
    ...turtle,
    angle: commandToAnimate.newAngle
  }));
  setAnimatingCommandIndex(
    (animatingCommandToIndex) =>
      animatingCommandToIndex + 1
  );
}
}

```

That's it! If you haven't done so already, it's worth running the app to try it out.

Summary

In this chapter, we've explored how to test the `requestAnimationFrame` browser API. It's not a straightforward process, and there are multiple tests that need to be written if you wish to be fully covered.

Nevertheless, you've seen that it is entirely possible to write automated tests for onscreen animation. The benefit of doing so is that the complex production code is fully documented via the tests.

In the next chapter, we'll look at adding WebSocket communication into Spec Logo.

Exercises

1. Update `Drawing` so that it resets the turtle position when the user clears the screen with the **Reset** button.
2. Our tests have a lot of duplication due to the repeated calls to `triggerRequestAnimationFrame`. Simplify how this is called by creating a wrapper function called `triggerAnimationSequence` that takes an array of frame times and calls `triggerRequestAnimationFrame` for each of those times.
3. Loading an existing script (for example, on startup) will take a long time to animate all instructions, and so will pasting in code snippets. Add a **Skip animation** button that can be used to skip all the queued animations.
4. Ensure that the **Undo** button works correctly when animations are in progress.

16

Working with WebSockets

In this chapter, we'll look at how to test-drive the WebSocket API within our React app. We'll use it to build a teaching mechanism whereby one person can share their screen and others can watch as they type out commands.

The WebSocket API isn't straightforward. It uses a number of different callbacks and requires functions to be called in a certain order. To make things harder, we'll do this all within a Redux saga: that means we'll need to do some work to convert the callback API to one that can work with generator functions.

Because this is the last chapter covering unit testing techniques, it does things a little differently. It doesn't follow a strict TDD process. The starting point for this chapter has a skeleton of our functions already completed. You'll flesh out these functions, concentrating on learning test-driven techniques for WebSocket connections.

This chapter covers the following topics:

- Designing a WebSocket interaction
- Test-driving a WebSocket connection
- Streaming events with redux-saga
- Updating the app

By the end of the chapter, you'll have learned how the WebSocket API works along with its unit testing mechanisms.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter16>

Designing a WebSocket interaction

In this section, we'll start by describing the sharing workflow, then we'll look at the new UI elements that support this workflow, and finally we'll walk through the code changes you'll make in this chapter.

The sharing workflow

A sharing session is made up of one presenter and zero or more watchers. That means there are two modes that the app can be in: either **presenting** or **watching**.

When the app is in presenting mode, then everyone watching will get a copy of your Spec Logo instructions. All your instructions are sent to the server via a WebSocket.

When your app is in watching mode, a WebSocket receives instructions from the server and immediately outputs them onto your screen.

The messages sent to and from the server are simple JSON-formatted data structures.

Figure 16.1 shows how the interface looks when it's in presenter mode.

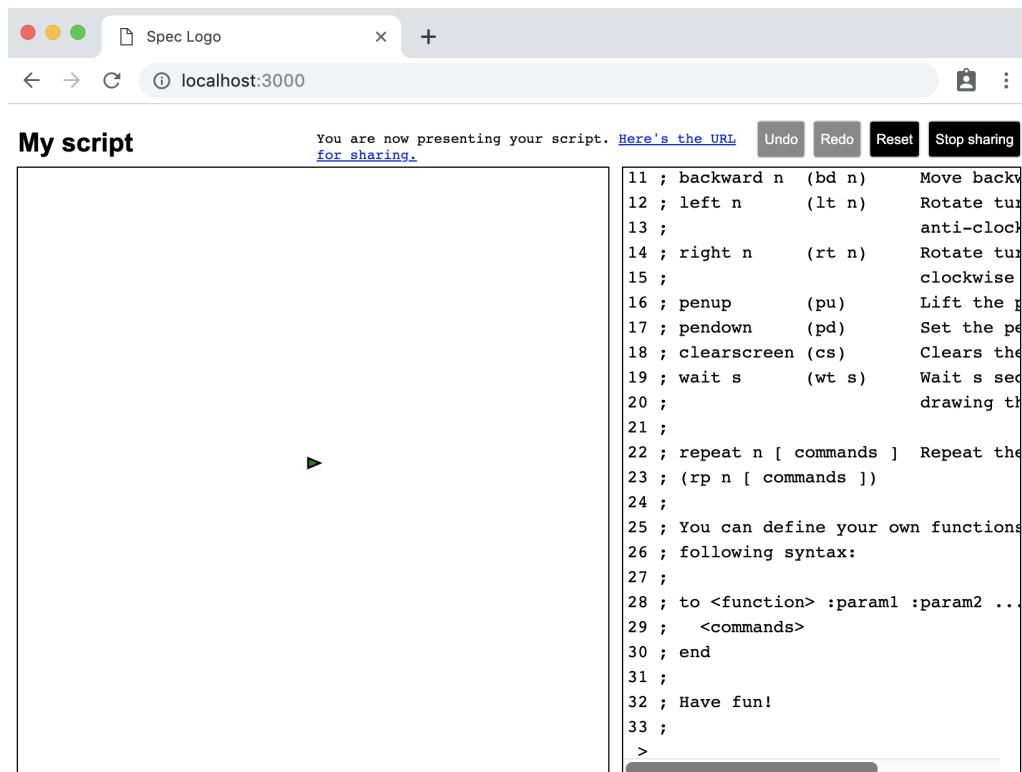


Figure 16.1 – Spec Logo in presenter mode

So, how does it work?

1. The presenter clicks the **Start sharing** button. The server is sent the following message:

```
{ type: "START_SHARING" }
```

2. The server then responds with the ID of the session:

```
{ status: "STARTED", id: 123 }
```

3. This ID is used to construct a URL that opens the application in watching mode, for example:

```
http://localhost:3000/index.html?watching=123
```

4. The URL can be shared and opened anywhere. When the application is opened in this mode, the application immediately opens a WebSocket to the server and sends this message:

```
{ type: "START_WATCHING", id: 123 }
```

5. There can be any number of watchers that connect. On an initial connection, any commands that the presenter has already sent to the server will be replayed. Those commands are sent from the presenter for any Redux action of type `SUBMIT_EDIT_LINE`, and they are sent via the WebSocket to the server like this:

```
{
  type: "NEW_ACTION",
  innerAction: {
    type: "SUBMIT_EDIT_LINE",
    text: "forward 10\n"
  }
}
```

6. When the server receives actions on the presenter's WebSocket, it immediately forwards the actions to every subscriber:

```
{ type: "SUBMIT_EDIT_LINE", text: "forward 10\n" } }
```

7. The server also stores received actions in history, so the actions can be replayed by new joiners.
8. When the watcher is done, they simply need to close the browser window and their WebSocket will close.
9. When the presenter is done, they can either close the browser window or hit the **Stop sharing** button. This closes the connection and the server clears out its internal state.

The new UI elements

Here's what you'll find in the UI; all of this has already been built for you:

- A new menu button to toggle sharing on and off. It is named **Start sharing**, but once sharing has started, the name switches to **Stop sharing**.
- There is a new message that appears as part of the menu buttons bar when Spec Logo is in sharing mode. It contains a message telling the user whether they are presenting or watching. If they are presenting, it also contains a URL that they can copy and share with others.
- You can now launch the app in watching mode by adding the search parameter `?watching=<id>` to the end of the Spec Logo URL.

Next, let's have a look at the skeleton of the Redux saga that you'll be fleshing out.

Splitting apart the saga

A new piece of Redux middleware exists in the file `src/middleware/sharingSagas.js`. This file has two parts to it. First, there's a middleware function named `duplicateForSharing`. This is a filter that provides us with all the actions that we wish to broadcast:

```
export const duplicateForSharing =
  store => next => action => {
  if (action.type === "SUBMIT_EDIT_LINE") {
    store.dispatch({
      type: "SHARE_NEW_ACTION",
      innerAction: action,
    });
  }
  return next(action);
};
```

Second, there's the root saga itself. It's split into four smaller functions, and these are the functions we'll fill out in this chapter, using a test-driven approach:

```
export function* sharingSaga() {
  yield takeLatest("TRY_START_WATCHING", startWatching);
  yield takeLatest("START_SHARING", startSharing);
  yield takeLatest("STOP_SHARING", stopSharing);
  yield takeLatest("SHARE_NEW_ACTION", shareNewAction);
}
```

With enough of the design done, let's get cracking with the implementation.

Test-driving a WebSocket connection

We start by filling out that first function, `startSharing`. This function is invoked when the `START_SHARING` action is received. That action is triggered when the user clicks the **Start sharing** button:

1. Open `test/middleware/sharingSagas.test.js` and add the following imports at the top:

```
import { storeSpy, expectRedux } from "expect-redux";
import { act } from "react-dom/test-utils";
import { configureStore } from "../../src/store";
```

2. At the bottom of the file, add a new `describe` block and its setup. We'll break this into a couple of steps: first, set up the Redux store and the WebSocket spy. Because `window.WebSocket` is a constructor function, we use `mockImplementation` to stub it out:

```
describe("sharingSaga", () => {
  let store;
  let socketSpyFactory;

  beforeEach(() => {
    store = configureStore([storeSpy]);
    socketSpyFactory = spyOn(window, "WebSocket");
    socketSpyFactory.mockImplementation(() => {
      return {};
    });
  });
}) ;
```

Understanding the WebSocket API

The `WebSocket` constructor returns an object with `send` and `close` methods, plus `onopen`, `onmessage`, `onclose`, and `onerror` event handlers. We'll implement most of these on our test double as we build out our test suite. If you'd like to learn more about the `WebSocket` API, check out the *Further reading* section at the end of this chapter.

3. Next, because we're also concerned with the window location, we also need to stub out the `window.location` object. Because this is a read-only object in the JSDOM environment, we need to use the `Object.defineProperty` function to override it. This is a little clunky, so you may prefer to extract that into its own function with a good name. Add the following into the same `beforeEach` block:

```
beforeEach(() => {
  ...
  Object.defineProperty(window, "location", {
    writable: true,
    value: {
      protocol: "http:",
      host: "test:1234",
      pathname: "/index.html",
    },
  });
});
```

4. Add the first test in a nested `describe` block. This checks that we make the WebSocket connection with the right URL:

```
describe("START_SHARING", () => {
  it("opens a websocket when starting to share", () => {
    store.dispatch({ type: "START_SHARING" });
    expect(socketSpyFactory).toBeCalledWith(
      "ws://test:1234/share"
    );
  });
});
```

5. Make that pass by filling in the `startSharing` generator function in the file `src/middleware/sharingSagas.js` (remembering that the skeleton has already been created for you). This code constructs a new URL with the right host:

```
function* startSharing() {
  const { host } = window.location;
  new WebSocket(`ws://${host}/share`);
}
```

6. Back in the test suite, modify the WebSocket stub implementation to add an inner spy, `sendSpy`, which will be called when the user calls the `send` function on the WebSocket. We also need to store a reference to the `socketSpy` function that's created, so we can call the callbacks that the user attaches to its event handlers (such as `onopen` and `onmessage`). This will make sense when we write the next test:

```
let sendSpy;
let socketSpy;

beforeEach(() => {
  sendSpy = jest.fn();
  socketSpyFactory = spyOn(window, "WebSocket");
  socketSpyFactory.mockImplementation(() => {
    socketSpy = {
      send: sendSpy,
    };
    return socketSpy;
  });
  ...
})
```

7. When test-driving an API with callbacks, such as the WebSocket API, it's important that we mimic the exact behavior of each callback. We will start with the `onopen` callback. The next test will trigger this as if it was a server sending a message. Because we expect a bunch of asynchronous actions to occur when `onopen` is received, we can use `async act` to wait for the actions to be completed. So, before the next test, define the following function, which triggers the `onopen` callback:

```
const notifySocketOpened = async () => {
  await act(async () => {
    socketSpy.onopen();
  });
};
```

Using `act` with non-React code

The `async act` function helps us even when we're not dealing with React components because it waits for promises to run before returning.

8. We can then use the `notifySocketOpened` function in our next test, which checks that when the client receives a `START_SHARING` action, it immediately forwards it onto the server:

```
it("dispatches a START_SHARING action to the socket",
  async () => {
    store.dispatch({ type: "START_SHARING" });
    await notifySocketOpened();
    expect(sendSpy).toBeCalledWith(
      JSON.stringify({ type: "START_SHARING" })
    );
  });
}
```

9. To make that pass, start by extracting the existing code in the `startSharing` function into a new function named `openWebsocket`. Then, add in code that invokes a `Promise` object that resolves when the `onopen` message is received on the socket. This code is fairly difficult—we’re building a `Promise` object specifically to adapt the callback-based API into something that we can use with the generator `yield` keyword:

```
const openWebSocket = () => {
  const { host } = window.location;
  const socket = new WebSocket(`ws://${host}/share`);
  return new Promise(resolve => {
    socket.onopen = () => {
      resolve(socket);
    };
  });
};
```

10. You can now make use of that `openWebSocket` function in `startSharing`. After this, your test should pass:

```
function* startSharing() {
  const presenterSocket = yield openWebSocket();
  presenterSocket.send(
    JSON.stringify({ type: "START_SHARING" })
  );
}
```

11. The next test will send a message over the socket from the server to the app. For this, we need a helper function to mimic sending a message and wait to empty the current task queue of

tasks. Add this helper to `test/middleware/sharingSagas.test.js`, just below `notifySocketOpened`:

```
const sendSocketMessage = async message => {
  await act(async () => {
    socketSpy.onmessage({
      data: JSON.stringify(message)
    });
  });
};
```

12. Add the next test, using the function you've just defined:

```
it("dispatches an action of STARTED_SHARING with
a URL containing the id that is returned from the
server",  async () => {
  store.dispatch({ type: "START_SHARING" });
  await notifySocketOpened();
  await sendSocketMessage({
    type: "UNKNOWN",
    id: 123,
  });
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({
      type: "STARTED_SHARING",
      url: "http://test:1234/index.html?watching=123",
    });
});
```

13. To make this pass, we'll read the message from the socket. Once that's done, we can pass the retrieved information back to the Redux store. Start by adding the following new functions at the top of `src/middleware/sharingSagas.js`:

```
const receiveMessage = (socket) =>
  new Promise(resolve => {
    socket.onmessage = evt => {
      resolve(evt.data)
    };
});
```

```

    });

const buildUrl = (id) => {
  const {
    protocol, host, pathname
  } = window.location;
  return (
    `${protocol}//${host}${pathname}?watching=${id}`
  );
}

```

14. Now you can use those functions to finish the implementation of `startSharing`:

```

function* startSharing() {
  const presenterSocket = yield openWebSocket();
  presenterSocket.send(
    JSON.stringify({ type: "START_SHARING" })
  );
  const message = yield receiveMessage(
    presenterSocket
  );
  const presenterSessionId = JSON.parse(message).id;
  yield put({
    type: "STARTED_SHARING",
    url: buildUrl(presenterSessionId),
  });
}

```

That's it for the process of starting to share. Now let's deal with what happens when the user clicks the **Stop sharing** button:

1. Create a helper function inside the `describe` block named `sharingSaga`, as shown. This function will change the system to a state of `STARTED_SHARING`:

```

const startSharing = async () => {
  store.dispatch({ type: "START_SHARING" });
  await notifySocketOpened();
  await sendSocketMessage({
    type: "UNKNOWN",
  });
}

```

```
    id: 123,  
  } );  
};
```

2. Update the spy to include a `closeSpy` variable, which we set up in the same way as `sendSpy`:

```
let closeSpy;  
  
beforeEach(() => {  
  sendSpy = jest.fn();  
  closeSpy = jest.fn();  
  socketSpyFactory = spyOn(window, "WebSocket");  
  socketSpyFactory.mockImplementation(() => {  
    socketSpy = {  
      send: sendSpy,  
      close: closeSpy,  
    };  
    return socketSpy;  
  });  
  ...  
});
```

3. Add the first test in a new nested context. It begins by starting sharing and then dispatches the `STOP_SHARING` action:

