O'REILLY®

React Cookbook

Recipes for Mastering the React Framework

Early Release
RAW & UNEDITED

David Griffiths & Dawn Griffiths

React Cookbook

Recipes for Mastering the React Framework

David Griffiths and Dawn Griffiths

React Cookbook

by David Griffiths and Dawn Griffiths

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Amanda Quinn
- Development Editor: Corbin Collins
- Production Editor: Daniel Elfanbaum
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- October 2021: First Edition

Revision History for the Early Release

- 2020-08-05: First Release
- 2020-11-20: Second Release
- 2021-01-20: Third Release
- 2021-03-15: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492085843 for release details.

978-1-492-08577-5

[LSI]

Preface

The following software versions have been used:

Tool/library	Description	Versions
Apollo client	GraphQL client	3.3.6
axios	HTTP library	0.21.0
Create React App	Tool for generating React apps	4.0.1
Cypress	Automated test system	6.1.0
Gatsby	Tool for generating React apps	2.26.1
GraphQL	API query language	15.4.0
Material-UI	Component library	4.11.2
Node	JavaScript runtime	v12.20.0
npm	The Node package manager	6.14.8
nvm	Tool for running multiple Node environments	0.33.2
nwb	Tool for generating React apps	0.25.x
Next.js	Tool for generating React apps	10.0.3
Preact	Lightweight React-like framework	10.3.2
Preact Custom Elements	Library to create custom elements	4.2.1
Rails	Web development framework	6.0.3.4
Razzle	Tool for generating React apps	3.3.8
React	Web framework	17.0.1
React Media	Media queries in React code	1.10.0
React Router (DOM)	Library for managing React routes	5.2.0
React Testing Library	Unit testing library for React	11.1.0

react-animations	React CSS animation library	1.0.0
react-md-editor	Markdown editor	2.0.3
React-Redux	React support library for Redux	7.2.2
Redux	State management library	4.0.5
Redux-Persist	Library to store Redux state	6.0.0
Ruby	Language used by Rails	2.7.0p0
Storybook	Component gallery system	6.1.11
TweenOne	React animation library	2.7.3
Typescript	Type-safe extension to JavaScript	4.0.3
Webpacker	Tool for adding React to Rails apps	4.3.0
Yarn	Another Node package manager	1.22.10

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *TBD*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*React Cookbook* by David Griffiths and Dawn Griffiths (O'Reilly). Copyright 2021 O'Reilly Media Inc., 978-1-492-08584-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781492085843*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Chapter 1. Creating Applications

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

React is a surprisingly adaptable development framework. Developers use it to create large JavaScript-heavy Single Page Applications or to build surprisingly small plug-ins. You can use it to embed code inside a Rails applications or generate a content-rich web site.

In this chapter, we look at the various ways of creating a React application. We also look at some of the more useful tools that you might want to add to your development cycle. Very few people now create their JavaScript projects from scratch. Doing so is now a very tedious process, involving an uncomfortable amount of tinkering and configuration. The good news is that in almost every case, you can use a tool to generate the code you need.

So let's take a whistle-stop tour of the many ways of starting your React journey, beginning with the one most frequently used: `create-react-app`….

1.1 Create a Vanilla App with create-react-app

Problem

React projects are difficult to create and configure from scratch. Not only are there numerous design choices to make–which libraries to include, which tools to use, which language features to enable–but manually created applications will, by their nature, differ from one another. Project idiosyncrasies increase the time it takes a new developer to become productive.

Solution

`create-react-app` is a tool for building SPAs with a standard structure and a reasonable set of default options. Generated projects use the `react-scripts` library to build, test, and run the code. Projects have a standard Webpack configuration and a standard set of language features enabled.

Any developer who has worked on one `create-react-app` application instantly feels at home with any other. They understand the project structure and know which language features they can use. It is simple to use and contains all the features that a typical application requires: from babel configuration and file loaders to testing libraries and a development server.

If you're new to React, or need to create a generic SPA with the minimum of fuss, then you should consider creating your app with `create-react-app`.

You can choose to install the `create-react-app` command globally on your machine, but this is now discouraged. Instead, you should create a new project by calling `create-react-app` via `npx`. Using `npx` ensures you're building your application with the latest version of `create-react-app`:

```
$ npx create-react-app my-app
```

This command creates a new project directory called `my-app`. By default, the application uses JavaScript. If you want to use TypeScript as your development language, `create-react-app` provides that as an option:

```
$ npx create-react-app --template typescript my-app
```

Facebook developed `create-react-app`, so it should come as no surprise that your new project uses the `yarn` package manager. To use `npm`, change into the directory and remove the *yarn.lock* file, and then re-run the install with `npm`:

```
$ cd my-app
$ rm yarn.lock
$ npm install
```

To start your application, you should run the `start` script:

```
npm run start # or yarn start
```

This command launches a server on port 3000, and opens a browser at the home page (see *figure 1-1*.)

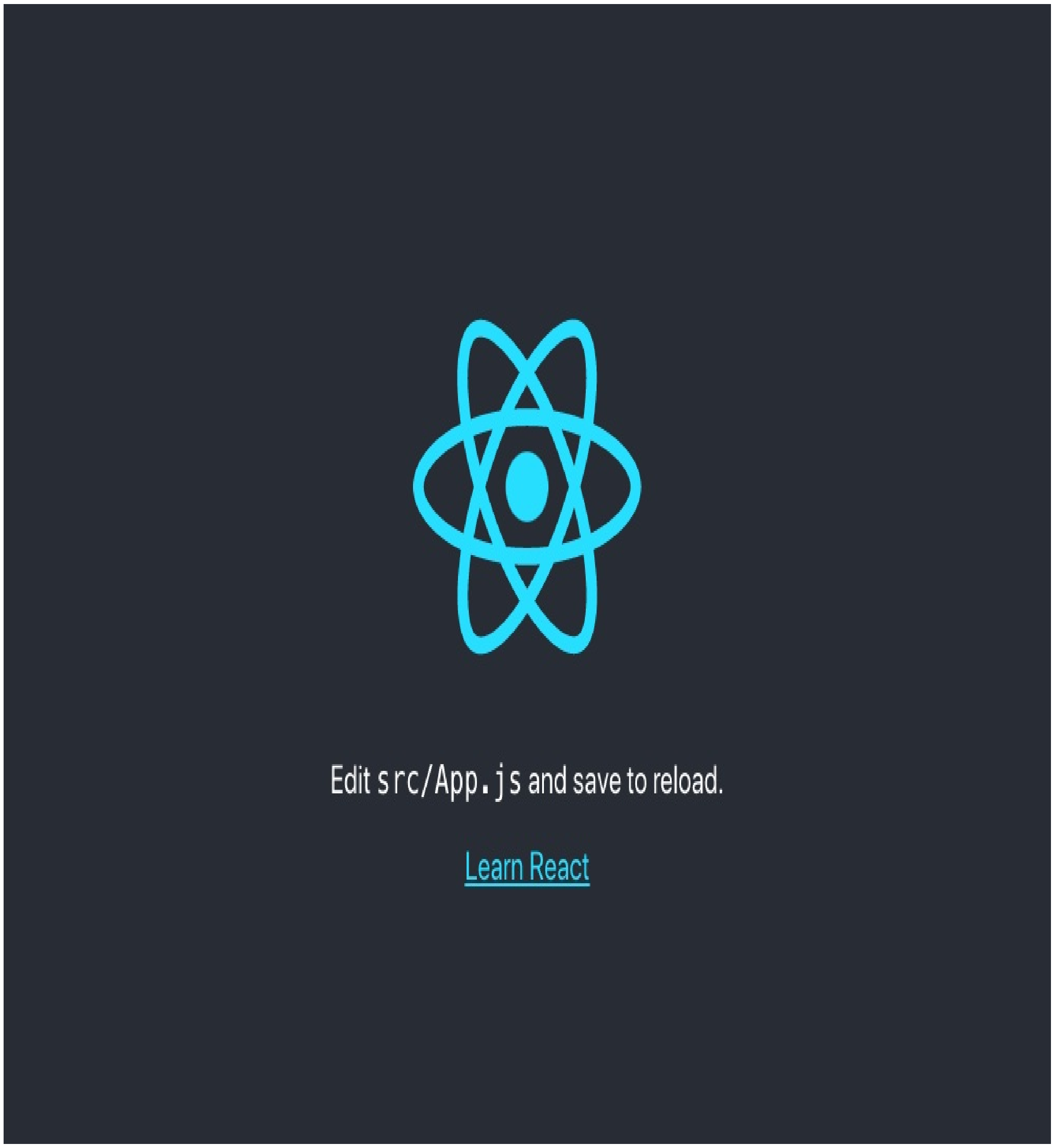


Figure 1-1. The generated front page

The server delivers your application as a single, large bundle of JavaScript. The code mounts all of its components inside this `<div/>` in *public/index.html*:

```
<div id="root"></div>
```

The code to generate the components begins in the *src/index.js* file (*src/index.tsx* if you're using TypeScript):

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

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

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

This file does little more than render a single component called `<App/>`, which is imported from *App.js* in the same directory:

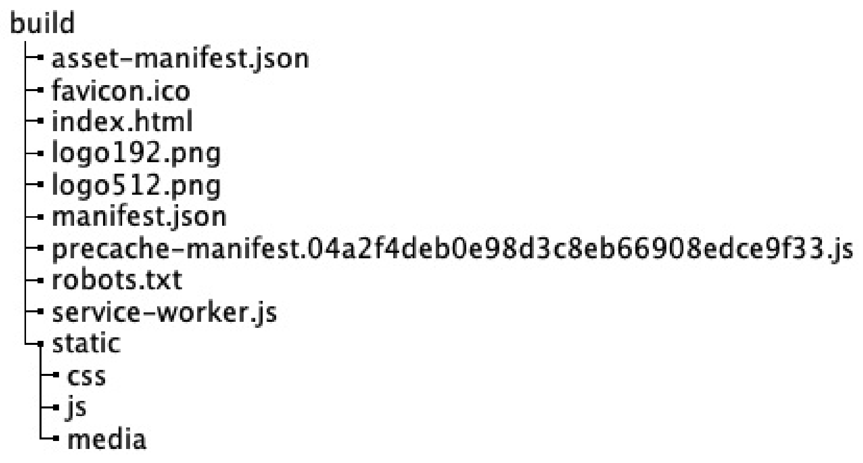
```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

If you edit this file while the application is `start`-ed, the page in the browser automatically updates.

When you’re ready to ship the code to production, you need to generate a set of static files that you can deploy on a standard web server. To do this, run the `build` script:

```
$ npm run build
```

The `build` script creates a *build/* directory and in there generates a set of static files (see *figure 1-2*.)