```
describe("STOP_SHARING", () => {  
  it("calls close on the open socket", async () => {  
    await startSharing();  
    store.dispatch({ type: "STOP_SHARING" });  
    expect(closeSpy).toBeCalled();  
  });  
});
```

4. To make this pass, we'll fill out the `stopSharing` generator function. First, however, we need to get access to the socket that we created within the `startSharing` function. Extract that variable into the top-level namespace:

```
let presenterSocket;  
  
function* startSharing() {
```

```

presenterSocket = yield openWebSocket();
...
}

```

- Then, add the following definition to the `stopSharing` function. You can then run your tests, and everything should pass; however, if you're running your entire test suite (with `npm test`), you'll see a couple of console errors appear. These are coming from one test in the `MenuButtons` test suite—we will fix this in the *Updating the app* section later:

```

function* stopSharing() {
  presenterSocket.close();
}

```

Running tests in just a single suite

To avoid seeing the console errors, remember you can opt to run tests for this test suite only using the command `npm test test/middleware/sharingSagas.test.js`.

- Moving on to the next test, we want to update the Redux store with the new `stopped` status. This will allow us to remove the message that appeared to the user when they began sharing:

```

it("dispatches an action of STOPPED_SHARING", async () =>
{
  await startSharing();
  store.dispatch({ type: "STOP_SHARING" });
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "STOPPED_SHARING" });
});

```

- That's a simple one-liner to make pass:

```

function* stopSharing() {
  presenterSocket.close();
  yield put({ type: "STOPPED_SHARING" });
}

```

Next up is broadcasting actions from the presenter to the server:

- Create a new nested `describe` block with the following test:

```

describe("SHARE_NEW_ACTION", () => {
  it("forwards the same action on to the socket", async

```

```
(() => {
  const innerAction = { a: 123 };
  await startSharing(123);
  store.dispatch({
    type: "SHARE_NEW_ACTION",
    innerAction,
  });
  expect(sendSpy).toHaveBeenCalledWith(
    JSON.stringify({
      type: "NEW_ACTION",
      innerAction,
    })
  );
});
});
```

2. Make it pass by filling in the following content for the `shareNewAction` function:

```
const shareNewAction = ({ innerAction }) => {
  presenterSocket.send(
    JSON.stringify({
      type: "NEW_ACTION",
      innerAction,
    })
  );
}
```

3. Add the next test, which checks that we do not send any actions if the user isn't presenting:

```
it("does not forward if the socket is not set yet", () =>
{
  store.dispatch({ type: "SHARE_NEW_ACTION" });
  expect(sendSpy).not.toBeCalled();
});
```

Using `not.toBeCalled` in an asynchronous environment

This test has a subtle issue. Although it will help you add to the design of your software, it's slightly less useful as a regression test because it *could* potentially result in false positives. This test guarantees that something doesn't happen between the start and the end of the test, but it makes no guarantees about what happens *after*. Such is the nature of the `async` environment.

- Making this test pass is simply a matter of adding a conditional around the code we have:

```
function* shareNewAction({ innerAction } ) {
  if (presenterSocket) {
    presenterSocket.send(
      JSON.stringify({
        type: "NEW_ACTION",
        innerAction,
      })
    );
  }
}
```

- We also don't want to share the action if the user has stopped sharing—so let's add that in:

```
it("does not forward if the socket has been closed",
  async () => {
    await startSharing();
    socketSpy.readyState = WebSocket.CLOSED;
    store.dispatch({ type: "SHARE_NEW_ACTION" });
    expect(sendSpy.mock.calls).toHaveLength(1);
  });
}
```

The WebSocket specification

The constant in the preceding test, `WebSocket.CLOSED`, and the constant in the following code, `WebSocket.OPEN`, are defined in the WebSocket specification.

- Move to the top of the test file and define the following two constants, underneath your imports. These are needed because when we spy on the `WebSocket` constructor, we overwrite these values. So, we need to add them back in. Start by saving the real values:

```
const WEB_SOCKET_OPEN = WebSocket.OPEN;
const WEB_SOCKET_CLOSED = WebSocket.CLOSED;
```

7. Update your spy to set these constants once WebSocket has been stubbed. While we're here, let's also set the default readyState for a socket to be `WebSocket.OPEN`, which means our other tests won't break:

```
socketSpyFactory = jest.spyOn(window, "WebSocket");
Object.defineProperty(socketSpyFactory, "OPEN", {
  value: WEB_SOCKET_OPEN
});
Object.defineProperty(socketSpyFactory, "CLOSED", {
  value: WEB_SOCKET_CLOSED
});
socketSpyFactory.mockImplementation(() => {
  socketSpy = {
    send: sendSpy,
    close: closeSpy,
    readyState: WebSocket.OPEN,
  };
  return socketSpy;
});
```

8. Finally, back in the production code, make the test pass by checking if `readyState` is `WebSocket.OPEN`, which is not exactly what the test specified, but it's good enough to make it pass:

```
const shareNewAction = ({ innerAction }) => {
  if (
    presenterSocket &&
    presenterSocket.readyState === WebSocket.OPEN
  ) {
    presenterSocket.send(
      JSON.stringify({
        type: "NEW_ACTION",
        innerAction,
      })
    );
  }
}
```

That's it for the presenter behavior: we have test-driven the `onopen`, `onclose`, and `onmessage` callbacks. In a real-world application, you would want to follow the same process for the `onerror` callback.

Now let's look at the watcher's behavior.

Streaming events with redux-saga

We'll repeat a lot of the same techniques in this section. There are two new concepts: first, pulling out the `search` param for the watcher ID, and second, using `eventChannel` to subscribe to the `onmessage` callback. This is used to continually stream messages from the WebSocket into the Redux store.

Let's begin by specifying the new URL behavior:

1. Write a new `describe` block at the bottom of `test/middleware/sharingSagas.test.js`, but still nested inside the main `describe` block:

```
describe("watching", () => {
  beforeEach(() => {
    Object.defineProperty(window, "location", {
      writable: true,
      value: {
        host: "test:1234",
        pathname: "/index.html",
        search: "?watching=234"
      }
    });
  });
}

it("opens a socket when the page loads", () => {
  store.dispatch({ type: "TRY_START_WATCHING" });
  expect(socketSpyFactory).toBeCalledWith(
    "ws://test:1234/share"
  );
});
```

2. Make it pass by filling out the `startWatching` function in your production code. You can make use of the existing `openWebSocket` function:

```
function* startWatching() {  
  yield openWebSocket();  
}
```

3. In the next test, we'll begin to make use of the `search` param:

```
it("does not open socket if the watching field is not set", () => {  
  window.location.search = "?";  
  store.dispatch({ type: "TRY_START_WATCHING" });  
  expect(socketSpyFactory).not.toBeCalled();  
});
```

4. Make it pass by extracting the `search` param using the `URLSearchParams` object:

```
function* startWatching() {  
  const sessionId = new URLSearchParams(  
    window.location.search.substring(1)  
  ).get("watching");  
  
  if (sessionId) {  
    yield openWebSocket();  
  }  
}
```

5. Before we write the next test, add the following helper function, which mimics the action that will occur on the real WebSocket, ensuring that `onopen` is called:

```
const startWatching = async () => {  
  await act(async () => {  
    store.dispatch({ type: "TRY_START_WATCHING" });  
    socketSpy.onopen();  
  });  
};
```

6. When a new watch session has started, we need to reset the user's output so that it is blank:

```
it("dispatches a RESET action", async () => {  
  await startWatching();
```

```

        return expectRedux(store)
            .toDispatchAnAction()
            .matching({ type: "RESET" });
    });
}

```

7. Make it pass by adding in a put function call:

```

function* startWatching() {
    const sessionId = new URLSearchParams(
        location.search.substring(1)
    ).get("watching");

    if (sessionId) {
        yield openWebSocket();
        yield put({ type: "RESET" });
    }
}

```

8. Next, we need to send a message to the server, including the ID of the session we wish to watch:

```

it("sends the session id to the socket with an action
type of START_WATCHING", async () => {
    await startWatching();
    expect(sendSpy).toBeCalledWith(
        JSON.stringify({
            type: "START_WATCHING",
            id: "234",
        })
    );
})
;
```

9. We already have our spy set up from the previous section, so this is a quick one to fix:

```

function* startWatching() {
    const sessionId = new URLSearchParams(
        window.location.search.substring(1)
    ).get("watching");

    if (sessionId) {
}

```

```

const watcherSocket = yield openWebSocket();
yield put({ type: "RESET" });
watcherSocket.send(
  JSON.stringify({
    type: "START_WATCHING",
    id: sessionId,
  })
);
}
}
}

```

10. The next test tells the Redux store that we have started watching. This will then allow the React UI to display a message to the user telling them that they are connected:

```

it("dispatches a STARTED_WATCHING action", async () => {
  await startWatching();
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "STARTED_WATCHING" });
});

```

11. Make that pass by adding a new call to `put`, as shown:

```

function* startWatching() {
  ...
  if (sessionId) {
    ...
    yield put({ type: "STARTED_WATCHING" });
  }
}

```

12. Now the big one. We need to add in the behavior that allows us to receive multiple messages from the server and read them in:

```

it("relays multiple actions from the websocket", async () => {
  const message1 = { type: "ABC" };
  const message2 = { type: "BCD" };
  const message3 = { type: "CDE" };
  await startWatching();
}

```

```

        await sendSocketMessage(message1);
        await sendSocketMessage(message2);
        await sendSocketMessage(message3);

        await expectRedux(store)
            .toDispatchAnAction()
            .matching(message1);
        await expectRedux(store)
            .toDispatchAnAction()
            .matching(message2);
        await expectRedux(store)
            .toDispatchAnAction()
            .matching(message3);
        socketSpy.onclose();
    });
}

```

Long tests

You may think it would help to have a smaller test that handles just one message. However, that won't help us for multiple messages, as we need to use an entirely different implementation for multiple messages, as you'll see in the next step.

13. We'll use the `eventChannel` function to do this. Its usage is similar to the earlier `Promise` object usage when we converted a callback to an operation that could be awaited with `yield`. With the `Promise` object, we called `resolve` when the callback was received. With `eventChannel`, when the callback is received, we invoke `emitter(END)`. The significance of this will become apparent in the next step:

```

import { eventChannel, END } from "redux-saga";

const webSocketListener = socket =>
  eventChannel(emitter => {
    socket.onmessage = emitter;
    socket.onclose = () => emitter(END);
    return () => {
      socket.onmessage = undefined;
      socket.onclose = undefined;
    }
  });
}

```

```
    } ;  
} ) ;
```

Understanding the eventChannel function

The `eventChannel` function from `redux-saga` is a mechanism for consuming event streams that occur outside of Redux. In the preceding example, the WebSocket provides the stream of events. When invoked, `eventChannel` calls the provided function to initialize the channel, then the provided `emitter` function must be called each time an event is received. In our case, we pass the message directly to the `emitter` function without modification. When the WebSocket is closed, we pass the special `END` event to signal to `redux-saga` that no more events will be received, allowing it to close the channel.

14. Now you can use the `websocketListener` function to create a channel that we can repeatedly take events from using a loop. This loop needs to be wrapped in a `try` construct. The `finally` block will be called when the `emitter(END)` instruction is reached. Create a new generator function that does that, as shown:

```
function* watchUntilStopRequest(chan) {  
  try {  
    while (true) {  
      let evt = yield take(chan);  
      yield put(JSON.parse(evt.data));  
    }  
  } finally {  
  }  
};
```

15. Link the `websocketListener` function and the `watchUntilStopRequest` generator function by calling them both from within `startWatching`. After this step, your test should pass:

```
function* startWatching() {  
  ...  
  if (sessionId) {  
    ...  
    yield put({ type: "STARTED_WATCHING" });  
    const channel = yield call(  
      websocketListener, watcherSocket  
    );
```

```

        yield call(watchUntilStopRequest(channel));
    }
}

```

16. The final test is to alert the Redux store that we've stopped watching so that it can then remove the message that appears in the React UI:

```

it("dispatches a STOPPED_WATCHING action when the
connection is closed", async () => {
    await startWatching();
    socketSpy.onclose();

    return expectRedux(store)
        .toDispatchAnAction()
        .matching({ type: "STOPPED_WATCHING" });
});

```

17. Make that pass by adding this one-liner to the `finally` block in `watchUntilStopRequest`:

```

try {
    ...
} finally {
    yield put({ type: "STOPPED_WATCHING" });
}

```

You've now completed the saga: your application is now receiving events, and you've seen how to use the `eventChannel` function to listen to a stream of messages.

All that's left is to integrate this into our React component.

Updating the app

We've completed the work on building the sagas, but we have just a couple of adjustments to make in the rest of the app.

The `MenuButtons` component is already functionally complete, but we need to update the tests to properly exercise the middleware, in two ways: first, we must stub out the `WebSocket` constructor, and second, we need to fire off a `TRY_START_WATCHING` action as soon as the app starts:

1. Open `test/MenuButtons.test.js` and start by importing the `act` function. We'll need this to await our socket saga actions:

```

import { act } from "react-dom/test-utils";

```

2. Next, find the `describe` block named `sharing button` and insert the following `beforeEach` block, which is similar to the same stubbed constructor you used in the saga tests:

```
describe("sharing button", () => {
  let socketSpyFactory;
  let socketSpy;

  beforeEach(() => {
    socketSpyFactory = jest.spyOn(
      window,
      "WebSocket"
    );
    socketSpyFactory.mockImplementation(() => {
      socketSpy = {
        close: () => {},
        send: () => {},
      };
      return socketSpy;
    });
  });
});
```

3. Next, in the same `describe` block, add the following `notifySocketOpened` implementation. This is different from the `notifySocketOpened` implementation in the saga tests, because it calls both `onopen` and then `onmessage`, with a sample message. All of this is necessary for the `startSharing` saga to run correctly: it mimics the WebSocket opening, then the server sending the first message, which should result in the `STARTED_SHARING` message being sent:

```
const notifySocketOpened = async () => {
  const data = JSON.stringify({ id: 1 });
  await act(async () => {
    socketSpy.onopen();
    socketSpy.onmessage({ data });
  });
};
```

4. We can now use this to update the test that is causing console errors. The test is the one with the description `dispatches an action of STOP_SHARING when stop sharing is clicked`. To avoid the errors, we must adjust a couple of lines. First, we dispatch a `START_SHARING` message, rather than a `STARTED_SHARING` message. Then, we use `notifySocketOpened` to mimic the server response to opening the socket. This will trigger the saga to send a `STARTED_SHARING` event, which causes the **Sharing** button in `MenuButtons` to change to be named **Stop sharing**. The test clicks it and waits for the `STOP_SHARING` event to be sent:

```
it("dispatches an action of STOP_SHARING when stop
sharing is clicked", async () => {
  renderWithStore(<MenuButtons />);
  dispatchToStore({ type: "START_SHARING" });
  await notifySocketOpened();
  click(buttonWithLabel("Stop sharing"));
  return expectRedux(store)
    .toDispatchAnAction()
    .matching({ type: "STOP_SHARING" });
});
```

5. With the test passing, update `src/index.js` to call the `TRY_START_WATCHING` action when the app first loads:

```
const store = configureStoreWithLocalStorage();
store.dispatch({ type: "TRY_START_WATCHING" });

ReactDOM
  .createRoot(document.getElementById("root"))
  .render(
    <Provider store={store}>
      <App />
    </Provider>);
```

You can now run the app and try it out. Here's a manual test you can try:

1. Open a session in a browser window and click **Start sharing**.
2. Right-click on the link that appears and choose to open it in a new window.
3. Move your two windows so that they are side by side.
4. In the original window, type some commands, such as `forward 100` and `right 90`. You should see the commands update.

5. Now, hit **Stop sharing** in the original window. You should see the sharing messages disappear from both screens.

That covers test-driving WebSockets.

Summary

In this chapter, we've covered how to test against the WebSocket API.

You've seen how to mock the WebSocket constructor function, and how to test-drive its `onopen`, `onclose`, and `onmessage` callbacks.

You've also seen how to use a `Promise` object to convert a callback into something that can be yielded in a generator function, and how you can use `eventChannel` to take a stream of events and send them into the Redux store.

In the next chapter, we'll look at using Cucumber tests to drive some improvements to the sharing feature.

Exercises

What tests could you add to ensure that socket errors are handled gracefully?

Further reading

The WebSocket specification:

<https://www.w3.org/TR/websockets/>

Part 4 – Behavior-Driven Development with Cucumber

This part is about **behavior-driven development (BDD)** using Cucumber tests. Whereas the first three parts were focused on building Jest unit tests at the component level, this part looks at writing tests at the *system* level—you might also think of these as end-to-end tests. The goal is to show how the TDD workflow applies beyond unit testing and can be used by the whole team, not just developers.

Finally, we end the book with a discussion of how TDD fits within the wider testing landscape and suggestions for how you can continue your TDD journey.

This part includes the following chapters:

- *Chapter 17, Writing Your First Cucumber Test*
- *Chapter 18, Adding Features Guided by Cucumber Tests*
- *Chapter 19, Understanding TDD in the Wider Testing Landscape*

Writing Your First Cucumber Test

Test-driven development is primarily a process for developers. Sometimes, customers and product owners want to see the results of automated tests too. Unfortunately, the humble unit test that is the foundation of TDD is simply too low-level to be helpful to non-developers. That's where the idea of **Behavior Driven Development (BDD)** comes in.

BDD tests have a few characteristics that set them apart from the unit tests you've seen so far:

- They are **end-to-end tests** that operate across the entire system.
- They are written in natural language rather than code, which is understandable by non-coders and coders alike.
- They avoid making references to internal mechanics, instead focusing on the outward behavior of the system.
- The test definition describes itself (with unit tests, you need to write a test description that matches the code).
- The syntax is designed to ensure that your tests are written as examples, and as discrete specifications of behavior.

BDD tools vs TDD vs unit tests

The style of TDD you've seen so far in this book treats (for the most part) its tests as examples that specify behavior. Also, our tests were always written in the **Arrange-Act-Assert (AAA)** pattern. However, notice that unit test tools such as Jest do not force you to write tests this way.

This is one reason why BDD tools exist: to force you to be very clear when you specify the behavior of your system.

This chapter introduces two new software packages: Cucumber and Puppeteer.

We'll use Cucumber to build our BDD tests. Cucumber is a system that exists for many different programming environments, including Node.js. It consists of a test runner that runs tests contained within **feature files**. Features are written in a plain-English language known as **Gherkin**. When Cucumber runs your tests, it translates these feature files into function calls; these function calls are written in JavaScript **support scripts**.

Since Cucumber has its own test runner, it doesn't use Jest. However, we will make use of Jest's `expect` package in some of our tests.

Cucumber is not the only way to write system tests

Another popular testing library is Cypress, which may be a better choice for you and/or your team. Cypress puts the emphasis on the visual presentation of results. I tend to avoid it because its API is very different from industry-standard testing patterns, which increases the amount of knowledge developers need to have. Cucumber is cross-platform and tests look very similar to the standard unit tests you've seen throughout this book.

Puppeteer performs a similar function to the JSDOM library. However, while JSDOM implements a fake DOM API within the Node.js environment, Puppeteer uses a real web browser, Chromium. In this book, we'll use it in *headless* mode, which means you won't see the app running onscreen; but you can, if you wish, turn headless mode off. Puppeteer comes with all sorts of bolt-ons, such as the ability to take screenshots.

Cross-browser testing

If you wish to test cross-browser support for your application, you may be better off looking at an alternative such as Selenium, which isn't covered in this book. However, the same testing principles apply when writing tests for Selenium.

This chapter covers the following topics:

- Integrating Cucumber and Puppeteer into your code base
- Writing your first Cucumber test
- Using data tables to perform setup

By the end of the chapter, you'll have a good idea of how a Cucumber test is built and run.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter17>

Integrating Cucumber and Puppeteer into your code base

Let's add the necessary packages to our project:

1. Start by installing the packages we're after. As well as Cucumber and Puppeteer, we'll also pull in `@babel/register`, which will enable us to use ES6 features within our support files:

```
$ npm install --save-dev @cucumber/cucumber puppeteer
$ npm install --save-dev @babel/register
```

2. Next, create a new file named `cucumber.json` with the following content. This has two settings; `publishQuiet` turns off a bunch of noise that would otherwise appear when you run tests, and `requireModule` hooks up `@babel/register` before tests are run:

```
{
  "default": {
    "publishQuiet": true,
    "requireModule": [
      "@babel/register"
    ]
  }
}
```

3. Create a new folder called `features`. This should live at the same level as `src` and `test`.
4. Create another folder within that called `features/support`.

You can now run tests with the following command:

```
$ npx cucumber-js
```

You'll see output like this:

```
0 scenarios
0 steps
0m00.000s
```

Throughout this chapter and the following one, it may be helpful to narrow down the tests you’re running. You can run a single scenario by providing the test runner with the filename and starting line number of the scenario:

```
$ npx cucumber-js features/drawing.feature:5
```

That’s all there is to getting set up with Cucumber and Puppeteer—now it’s time to write a test.

Writing your first Cucumber test

In this section, you’ll build a Cucumber feature file for a part of the Spec Logo application that we’ve already built.

Warning on Gherkin code samples

If you’re reading an electronic version of this book, be careful when copying and pasting feature definitions. You may find extra line breaks are inserted into your code that Cucumber won’t recognise. Before running your tests, please look through your pasted code snippets and remove any line breaks that shouldn’t be there.

Let’s get started!

1. Before running any Cucumber tests, it’s important to ensure that your build output is up to date by running `npm run build`. Your Cucumber specs are going to run against the code built in the `dist` directory, not your source in the `src` directory.

Use package.json scripts to your advantage

You could also modify your `package.json` scripts to invoke a build before Cucumber specs are run, or to run webpack in watch mode.

2. Create a new file named `features/sharing.feature` and enter the following text. A feature has a name and a short description, as well as a bunch of scenarios listed one after another. We’ll start with just one scenario for now:

```
Feature: Sharing
```

```
A user can choose to present their session to any
number of other users, who observe what the
presenter is doing via their own browser.
```

```
Scenario: Observer joins a session
  Given the presenter navigated to the application page
  And the presenter clicked the button 'startSharing'
  When the observer navigates to the presenter's
  sharing link
  Then the observer should see a message saying 'You
  are now watching the session'
```

Gherkin syntax

Given, **When**, and **Then** are analogous to the **Arrange**, **Act**, and **Assert** phases of your Jest tests: *given* all these things are true, *when* I perform this action, *then* I expect all these things to happen.

Ideally, you'd have a single **When** clause in each of your scenarios.

You'll notice that I've written the **Given** clauses in past tense and the **When** clause in the present tense, and the **Then** clause has a "should" in there.

3. Go ahead and run the feature by typing `npx cucumber-js` at the command line. You'll see a warning printed, as shown in the following code block. Cucumber has stopped processing at the first `Given...` statement because it can't find the JavaScript support function that maps to it. In the warning, Cucumber has helpfully given you a starting point for the definition:

```
? Given the presenter navigated to the application page
Undefined. Implement with the following snippet:
```

```
Given('the presenter navigated to the application
page', function () {
  // Write code here that turns the phrase above
  // into concrete actions
  return 'pending';
});
```

4. Let's do exactly what it suggested. Create the `features/support/sharing.steps.js` file and add the following code. It defines a step definition that calls Puppeteer's API to launch a new browser, then open a new page, and then navigate to the URL provided. The step definition description matches up with the **Given** clause in our test scenario.

5. The second parameter to **Given** is marked with the `async` keyword. This is an addition to what Cucumber tells us in its suggested function definition. We need `async` because Puppeteer's API calls all return promises that we'll need to `await`:

```
import {  
    Given, When, Then  
} from "@cucumber/cucumber";  
import puppeteer from "puppeteer";  
  
const port = process.env.PORT || 3000;  
const appPage = `http://localhost:${port}/index.html`;  
  
Given(  
    "the presenter navigated to the application page",  
    async function () {  
        const browser = await puppeteer.launch();  
        const page = await browser.newPage();  
        await page.goto(appPage);  
    }  
);
```

Anonymous functions, not lambda expressions

You may be wondering why we are defining anonymous functions (`async function (...) { ... }`) rather than lambda expressions (`async (...) => { ... }`). It allows us to take advantage of the implicit context binding that occurs with anonymous functions. If we used lambdas, we'd need to call `.bind(this)` on them.

6. Run your tests again. Cucumber now dictates the next clause that needs work. For this clause, `And the presenter clicked the button 'startSharing'`, we need to get access to the `page` object we just created in the previous step. The way to do this is by accessing what's known as the `World` object, which is the context for all the clauses in the current scenario. We must build this now. Create the `features/support/world.js` file and add the following content. It defines two methods, `setPage` and `getPage`, which allow us to save multiple pages within the world. The ability to save multiple pages is important for this test, where we have at least two pages—the presenter page and the observer page:

```
import {  
    setWorldConstructor  
}
```

```
    } from "@cucumber/cucumber";  
  
class World {  
  constructor() {  
    this.pages = {};  
  }  
  
  setPage(name, page) {  
    this.pages[name] = page;  
  }  
  
  getPage(name) {  
    return this.pages[name];  
  }  
};  
  
setWorldConstructor(World);
```

7. We can now use the `setPage` and `getPage` functions from within our step definitions. Our approach will be to call `setPage` from the first step definition—the one we wrote in *step 3*—and then use `getPage` to retrieve it in subsequent steps. Modify the first step definition now to include the call to `setPage`, as shown in the following code block:

```
Given(  
  "the presenter navigated to the application page",  
  async function () {  
    const browser = await puppeteer.launch();  
    const page = await browser.newPage();  
    await page.goto(appPage);  
    this.setPage("presenter", page);  
  }  
);
```

8. Moving on to the next step, the presenter clicked the button 'startSharing', we'll solve this by using the `Page.click` Puppeteer function to find a button with an ID of `startSharing`. As in the last test, we use a `buttonId` parameter so that this step definition can be used with other buttons in future scenarios:

```
Given(  
  "the presenter clicked the button {string}",
```

```
    async function (buttonId) {
      await this.getPage(
        "presenter"
      ).click(`button#${buttonId}`);
    }
  );
}
```

9. The next step, the observer navigates to the presenter's sharing link, is like the first step in that we want to open a new browser. The difference is that it's for the observer, and we first need to look up the path to follow. The path is given to us through the URL that the presenter is shown once they start searching. We can look that up using the `Page.$eval` function:

```
When(
  "the observer navigates to the presenter's sharing
link",
  async function () {
    await this.getPage(
      "presenter"
    ).waitForSelector("a");
    const link = await this.getPage(
      "presenter"
    ).$eval("a", a => a.getAttribute("href"));
    const url = new URL(link);
    const browser = await puppeteer.launch();
    const page = await browser.newPage();
    await page.goto(url);
    this.setPage("observer", page);
  }
);
```

Step definition duplication

There's some duplication building up between our step definitions. Later on, we'll extract this commonality into its own function.

10. The final step definition uses the `Page.$eval` Puppeteer function again, this time to find an HTML node and then transform it into a plain JavaScript object. We then test that object using the `expect` function in the normal way. Make sure to place the listed `import` statement at the top of your file:

```
import expect from "expect";  
  
...  
  
Then(  
  "the observer should see a message saying {string}",  
  async function (message) {  
    const pageText = await this.getPage(  
      "observer"  
    ) .$eval("body", e => e.outerHTML);  
    expect(pageText).toContain(message);  
  }  
) ;
```

11. Run your tests with `npx cucumber-js`. The output from your test run will look as follows. While our step definitions are complete, something is amiss:

```
1) Scenario: Observer joins a session  
  ✘ Given the presenter navigated to the application  
  page  
    Error: net::ERR_CONNECTION_REFUSED at http://  
    localhost:3000/index.html
```

12. Although our app has loaded, we still need to spin up the server to process our requests. To do that, add the following two functions to the `World` class in `features/support/world.js`, including the `import` statement for the `app` at the top of the file. The `startServer` function is equivalent to how we start the server in `server/src/server.js`. The `closeServer` function stops the server, but before it does this, it closes all Puppeteer browser instances. It's important to do this before closing the server. That's because the server does not kill any live websocket connections when the `close` method is called. We need to ensure they are closed first; otherwise, the server won't stop:

Starting a server from within the same project

We are lucky that all our code lives within the same project, so it can be started within the same process. If your code base is split over multiple projects, you may find yourself dealing with multiple processes.

```
import { app } from "../../server/src/app";

class World {
    ...
    startServer() {
        const port = process.env.PORT || 3000;
        this.server = app.listen(port);
    }

    closeServer() {
        Object.keys(this.pages).forEach(name =>
            this.pages[name].browser().close()
        );
        this.server.close();
    }
}
```

13. Make use of these new functions with the `Before` and `After` hooks. Create a new file, `features/support/hooks.js`, and add the following code:

```
import { Before, After } from "@cucumber/cucumber";

Before(function() {
    this.startServer();
});

After(function() {
    this.closeServer();
});
```

14. Run the `npx cucumber-js` command and observe the output. Your scenario should now be passing (if it isn't, double-check you've run `npm run build`):

```
> npx cucumber-js
.....
```

```
1 scenario (1 passed)
4 steps (4 passed)
0m00.848s
```

15. Let's go back and tidy up that repeated code. We'll extract a function called `browseToPageFor` and place it within our `World` class. Open `features/support/world.js` and add the following method at the bottom of the class:

```
async browseToPageFor(role, url) {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto(url);
  this.setPage(role, page);
}
```

16. Also, move the Puppeteer import statement across from `features/support/sharing.steps.js` into `features/support/world.js`:

```
import puppeteer from "puppeteer";
```

17. Finally, rewrite the two navigation steps in terms of `browseToPageFor`:

```
Given(
  "the presenter navigated to the application page",
  async function () {
    await this.browseToPageFor("presenter", appPage);
  }
);

When(
  "the observer navigates to the presenter's sharing
link",
  async function () {
    await this.getPage(
      "presenter"
    ).waitForSelector("a");
    const link = await this.getPage(
      "presenter"
    ).$eval("a", a => a.getAttribute("href"));
    const url = new URL(link);
```

```

        await this.browserToPageFor("observer", url);
    }
);

```

Observing within a browser and with console logging

The tests we've written run Puppeteer in headless mode, meaning that an actual Chrome browser window doesn't launch. If you'd like to see that happen, you can turn headless mode off by modifying the launch commands (remember there are two in the previous step definitions) to read as follows:

```

const browser = await puppeteer.launch(
  { headless: false }
);

```

If you're using console logging to assist in your debugging, you'll need to provide another parameter to dump console output to `stdout`:

```

const browser = await puppeteer.launch(
  { dumpio: true }
);

```

You've now written a BDD test with Cucumber and Puppeteer. Next, let's look at a more advanced Cucumber scenario.

Using data tables to perform setup

In this section, we'll look at a useful time-saving feature of Cucumber: data tables. We'll write a second scenario that, as with the previous one, will already pass given the existing implementation of Spec Logo:

1. Create a new feature file called `features/drawing.feature` with the following content. It contains a set of instructions to draw a square using a Logo function. A small side length of 10 is used; that's to make sure the animation finishes quickly:

```
Feature: Drawing
```

```
A user can draw shapes by entering commands
at the prompt.
```

```
Scenario: Drawing functions
```

```
  Given the user navigated to the application page
  When the user enters the following instructions at
  the prompt:
```

```

| to drawsquare |
|   repeat 4 [ forward 10 right 90 ] |
| end |
| drawsquare |

Then these lines should have been drawn:
| x1 | y1 | x2 | y2 |
| 0 | 0 | 10 | 0 |
| 10 | 0 | 10 | 10 |
| 10 | 10 | 0 | 10 |
| 0 | 10 | 0 | 0 |

```

- The first phrase does the same thing as our previous step definition, except we've renamed `presenter` to `user`. Being more generic makes sense in this case as the role of the presenter is no longer relevant to this test. We can use the `World` function `browseToPageFor` for this first step. In the sharing feature, we used this function together with an `appPage` constant that contained the URL to navigate to. Let's pull that constant into `World` now. In `features/support/world.js`, add the following constant at the top of the file, above the `World` class:

```
const port = process.env.PORT || 3000;
```

- Add the following method to the `World` class:

```
appPage() {
  return `http://localhost:${port}/index.html`;
}
```

- In `features/support/sharing.steps.js`, remove the definitions for `port` and `appPage` and update the first step definition, as shown:

```
Given(
  "the presenter navigated to the application page",
  async function () {
    await this.browseToPageFor(
      "presenter",
      this.appPage()
    );
  }
);
```

5. It's time to create a new step definition for a user page. Open the `features/support/drawing.steps.js` file and add the following code:

```
import {
  Given,
  When,
  Then
} from "@cucumber/cucumber";
import expect from "expect";

Given("the user navigated to the application page",
  async function () {
    await this.browserToPageFor(
      "user",
      this.appPage()
    );
  }
);
```

6. Now, what about the second line, with the data table? What should our step definition look like? Well, let's ask Cucumber. Run the `npx cucumber -js` command and have a look at the output. It gives us the starting point of our definition:

```
1) Scenario: Drawing functions
  ✓ Before # features/support/sharing.steps.js:5
  ✓ Given the user navigated to the application page
    ? When the user enters the following instructions at
      the prompt:
        | to drawsquare |
        |   repeat 4 [ forward 10 right 90 ] |
        | end |
        | drawsquare |
    Undefined. Implement with the following snippet:
```

```
When('the user enters the following instructions at the
prompt:',

  function (dataTable) {
    // Write code here that turns the phrase above
```

```
// into concrete actions
return 'pending';
}
);
```

7. Go ahead and add the suggested code to `features/supports/drawing.steps.js`. If you run `npx cucumber -js` at this point, you'll notice that Cucumber successfully notices that the step definition is pending:

```
When(
  "the user enters the following instructions at the
  prompt:",
  function (dataTable) {
    // Write code here that turns the phrase above
    //into concrete actions
    return "pending";
  }
);
```

8. The `dataTable` variable is a `DataTable` object with a `raw()` function that returns an array of arrays. The outer array represents each row, and the inner arrays represent the columns of each row. In the next step definition, we want to take every single line and insert it into the edit prompt. Each line should be followed by a press of the *Enter* key. Create that now:

```
When(
  "the user enters the following instructions at the
  prompt:",
  async function (dataTable) {
    for (let instruction of dataTable.raw()) {
      await this.getPage("user").type(
        "textarea",
        `${instruction}\n`
      );
    }
  }
);
```

9. The final step requires us to look for line elements with the right attribute values and compare them to the values in our second data table. The following code does exactly that. Copy it out now and run your tests to ensure that it works and that the test will pass. An explanation of all the detailed points will follow:

```
Then("these lines should have been drawn:",  
    async function(dataTable) {  
        await this.getPage("user").waitForTimeout(2000);  
        const lines = await this.getPage("user").$$eval(  
            "line",  
            lines => lines.map(line => {  
                return {  
                    x1: parseFloat(line.getAttribute("x1")),  
                    y1: parseFloat(line.getAttribute("y1")),  
                    x2: parseFloat(line.getAttribute("x2")),  
                    y2: parseFloat(line.getAttribute("y2"))  
                };  
            })  
        );  
        for (let i = 0; i < lines.length; ++i) {  
            expect(lines[i].x1).toBeCloseTo(  
                parseInt(dataTable.hashes()[i].x1)  
            );  
            expect(lines[i].y1).toBeCloseTo(  
                parseInt(dataTable.hashes()[i].y1)  
            );  
            expect(lines[i].x2).toBeCloseTo(  
                parseInt(dataTable.hashes()[i].x2)  
            );  
            expect(lines[i].y2).toBeCloseTo(  
                parseInt(dataTable.hashes()[i].y2)  
            );  
        }  
    }  
});
```

That last test contained some complexity that's worth diving into:

- We used `Page.waitForTimeout` to wait for 2 seconds, which gives the system time to complete animations. Including a timeout like this is not a great practice, but it'll work for now. We'll look at a way of making this more specific in the next chapter.
- The `Page.$$eval` function is like `Page.eval` but returns an array under the hood, and calls `document.querySelector` rather than `document.querySelectorAll`.
- It's important that we do all of the attribute transformation logic—moving from HTML line elements and attributes to “plain” integer values of `x1`, `y1`, and so on—withn the `transform` function of `Page.$$eval`. This is because Puppeteer will garbage collect any DOM node objects once the `$$eval` call is done.
- Our line values need to be parsed with `parseFloat` because the `requestAnimationFrame` logic we coded doesn't perfectly line up with the integer endpoints—they are out by very slight fractional amounts.
- That also means we need to use the `toBeCloseTo` Jest matcher rather than `toBe`, which we need because of the fractional value difference described previously.
- Finally, we use the `DataTable hashes()` function here to pull out an array of objects that has a key for each of the columns in the data table, based on the header row that we provided in the feature definition. So, for example, we can call `hashes()[0].x1` to pull out the value in the `x1` column for the first row.

Go ahead and run your tests again with `npx cucumber-jsc`. Everything should be passing.

You've now got a good understanding of using Cucumber data tables to make more compelling BDD tests.

Summary

Cucumber tests (and BDD tests in general) are similar to the unit tests we've been writing in the rest of the book. They are focused on specifying *examples* of behavior. They should make use of real data and numbers as means to test a general concept, like we've done in the two examples in this chapter.

BDD tests differ from unit tests in that they are system tests (having a much broader test surface area) and they are written in natural language.

Just as with unit tests, it's important to find ways to simplify the code when writing BDD tests. The number one rule is to try to write generic **Given**, **When**, and **Then** phrases that can be reused across classes and extracted out of step definition files, either into the `World` class or some other module. We've seen an example of how to do that in this chapter.

In the next chapter, we'll use a BDD test to drive the implementation of a new feature in Spec Logo.

18

Adding Features Guided by Cucumber Tests

In the last chapter, we studied the basic elements of writing Cucumber tests and how to use Puppeteer to manipulate our UI. But we haven't yet explored how these techniques fit into the wider development process. In this chapter, we'll implement a new application feature, but starting the process with Cucumber tests. These will act as acceptance tests that our (imaginary) product owner can use to determine whether the software works as required.

Acceptance testing

An **acceptance test** is a test that a product owner or customer can use to decide whether they accept the delivered software. If it passes, they accept the software. If it fails, the developers must go back and adjust their work.

We can use the term **Acceptance-Test-Driven Development (ATDD)** to refer to a testing workflow that the whole team can participate in. Think of it as like TDD but it is done at the wider team level, with the product owner and customer involved in the cycle. Writing BDD tests using Cucumber is one way—but not the only way—that you can bring ATDD to your team.

In this chapter, we'll use our BDD-style Cucumber tests to act as our acceptance tests.

Imagine that our product owner has seen the great work that we've done building **Spec Logo**. They have noted that the share screen functionality is good, but it could do with an addition: it should give the presenter the option of resetting their state before sharing begins, as shown:

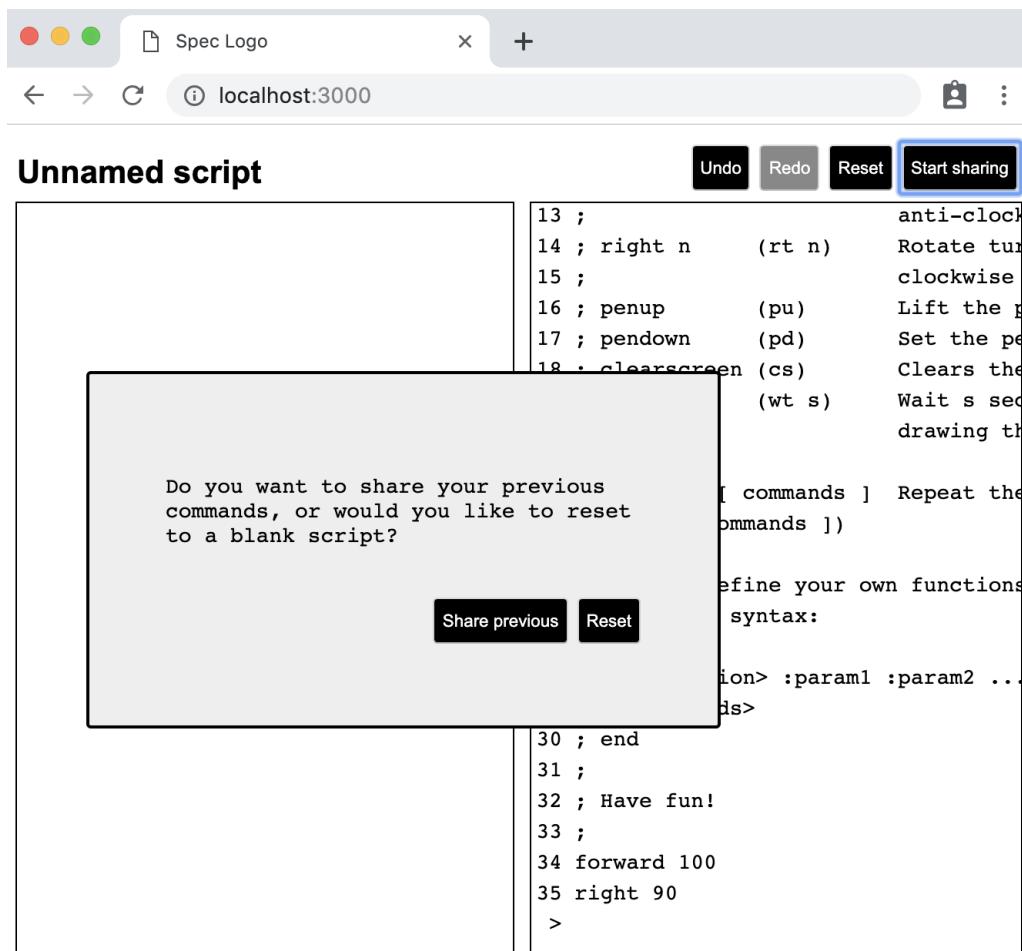


Figure 18.1 – The new sharing dialog

The product owner has provided us with some Cucumber tests that are currently red for implementation—both the step definitions and the production code.

This chapter covers the following topics:

- Adding Cucumber tests for a dialog box
- Fixing Cucumber tests by test-driving production code
- Avoiding timeouts in test code

By the end of the chapter, you'll have seen more examples of Cucumber tests and how they can be used as part of your team's workflow. You'll also have seen how to avoid using specific timeouts within your code.

Technical requirements

The code files for this chapter can be found here:

<https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-Second-Edition/tree/main/Chapter18>

Adding Cucumber tests for a dialog box

In this section, we'll add a new Cucumber test that won't yet pass.

Let's start by taking a look at the new feature:

1. Open the `features/sharing.feature` file and take a look at the first feature that you've been given. Read through the steps and try to understand what our product owner is describing. The test covers quite a lot of behavior—unlike our unit tests. It tells a complete story:

```
Scenario: Presenter chooses to reset current state when sharing
  Given the presenter navigated to the application page
  And the presenter entered the following instructions at the prompt:
    | forward 10 |
    | right 90 |
  And the presenter clicked the button 'startSharing'
  When the presenter clicks the button 'reset'
  And the observer navigates to the presenter's sharing link
    Then the observer should see no lines
    And the presenter should see no lines
    And the observer should see the turtle at x = 0, y = 0,
      angle = 0
    And the presenter should see the turtle at x = 0, y =
      0, angle = 0
```

2. The first **Given** phrase, the presenter navigated to the application page, already works, and you can verify this if you run `npx cucumber-js`.
3. The next step, the presenter entered the following instructions at the prompt, is very similar to a step from the last chapter. We could choose to extract the commonality here, just as we did with the `browseToPageFor` function; however, we'll wait until our test and implementation are complete before refactoring. For now, we'll

just duplicate the code. Open `features/support/sharing.steps.js` and add the following step definition at the bottom of the code:

```
When(
  "the presenter entered the following instructions at
  the prompt:",
  async function(dataTable) {
    for (let instruction of dataTable.raw()) {
      await this.getPage("presenter").type(
        "textarea",
        `${instruction}\n`
      );
      await this.getPage(
        "presenter"
      ).waitForTimeout(3500);
    }
  }
);
```

4. Next up is a **Given** clause that we have already: `the presenter clicked the button 'startSharing'`. The line that appears after this is the first **When** clause, which will need to be implemented. Run `npx cucumber-js` and you'll be given template code for this function. Copy and paste the template code into your step definition file, as shown in the following code block:

```
When(
  "the presenter clicks the button {string}",
  function (string) {
    // Write code here that turns the phrase above
    // into concrete actions
    return "pending";
  }
);
```

Two When phrases

This scenario has *two* **When** phrases, which is unusual. Just as with your unit tests in the **Act** phase, you generally want just one **When** phrase. However, since there are two users working together at this point, it makes sense to have a single action for both of them, so we'll let our product owner off the hook on this occasion.

5. This step definition is very similar to the ones we've written before. Fill out the function as shown in the following code block. There is a new call to `waitForSelector`. This waits for the button to appear on the page before we continue, which gives the dialog time to render:

```
When(  
  "the presenter clicks the button {string}",  
  async function (  
    buttonId  
  ) {  
    await this.getPage(  
      "presenter"  
    ).waitForSelector(`button#${buttonId}`);  
    await this.getPage(  
      "presenter"  
    ).click(`button#${buttonId}`);  
  }  
) ;
```

6. The second **When** clause already has a definition from our previous test, so we move on to the **Then** clauses. The first is `the observer should see no lines`; run `npx cucumber-js` and copy and paste the template function that Cucumber provides, as shown in the following code block:

```
Then("the observer should see no lines", function () {  
  // Write code here that turns the phrase above  
  // into concrete actions  
  return "pending";  
) ;
```

7. For this step, we want to assert that there are no line elements on the page:

```
Then(  
  "the observer should see no lines",  
  async function () {  
    const numLines = await this.getPage(  
      "observer"  
    ).$$eval("line", lines => lines.length);  
    expect(numLines).toEqual(0);  
  }  
) ;
```

8. Running `npx cucumber-js`, you should see that this step passes, and the next one is very similar. Copy the step definition you just wrote and modify it to work for the presenter, as shown in the following code block. Again, we can clean up the duplication later:

```
Then(  
  "the presenter should see no lines",  
  async function () {  
    const numLines = await this.getPage()  
    "presenter"  
    ).$$_eval("line", lines => lines.length);  
    expect(numLines).toEqual(0);  
  }  
);
```

9. Run Cucumber now, and you'll see that this step fails; this is the first failure that we've got. It points to the specific change that we'll need to make in the code base:

✗ And the presenter should see no lines
Error: `expect(received).toEqual(expected)`

Expected value to equal:
0
Received:
1

10. Since we have hit a red step, we could now go back and start working on our code to make this green. However, because we just have two almost identical clauses to complete, I'm going to choose to complete these definitions before continuing. Cucumber tells us the template function that we should use, so add that now, as follows:

```
Then(  
  "the observer should see the turtle at x = {int}, y =  
  {int}, angle = {int}",  
  function (int, int2, int3) {  
    // Write code here that turns the phrase above  
    // into concrete actions  
    return "pending";  
});
```

11. We need to define a couple of helpers that can tell us the current x , y , and angle values of the turtle. We need this because all we have is the SVG polygon element, which uses a `points` string and a `transform` string to position the turtle. Our helpers will take these strings and convert them back to numbers for us. As a reminder, here's how the turtle is initially positioned:

```
<polygon  
  points="-5,5, 0,-7, 5,5"  
  fill="green"  
  stroke-width="2"  
  stroke="black"  
  transform="rotate(90, 0, 0)" />
```

We can use the first `points` coordinate to calculate x and y , by adding 5 to the first number and subtracting 5 from the second. The angle can be calculated by taking the first parameter to rotate and subtracting 90. Create a new file named `features/support/turtle.js`, and then add the following two definitions:

```
export const calculateTurtleXYFromPoints = points => {  
  const firstComma = points.indexOf(",");  
  const secondComma = points.indexOf(  
    ",",  
    firstComma + 1  
  );  
  return {  
    x:  
      parseFloat(  
        points.substring(0, firstComma)  
      ) + 5,  
    y:  
      parseFloat(  
        points.substring(firstComma + 1, secondComma)  
      ) - 5  
  };  
  
export const calculateTurtleAngleFromTransform = (  
  transform  
) => {  
  const firstParen = transform.indexOf("(");
```