```
build
├─ asset-manifest.json
├─ favicon.ico
├─ index.html
├─ logo192.png
├─ logo512.png
├─ manifest.json
├─ precache-manifest.04a2f4deb0e98d3c8eb66908edce9f33.js
├─ robots.txt
├─ service-worker.js
└┬ static
 ├─ css
 ├─ js
 └─ media
```

Figure 1-2. The generated contents in the build directory.

The build copies many of these files from the *public/* directory. The code for the app is transpiled into browser-compatible JavaScript and stored in one or more files in the `static/js` directory. Stylesheets used by the application, are stitched together and stored in *static/css*. Several of the files have randomized ids added to them so that when you deploy your application, browsers download the latest code, rather than some old cached version.

Discussion

`create-react-app` is not just a tool for generating a new application, but also a platform to keep your React application up-to-date with the latest tools and libraries. You can upgrade the `react-scripts` library as you would any other: by changing the version number and re-running `npm install`. You don't need to manage a list of babel plug-ins, postcss libraries, or maintain a complex *webpack.config.js* file. The `react-scripts` library manages them all for you.

If, however, you later decide to manage all of this yourself, you're free to do so. If you eject the application, then everything comes back under your control:

```
npm run eject
```

However, this is a one-time-only change. Once you have ejected your application, there is no going back. You should think carefully before ever ejecting an application. You may find that the configuration you need is already available. For example, developers would often eject an application to switch to using TypeScript. The `--template typescript` option now removes the need for that.

Another common reason for ejecting was to proxy web services. React apps often need to connect to some separate API backend. Developers used to do this by configuring Webpack to proxy a remote server through the local development server. You can now avoid do this by setting a proxy in the `package.json` file:

```
"proxy": "http://myapiserver",
```

If your code now contacts a URL that the server cannot find locally (`/api/thing`), the `react-scripts` automatically proxy these requests to `http://myapiserver/api/thing`.

TIP

If you possibly can, avoid ejecting your application. Look through the `create-react-app` documentation at *https://create-react-app.dev/* to see if you can make the change some other way.

You can download the source for this recipe in JavaScript or TypeScript from the Github site.

1.2 Build Content-Rich Apps with Gatsby

Problem

Content-rich sites like blogs and online stores need to serve large amounts of complex content efficiently. A tool like `create-react-app` is not suitable for this kind of web site because it delivers everything as a single large bundle of JavaScript that a browser must download before anything displays.

Solution

If you are building a content-rich site, consider using Gatsby.

Gatsby focuses on loading, transforming, and delivering content in the most efficient way possible. It can generate static versions of web pages, which means that the response times of Gatsby sites are often significantly lower than, say, those built with `create-react-app`.

Gatsby has a large number of plug-ins that can load and transform data efficiently from static local data, GraphQL sources, and third-party CMS systems such as Wordpress.

You can install `gatsby` globally, but you can also run it via the `npx` command:

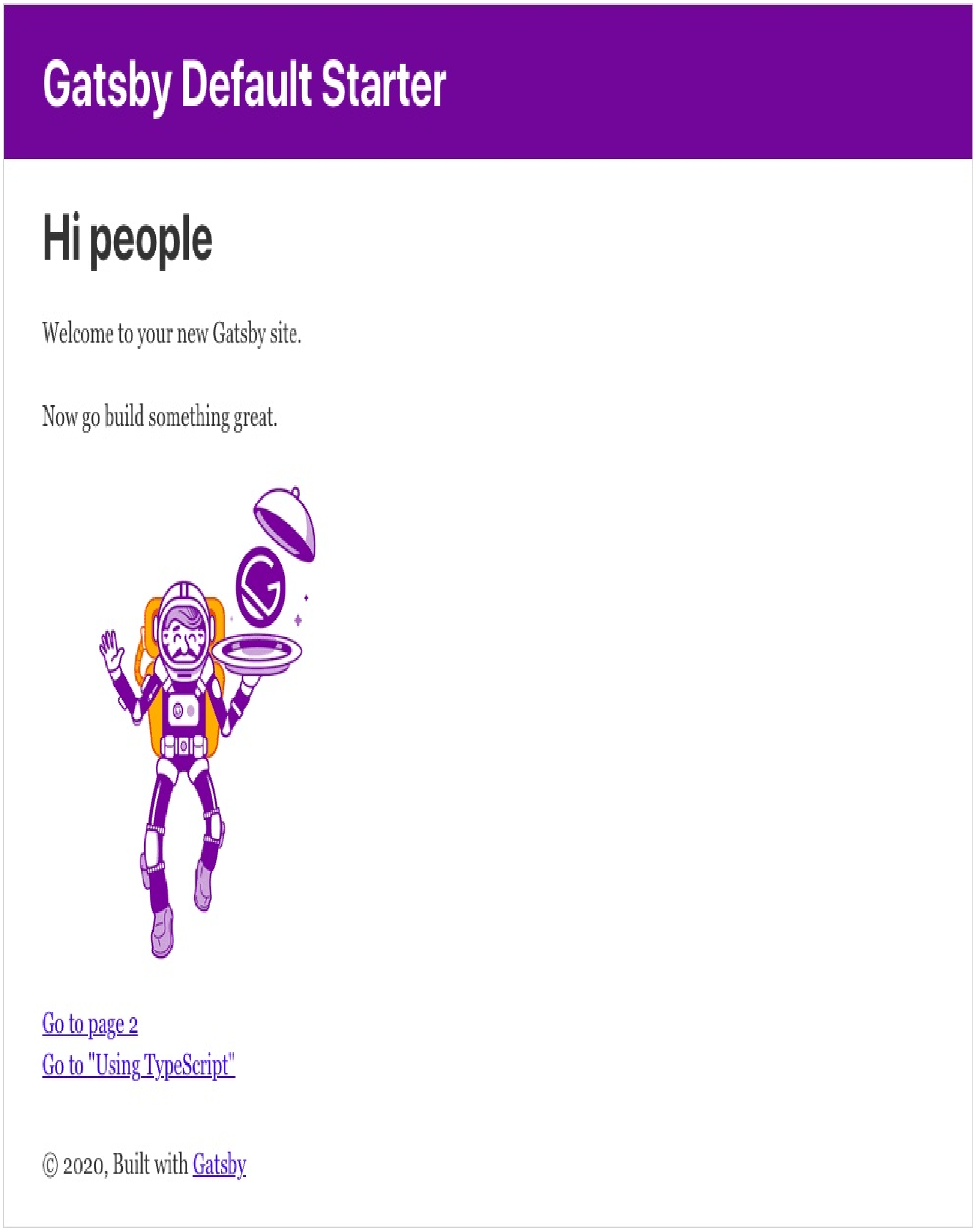
```
$ npx gatsby new my-app
```

The `gatsby new` command creates a new project in a subdirectory called *my-*

app. The first time you run this command, it asks which package manager to use: either `yarn` or `npm`.

To start your application, change into the new directory and run it in development mode:

```
$ cd my-app
$ npm run develop
```


Gatsby Default Starter

Hi people

Welcome to your new Gatsby site.

Now go build something great.

Go to page 2
Go to "Using TypeScript"

© 2020, Built with Gatsby

Figure 1-3. Gatsby page at http://localhost:8000

Gatsby projects have a straightforward structure, as shown in *figure 1-4*.

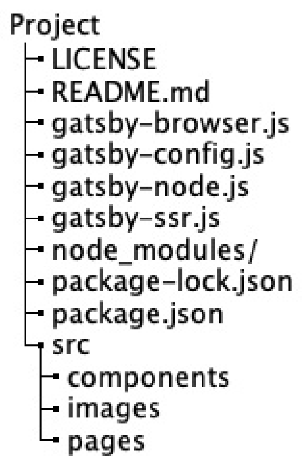


Figure 1-4. The Gatsby directory structure.

The core of the application lives under the *src/* directory. Each page within a Gatsby app has its own React component. This is the front page of the default application:

```
import React from "react"
import { Link } from "gatsby"

import Layout from "../components/layout"
import Image from "../components/image"
import SEO from "../components/seo"

const IndexPage = () => (
  <Layout>
    <SEO title="Home" />
    <h1>Hi people</h1>
    <p>Welcome to your new Gatsby site.</p>
    <p>Now go build something great.</p>
    <div style={{ maxWidth: `300px`, marginBottom: `1.45rem` }}>
      <Image />
    </div>
    <Link to="/page-2/">Go to page 2</Link> <br />
    <Link to="/using-typescript/">Go to "Using TypeScript"</Link>
  </Layout>
)

export default IndexPage
```

There is no need to create a route for the page. Each page component is automatically assigned a route. For example, the page at *src/pages/using-typescript.tsx* [1] is automatically available at `/using-typescript/`. This approach has multiple advantages. First, if you have a lot of pages, you don’t need to manage the routes for them manually. Second, it means that Gatsby can deliver much more rapidly. To see why let’s look at how to generate a production build for a Gatsby application.

If you stop the Gatsby development server[2], you can generate a production build with the following:

```
$ npm run build
```

This command runs a `gatsby build` command, which creates a *public/* directory. And it is the *public/* directory that contains the real magic of Gatsby. For each page, you find two files. First, a generated JavaScript file:

```
1389 06:48 component---src-pages-using-typescript-tsx-
93b78cfadc08d7d203c6.js
```

Here you can see that the code for *using-typescript.tsx* is just 1389 bytes long and which, with the core framework, is just enough JavaScript to build the page. It is not the kind of include-everything script that you find in a `create-react-app` project.

Secondly, there is a subdirectory for each page, containing a generated HTML file. For *using-typescript.tsx* the file is called *public/using-typescript/index.html*, which is a statically generated version of the web page. It contains the HTML that the *using-typescript.tsx* component would otherwise render dynamically. At the end of the web page, it loads the JavaScript version of the page in case it needs to generate some dynamic content.

This file structure means that Gatsby pages load in around the same time that it takes to load a static web page. Using the bundled `react-helmet` library, you can also generate `<meta/>` header tags with additional features about your site. Both features are great for Search Engine Optimization.

Discussion

How will the content get into your Gatsby application? You might use a headless

CMS system, a GraphQL service, a static data source, or something else. Fortunately, Gatsby has many plug-ins which allow you to connect data sources to your application, and then transform the content from a format such as Markdown into HTML.

You can find a full set of plug-ins on the Gatsby web site.

Most of the time, you choose the plug-ins you need when you first create the project. To give you a head-start, Gatsby also supports *start templates*. The template provides the initial application structure and configuration. The app we built above uses the default started template, which is quite simple. The *gatsby-config.js* file in the root of the application configures which plug-ins your application uses.

But there are masses of Gatsby starters available, pre-configured to build applications that connect to a variety of data sources, with pre-configured options for SEO, styling, offline caching, PWA (Progressive Web Applications), and more. Whatever kind of content-rich application you are building, there is a starter that is close to what you need.

There is more information on the Gatsby web site about Gatsby starters, as well as a cheat sheet for the most useful tools and commands.

You can download the source for this recipe from the Github site.

1.3 Build Universal Apps with Razzle

Problem

Sometimes when you start to build an application, it is not always clear what the main architectural decisions will be. Should you create a SPA? If performance is critical, should you use Server Side Rendering? You will need to decide what your deployment platform will be, and whether you are going to write your code in JavaScript or TypeScript.

Many tools require that you answer these questions early on. If you later change your mind, modifying the way you build and deploy your application can be complicated.

Solution

If you want to defer decisions about how you build and deploy your application, you should consider using Razzle.

Razzle is a tool for building Universal applications: that is, applications that can execute their JavaScript on the server. Or the client. Or both.

Razzle uses a plug-in architecture that allows you to change your mind about how you build your application. It will even let you change your mind about whether you are building your code in React, or Preact or some other framework entirely, like Elm or Vue.

You can create a Razzle application with the `create-razzle-app` command[3]:

```
$ npx create-razzle-app my-app
```

This command creates a new Razzle project in the *my-app* subdirectory. You can start the development server with the `start` script:

```
$ cd my-app
$ npm run start
```

The `start` script will dynamically build both client code and server code, and then run the server on port 3000, as shown in *figure 1-5*.

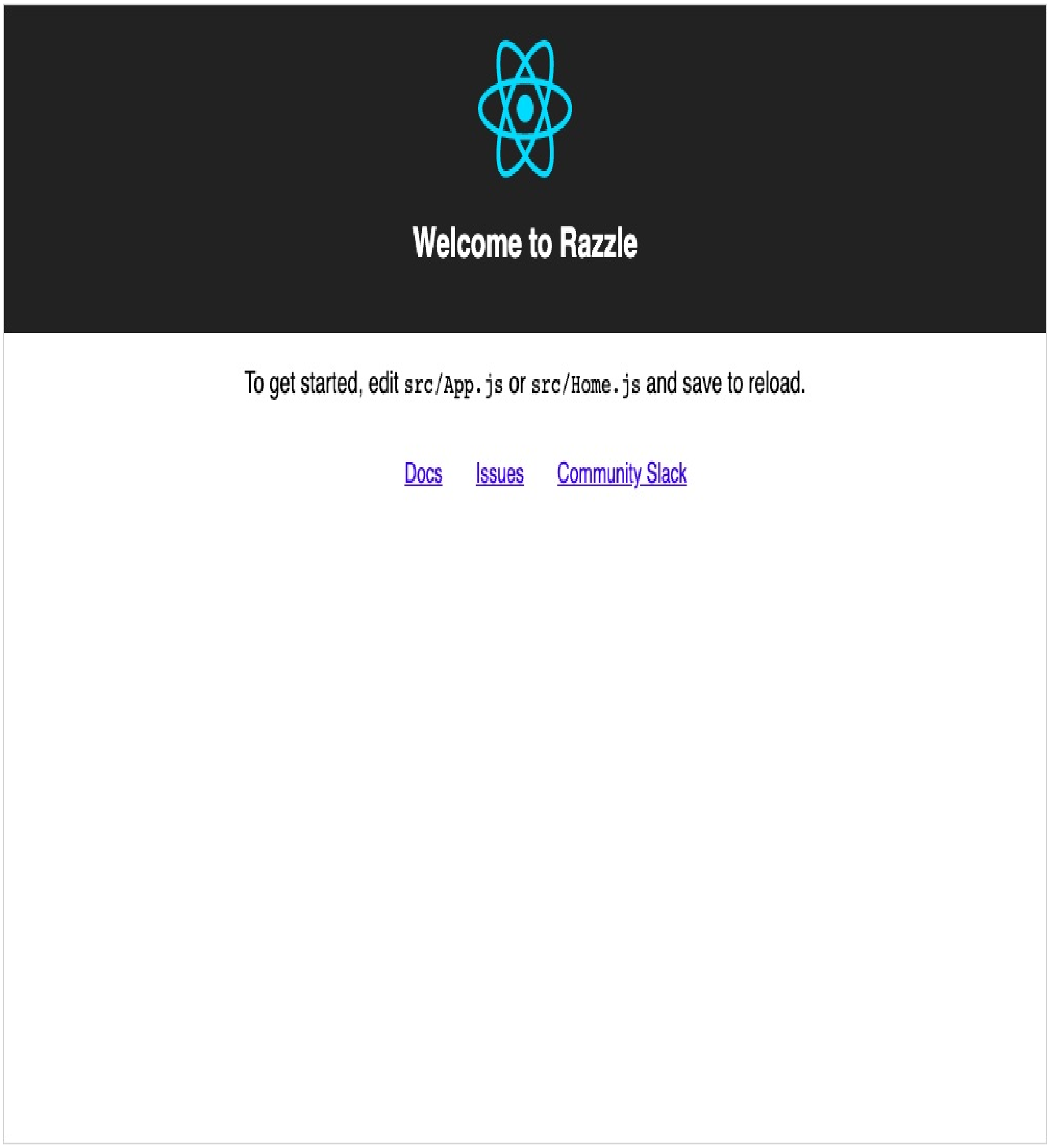

Welcome to Razzle
To get started, edit src/App.js or src/Home.js and save to reload.
Docs
Issues
Community Slack

Figure 1-5. The Razzle front page at http://localhost:3000

When you want to deploy a production version of your application, you can then run the `build` script:

```
$ npm run build
```

Unlike `create-react-app`, this will build not just the client code, but also a node server. Razzle generates the code in the *build/* subdirectory. The server code will continue to generate static code for your client at runtime. You can start a production server by running the *build/server.js* file with node using the `start:prod` script:

```
$ npm run start:prod
```

You can deploy the production server anywhere that node is available.

The server and the client can both run the same code, which is what makes it *Universal*. But how does it do this?

The client and the server have different entry points. The server runs the code in *src/server.js*; the browser runs the code in *src/client.js*. Both *server.js* and *client.js* then render the same app using *src/App.js*.

If you want to run your app as a SPA, remove the *app/index.js* and *app/server.js* files. Then create an *index.html* in the *public/* folder containing a `<div/>` with id `root`, and re-build the application with:

```
$ node_modules/.bin/razzle build --type=spa
```

You will generate a full SPA in *build/public/* that you can deploy on any web server.

Discussion

Razzle is so adaptable because it is built from a set of highly configurable plug-ins. Each plug-in is a higher-order function that receives a webpack configuration and returns a modified version. One plug-in might transpile TypeScript code, another might bundle the React libraries.

If you want to switch your application to Vue, you only need to replace the plug-ins you use.

You can find a list of available plug-ins on the Razzle web site.

You can download the source for this recipe from the Github site.

1.4 Manage Server and Client Code with Next.js

Problem

The focus of React is on client code–even if that client code is generated on the server. Sometimes, however, you might have a relatively small amount of API code that you would prefer to manage as part of the same React application.

Solution

Next.js is a tool for generating React applications that include their own server code. The api end-points and the client pages use default routing conventions, which makes them simpler to build and deploy than they would be if you manage them yourself. You can find full details about Next.js on the web site.

At the time of writing, you cannot create a Next.js application using `npx`. Instead, you should first install `create-next-app` globally:

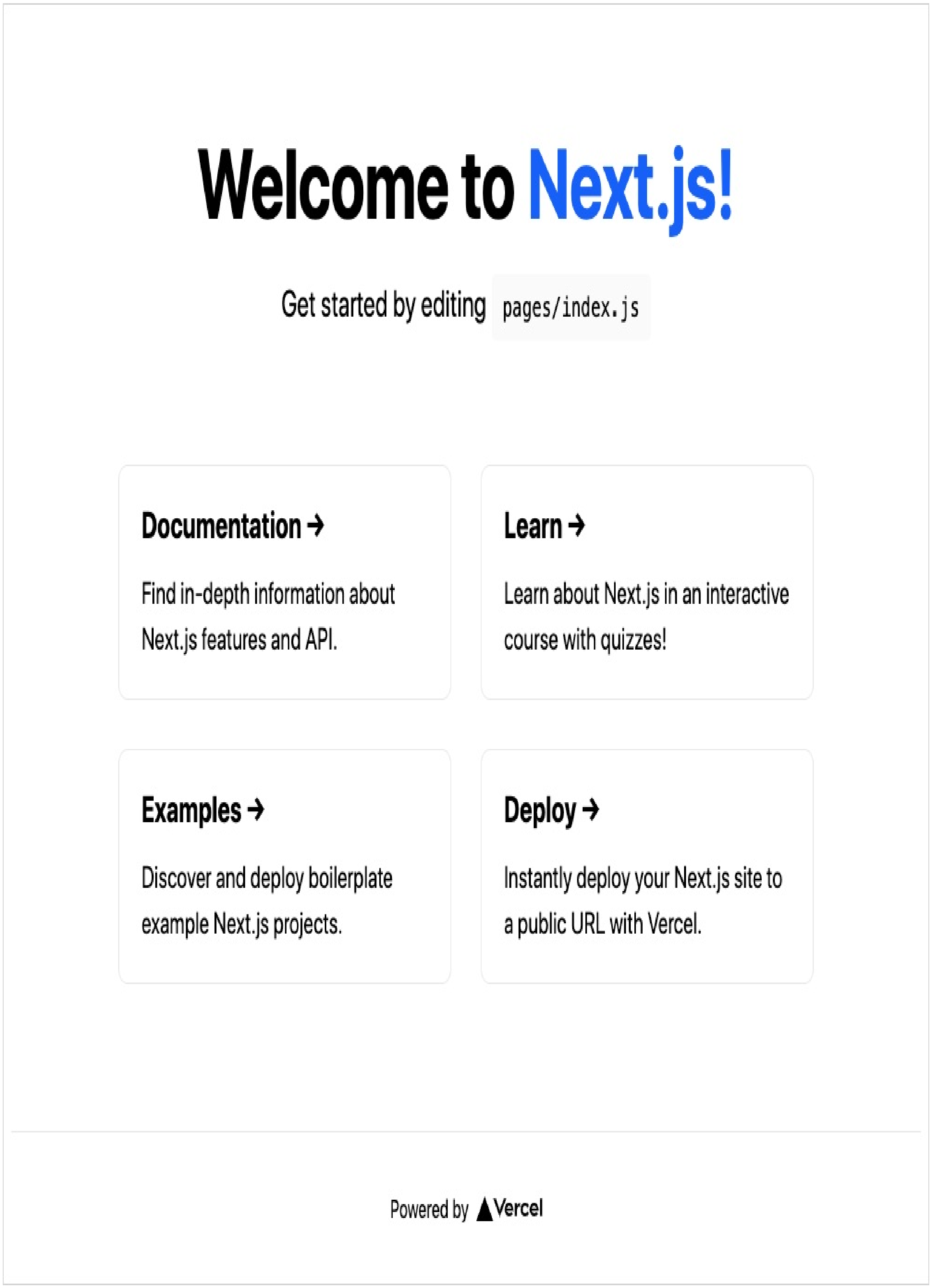
```
$ npm install -g create-next-app
```

Then, you can generate a new application:

```
$ create-next-app my-app
```

This will create a Next.js application in the `my-app` subdirectory. To start the app, run the `start` script:

```
$ cd my-app
$ npm run start
```



Welcome to Next.js!

Get started by editing `pages/index.js`

Documentation →

Find in-depth information about Next.js features and API.

Learn →

Learn about Next.js in an interactive course with quizzes!

Examples →

Discover and deploy boilerplate example Next.js projects.

Deploy →

Instantly deploy your Next.js site to a public URL with Vercel.

Powered by Vercel

Figure 1-6. A NextJS page running at http://localhost:3000

Next.js allows you to create pages without the need to manage any routing configuration. If you add a component script to the `pages/` folder, it will instantly become available through the server. For example, the `pages/index.js` component is used to generate the home page of the default application.

This approach is similar to the one taken by Gatsby[4] but is taken further in Next.js, to include server-side code as well.

Next.js applications usually include some API server code. This is unusual for React applications, which are often built quite separately from server code. But if you look inside `pages/api` you will find an example server end-point called `hello.js`:

```
// Next.js API route support: https://nextjs.org/docs/api-
routes/introduction

export default (req, res) => {
  res.statusCode = 200
  res.json({ name: 'John Doe' })
}
```

The routing which mounts this to the end-point `api/hello` happens automatically.

The code that you write in Next.js is automatically built into a hidden directory called *.next/*. This code can then be deployed to a service such as Next.js' own Vercel platform.

If you want, you generate a static build of your application with:

```
$ node_modules/.bin/next export
```

This will build your client code in a directory called `out/`. Each page of your site will be converted into a statically rendered HTML file, which will load very quickly in the browser. At the end of the page, it will load the JavaScript version in case the React code needs to modify the DOM.

WARNING

If you create an exported version of a Next.js application, it won't include any server-side APIs.

Next.js comes with a bunch of data-fetching options, which allow you to get data from static content, or via headless CMS sources.

Discussion

Next.js is in many ways similar to Gatsby. Its focus is on the speed of delivery, with a small amount of configuration. It's probably most useful for teams who will have a small amount of server code that they want to manage simply.

You can download the source for this recipe from the Github site.

1.5 Create a Tiny App with Preact

Problem

React applications can be large. It's quite easy to create a simple React application which is transpiled into bundles of JavaScript code that are several hundred Kbs in size. There are times when you might want to build an app with React-like features, but without having to download a large amount of JavaScript code.

Solution

If you want React-features, but don't want to pay the price of a React-size JavaScript bundle, you might want to consider using Preact.

Preact is *not* React. It is a separate library, but it is designed to be as close to React as possible while being much, much smaller.

The reason that the React framework is so big is because of the way it works. React components don't generate elements in the Document Object Model (DOM) of the browser directly. Instead, they build elements within a *virtual DOM*, which is then used to update the actual DOM at frequent intervals. Doing so allows basic DOM-rendering to be fast because the actual DOM only needs to be updated when there are real changes. However, it does have a downside.

React’s virtual DOM requires a lot of code to keep it up to date. It needs to manage an entire synthetic event model, which parallels the one in the browser. For this reason, the React framework is large and can take some time to download.

One way around this is to use techniques such as Server-Side Rendering[5], but SSR can be complex to configure. Sometimes, you just want to download a small amount of code. And that’s why Preact exists.

The Preact library, although similar to React, is tiny. At the time of writing, the main Preact library is around 4Kb. This is small enough that it’s possible to add React-like features to web pages in barely more code than is required to write native JavaScript.

Preact lets you choose how to use it: as a small JavaScript library included in a web page (the *no tools* approach) or as a full-blown JavaScript application.

The no-tools approach is really very basic. The core Preact library does not support JSX, and you will have no Babel support and so you will not be able to use modern JavaScript. This is an example web page using the raw Preact library:

```
<html>
    <head>
        <title>No Tools!</title>
        <script src="https://unpkg.com/preact?umd"></script>
    </head>
    <body>
        <h1>No Tools Preact App!</h1>
        <div id="root"></div>
        <script>
         var h = window.preact.h;
         var render = window.preact.render;

         var mount = document.getElementById('root');

         render(
             h('button',
               {
                   onClick: function() {
                       render(h('div', null, 'Hello'), mount);
                   }
               },
               'Click!'),
             mount
        );
      </script>
    </body>
</html>
```

This application will mount itself at the `<div/>` with id `root`, where it will display a button. When you click the button, it will replace the contents of the root `div` with the string `"Hello"`. This is about as basic as a Preact app can be.

You would rarely write an application in this way. In reality, you would create a simple build-chain that would, at the very least, support modern JavaScript.

In fact, Preact supports the entire spectrum of JavaScript applications. At the other extreme, you can create a full Preact application, with the `preact-cli`.

`preact-cli` is a tool for creating Preact projects and is analogous to tools like `create-react-app`. You can install `preact-cli` globally like this:

```
npm install -g preact-cli
```

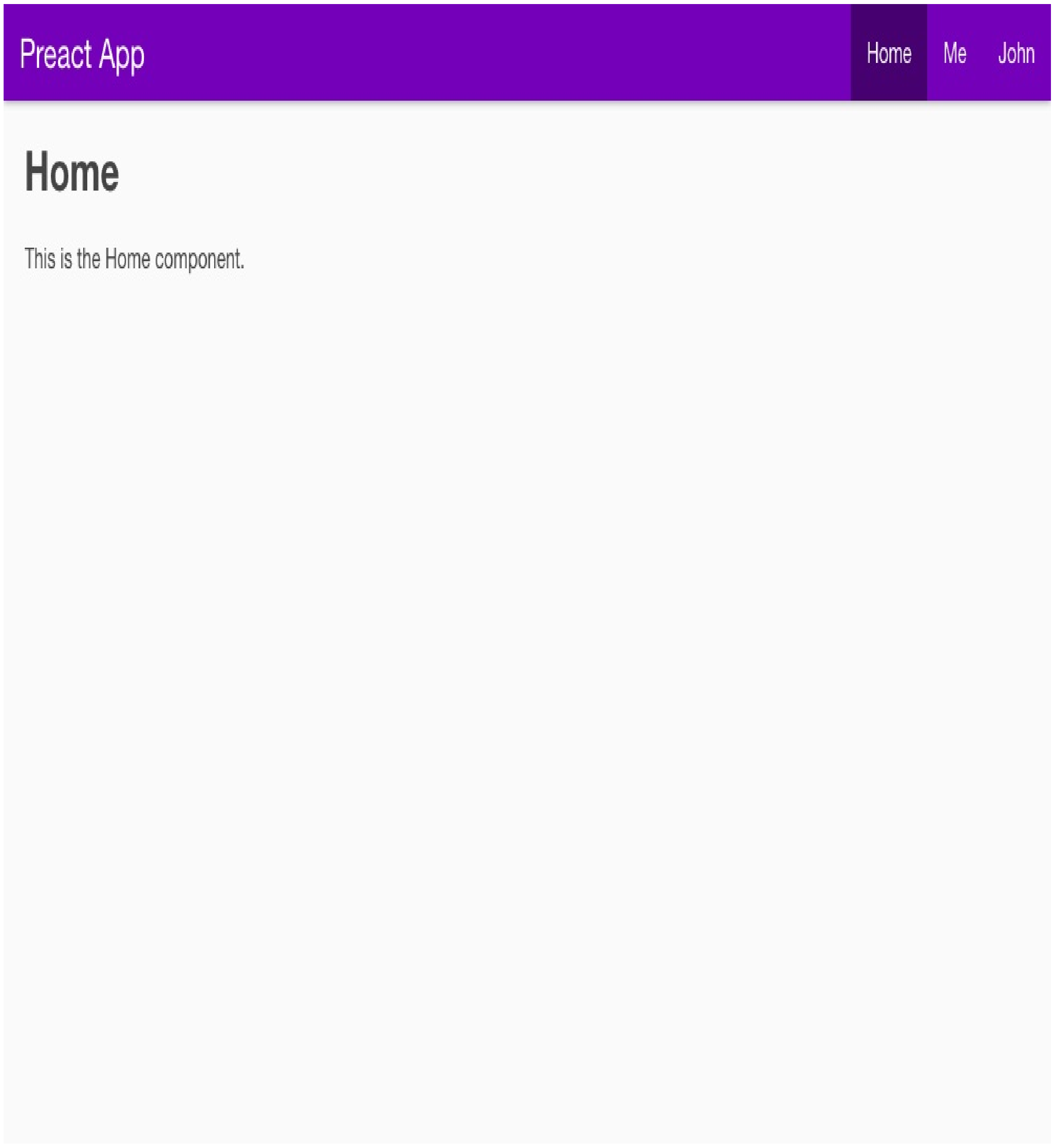
Then you can create a Preact application with:

```
preact create default my-app
```

This will create your new Preact application in the *my-app/* subdirectory. To start it, run the `dev` script:

```
cd my-app
npm run dev
```

This will start the server on port 8080, as shown in *figure 1-7*.



Home

This is the Home component.

Figure 1-7. A page from Preact

The server generates a web page, which calls back for a JavaScript bundle made from the code in *src/index.js*.

You now have a full-scale React-like application. The code inside the `Home` component, for example, looks very react-like, with full JSX support.

```
import { h } from 'preact';
import style from './style.css';

const Home = () => (
    <div class={style.home}>
        <h1>Home</h1>
        <p>This is the Home component.</p>
    </div>
);

export default Home;
```

The only significant difference from a standard React component, is that a function called `h` is imported from the `preact` library, instead of importing `React` from the `react` library.

However, the size of the application has increased: it is now a little over 300Kb. That's pretty large, but we are still in dev-mode. To see the real power of Preact, stop the dev server[6] and then run the `build` script:

```
npm run build
```

This will generate a static version of the application in the *build/* directory. First of all, this will have the advantage of creating a static copy of the front page, which will render very quickly. Secondly, it will remove all unused code from the application and shrink everything down. If you serve this built version of the app on a standard web server, the browser will transfer only about 50-60Kb when it's opened.

Discussion

Preact is a remarkable project. Despite working in a very different way to React, it provides virtually the same power, at a fraction of the size. And the fact that it can be used for anything between the lowliest inline code to a full-blown SPA means it is well worth considering if code-size is critical to your project.

You can find out more about Preact on the Preact web site.

You can download the source for the no-tools example and the larger Preact example from the Github site.

If you would to make Preact look even more like React, see the preact-compat library.

Finally, for a project that takes a similar approach to Preact, look at InfernoJS.

1.6 Build Libraries with NWB

Problem

Large organizations often develop several React applications at the same time. If you're a consultancy, you might create applications for multiple organizations. If you're a software house, you might create various applications that require the same look and feel, so you will probably want to build shared components that can be used across several applications.

When you create a component project, you need to create a directory structure, select a set of tools, choose a set of language features and create a build chain that can bundle your component in a deployable format. This can be just as tedious as manually creating a project for an entire React application.

Solution

The NWB toolkit can be used to create full React applications, but can also create projects that are specifically intended to create a single React component. In fact, it can also create components for use within Preact and InjernoJS projects, but we shall concentrate on React components here.

To create a new React component project, you will first need to install the `nwb` tool globally:

```
npm install -g nwb
```

You can then create a new project with the `nwb` command:

```
nwb new react-component my-component
```

NOTE

If instead of creating a single component, you want to create an entire NWB application, you can replace `react-component` in this command with `react-app`, `preact-app`, or `inferno-app` to create an application in the given framework. You can also use `vanilla-app` if you want to create a basic

JavaScript project without a framework.

When you run this command, you will be asked several questions about the type of library you want to build. You will be asked if you're going to build ECMAScript modules:

```
Creating a react-component project...
? Do you want to create an ES modules build? (Y/n)
```

This will allow you to build a version including an `export` statement, which can use by WebPack to decide whether or not the module is required in a client application. You will also be asked if you want to create a Universal Module Definition:

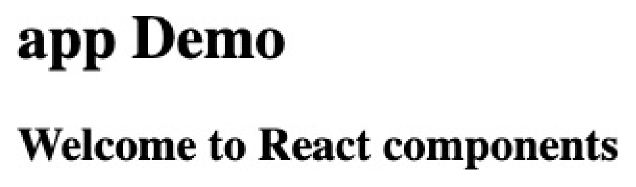
```
? Do you want to create a UMD build? (y/N)
```

That's useful if you want to include your component in a `<script/>` within a web page. For our example, we won't create a UMD build.

This will create an NWB component project inside the *my-component/* subdirectory. The project comes with a simple wrapper application that you can start with the `start` script:

```
cd my-component
npm run start
```

The demo application runs on port 3000, as shown in *figure 1-8*.



app Demo

Welcome to React components

Figure 1-8. An NWB component

The application will contain a single component defined in *src/index.js*.

```
import React, {Component} from 'react'

export default class extends Component {
  render() {
    return <div>
      <h2>Welcome to React components</h2>
    </div>
  }
}
```

You can now build the component as you would any React project. When you are reading to create a publishable version, simply type:

```
npm run build
```

This will now create a built version of your component in *lib/index.js*, which you can deploy to a repository for use within other projects.

Discussion

For further details on creating NWB components, see the NWB guide to developing components and libraries.

You can download the source for this recipe from the Github site.

1.7 Add React to Rails with Webpacker

Problem

The Rails framework was created before interactive JavaScript applications became popular. Rails applications follow a more traditional model for web application development, in which HTML pages are rendered on the server in response to browser requests. But sometimes you may want to include more interactive elements inside a Rails application.

Solution

The Webpacker library can be used to insert React applications into Rails generated web pages. To see how it works, let's first generate a Rails application which includes Webpacker:

```
$ rails new my-app --webpack=react
```

This will create a Rails application in a directory called *my-app/* that is preconfigured to run a Webpacker server. Before we start the application, let's go into it and generate an example page/controller:

```
$ cd my-app
$ rails generate controller Example index
```

That will generate this template page at *app/views/example/index.html.erb*:

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
```

Next, we need to create a small React application that we can insert into this page. Webpacker applications are inserted as *packs*: small JavaScript bundles, within Rails. We'll create a new pack in *app/javascript/packs/counter.js* containing a simple counter component:

```
import React, {useState} from 'react';
import ReactDOM from 'react-dom';

const Counter = props => {
  const [count, setCount] = useState(0);
  return <div className='Counter'>
    You have clicked the button {count} times.
    <button onClick={() => setCount(c => c + 1)}>Click!</button>
  </div>;
};

document.addEventListener('DOMContentLoaded', () => {
  ReactDOM.render(
    <Counter />,
    document.body.appendChild(document.createElement('div')),
  )
});
```

This application updates a counter every time the button is clicked.