```
const firstComma = transform.indexOf(",") ;
return (
  parseFloat(
    transform.substring(
      firstParen + 1,
      firstComma
    )
  ) - 90
);
}
```

12. In `feature/sharing.steps.js`, update the step definition, as shown in the following code block:

```
Then(
  "the observer should see the turtle at x = {int}, y =
{int}, angle = {int}",
  async function (
    expectedX, expectedY, expectedAngle
  ) {
    await this.getPage(
      "observer"
    ).waitForTimeout(4000);
    const turtle = await this.getPage(
      "observer"
    ).$eval(
      "polygon",
      polygon => ({
        points: polygon.getAttribute("points"),
        transform: polygon.getAttribute("transform")
      })
    );
    const position = calculateTurtleXYFromPoints(
      turtle.points
    );
    const angle = calculateTurtleAngleFromTransform(
      turtle.transform
    );
  }
);
```

```
    expect(position.x).toBeCloseTo(expectedX);
    expect(position.y).toBeCloseTo(expectedY);
    expect(angle).toBeCloseTo(expectedAngle);
  }
);
```

13. Finally, repeat this step definition for the presenter, as follows:

```
Then(
  "the presenter should see the turtle at x = {int}, y =
  {int}, angle = {int}",
  async function (
    expectedX, expectedY, expectedAngle
  ) {
    await this.getPage(
      "presenter"
    ).waitForTimeout(4000);
    const turtle = await this.getPage(
      "presenter"
    ).$eval(
      "polygon",
      polygon => ({
        points: polygon.getAttribute("points"),
        transform: polygon.getAttribute("transform")
      })
    );
    const position = calculateTurtleXYFromPoints(
      turtle.points
    );
    const angle = calculateTurtleAngleFromTransform(
      turtle.transform
    );
    expect(position.x).toBeCloseTo(expectedX);
    expect(position.y).toBeCloseTo(expectedY);
    expect(angle).toBeCloseTo(expectedAngle);
  }
);
```

That's the first test; now, let's move on to the second scenario:

1. Nearly all of the step definitions for our second scenario are already implemented; there are only two that aren't:

Then these lines should have been drawn for the observer:

x1	y1	x2	y2
0	0	10	0

And these lines should have been drawn for the presenter:

x1	y1	x2	y2
0	0	10	0

We already have a step definition that is very similar to these two in `features/support/drawing.steps.js`. Let's extract that logic into its own module so that we can reuse it. Create a new file named `features/support/svg.js` and then duplicate the following code from the drawing step definitions:

```
import expect from "expect";

export const checkLinesFromDataTable = page =>
  return async function (dataTable) {
    await this.getPage(page).waitForTimeout(2000);
    const lines = await this.getPage(page).$$eval(
      "line",
      lines =>
        lines.map(line => ({
          x1: parseFloat(line.getAttribute("x1")),
          y1: parseFloat(line.getAttribute("y1")),
          x2: parseFloat(line.getAttribute("x2")),
          y2: parseFloat(line.getAttribute("y2"))
        }))
    );
    for (let i = 0; i < lines.length; ++i) {
      expect(lines[i].x1).toBeCloseTo(
        parseInt(dataTable.hashes() [i].x1)
      );
      expect(lines[i].y1).toBeCloseTo(
        parseInt(dataTable.hashes() [i].y1)
      );
    }
  };
}
```

```
) ;  
expect(lines[i].x2).toBeCloseTo(  
    parseInt(dataTable.hashes()[i].x2)  
) ;  
expect(lines[i].y2).toBeCloseTo(  
    parseInt(dataTable.hashes()[i].y2)  
) ;  
}  
};
```

2. In `features/support/drawing.steps.js`, modify the these lines should have been drawn step definition so that it now uses this function:

```
import { checkLinesFromDataTable } from "./svg";  
  
Then(  
    "these lines should have been drawn:",  
    checkLinesFromDataTable("user")  
) ;
```

3. The two new step definitions for our latest sharing scenario are now straightforward. In `features/support/sharing.steps.js`, add the following import statement and step definitions:

```
import { checkLinesFromDataTable } from "./svg";  
  
Then(  
    "these lines should have been drawn for the  
    presenter:",  
    checkLinesFromDataTable("presenter")  
) ;  
Then(  
    "these lines should have been drawn for the observer:",  
    checkLinesFromDataTable("observer")  
) ;
```

You've now seen how to write longer step definitions and how to extract common functionality into support functions.

With the step definitions complete, it's time to make both these scenarios pass.

Fixing Cucumber tests by test-driving production code

In this section, we'll start by doing a little up-front design, then we'll write unit tests that cover the same functionality as the Cucumber tests, and then use those to build out the new implementation.

Let's do a little up-front design:

- When the user clicks on **Start sharing**, a dialog should appear with a **Reset** button.
- If the user chooses to reset, the Redux store is sent a `START_SHARING` action with a new `reset` property that is set to `true`:

```
{ type: "START_SHARING", reset: true }
```

- If the user chooses to share their existing commands, then the `START_SHARING` action is sent with `reset` set to `false`:

```
{ type: "START_SHARING", reset: false }
```

- When the user clicks on **Reset**, a `RESET` action should be sent to the Redux store.
- Sharing should not be initiated until *after* the `RESET` action has occurred.

That's all the up-front design we need. Let's move on to integrating the `Dialog` component.

Adding a dialog box

Now that we know what we're building, let's go for it! To do so, perform these steps:

1. Open `test/MenuButtons.test.js` and skip the test that is titled `dispatches an action of START_SHARING when start sharing is clicked`. We're going to sever this connection for the moment. But we'll come back to fix this later:

```
it.skip("dispatches an action of START_SHARING when start
sharing is clicked", () => {
  ...
});
```

2. In the same file, add a new `import` statement for the `Dialog` component, and mock it out using `jest.mock`. The `Dialog` component already exists in the code base but has remained unused until now:

```
import { Dialog } from "../src/Dialog";
jest.mock("../src/Dialog", () => ({
  Dialog: jest.fn(() => <div id="Dialog" />),
});
```

3. Add this new test just below the one you've skipped. Very simply, it checks that we display the dialog when the appropriate button is clicked:

```
it("opens a dialog when start sharing is clicked", () =>
{
  renderWithStore(<MenuButtons />);
  click(buttonWithLabel("Start sharing"));
  expect(Dialog).toBeCalled();
}) ;
```

4. In `src/MenuButtons.js`, add a new `Dialog` element to the JSX, including the `import` statement at the top of the file. The new component should be placed at the very bottom of the returned JSX. The test should then pass:

```
import { Dialog } from "./Dialog";

export const MenuButtons = () => {
  ...
  return (
    <>
    ...
    <Dialog />
    </>
  ) ;
};
```

5. Next, let's set the `message` prop to something useful for the user. Add this to your test suite:

```
it("prints a useful message in the sharing dialog", () =>
{
  renderWithStore(<MenuButtons />);
  click(buttonWithLabel("Start sharing"));
  expect(propsOf(Dialog).message).toEqual(
    "Do you want to share your previous commands, or
would you like to reset to a blank script?"
  );
}) ;
```

6. To make that pass, add the message prop to your implementation:

```
<Dialog  
  message="Do you want to share your previous commands,  
  or would you like to reset to a blank script?"  
/>
```

7. Now, we need to make sure the dialog isn't shown until the sharing button is clicked; add the following test:

```
it("does not initially show the dialog", () => {  
  renderWithStore(<MenuButtons />);  
  expect(Dialog).not.toBeCalled();  
});
```

8. Make this pass by adding a new state variable, `isSharingDialogOpen`. The sharing button will set this to `true` when it's clicked. You'll need to add the `import` statement for `useState` at the top of the file:

```
import React, { useState } from "react";  
  
export const MenuButtons = () => {  
  const [  
    isSharingDialogOpen, setIsSharingDialogOpen  
  ] = useState(false);  
  
  const openSharingDialog = () =>  
    setIsSharingDialogOpen(true);  
  
  ...  
  
  return (  
    <>  
    ...  
    {environment.isSharing ? (  
      <button  
        id="stopSharing"  
        onClick={() => dispatch(stopSharing())}  
      >
```

```

        Stop sharing
    </button>
) : (
    <button
        id="startSharing"
        onClick={openSharingDialog}
    >
        Start sharing
    </button>
) }
{isSharingDialogOpen ? (
    <Dialog
        message="..."
    />
) : null}
</>
);
};

```

- Now, let's add a test for adding buttons to the dialog. This is done by specifying the `buttons` prop on the `Dialog` component:

```

it("passes Share and Reset buttons to the dialog", () =>
{
    renderWithStore(<MenuButtons />);
    click(buttonWithLabel("Start sharing"));
    expect(propsOf(Dialog).buttons).toEqual([
        { id: "keep", text: "Share previous" },
        { id: "reset", text: "Reset" }
    ]);
});

```

- Make this pass by adding a `buttons` prop to the `Dialog` component, as follows:

```

{isSharingDialogOpen ? (
    <Dialog
        message="..."
        buttons={[
            { id: "keep", text: "Share previous" },

```

```
        { id: "reset", text: "Reset" }
    ]
  />
) : null}
```

11. For the next test, we'll test that the dialog closes. Start by defining a new `closeDialog` helper in your test suite:

```
const closeDialog = () =>
  act(() => propsOf(Dialog).onClose());
```

12. Add the next test, which checks that the `Dialog` component disappears once the dialog has had its `onClose` prop invoked:

```
it("closes the dialog when the onClose prop is called",
() => {
  renderWithStore(<MenuButtons />);
  click(buttonWithLabel("Start sharing"));
  closeDialog();
  expect(element("#dialog")).toBeNull();
});
```

13. Make that pass by adding the following line to the `Dialog` JSX:

```
<Dialog
  onClose={() => setIsSharingDialogOpen(false)}
  ...
/>
```

14. Now go back to the test that you skipped and modify it so that it reads the same as the following code block. We're going to modify the `START_SHARING` Redux action to take a new `reset` Boolean variable:

```
const makeDialogChoice = button =>
  act(() => propsOf(Dialog).onChoose(button));

it("dispatches an action of START_SHARING when dialog
onChoose prop is invoked with reset", () => {
  renderWithStore(<MenuButtons />);
  click(buttonWithLabel("Start sharing"));
```

```
makeDialogChoice("reset");

return expectRedux(store)
  .toDispatchAnAction()
  .matching({ type: "START_SHARING", reset: true });
}) ;
```

15. To make this pass, move to `src/MenuButtons.js` and modify the `startSharing` function to add a `reset` property to the created Redux action. Notice how we hardcode the value to `true` for now—we'll need to **triangulate** in the upcoming test:

```
const startSharing = () => ({
  type: "START_SHARING",
  reset: true,
}) ;
```

Triangulation within tests

See *Chapter 1, First Steps with Test-Driven Development*, for a reminder on triangulation and why we do it.

16. In the `MenuButtons` component, set the `onChoose` prop on the `Dialog` component:

```
return (
  <>
  ...
  {isSharingDialogOpen ? (
    <Dialog
      onClose={() => setIsSharingDialogOpen(false)}
      onChoose={() => dispatch(startSharing())}
      ...
    />
  ) : null}
  </>
) ;
```

17. Finally, we need to add a new test for sending a value of `false` through for the `reset` action property:

```
it("dispatches an action of START_SHARING when dialog
onChoose prop is invoked with share", () => {
```

```

    renderWithStore(<MenuButtons />);
    click(buttonWithLabel("Start sharing"));

    makeDialogChoice("share");

    return expectRedux(store)
        .toDispatchAnAction()
        .matching({
            type: "START_SHARING",
            reset: false
        });
    });
}
);

```

18. To make that pass, modify `startSharing` to take a `button` parameter and then use that to set the `reset` property:

```

const startSharing = (button) => ({
    type: "START_SHARING",
    reset: button === "reset",
})
;
```

19. Then, finally, in the `MenuButtons` component JSX, set the `onChoose` prop on the `Dialog` element:

```
onChoose={(button) => dispatch(startSharing(button))}
```

You've now completed the first new piece of functionality specified in the Cucumber test. There's a dialog box being displayed and a `reset` Boolean flag being sent through to the Redux store. We are inching toward a working solution.

Updating sagas to a reset or replay state

Now, we need to update the sharing saga to handle the new `reset` flag:

1. Open `test/middleware/sharingSagas.test.js` and add the following test to the end of the `START_SHARING` nested describe block:

```

it("puts an action of RESET if reset is true", async () => {
    store.dispatch({
        type: "START_SHARING",

```

```
        reset: true,
    });
    await notifySocketOpened();
    await sendSocketMessage({
        type: "UNKNOWN",
        id: 123,
    });
    return expectRedux(store)
        .toDispatchAnAction()
        .matching({ type: "RESET" });
}) ;
```

2. In `src/middleware/sharingSagas.js`, modify `startSharing` so that it reads the same as the following code block. Don't forget to add the new `action` parameter to the top line:

```
function* startSharing(action) {
    ...
    if (action.reset) {
        yield put({ type: "RESET" });
    }
}
```

3. Now for the tricky second test. If `reset` is `false`, we want to replay all the current actions:

```
it("shares all existing actions if reset is false", async () => {
    const forward10 = {
        type: "SUBMIT_EDIT_LINE",
        text: "forward 10",
    };
    const right90 = {
        type: "SUBMIT_EDIT_LINE",
        text: "right 90"
    };
    store.dispatch(forward10);
    store.dispatch(right90);
    store.dispatch({
        type: "START_SHARING",
    });
    const sharedForward10 = {
        type: "SUBMIT_EDIT_LINE",
        text: "forward 10"
    };
    const sharedRight90 = {
        type: "SUBMIT_EDIT_LINE",
        text: "right 90"
    };
    expect(store.getState().sharing).toEqual([sharedForward10, sharedRight90]);
});
```

```

        reset: false,
    });
    await notifySocketOpened();
    await sendSocketMessage({
        type: "UNKNOWN",
        id: 123,
    });
    expect(sendSpy).toBeCalledWith(
        JSON.stringify({
            type: "NEW_ACTION",
            innerAction: forward10,
        })
    );
    expect(sendSpy).toBeCalledWith(
        JSON.stringify({
            type: "NEW_ACTION",
            innerAction: right90
        })
    );
}
);

```

4. To make this pass, we can use the `toInstructions` function from the `export` namespace. We also need to make use of two new `redux-saga` functions: `select` and `all`. The `select` function is used to retrieve the state and the `all` function is used with `yield` to ensure that we wait for all the passed calls to complete before continuing. Add those `import` statements now to `src/middleware/sharingSagas.js`:

```

import {
    call,
    put,
    takeLatest,
    take,
    all,
    select
} from "redux-saga/effects";

import { eventChannel, END } from "redux-saga";
import { toInstructions } from "../language/export";

```

- Now, modify the `startSharing` function by tacking on an `else` block to the conditional:

```
if (action.reset) {  
    yield put({ type: "RESET" });  
} else {  
    const state = yield select(state => state.script);  
    const instructions = toInstructions(state);  
    yield all(  
        instructions.map(instruction =>  
            call(shareNewAction, {  
                innerAction: {  
                    type: "SUBMIT_EDIT_LINE",  
                    text: instruction  
                }  
            })  
        )  
    );  
}
```

- If you run the tests now, you'll notice that there are a couple of unrelated failures. We can fix these by adding a default value for the `reset` property to the `startSharing` helper method in our tests:

```
const startSharing = async () => {  
    store.dispatch({  
        type: "START_SHARING",  
        reset: true  
    });  
    ...  
};
```

That completes the feature; both the unit tests and the Cucumber tests should be passing. Now would be a great time to try things out manually, too.

In the next section, we'll focus on reworking our Cucumber tests to make them run much faster.

Avoiding timeouts in test code

In this section, we'll improve the speed at which our Cucumber tests run by replacing `waitForTimeout` calls with `waitForSelector` calls.

Many of our step definitions contain waits that pause our test script interaction with the browser while we wait for the animations to finish. Here's an example from our tests, which waits for a period of 3 seconds:

```
await this.getPage("user").waitForTimeout(3000);
```

Not only will this timeout slow down the test suite, but this kind of wait is also brittle as there are likely to be occasions when the timeout is slightly too short and the animation hasn't finished. In this case, the test will intermittently fail. Conversely, the wait period is actually quite long. As more tests are added, the timeouts add up and the test runs suddenly take forever to run.

Avoiding timeouts

Regardless of the type of automated test, it is a good idea to avoid timeouts in your test code. Timeouts will substantially increase the time it takes to run your test suite. There are almost always methods you can use to avoid using them, such as the one highlighted in this section.

What we can do instead is modify our production code to notify us when it is animating, by setting an `isAnimating` class when the element is animating. We then use the Puppeteer `waitForSelector` function to check for a change in the value of this class, replacing `waitForTimeout` entirely.

Adding HTML classes to mark animation status

We do this by adding an `isAnimating` class to the `viewport` `div` element when an animation is running.

Let's start by adding the `isAnimating` class when the `Drawing` element is ready to animate a new Logo command:

1. In `test/Drawing.test.js`, add a new nested `describe` block within the main `Display` context, just below the context for resetting. Then, add the following test:

```
describe("isAnimating", () => {
  it("adds isAnimating class to viewport when animation begins", () => {
    renderWithStore(<Drawing />, {
      script: { drawCommands: [horizontalLine] }
    });
    triggerRequestAnimationFrame(0);
    expect(
      element("#viewport")
    ).toHaveClass("isAnimating");
  });
});
```

```
    }) ;
}) ;
```

2. In `src/Drawing.js`, update the JSX to include this class name on the `viewport` element:

```
return (
  <div
    id="viewport"
    className="isAnimating"
  >
  ...
</div>
);
```

3. Let's triangulate in order to get this state variable in place. To do this, add the following test:

```
it("initially does not have the isAnimating class set",
() => {
  renderWithStore(<Drawing />, {
    script: { drawCommands: [] }
  });
  expect(
    element("#viewport")
    ).not.toHaveClass("isAnimating");
});
```

4. To make this pass, update `className` to only set `isAnimating` if `commandToAnimate` is not null:

```
className={commandToAnimate ? "isAnimating" : ""}>
```

5. As a final flourish, we'll add an arguably unnecessary test. We want to be careful about removing the `isAnimating` class once the animation is finished. However, our implementation already takes care of this as `commandToAnimate` will be set to `undefined` when that happens. In other words, we don't need an explicit test for this, and we're done with this addition. However, for completeness' sake, you can add the test:

```
it("removes isAnimating class when animation is
finished", () => {
  renderWithStore(<Drawing />, {
    script: { drawCommands: [horizontalLine] },
    ...
```

```
    });
    triggerAnimationSequence([0, 500]);
    expect(element("#viewport")).not.toHaveClass(
      "isAnimating"
    );
}) ;
```

That completes adding the `isAnimating` class functionality. Now we can use this class as a means of replacing the `waitForTimeout` calls.

Updating step definitions to use `waitForSelector`

We're ready to use this new behavior in our step definitions, bringing in a new call to `waitForSelector` that waits until the `isAnimating` class appears (or disappears) on an element:

1. In `features/support/world.js`, add the following two methods to the `World` class. The first waits for the `isAnimating` selector to appear within the DOM and the second waits for it to disappear:

```
waitForAnimationToBegin(page) {
  return this.getPage(page).waitForSelector(
    ".isAnimating"
  );
}

waitForAnimationToEnd(page) {
  return this.getPage(page).waitForSelector(
    ".isAnimating",
    { hidden: true }
  );
}
```

2. In `features/support/drawing.steps.js`, search for the single `waitForTimeout` invocation in this file and replace it with the code in the following block:

```
When(
  "the user enters the following instructions at the
  prompt:",
  async function (dataTable) {
    for (let instruction of dataTable.raw()) {
```

```

        await this.getPage("user").type(
            "textarea",
            `${instruction}\n`
        );
        await this.waitForAnimationToEnd("user");
    }
}
);

```

Being careful about class transitions

We're waiting for animation after *each* instruction is entered. This is important as it mirrors how the `isAnimating` class will be added and removed from the application. If we only had one `waitForAnimationToEnd` function as the last instruction on the page, we may end up exiting the step definition early if the wait catches the removal of an `isAnimating` class in the *middle* of a sequence of instructions, rather than catching the *last* one.

- Now, open `features/support/sharing.steps.js`; this file has a similar step in it as the previous one, so update that one now, in the same way:

```

When(
    "the presenter entered the following instructions at
    the prompt:",
    async function(dataTable) {
        for (let instruction of dataTable.raw()) {
            await this.getPage("presenter").type(
                "textarea",
                `${instruction}\n`
            );
            await this.waitForAnimationToEnd("presenter");
        }
    }
);

```

- Toward the bottom of the file, update the two step definitions that check the turtle position:

```

Then(
    "the observer should see the turtle at x = {int}, y =
    {int}, angle = {int}",
    async function (

```

```
    expectedX, expectedY, expectedAngle
  ) {
  await this.waitForAnimationToEnd("observer");
  ...
}
);

Then(
  "the presenter should see the turtle at x = {int}, y =
  {int}, angle = {int}",
  async function (
    expectedX, expectedY, expectedAngle
  ) {
  await this.waitForAnimationToEnd("presenter");
  ...
}
);

```

5. Open `features/support/svg.js` and update the function within it, as follows:

```
export const checkLinesFromDataTable = page => {
  return async function (dataTable) {
    await this.waitForAnimationToEnd(page);
    ...
  }
};

```

6. If you run `npx cucumber-js` now, you'll see that we have one test failure, which is related to the output on the observer's screen. It indicates that we need to wait for the animations when we load the observer's page. In this case, we need to wait for the animation to start before we can wait for it to finish. We can fix this by adding a new step to the feature. Open `features/sharing.feature` and modify the last test to include a *third* entry in the **When** section:

```
When the presenter clicks the button 'keep'
And the observer navigates to the presenter's sharing
link
And the observer waits for animations to finish
```

Encapsulating multiple When clauses

If you aren't happy with having three **When** clauses, then you can always combine them into a single step.

7. Back in `features/support/sharing.steps.js`, add this new step definition just underneath the other **When** step definitions:

```
When(  
  "the observer waits for animations to finish",  
  async function () {  
    await this.waitForAnimationToBegin("observer");  
    await this.waitForAnimationToEnd("observer");  
  }  
);
```

Your tests should now be passing, and they should be much faster. On my machine, they now only take a quarter of the time than they did before.

Summary

In this chapter, we looked at how you can integrate Cucumber into your team's workflow.

You saw some more ways that Cucumber tests differ from unit tests. You also learned how to avoid using timeouts to keep your test suites speedy.

We're now finished with our exploration of the **Spec Logo** world.

In the final chapter of the book, we'll look at how TDD compares to other developer processes.

Exercise

Remove as much duplication as possible from your step definitions.

19

Understanding TDD in the Wider Testing Landscape

Besides the mechanics of test-driven development, this book has touched on a few ideas about the mindset of the TDD practitioner: how and when to “cheat,” systematic refactoring, *strict* TDD, and so on.

Some dev teams like to adopt the mantra of *move fast and break things*. TDD is the opposite: go slow and think about things. To understand what this means in practice, we can compare TDD with various other popular testing techniques.

The following topics will be covered in this chapter:

- Test-driven development as a testing technique
- Manual testing
- Automated testing
- Not testing at all

By the end of this chapter, you should have a good idea of why and how we practice TDD compared to other programming practices.

Test-driven development as a testing technique

TDD practitioners sometimes like to say that TDD is not about testing; rather, it’s about design, behavior, or specification, and the automated tests we have at the end are simply a bonus.

Yes, TDD is about design, but TDD is certainly about testing, too. TDD practitioners care that their software has a high level of *quality*, and this is the same thing that testers care about.

Sometimes, people question the naming of TDD because they feel that the notion of a “test” confuses the actual process. The reason for this is that developers misunderstand what it means to build a “test.” A typical unit testing tool offers you practically no guidance on how to write *good* tests. And it turns out that reframing tests as specifications and examples is a good way to introduce testing to developers.

All automated tests are hard to write. Sometimes, we forget to write important tests, or we build brittle tests, write loose expectations, over-complicate solutions, forget to refactor, and so on.

It’s not just novices who struggle with this – everyone does it, experts included. People make a mess all the time. That’s also part of the fun. Discovering the joy of TDD requires a certain degree of humility and accepting that you aren’t going to be writing pristine test suites most of the time. Pristine test suites are very rare indeed.

If you are lucky enough to have a tester on your team, you may think that TDD encroaches on their work, or may even put them out of a job. However, if you ask them their opinion, you’ll undoubtedly find they are only too keen for developers to take an interest in the quality of their work. With TDD, you can catch all those trivial logic errors yourself without needing to rely on someone else’s manual testing. The testers can then better use their time by focusing on testing complex use cases and hunting down missed requirements.

Best practices for your unit tests

The following are some great unit tests:

- **Independent:** Each test should test just one thing, and invoke only one unit. There are many techniques that we can employ to achieve this goal. To take just two examples, collaborators are often (but not always) mocked, and example data should be the minimum set of data required to correctly describe the test.

Classictist versus mockist TDD

You may have heard of the great TDD debate of *classictist* versus *mockist* TDD. The idea is that the classictist will not use mocks and stubs, while the mockist will mock all collaborators. In reality, both techniques are important. You have seen both in use in this book. I encourage you to not limit yourself to a single approach, but instead experiment and learn to be comfortable with both.

- **Short, with a high level of abstraction:** The test description should be concise. The test code should highlight all the pieces of code that are important to the test, and hide any apparatus that’s required but not relevant.
- **Quick to run:** Use test doubles instead of interacting with system resources (files, network connections, and so on) or other processes. Do not use timeouts in your code, or rely on the passing of time.

- **Focused on observable behavior:** The system is interesting for what it does to the outside world, not for how it does it. In the case of React, we care about DOM interaction.
- **In three parts:** These parts are **Arrange**, **Act**, and **Assert**, also known as the **AAA** pattern. Each test should follow this structure.
- **Don't Repeat Yourself (DRY):** Always take the time to refactor and clean up your tests, aiming for readability.
- **A design tool:** Great tests help you figure out how to design your system. That's not to say that up-front design isn't important. In almost every chapter in this book, we started with a little design before we embarked on our tests. It's important to do some thinking so that you have an idea of the general direction you're headed. Just don't try to plan too far ahead, and be prepared to throw out your design entirely as you proceed.

TDD is not a replacement for great design. To be a great TDD practitioner, you should also learn about and practice software design. There are many books about software design. Do not limit yourself to books about JavaScript or TypeScript; good design transcends language.

Improving your technique

The following are some general tips for improving:

- **Work with others:** Beyond reading this book, the best way to level up in TDD is to work with experts. Since TDD lends itself so well to pair and mob programming, it can give structure to teams of mixed experience. More experienced developers can use the granularity of small tests to help improve the work of less experienced developers.
- **Experiment with design:** TDD gives you a safety net that allows you to experiment with the style and shape of your programs. Make use of that safety net to learn more about design. Your tests will keep you safe.
- **Learn to slow down:** TDD requires a great deal of personal discipline. Unfortunately, there is no room for sloppiness. You must not cut corners; instead, take every opportunity to refactor. Once your test passes, sit with your code. Before moving on to the next test, stare at your current solution and think carefully about whether it is the best it can be.
- **Don't be afraid to defer design decisions:** Sometimes, we're faced with several design options, and it can be tricky to know which option to choose. Even a simple act such as naming variables can be difficult. Part of having a sense of design is knowing when to defer your thinking. If you're in the refactor stage and feel yourself weighing up two or more options, move on and add another test, and then revisit your design. You'll often find that you have more design knowledge and will be closer to the right answer.
- **Solve a kata each day:** A kata is a short exercise designed to be practiced repeatedly to teach you a certain technique. Two basic katas are *Coin Changer* and *Roman Numerals*. More complex

katas include the bowling game kata, the bank kata, and Conway's *Game of Life*. The diamond kata is a favorite of mine, as are sorting algorithms.

- **Attend a Coderetreat:** Coderetreats involve a day of pairing and TDD that revolves around the *Game of Life* kata. The *Global Day of Coderetreat* is held in November. Groups from all around the world get together to solve this problem. It's not only fun but a great way to expand your TDD horizons.

That covers the general advice on TDD. Next, let's look at manual testing techniques.

Manual testing

Manual testing, as you may have guessed, means starting your application and actually using it.

Since your software is your creative work, naturally, you are interested to find out how it performs. You should certainly take the time to do this but think of it as downtime and a chance to relax, rather than a formal part of your development process.

The downside to *using* your software as opposed to *developing* your software is that using it takes up a lot of time. It sounds silly but pointing, clicking, and typing all take up valuable time. Plus, it takes time to get test environments set up and primed with the relevant test data.

For this reason, it's important to avoid manual testing where possible. There are, however, times when it's necessary, as we'll discover in this section.

There is always a temptation to manually test the software after each feature is complete, just to verify that it works. If you find yourself doing this a lot, consider how much confidence you have in your unit tests.

If you can claim, "I have 100% confidence in my unit tests," why would you ever need to *use* your software to prove it?

Let's look at some specific types of manual testing, starting with demonstrating software.

Demonstrating software

There are at least two important occasions where you should *always* manually test: when you are demonstrating your software to your customers and users, and when you are preparing to demonstrate your software.

Preparing means writing down a demo script that lists every action you want to perform. Rehearse your script at least a couple of times before you perform it live. Very often, rehearsals will bring about changes to the script, which is why rehearsals are so important. Always make sure you've done at least one full run-through that didn't require changes before you perform a live demo.

Testing the whole product

Frontend development includes a lot of moving parts, including the following:

- Multiple browser environments that require support
- CSS
- Distributed components, such as proxies and caches
- Authentication mechanisms

Manually testing is necessary because of the interaction of all these moving parts. We need to check that everything sits together nicely.

Alternatively, you can use end-to-end tests for the same coverage; however, these are costly to develop and maintain.

Exploratory testing

Exploratory testing is what you want your QA team to do. If you don't work with a QA team, you should allocate time to do this yourself. Exploratory testing involves exploring software and hunting for missing requirements or complex use cases that your team has not thought about yet.

Because TDD works at a very low level, it can be easy to miss or even misunderstand requirements. Your unit tests may cover 95% of cases, but you can accidentally forget about the remaining 5%. This happens a lot when a team is new to TDD, or is made up of novice programmers. It happens all the time with experienced TDDers, too – even those of us who write books on TDD! We all make errors from time to time.

A very common error scenario involves mocks. When a class or function signature is changed, any mocks of that class or function must also be updated. This step is often forgotten; the unit tests still pass, and the error is only discovered when you run the application for real.

Bug-free software

TDD can give you more confidence, but there is absolutely no way that TDD guarantees bug-free software.

With time and experience, you'll get better at spotting all those pesky edge cases before they make it to the QA team.

An alternative to exploratory testing is automated acceptance tests, but as with end-to-end tests, these are costly to develop and maintain, and they also require a high level of expertise and team discipline.

Debugging in the browser

Debugging is always an epic time sink. It can be an incredibly frustrating experience, with a lot of hair-pulling. That's a big reason we test-drive: so that we never have to debug. Our tests do the debugging for us.

Conversely, a downside of TDD is that your debugging skills will languish.

For the TDD practitioner, debugging should, in theory, be a very rare experience, or at least something that is actively avoided. But there are always occasions when debugging is necessary.

Print-line debugging is the name given to the debugging technique where a code base is littered with `console.log` statements in the hope that they can provide runtime clues about what's going wrong. I've worked with many programmers who began their careers with TDD; for many of them, print-line debugging is the only form of debugging they know. Although it's a simple technique, it's also time-consuming, involves a lot of trial and error, and you have to remember to clean up after yourself when you're done. There's a risk of accidentally forgetting about a stray `console.log` and it then going live in production.

Modern browsers have very sophisticated debugging tools that, until just recently, would have been imaginable only in a “full-fat” IDE such as Visual Studio or IntelliJ. You should make time to learn about all of the standard debugging techniques, including setting breakpoints (including conditional breakpoints), stepping in, out, and over, watching variables, and so on.

A common anti-pattern is to use debugging techniques to track down a bug, and once it's discovered, fix it and move on to the next task. What you should be doing instead is writing a failing test to prove the existence of a bug. As if by magic, the test has done the debugging for you. Then, you can fix the bug, and immediately, the test will tell you whether the issue has been fixed, without you needing to manually re-test. Think of all the time you'll save!

Check out the *Further reading* section for resources on the Chrome debugger.

That covers the main types of manual testing that you'll perform. Next, let's take a look at automated testing techniques.

Automated testing

TDD is a form of automated testing. This section lists some other popular types of automated testing and how they compare to TDD.

Integration tests

These tests check how two or more independent processes interact. Those processes could either be on the same machine or distributed across a network. However, your system should exercise the same communication mechanisms as it would in production, so if it makes HTTP calls out to a web service, then it should do so in your integration tests, regardless of where the web service is running.

Integration tests should be written in the same unit test framework that you use for unit tests, and all of the same rules about writing good unit tests apply to integration tests.

The trickiest part of integration testing is the orchestration code, which involves starting and stopping processes, and waiting for processes to complete their work. Doing that reliably can be difficult.

If you're choosing to mock objects in your unit tests, you will need at least *some* coverage of those interactions when they *aren't* mocked, and integration tests are one way to do that. Another way is system testing, as discussed below.

System tests and end-to-end tests

These are automated tests that exercise the entire system, usually (but not necessarily) by driving a UI.

They are useful when manual exploratory testing starts taking an inordinate amount of time. This happens with codebases as they grow in size and age.

End-to-end tests are costly to build and maintain. Fortunately, they can be introduced gradually, so you can start small and prove their value before increasing their scope.

Acceptance tests

Acceptance tests are written by the customer, or a proxy to the customer such as a product owner, where *acceptance* refers to a quality gate that must be passed for the released software to be accepted as complete. They may or may not be automated, and they specify behavior at a system level.

How should the customer write these tests? For automated tests, you can often use system testing tools such as Cucumber and Cypress. The Gherkin syntax that we saw in *Chapter 17, Writing Your First Cucumber Test*, and *Chapter 18, Adding Features Guided by Cucumber Tests*, is one way to do it.

Acceptance tests can be used to build trust between developers and product stakeholders. If the customer is endlessly testing your software looking for bugs, that points to a low level of trust between the development team and the outside world. Acceptance tests could help improve that trust if they start catching bugs that would otherwise be found by your customer. At the same time, however, you should be asking yourself why TDD isn't catching all those bugs in the first place and consider how you can improve your overall testing process.

Property-based and generative testing

In traditional TDD, we find a small set of specifications or examples to test our functions against. Property-based testing is different: it generates a large set of tests based on a definition of what the inputs to those functions should be. The test framework is responsible for generating the input data and the tests.

For example, if I had a function that converted Fahrenheit to Celsius, I could use a generative test framework to generate tests for a large, random sample of integer-valued Fahrenheit measurements and ensure that each of them converts into the correct Celsius value.

Property-based testing is just as hard as TDD. It is no magic bullet. Finding the right properties to assert is challenging, particularly if you aim to build them up in a test-driven style.

This kind of testing does not replace TDD, but it is another tool in any TDD practitioner's toolbox.

Snapshot testing

This is a popular testing technique for React applications. React component trees are serialized to disk as a JSON string and then compared between test runs.

React component trees are useful in a couple of important scenarios, including the following:

- When your team has a low level of experience with TDD and general program design and could become more confident with a safety net of snapshot testing
- When you have zero test coverage of software that is already being used in production, and you would like to quickly gain some level of confidence before making any changes

QA teams are sometimes interested in how software changes visually between releases, but they will probably not want to write tests in your unit test suites; they'll have their own specialized tool for that.

Snapshot testing is certainly a useful tool to know about, but be aware of the following issues:

- Snapshots are not descriptive. They do not go beyond saying, "this component tree looks the same as it did before." This means that if they break, it will not be immediately clear why they broke.
- If snapshots are rendered at a high level in your component tree, they are brittle. Brittle tests break frequently and therefore take a lot of time to correct. Since the tests are at a high level, they do not pinpoint where the error is, so you'll spend a lot of time hunting down failures.
- Snapshot tests can pass in two scenarios: first, when the component tree is the same as the previous version that was tested, and second, when no snapshot artifacts from the previous test run are found. This means that a green test does not give you full confidence – it could simply be green because previous artifacts are missing.

When writing good tests (of any kind), you want the following to be true of any test failure that occurs:

- It is very quick to ascertain whether the failure is due to an error or a change in specification
- In the case of errors, it is very quick to pinpoint the problem and the location of the error

TDD is an established technique that the community has learned enough about to know how to write good tests. We aren't quite there with snapshot testing. If you absolutely must employ snapshot testing in your code base, make sure that you measure how much value it is providing you and your team.

Canary testing

Canary testing is when you release your software to a small proportion of your users and see what happens. It can be useful for web applications with a large user base. One version of canary testing involves sending each request to two systems: the live system and the system under test. Users only sense the live system but the test system results are recorded and analyzed by you. Differences in functionality and performance can then be observed, while your users are never subjected to test software.

Canary testing is attractive because, on the surface, it seems very cost-effective, and also requires next to no *thinking* from the programmer.

Unlike TDD, canary testing cannot help you with the design of your software, and it may take a while for you to get any feedback.

That completes our look at the automated testing landscape. We started this chapter by looking at manual testing techniques. Now, let's round this chapter off with a final technique: not testing at all!

Not testing at all

There is a belief that TDD doesn't apply to some scenarios in which it does – for example, if your code is throwaway or if it's presumed to never need modification once it's deployed. Believing this is almost ensuring the opposite is true. Code, particularly code without tests, has a habit of living on beyond its intended lifespan.

Fear of deleting code

In addition to reducing the fear of changing code, tests also reduce the fear of *removing* code. Without tests, you'll read some code and think "maybe something uses this code for some purpose I don't quite remember." With tests in place, this won't be a concern. You'll read the test, see that the test no longer applies due to a changed requirement, and then delete the test and its corresponding production code.

However, there are several scenarios in which not writing tests is acceptable. The two most important ones are as follows.

When quality doesn't matter

Unfortunately, in many environments, quality doesn't matter. Many of us can relate to this. We've worked for employers who actively disregard quality. These people make enough profit that they don't *need* or *want* to care. Caring about quality is, unfortunately, a personal choice. If you are in a team that does not value quality, it will be hard to convince them that TDD is worthwhile.

If you're in this situation and you have a burning desire to use TDD, then you have a few options. First, you can spend time convincing your colleagues that it is a good idea. This is never an easy task. You could also play the TDD-by-stealth game, in which you don't ask permission before you start. Failing these options, some programmers will be fortunate enough that they can take the risk of finding an alternative employer that *does* value quality.

Spiking and deleting code

Spiking means coding without tests. We spike when we're in uncharted territory. We need to find a workable approach to a problem we've never solved before, and there is likely to be a great deal of trial and error, along with a lot of backtracking. There is a high chance of finding *unworkable* approaches before a workable one. Writing tests doesn't make much sense in this situation because many of the tests written along the way will ultimately end up being scrapped.

Let's say, for example, that I'm building a web socket server and client, but it's the first time I've used WebSockets. This would be a good candidate for spiking – I can safely explore the WebSocket API until I'm comfortable baking it into my application.

It's important to stop spiking when you feel that you've hit on a workable approach. You don't need a complete solution, just one that teaches you enough to set you off on the right path.

In the purist vision of TDD, spiking must be followed by deleting. If you're going to spike, you must be comfortable with deleting your work. Unfortunately, that's easier said than done; it's hard to scrub out creative output. You must shake off the belief that your code is sacred. Be happy to chuck it away.

In the pragmatic vision of TDD, spiking can often be followed by writing tests around the spiked code. I use this technique all the time. If you're new to TDD, it may be wise to avoid this particular cheat until you're confident that you can think out a test sequence of required tests that will cover all the required functionality within spike code.

A purist may say that your spike code can include redundant code, and it may not be the simplest solution because tests will not have driven the implementation. There is some merit to this argument.

Spiking and test-last development

Spiking is related to the practice of **test last**, but there's a subtle difference. Writing code around a spike is a TDD cheat in that you want your finished tests to look as if you used TDD in the first place. Anyone else coming along after you should never know that you cheated.

Test last, however, is a more loosely defined way of testing where you write all the production code and then write some unit tests that prove *some* of the more important use cases. Writing tests like this gives you some level of regression coverage but none of the other benefits of TDD.

Summary

Becoming a great practitioner of TDD takes great effort. It requires practice, experience, determination, and discipline.

Many people have tried TDD and failed. Some of them will conclude that TDD is broken. But I don't believe it's broken. It just takes effort and patience to get right.

But what is getting it right, anyway?

All software development techniques are subjective. Everything in this book is subjective; it is not the *right way*. It is a collection of techniques that I like to use, and that I have found success with. Other people have found success with other techniques.

The exciting part of TDD is not the black-and-white, strict form of the process; it is the grays in which we can define (and refine) a development process that works for us and our colleagues. The TDD cycle gives us *just enough* structure that we can find joy in fleshing it out with our rules and our own dogma.

I hope you have found this book valuable and enjoyable. There are many, many ways to test-drive React applications and I hope that this is the launchpad for you to evolve your testing practice.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Useful Kata resources:

<http://codingdojo.org/kata/>

<http://codekata.com>

<http://kata-log.rocks>

<https://github.com/sandromancuso/Bank-kata>

<http://www.natpryce.com/articles/000807.html>

- Global Day of Coderetreat: <https://www.coderetreat.org>
- Getting Started with Debugging JavaScript in Chrome DevTools: <https://developers.google.com/web/tools/chrome-devtools/javascript/>
- Property-based testing for JavaScript: <https://github.com/leebryon/testcheck-js>

Index

A

acceptance tests 519
accessibility
 alert role 174, 249
 element description 249
Accessible Rich Internet Applications (ARIA)
 about 109, 174
 aria-describedby 234, 243
act function
 about 61, 255, 460
 reference link 19
 using 18-20
Act phase 148, 150
addAppointment 212
after parameter 268
alert role
 about 174, 249
 using, on multiple elements 249
AnimatedLine component
 building 414-417
animation
 behavior, varying 431-437
 canceling, with cancelAnimationFrame
 429-431
 class 506-508

designing 412, 413
turtle position, using for 422
animation status
anonymous functions 472
App component
 about 209
 plan, formulating 210, 211
 state, using to control active view 211-219
 updating 460-463
appointments
 data structure 5, 15
Arrange, Act, and Assert (AAA) pattern
 about 28, 61, 467, 515
 untangling 150-152
 within beforeEach 382, 383
Array.from function 270
array patterns
 asserting on 114
ASCII escape codes
 testing 66
Assert phase 150
async act 445
asynchronous requests
 creating, with sagas 328- 334
automated testing
 about 518
 acceptance tests 519

canary testing 521
end-to-end tests 519
generative testing 519
integration tests 518
property-based testing 520
snapshot testing 520
system tests 519

B

Babel
about 7
.babelrc configuration 7
configuring 7
beforeEach function 30
Behavior Driven Development (BDD) 467
black box 188
BrowserRouter component 294
builders
 fetchResponseError 173, 177, 183
 fetchResponseOk 173, 183
 todayAt 137, 138

C

calendar view
 constructing 111
header column, adding 113-117
header row, adding 117-120
table view, adding 112, 113
callback props, test-driving 219-224
callback values
 using 225-227
call function
 328-331
canary testing 521
cancelAnimationFrame 429- 431

child components
initial props, testing 188-191
mocking 186
mocking, need for 186-188
Chromium 468
classicist TDD
 versus mockist TDD 514
client-side validation
 form, submitting 246-249
 generalizing, for multiple fields 237-246
 non-React functionality, extracting
 into module 249-252
 performing 232, 233
 required field, validating 233-237
collaborating objects 148
combineReducers function 323
commit early and often 6
components
 creating 35-40
 placing 35
component state
 switching, for Redux state 334
constants
 defining 126
const keyword 26
create-react-app 4, 9
createRoot function 14
createStore function 323
CSS selectors
 first-of-type 214, 215
Cucumber tests
 about 470-478
 adding, to dialog box 487-495
 Chromium 468
 code, timeouts avoiding 505, 506
 data tables 478
 data tables, using to perform setup 478-483
 debugging output 478

dialog box, adding 496-502
 double When phrase 488
 encapsulating When phrases 511
 feature 469, 472
 feature file 468, 470, 478
 fixing, by test-driving production code 496
 Gherkin syntaxCucumber 468-471
 integrating, into code base 469
 sagas, updating to reset or
 replay state 502-505
 World object 472
 Cypress 468

D

data
 filtering 279-282
 initial selection 43, 44
 refactoring, to simplify component
 design 283, 284
 selecting 43
 data-testid 190
 Date constructor 38, 40, 41
 date handling
 getTime 121, 139
 setHours 118, 121, 123, 125, 137, 138, 139
 today 186, 191-199
 todayAt 137-139
 tomorrow 198, 199
 tomorrowAt 198, 199
 toShortDate 119
 default exports 16
 deleting code 521
 describe function 9, 12, 30
 dialog box
 adding 496-502
 Cucumber tests, adding to 487-495

Document Object Model (DOM)
 about 12
 appendChild 15, 20-23
 className 72
 defaultPrevented 87
 dispatchEvent 87
 document.body 10-13
 event
 click 46
 events
 blur 232-235
 change 90-94
 focus 232-235
 submit 87, 88
 Form API 79
 helpers, extracting 70-73
 preventDefault 87, 88
 querySelector 35, 36
 querySelectorAll 38, 41, 46, 47
 replaceChildren 22-26
 Don't Repeat Yourself (DRY) 25

E

ECMAScript 206
 element positioning
 testing 46
 element testing
 button 43-49
 form 79
 input 79
 label 82, 83, 97
 li 38-43
 menu 211, 213
 select 105, 106, 109
 span 233, 254, 256
 table 111-113
 tbody, tr, th 111-119, 122

end-to-end tests 519
Environment object construction
 test-driving 351-355
errorFor 240
errors
 displaying, to user 174-177
 server 422 error 253
errorSend fake 367
eventChannel function 459
expect function
 about 10, 12, 14
 expect.anything 156, 161, 166, 179
 expect.arrayContaining 156, 161
 expect.objectContaining 156, 161, 166
expected value 11
expect-redux
 using, to write expectations 325-328
expect-redux package 324
expect-utils package 203
exploratory testing 517

F

failing test
 about 13-16
expectation, writing 10-13
making, to pass 16, 17
writing 8, 9
fake object 149
fakes
 avoiding 149
false positive 21
fetch
 test-driving arguments 156
 test-driving return value 159, 169
fetch API
 extracting 182, 183

global functions, replacing
 with spies 158, 159
side-by-side implementation, used for
 reworking on existing tests 162-166
spy expectations, improving with
 helper functions 166, 167
spying on 157, 158
test-driving fetch argument values 159-162
fetchQuery 357-366
fetch responses
 acting on 168
asynchronous form, of act 169
async tasks, adding to existing
 components 169- 174
errors, displaying to user 174-177
spies, upgrading to stubs 168
stubbing 167, 168
stub scenarios, grouping in nested
 describe contexts 177, 178
focus 407
form
 changed values, submitting 90-94
 default submit action, preventing 87-90
 submitting 84
 submitting, without any changes 84-87
form element
 adding 76-78
forms
 about 258
 submitting, with spies 149, 150
 submitting indicator 254
 testing, before promise completion 255-258
 validation 231
functional component
 events, adding 45-48
functional update variant
 using 422

G

generative testing 520
 generator functions
 pulling out, for reducer actions 322, 323
`getOwnPropertyDescriptor` 90, 92
`gitignore` file 53
 Given clause 471
 Given phrase 487
 Given, When, Then 472
`global.fetch` (see `fetch`)
 Jest configuration 179
 global functions
 replacing, with spies 158, 159
 GraphQL
 about 345
 data, fetching from within
 component 356-369

H

hardcoded string value
 removing 20, 21
 helper functions
 spy expectations, improving with 166, 167
 history package 341
 HistoryRouter component 294
 HTML classes
 adding, to mark animation status 506-508
 hyperlinks
 page, checking for 305, 306
 HyperText Transfer Protocol (HTTP) 148

I

ID attribute
 use 190
`initializeReactContainer` 58

input controls
 hiding 122-127
 integration tests 518
 intermediate components
 using, to translate URL state 300-303
`it` function 10
`it.only` function 151, 152, 166
`it.skip` function 22, 23

J

JavaScript Object Notation (JSON) 157
 JavaScript Syntax Extension (JSX) 7
 Jest
 about 212, 214, 221
 alternatives 6
 `beforeEach` function 30
 commit early and often 6
 configuration
 `restoreMocks` 179, 181
 `setupFilesAfterEnv` 182
 creating 5
 `describe` function 9
 `expect` function 12-18, 21-24, 27
 `expect.hasAssertions` 85
 global functions 9
 installing 6
 `it` function 10
 `it.only` function 151, 152, 166
 `it.skip` function 22, 23
 `jest.mock` function 187
 `matcherHint` 66
 `mockResolvedValue` 178, 180, 183
 `printExpected` 66, 68
 running tests 9
 `spyOn` 178, 179
 test environment 11, 12
 test function 10

this.equals 156
watchAll flag 30
Jest matcher
 creating, with TDD 61-70
jest.mock call
 mocked components children,
 rendering 205
module mocks, alternatives 206
multiple instances of rendered
 component, checking 205
spy function, removing 205
variants 204, 205
Jest test-double support
 about 178-182
 fetch test functionality, extracting 182, 183
 migrating to 178
jsdom 12

K

kata 515
keyboard focus
 modifying 402
prompt, focusing 405-408
reducer, writing 402-405
requesting, in components 408, 409

L

let keyword 26
Link component
 about 294
 mocking 306-309
list item content
 specifying 40-43
list validator function 245
LocalStorage 393, 394, 400

M

manual testing
 about 49, 411, 516
 debugging, in browser 518
 exploratory testing 517
 software, demonstrating 516
 whole product, testing 517
manual testing, of changes
 about 49
 entry point, adding 49-51
 webpack, using 51-53
matchers
 about 11, 61, 62, 70
 building, for component mocks 200-204
 toBeCalledWith 154, 156, 166,
 173, 178, 179, 181
 toBeCloseTo 483
 toBeDefined 153-155
 toBeFirstRenderedWithProps 200, 204
 toBeNull 35-37
 toBeRenderedWithProps 200-204
 toContain 41, 46, 47
 toContain function 11
 toContainText 62-69
 toEqual 40, 46
 toHaveBeenCalledWith 196-200
 toHaveClass 73, 74
 toHaveLength 39, 45
 toMatchObject 317-321, 378, 381
 using, to simplify spy expectations 154-157
match validator function 244
memoized callback 129
MemoryRoute
 versus HistoryRouter 297
message function 65
middleware, test-driving 328
Mocha 6

mocked components
 alternatives 206
 children, rendering 205
mockImplementation 443
 mocking constructors 354
mockist TDD
 versus classicist TDD 514
mockResolvedValue 180, 183
mock-up
 sketching 34
module factory parameter 187
multiple form fields
 batch of tests, solving 98, 99
 describe blocks, nesting 94, 95
handleChange, modifying so
 that it works with 100
 parameterized tests, generating 95-98
 testing 100
 tests, duplicating 94
multiple pages dataset
 design, adjusting 272, 273
 design changes, forcing with tests 277, 278
 moving, between 268
 Next button, adding to move
 to next page 268-272
 Previous button, adding to move
 to previous page 274-277

N

name property 126
nested describe block 101
nested describe contexts
 stub, scenarios grouping in 177, 178
Node.js
 URL 5
Node Package Manager (npm)
 installation 5

package.json configuration 5, 10, 12, 19, 30
project initialization 5, 6
version 5
non-React functionality
 extracting, into module 249-252
noSend fake 366
not.toBeCalled 452

P

package installs
 Babel 7
expect-redux 324, 325
Jest 6
JSDOM 12
React 7
react-redux 324, 336
react-router-dom 296, 300, 303, 306, 307, 310
redux-saga 313, 327, 329
webpack 51-53
package.json scripts
 using, advantages 470
pair programming 20
parameterized tests. *See test*
 generator functions
 about 75
 generating 95-98
performFetch function
 building 347-351
ping pong programming 20
predictable state container 313
programmatic navigation
 testing 310-312
property-based testing 520
Provider component 324
Puppeteer
 console logging 478

headless mode, turning off 468, 478
integrating, into code base 469
Page.\$eval 474, 475, 483
Page.click 473
Page.waitForSelector 489, 505, 508
Page.waitForTimeout 483, 506
 avoiding 505
put function 325, 327

Q

queryByTestId function renderAndWait 190
query strings 276-283, 295, 301, 303

R

radio button
 selecting, in group 127-129
radio button groups
 field changes, handling 129-136
 input controls, hiding 122-127
 test-driving 120-122
React
 about 80
 act function 19
 component 8, 9, 13, 14
 container 14
 createRoot function 14
 defaultProps 106-110, 117, 120
 hooks 45
 htmlFor attribute 83
 installing 7, 8
 key values, testing 40
 readOnly attribute 81, 84, 93
 render function 18
 Simulate 47

React component trees
 uses 520
React form
 submitting, by dispatching
 Redux action 334-336
React project
 creating 4
 data, displaying with first test 8
React Router applications
 designing, from test-first perspective 294
 pieces, using 294
 tests, splitting 294
 up-front design, for new routes 295
React Testing Library 190
red, green, refactor cycle 28, 29
REDO action
 handling 386-390
reducer
 design 314
reducer, test-driving
 about 316-321
 entry point, setting up 323, 324
 generator functions, pulling out for 322, 323
 store, setting up 323, 324
Redux
 about 313
 decorating reducer 377-380
 environment reducer 402, 403
 middleware 394, 402
 need for 314
 store actions, designing 315, 316
 store state, designing 315, 316
 user actions, undoing and redoing 376
 withUndoRedo reducer 376, 383, 389
Redux action
 React form, submitting by
 dispatching 334-336

-
- Redux middleware
 - building 394-402
 - local storage, saving via 393, 394
 - Redux saga
 - events, streaming with 454-460
 - router history, navigating in 341-343
 - splitting apart 442, 443
 - Redux state
 - component state, switching for 334
 - refactoring process
 - about 25
 - methods, extracting 26, 27
 - test setup, sharing 25, 26
 - regular expressions 245, 246
 - Relay environment, testing
 - about 346
 - Environment object construction,
 - test-driving 351-355
 - performFetch function, building 347-351
 - singleton instance of Environment,
 - test-driving 355, 356
 - Relay library 345
 - render
 - render function 18
 - renderAndWait helper
 - adding 193
 - render props
 - about 285
 - actions, performing with 284-287
 - testing, in additional render contexts 287-290
 - renderWithStore test extension
 - adding 324, 325
 - requestAnimationFrame
 - about 411, 417, 432, 433, 435
 - animating with 417, 418-429
 - required validator function 240
 - reusable rendering logic
 - pulling out 58-61
 - reusable spy function
 - creating 152-154
 - rich internet applications (ARIA) labels 109
 - root saga 327
 - Route component 294
 - router
 - components, testing 296
 - Router component
 - about 294
 - test equivalent 296, 297
 - router history
 - navigating, in Redux saga 341-343
 - router links
 - Link component, mocking 306-309
 - page, checking for hyperlinks 305, 306
 - testing 304
 - Routes component
 - about 294
 - using, to replace switch statement 297-299

S

- saga
 - design 314
 - updating, to reset or replay state 502-505
- saga, test-driving
 - about 324
 - asynchronous requests, creating 328-334
 - expect-redux, using to write expectations 325-328
- renderWithStore test extension,
 - adding 324, 325
- sample data
 - appointments 50, 54

- Scalable Vector Graphics (SVG)
 - about 412
 - polygon element 491
- scenarios, for not writing code
 - code, deleting 522
 - code, spiking 522
 - quality 521, 522
- schema
 - compiling 346
- select box
 - options, providing 105-108
 - real values, verifying 106
 - value, pre-selecting 108-110
 - value, selecting 104
- sendCustomer fake 358
- server errors
 - handling 252-254
- shared state 22
- side-by-side implementation
 - about 161, 167
 - used, for reworking on existing tests 162-166
- singleton instance, Environment
 - test-driving 355, 356
- Spec Logo environment 393
- Spec Logo user interface 374, 375
- spies
 - global functions, replacing with 158, 159
 - upgrading, to stubs 168
 - used, for submitting forms 149, 150
- spike 154
- spiking 522
- spy
 - about 150
 - Jest, using to 178, 179
 - removing 205
- spy expectations
 - improving, with helper functions 166, 167
 - simplifying, with matcher 154-157
- state
 - using, to control active view 211-219
- store state
 - using, within component 337-341
- Structured Query Language (SQL) 149
- stub
 - about 167
 - Jest, using to 178, 179
 - scenarios, grouping in nested
 - describe contexts 177, 178
 - spies, upgrading to 168
- stubbing
 - window.location 444
- submitting indicator 255, 257
- switch statement
 - replacing, with Routes component 297-299
- synthetic event 47
- system tests 519

T

- tabular data fetched
 - displaying, from endpoint 263-268
- technical debt 5
- temporal coupling 258
- terminal colors 62, 65, 66
- test data
 - simplifying 14
- test data builders
 - extracting, for time and date
 - functions 137-139
- test descriptions 10
- test double
 - about 148, 149
 - fakes, avoiding 149
- test-driven development
 - using, as testing technique 513, 514

Test-Driven Development (TDD)

- about 3
- code, refactoring 29
- failing test, writing 28
- test, making to pass 28
- used, for creating Jest matcher 61-70
- test-driving fetch argument values 159-162
- test environment 12
- test extension
 - about 60
 - change 90, 93
 - click 60, 70
 - clickAndWait 169, 171, 172
 - dispatchToStore 335
 - element 58, 60, 64, 68, 70-72
 - elements 70-72
 - initializeReactContainer 60
 - propsOf 219-221
 - render 58, 59, 70, 72
 - renderAndWait 193-199, 204
 - renderWithRouter 296-299
 - renderWithStore 324, 335
 - submitAndWait 172
 - submitButton 91
 - textOf 70-72
 - typesOf 70-72
 - withFocus 232-235
- test generator functions 95
- testing process
 - streamlining 29, 30
- testing technique
 - improvement tips 515, 516
- test-last development 522
- testProps objects 125, 140, 141, 144
- testProps object
 - extracting 140-144

tests

- Arrange, Act, Assert (AAA) pattern 28
- locations, for storage 9
- red, green, refactor cycle 28
- writing 27, 28
- test suite
 - migrating, to use Jest test-double support 179-182
- text extensions
 - buttonWithLabel 268
 - changeAndWait 280, 284
 - renderAdditional 288, 290
- text input
 - accepting 78-84
- Then clause 471
- time origin 425
- timeouts
 - avoiding 506
 - avoiding, in test code 505, 506
- toEqual matcher 41
- touch command 8, 16
- triangulation 20, 501
- turtle position
 - using, for animation 422
- TypeScript 238

U

- UI elements 442
- UNDO action
 - handling 382-385
- Uniform Resource Locator (URL) 273
- unit testing 148, 159
- unit tests
 - AAA pattern 515
 - design tool 515
 - Don't Repeat Yourself (DRY) 515

focused on observable behavior 515
independent 514
quick to run 514
short, with high level of abstraction 514
unmounting component 431
unused function arguments
 ignoring 39
up-front design 34
URL state
 translating, with intermediate
 components 300-303
useCallback hook 129, 133, 134
useDispatch hook 336
useEffect hook
 about 191, 192
 adding 194-197
 dependency list, testing 198, 199
 setters, using 192
 used, for fetching data on mount 191
useLocation hook 294
useNavigate hook 294
user actions, in Redux
 reducer, building 376
 Undo and Redo buttons, building 390-393
user actions in Redux, reducer
 initial state, setting 377-381
 redo action, handling 386-390
 undo action, handling 382-385
useRef 405
useSearchParams hook 294
useSelector hook 337
useState hook
 about 45, 47
 functional update variant 422

W

waitForSelector
 step definitions, updating to use 508-511
WebSocket
 interaction, designing 440
 onclose 443, 454
 onerror 443, 454
 onopen 443, 454
WebSocket connection
 test-driving 443-454
WebSocket, interaction
 saga, splitting apart 442, 443
 sharing workflow 440, 441
 UI elements 442
When clause 471

Y

You Ain't Gonna Need It (YAGNI) 4



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

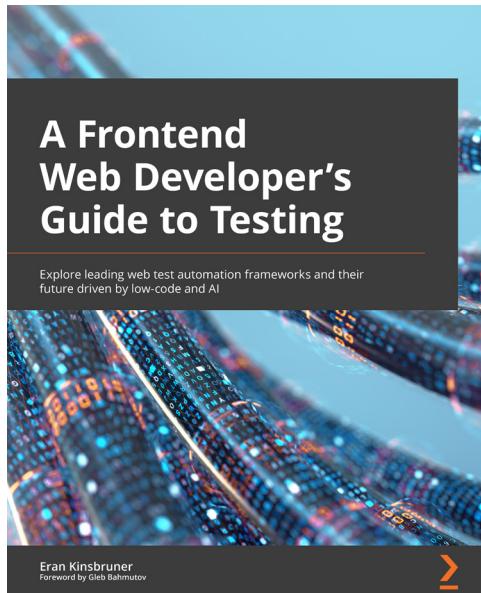
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

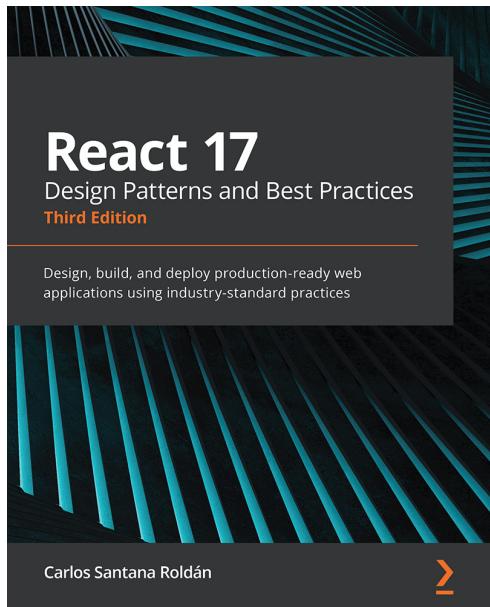


A Frontend Web Developer's Guide to Testing

Eran Kinsbruner

ISBN: 978-1-80323-831-9

- Choose the ideal tool or combination of tools for testing your app
- Continuously monitor the market and ensure that your developers are using the right tools
- Advance test automation for your web app with sophisticated capabilities
- Measure both code coverage and test coverage to assess your web application quality
- Measure the success and maturity of web application quality
- Understand the trade-offs in tool selection and the associated risks
- Build Cypress, Selenium, Playwright, and Puppeteer projects from scratch
- Explore low-code testing tools for web apps



React 17 Design Patterns and Best Practices – Third Edition

Carlos Santana Roldán

ISBN: 978-1-80056-044-4

- Get to grips with the techniques of styling and optimizing React components
- Create components using the new React Hooks
- Use server-side rendering to make applications load faster
- Get up to speed with the new React Suspense technique and using GraphQL in your projects
- Write a comprehensive set of tests to create robust and maintainable code
- Build high-performing applications by optimizing components

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Hi!

This is Daniel Irvine, author of *Mastering React Test-Driven Development*. I hope you enjoyed reading this book. It would really help me (and other potential readers) if you could leave a review on Amazon sharing your thoughts.

Go to the link below to leave your review:

<https://packt.link/r/1803247126>

Your review will help us to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best wishes,

A handwritten signature in black ink, appearing to read "Daniel Irvine".