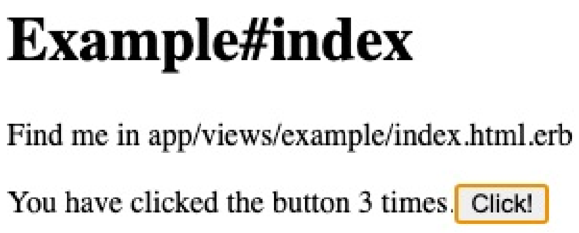
We can now insert the pack into the web page by adding a single line of code to the template page:

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
<%= javascript_pack_tag 'counter' %>
```

Finally, we can run the rails server on port 3000:

```
$ rails server
```

Which will show the page you can see in *figure 1-9*.



Example#index

Find me in app/views/example/index.html.erb

You have clicked the button 3 times.

Figure 1-9. A React app embedded in Rails

Discussion

Behind the scenes, as you have probably guessed, Webpacker transforms the application using a copy of webpack, which can be configured with the *app/config/webpacker.yml* config file.

Webpacker is intended to be used alongside Rails code, rather than as a replacement of it. It’s useful if your Rails application requires a small amount of additional interactivity.

To find out more about Webpacker on the Webpacker Github site.

You can download the source for this recipe from the Github site.

1.8 Create Custom Elements with Preact

Problem

There are sometimes circumstances where it is challenging to add React code into existing content. For example, in some CMS configurations, users are not allowed to insert additional JavaScript into the body of a page. In these cases, it would be useful to have some standardized way to insert JavaScript applications safely into a page.

Solution

Custom elements are a standard way of creating new HTML elements that can be used in a web page. In effect, they are a way of extending the HTML language by making more tags available to a user.

This recipe looks at how a lightweight framework like Preact can be used to create custom elements, which themselves can be served from a third-party service.

Let's begin by creating a new Preact application. This application will serve the custom element that we will be able to use elsewhere:[7]

```
$ preact create default my-element
```

Now we will change into the app's directory and add the `preact-custom-element` library to the project:

```
$ cd my-element
$ npm install preact-custom-element --save
```

The `preact-custom-element` library will allow us to register a new custom HTML element in a browser.

Next, we need to modify the *app/index.js* of the Preact project so that it registers a new custom element, which we will call *components/Converter/index.js*

```
import register from 'preact-custom-element';
import Converter from './components/Converter';

register(Converter, 'x-converter', ['currency']);
```

The `register` method tells the browser that we want to create a new custom HTML element called `<x-converter/>`, which has a single property called `currency` and which will be built using a component defined in *./components/Converter/index.js*, which we will define like this:

```
import {h} from 'preact';
import {useEffect, useState} from "preact/hooks";
import 'style/index.css';

const rates = {gbp: 0.81, eur: 0.92, jpy: 106.64};

export default ({currency = 'gbp'}) => {
    const [curr, setCurr] = useState(currency);
    const [amount, setAmount] = useState(0);

    useEffect(() => {
        setCurr(currency);
    }, [currency]);

    return <div className='Converter'>
        <p>
            <label htmlFor='currency'>Currency: </label>
            <select
                name='currency'
                value={curr}
                onChange={evt => setCurr(evt.target.value)}
            >
                {
                    Object.keys(rates).map(r => <option value={r}>{r}
</option>)
                }
            </select>
        </p>
        <p className='Converter-amount'>
            <label htmlFor='amount'>Amount: </label>
            <input
                name='amount'
                size={8}
                type="number"
                value={amount}
                onInput={evt =>
setAmount(parseFloat(evt.target.value))}
            />
        </p>
        <p>
            Cost:
            {((amount || 0) / rates[curr]).toLocaleString('en-US', {
                style: 'currency',
                currency: 'USD'
            })}
        </p>
    </div>
};
```

NOTE

To be compliant with the custom elements specification[8] we must choose a name for our element that begins with a lowercase letter, does not include any uppercase letters, and contains a hyphen. This ensures the name does not clash with any standard element name.

Our `Converter` component is a very simple currency converter, which in our example, is using a fixed set of exchange rates. If we now start our preact server:

```
$ npm run dev
```

The JavaScript for the custom element will be available at *http://localhost:8080/bundle.js*

In order to use this new custom element, let’s create a static web page somewhere with this HTML:

```
<html>
    <head>
        <script src="https://unpkg.com/babel-
polyfill/dist/polyfill.min.js"></script>
        <script
src="https://unpkg.com/@webcomponents/webcomponentsjs">
        </script>
        <!-- Replace this with the address of your custom element -->
        <script type="text/javascript"
src="http://localhost:8080/bundle.js"></script>
    </head>
    <body>
        <h1>Custom Web Element</h1>
        <div style="float: right; clear: both">
            <!-- This tag will insert the Preact app -->
            <x-converter currency="jpy"/>
        </div>
        <p>This page contains an example custom element called
            <code>&lt;x-converter/&gt;</code>,
            which is being served from a different location</p>
    </body>
</html>
```

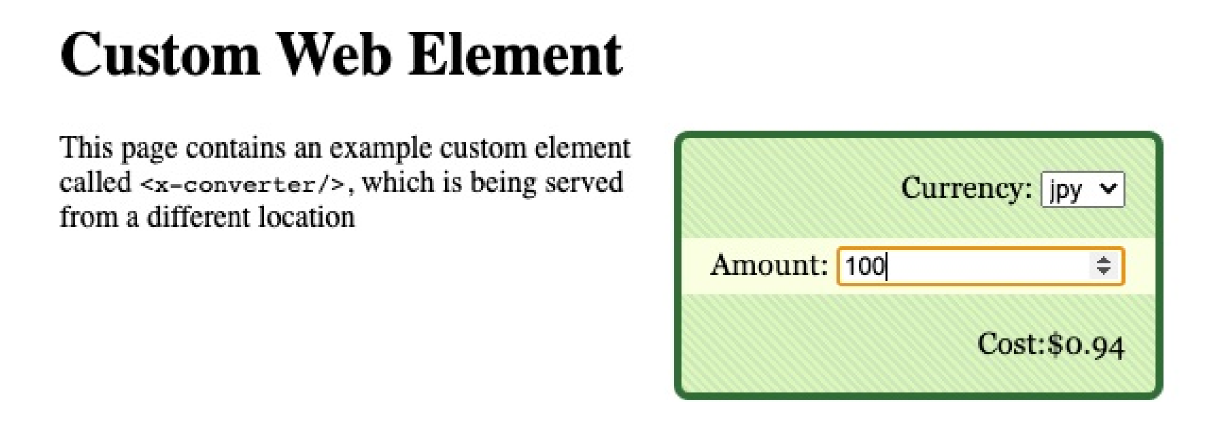
This web page is including the definition of the custom element in the final `<script/>` of the `<head/>` element. In order to make sure that the custom element is available across both new and old browsers, we also include a couple of shims from `unpkg.com`.

Now that the custom element code is included in the web page, we can insert `<x-converter/>` tags into the code, as if they are part of standard HTML. In our example, we are also passing a `currency` property, which will be passed through to the underlying Preact component.

WARNING

Custom element properties are passed to the underlying component with lowercase names, regardless of how they are defined in the HTML.

We can run this page through a web server, separate from the Preact server. The new custom element is shown in *figure 1-10*.



Custom Web Element

This page contains an example custom element called `<x-converter/>`, which is being served from a different location

Figure 1-10. The custom element embedded in a static page

Discussion

The custom element does not need to be served from the same server as the web page that uses it. This means that custom elements are a way of making embeddable widgets available for online services. Because they can be accessed from elsewhere, you might want to check the `Referer` header on any incoming request to the component, to prevent any unauthorized usage.

Our example is serving the custom element from Preact’s development server. For a production release, you would probably want to create a static build of the component, which can then be placed on any web server, and will likely be significantly smaller.[9]

Think about security. You may want to tie down which domains can access the element by checking the forward header

You can download the source for this recipe from the Github site.

1.9 Use Storybook for Component Development

Problem

React components are the stable building material of React applications. If they are written carefully, they can be re-used cleanly within and between React applications. But when you are building components, it is sometimes challenging to create the entire set of circumstances that the component will have to deal with. For example, in an asynchronous application, components might frequently be rendered with undefined properties. Will the component still render correctly? Will they show errors when they're misused?

But if you are building components as part of a complex application, it can be tough to create all of the situations with which your component will need to cope.

Also, if you have specialized UX developers working on your team, it can waste a lot of time if they have to navigate through an application to view the single component they have in development.

It would be useful if there was some way of displaying a component in isolation and passing it example sets of properties.

Solution

Storybook is a tool for displaying libraries of components in various states. It could be described as a gallery for components, but that's probably selling it short. In reality, Storybook is a tool for component development.

How do we add Storybook to a project? Let's begin by creating a React application with `create-react-app`:

```
$ npx create-react-app my-app
$ cd my-app
```

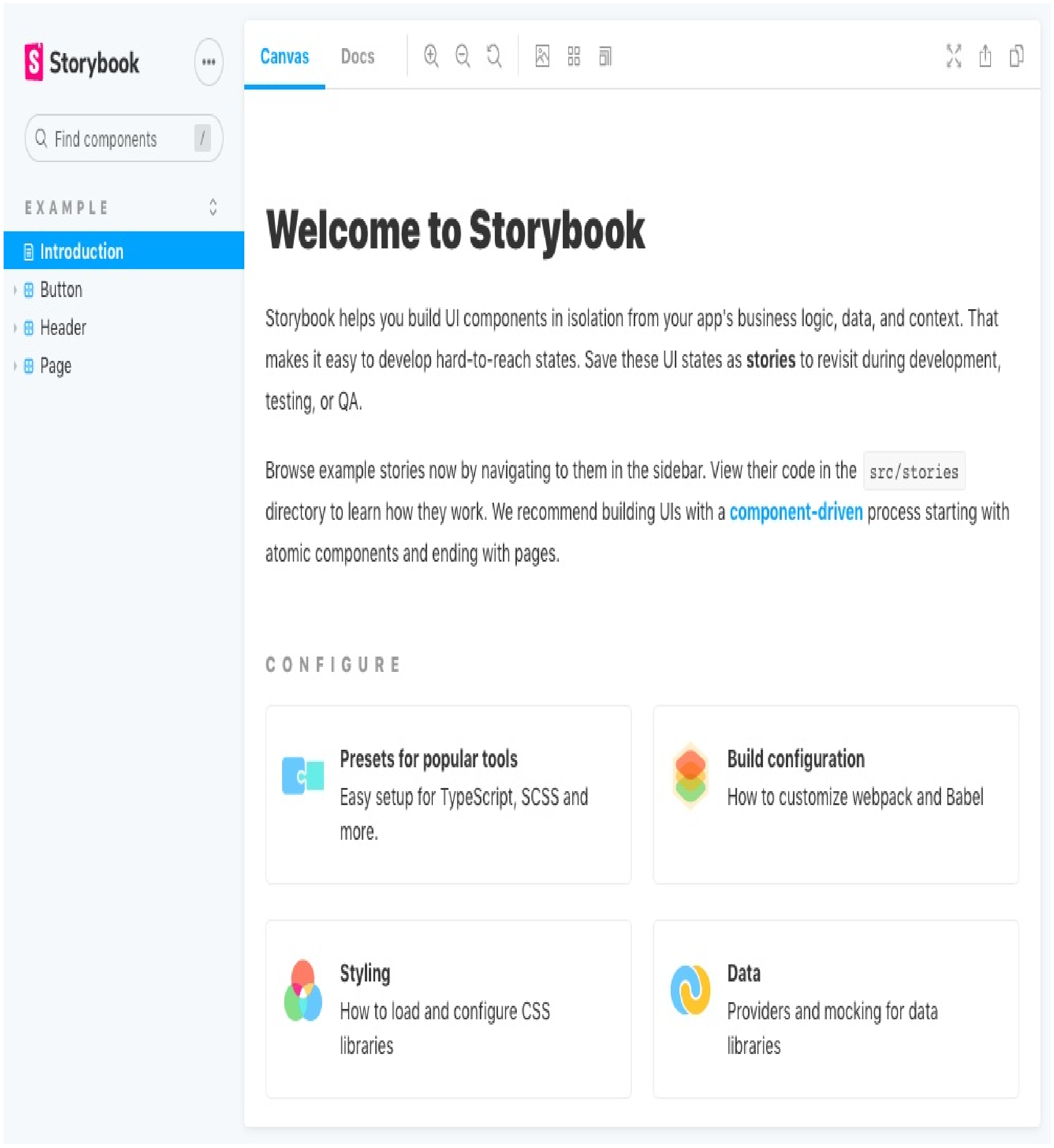
Now we can add Storybook to the project:

```
$ npx -p @storybook/cli sb init
```

And then start the Storybook server:

```
$ npm run storybook
```

Storybook runs its own server: in this case, we are running it on port 9000, as you can see in *figure 1-11*. When you are using Storybook, there is no need to run the actual React application.



Welcome to Storybook

Storybook helps you build UI components in isolation from your app's business logic, data, and context. That makes it easy to develop hard-to-reach states. Save these UI states as **stories** to revisit during development, testing, or QA.

Browse example stories now by navigating to them in the sidebar. View their code in the `src/stories` directory to learn how they work. We recommend building UIs with a component-driven process starting with atomic components and ending with pages.

CONFIGURE

Presets for popular tools
Easy setup for TypeScript, SCSS and more.

Build configuration
How to customize webpack and Babel

Styling
How to load and configure CSS libraries

Data
Providers and mocking for data libraries

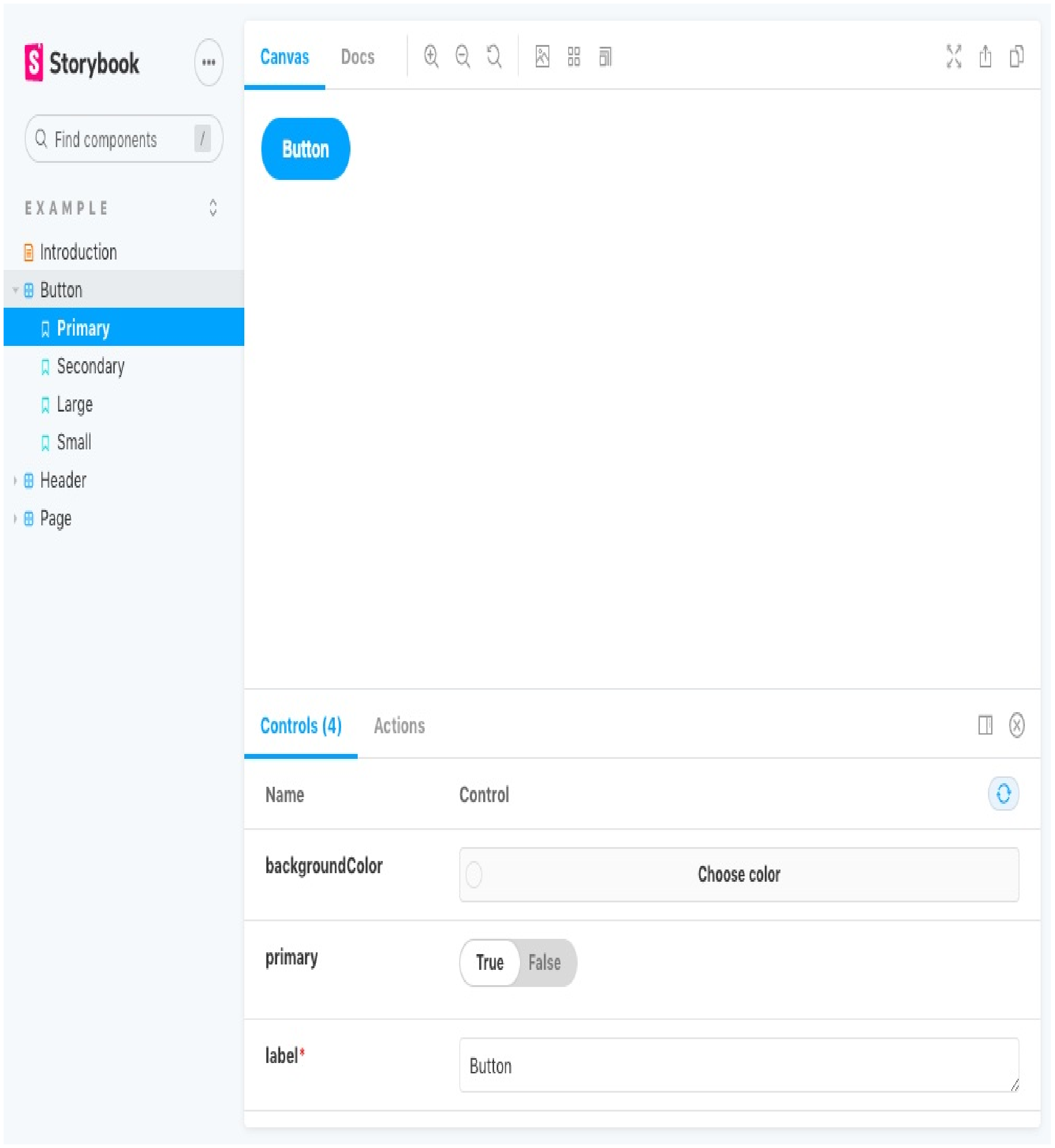
Figure 1-11. The welcome page in Storybook

Storybook calls a single component rendered with example properties a *story*. The default installation of Storybook generates sample stories in the *src/stories/* directory of the application. This is *src/stories/Button.stories.js*:

```
import React from 'react';

import { Button } from './Button';

export default {
  title: 'Example/Button',
  component: Button,
  argTypes: {
    backgroundColor: { control: 'color' },
  },
};

const Template = (args) => <Button {...args} />;

export const Primary = Template.bind({});
Primary.args = {
  primary: true,
  label: 'Button',
};

export const Secondary = Template.bind({});
Secondary.args = {
  label: 'Button',
};

export const Large = Template.bind({});
Large.args = {
  size: 'large',
  label: 'Button',
};

export const Small = Template.bind({});
Small.args = {
  size: 'small',
  label: 'Button',
};
```

Storybook watches for files named **.stories.js* in your source folder, and it doesn’t care where they are, so you are free to create them where you like. One typical pattern places the stories in a folder alongside the component they are showcasing. So if you copy the folder to a different application, you can take stories with it as a form of living documentation.

Figure 1-12 shows what *Button.stories.js* looks like inside Storybook.



Name	Control
backgroundColor	Choose color
primary	True False
label*	Button

Figure 1-12. An example story

Discussion

Despite its simple appearance, Storybook is actually a very productive development tool. It allows you to focus on one component at a time. Like a kind of visual unit test, it enables you to try out a component in a series of possible scenarios to check that it behaves appropriately.

Storybook also has a large selection of additional add-ons.

The add-ons allow you to:

- Add interactive controls for setting properties (*Knobs*)
- Include inline documentation for each story (*Docs*)
- Record snapshots of the HTML to test the impact of changes (*Storyshots*)

And many, many more.

For further information about Storybook see their web site.

You can download the source for this recipe from the Github site.

1.10 Test Your Code in a Browser with Cypress

Problem

Most React projects include a testing library. The most common is probably `@testing-library/react`, which comes bundled with `create-react-app`, or `enzyme` which is used by `preact`.

But nothing quite beats testing code inside a real browser, with all of the additional complications that that entails. Traditionally browser testing can be unstable and prone to frequent upgrade problems as browser drivers (such as `chromedriver`) have to be upgraded every time a browser is.

Add to that the issue of generating test data on a backend server and browser-based testing can be complex to set up and manage.

Solution

The Cypress testing framework avoids many of the downsides of traditional browser testing. It runs in a browser but avoids the need for an external webdriver tool. Instead, it communicates directly with a browser, like Chrome or Electron, over a network port and then injects JavaScript to run much of the test code.

Let’s create an application `create-react-app` to see how it works:

```
$ npx create-react-app my-app
```

Now let’s go into the app directory and install Cypress:

```
$ cd my-app
$ npm install cypress --save-dev
```

Before we run Cypress, we need to configure it so that it knows how to find our application. We can do this by editing the *cypress.json* file in the application directory, and tell if the URL of our app:

```
{
  "baseUrl": "http://localhost:3000/"
}
```

Once we have started the main application:

```
$ yarn start
```

We can then open Cypress:

```
$ npx cypress open
```

The first time you run Cypress it will install all of the dependencies it needs. We’ll now create a test in the *cypress/integration/* directory called *screenshot.js*. This will be a very simple test which opens the home page and takes a screenshot:

```
describe('screenshot', () => {
    it('should be able to take a screenshot', () => {
        cy.visit('/');
        cy.screenshot('frontpage');
    });
});
```

You’ll notice that tests are written in Jest format. Once you save the test, it will appear in the main Cypress window, shown in *figure 1-13*.

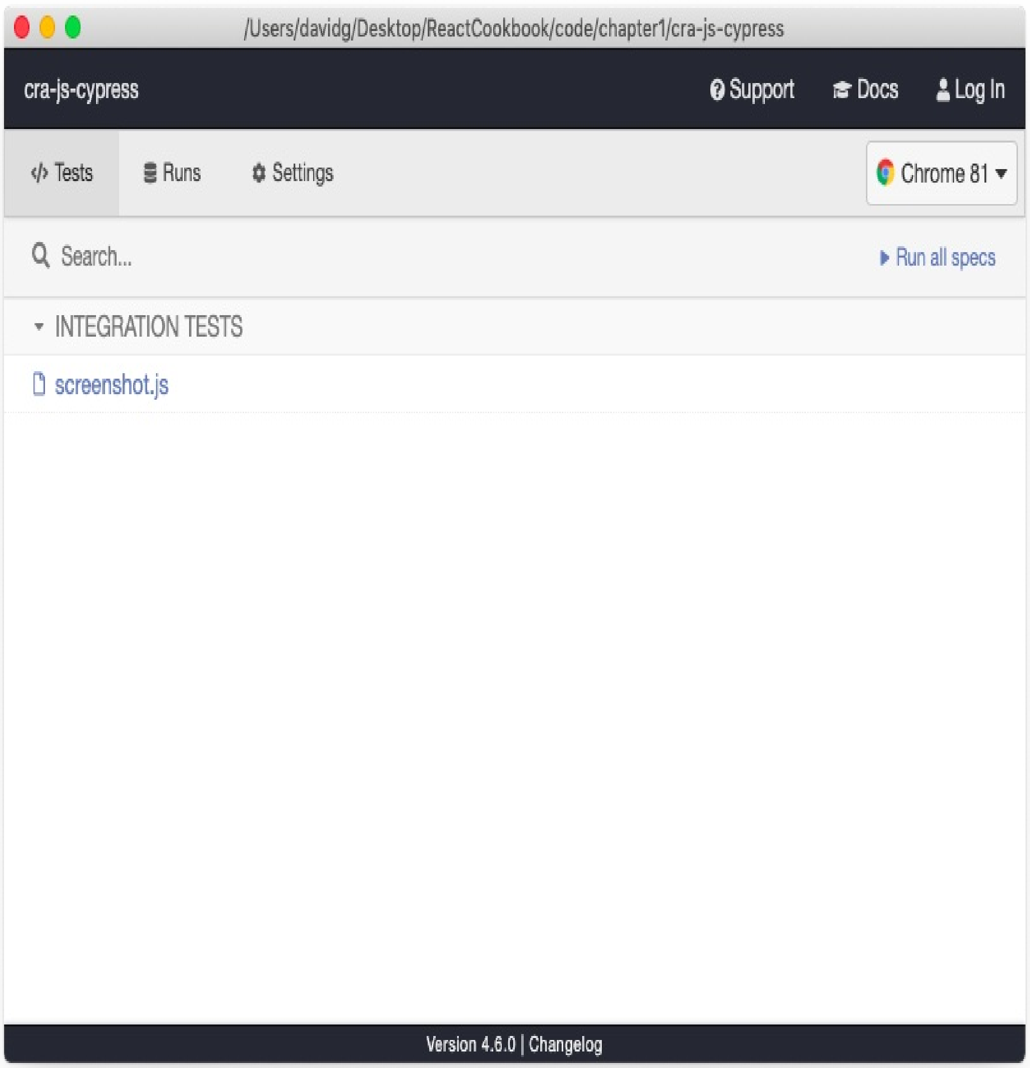


Figure 1-13. The Cypress window

If you double-click on the test, it will run it in a browser. The front page of the application will open, and a screenshot will be saved as *cypress/screenshots/screenshot.js/frontpage.png*.

Discussion

Here are some example commands you can perform with Cypress:

Command	Description
`cy.contains('Fred')`	Find the element containing `Fred`
`cy.get('.Norman').click()`	Click the element with class `Norman`
`cy.get('input').type('Hi!')`	Type `"Hi!"` into the input field
`cy.get('h1').scrollIntoView()`	Scroll the `<h1/>` into view

These are just the commands that interact with the web page. But Cypress has another trick up its sleeve. Cypress can also modify the code inside the browser to change the time (`cy.clock()`), the cookies (`cy.setCookie()`), the local-storage (`cy.clearLocalStorage`) and–most impressively–it can even fake requests and responses to an API server.

It does this by modifying the networking functions that are built into the browser so that this code:

```
cy.route("/api/server?*", [{some: 'Data'}])
```

Will cause any networking code in the application that makes a call to a server endpoint beginning `/api/server?...` will return the JSON array `[{some: 'Data'}]`.

This can completely change the way times can develop applications because it decouples the front-end development from the back end. The browser tests can specify what they want data they need without having to create a real server and database.

To learn more about Cypress, visit the documentation site.

You can download the source for this recipe from the Github site.

---

1 And yes, this means that Gatsby has TypeScript support built-in.

2 You can do this in most operating systems by pressing *CTRL-C*.

3 The name is intentionally similar to `create-react-app`. The maintainer of Razzle, Jared Palmer, lists `create-react-app` as one of the inspirations for Razzle.

[4] See recipe 2 in this chapter

[5] See the Gatsby and Razzle recipes elsewhere in this chapter.

[6] By pressing *CTRL=-C*

[7] For more information on creating Preact applications, see the Preact recipe earlier in this chapter.

[8] See the WHATWG specification for further details on custom elements and naming conventions.

[9] For further details on shrinking Preact downloads, see the Preact recipe earlier in this chapter.

Chapter 2. Routing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In this chapter, we are going to look at recipes using React routes and the *react-router-dom* library.

react-router-dom uses *declarative routing*: that means you treat routes as you would any other React component. React routes are obviously different from buttons, text fields, and blocks of text because they have no visual appearance. But in most other ways, they are very similar to those buttons and blocks of text. Routes live in the virtual DOM tree of components. They listen for changes in the current browser location and allow you to switch on and switch off parts of the interface. They are what give Single Page Applications the appearance of multi-page applications.

Used well, they can make your application feel like any other web site. Users will be able to bookmark sections of your application, as they might bookmark a page from Wikipedia. They can go backward and forwards in their browser history, and your interface will behave properly. If you are new to React, then it is well worth your time looking deeply into the power of routing.

2.1 Create Desktop/Mobile Interfaces with Responsive Routes

Problem

Most apps will be consumed by people on both mobile and laptop computers, which means you probably want your React application to work well across all screen sizes. This might involve relatively simple CSS changes to adjust the sizing of text and screen layout, as well as more substantial changes, which can give mobile and desktop users very different experiences when navigating around your site.

Our example application shows the names and addresses of a list of people. On a desktop, it looks like this:

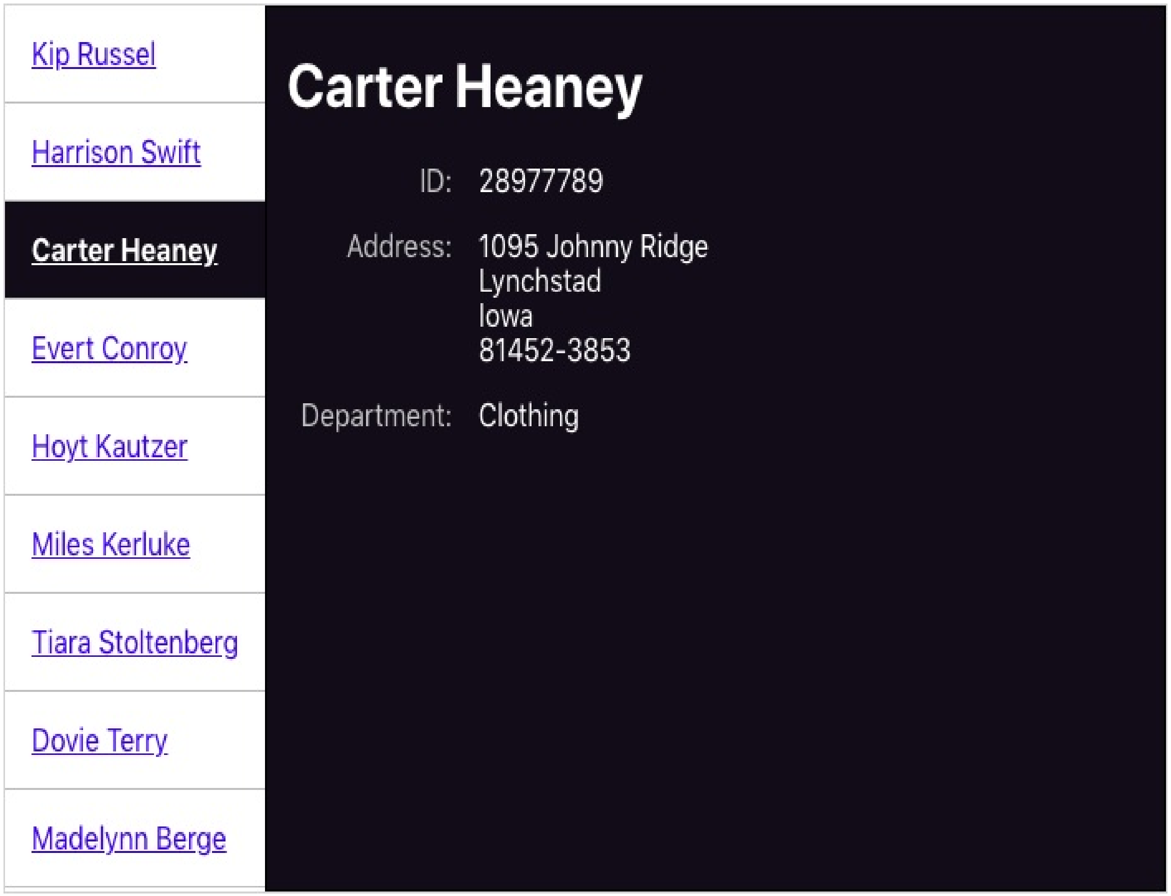


Figure 2-1. The desktop view of the app

But this won’t work very well on a mobile device, which might only have space to display either the list of people, or the details of one person, but not both.

What can we do in React to provide a custom navigation experience for both mobile and desktop users, without having to create two completely separate versions of the application?

Solution

We're going to use *responsive routes*. A responsive route changes according to the size of the user's display. Our existing application uses a single route for displaying the information for a person: */people/:id*

When you navigate to this route, the browser shows a page in *Figure 2-1*. All people are listed down the left-hand side, with the person matching *:id* highlighted. And the selected person's details are displayed on the right.

We're going to modify our application so that it will also cope with an additional route at */people*. Then we will make the routes responsive so that the user will see different things on different devices:

Route	Mobile	Desktop
/people	Show list of people	Redirect to /people/<some-id>
/people/<id>	Show details for <id>	Show list of people and details of <id>

What ingredients will we need to do this? First, we need to install *react-router-dom* if our application does not already have it:

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

The *react-router-dom* library allows us to coordinate the current location of the browser with the state of our application. Next, we will install the *react-media* library, which allows us to create React components that respond to changes in the size of the display screen.

```
npm install react-media --save
```

Now we're going to create a responsive *PeopleContainer* component that will manage the routes that we want to create. On small screens, our component will display *either* a list of people or the details of a single person. On large screens, it will show a combined view of a list of people on the left and the details of a single person on the right.

The *PeopleContainer* will use the *Media* component from *react-media*. The *Media* component performs a similar job to the CSS *@media* rule: it allows you to generate output for a specified range of screen sizes. The *Media* component accepts a *queries* property which allows you to specify a set of screen sizes. We're going to define a single screen size–small–that we'll use as the break between mobile and desktop screens:

```
<Media queries={{
        small: "(max-width: 700px)"
    }}>
  ...
</Media>
```

The *Media* component takes a single child component, which it expects to be a function. This function is given a *size* object which can be used to tell what the current screen size is. In our example, the *size* object will have a *small* attribute, which we can use to decide what other components to display:

```
<Media queries={{
        small: "(max-width: 700px)"
    }}>
  {
    size => size.small ? [SMALL SCREEN COMPONENTS] : [BIG SCREEN
COMPONENTS]
  }
</Media>
```

Before we look at the details of what code we are going to return for large and small screens, it's worth taking a look at how we will mount the *PeopleContainer* in our application. This is going to be our main *App* component:

```
import {BrowserRouter, Link, Route, Switch} from 'react-router-dom';
import PeopleContainer from "./PeopleContainer";

function App() {
  return (
    <BrowserRouter>
        <Switch>
            <Route path='/people'>
                <PeopleContainer/>
            </Route>
            <Link to='/people'>People</Link>
        </Switch>
    </BrowserRouter>
  );
}

export default App;
```

We are using the *BrowserRouter* from *react-router-dom*. This is the link between our code and the HTML5 history API in the browser. We need to wrap all of our routes in a *Router* so that they have access to the browser's current address.

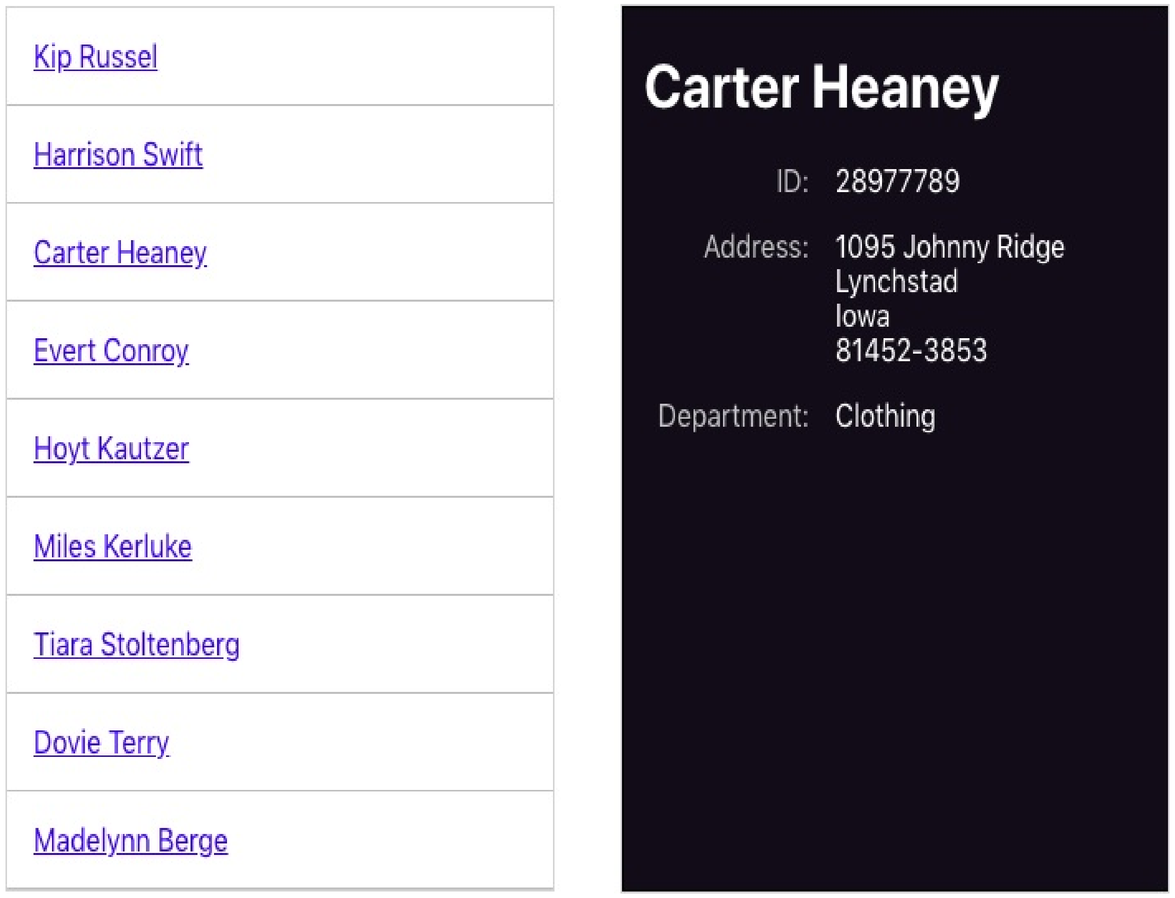
Inside the *BrowserRouter*, we have a *Switch*. The *Switch* looks at the components inside it, looking for a *Route* that matches the current location. Here we have a single *Route* matching paths that begin with */people*. If that's true, we display the *PeopleContainer*. If no route matches, we fall through to the end of the *Switch* and just render a *Link* to the */people* path. So when someone goes to the front page of the application, they only see a link to the *People* page.

So we know if we're in the *PeopleContainer*, we're already on a route that begins with */people/…*. If we're on a small screen, we need to either show a list of people or display the details of a single person, but not both. We can do this with *Switch*:

```
<Media queries={{
         small: "(max-width: 700px)"
    }}>
  {
    size => size.small ? [SMALL SCREEN COMPONENTS]
        <Switch>
          <Route path='/people/:id'>
            <Person/>
          </Route>
          <PeopleList/>
        </Switch>
        : [BIG SCREEN COMPONENTS]
  }
</Media>
```

On a small device, the *Media* component will call its child function with a value that means *size.small* is *true*. Our code will render a *Switch* that will show a *Person* component if the current path contains an *id*. Otherwise, the *Switch* will fail to match that *Route* and will instead render a *PeopleList*.

Ignoring the fact that we've yet to write the code for large screens, if we were to run this code right now on a mobile device, and hit the *People* link on the front page, we would navigate to */people* which could cause the application to render the *PeopleList* component. The *PeopleList* component[1] displays a set of links to people with paths of the form */people/id*. When someone selects a person from the list, our components are re-rendered, and this time *PeopleContainer* displays the details of a single person.


Kip Russel
Harrison Swift
Carter Heaney
Evert Conroy
Hoyt Kautzer
Miles Kerluke
Tiara Stoltenberg
Dovie Terry
Madelynn Berge

Carter Heaney

ID: 28977789

Address: 1095 Johnny Ridge
Lynchstad
Iowa
81452-3853

Department: Clothing

Figure 2-2. In mobile view: the list of people (left) which links to a person's details (right)

So far, so good. Now we need to make sure that our application still works for larger screens. We need to generate responsive routes in *PeopleContainer* for when *size.small* is *false*. If the current route is of the form */people/id* we can display the *PeopleList* component on the left, and the *Person* component on the right:

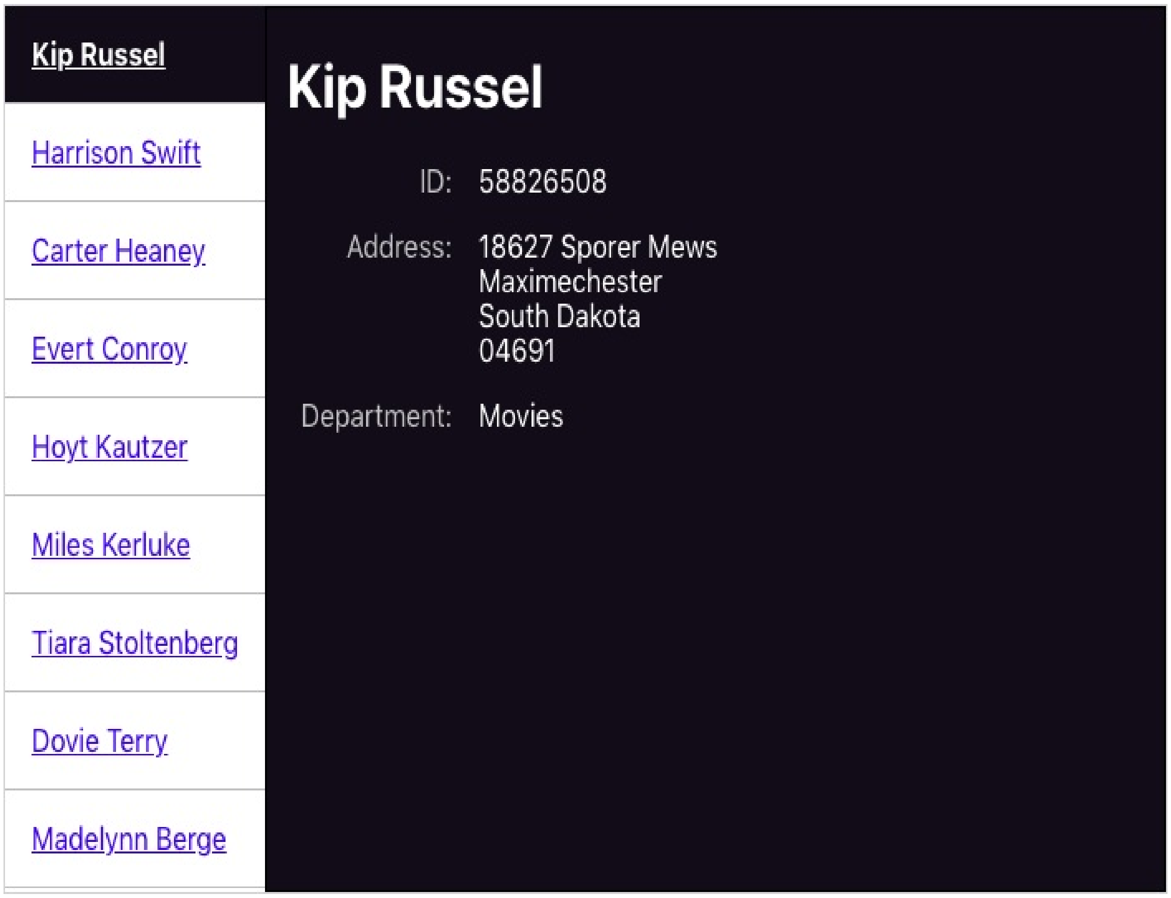
```
<div style={{display: 'flex'}}>
  <PeopleList/>
  <Person/>
</div>
```

Unfortunately that doesn't handle the case where the current path is */people*. For that, we need another *Switch* which will either display the details for a single person, or will *redirect* to */people/first-person-id* for the first person in the list of people.

```
<div style={{display: 'flex'}}>
    <PeopleList/>
    <Switch>
        <Route path='/people/:id'>
            <Person/>
        </Route>
        <Redirect to={`/people/${people[0].id}`}/>
    </Switch>
</div>
```

The *Redirect* component doesn't perform an **actual** browser redirect. It simply updates the current path to */people/first-person-id*, which causes the *PeopleContainer* to re-render. It's similar to making a call to *history.push()* in JavaScript, except it doesn't add an extra page to the browser history. If a person navigates to */people*, the browser will simply change it's location to */people/first-person-id*.

If we were now to go to */people* on a laptop or larger tablet, we would see the list of people next to the details for one person.



ID: 58826508

Address: 18627 Sporer Mews
Maximechester
South Dakota
04691

Department: Movies

Figure 2-3. What you see at http://localhost:3000/people on a large display

Here is the final version of our *PeopleContainer*

```
import Media from "react-media";
import {Redirect, Route, Switch} from "react-router-dom";
import Person from "./Person";
import PeopleList from "./PeopleList";
import people from './people';

export default () => {
    return <Media queries={{
        small: "(max-width: 700px)"
    }}>
        {
            size => size.small ?
                <Switch>
                    <Route path='/people/:id'>
                        <Person/>
                    </Route>
                    <PeopleList/>
```

```
                  </Switch>
                  :
                  <div style={{display: 'flex'}}>
                      <PeopleList/>
                      <Switch>
                          <Route path='/people/:id'>
                              <Person/>
                          </Route>
                          <Redirect to={`/people/${people[0].id}`}/>
                      </Switch>
                  </div>
          }
      </Media>;
  };
```

Discussion

Declarative routing inside components can seem an odd thing when you first meet it. If you've used a centralized routing model before, declarative routes may at first seem messy. They do, after all, spread the wiring of your application across who-knows-how-many components, rather than keeping it all neatly in a single file. Rather than creating clean components that know nothing of the outside world, you are suddenly giving the intimate knowledge of the paths used in the application, which might make them less portable.

However, responsive routes show the real power of declarative routing. If you're concerned about your components knowing too much about the paths in your application, consider extracting the path-strings into a shared file. That way, you will have the best of both worlds: components that modify their behavior based upon the current path, and a centralized set of path configurations.

You can download the source for this recipe from the Github site.

2.2 Move State into Routes to Create Deep Links

Problem

It is often useful to manage the internal state of a component with the route that the component is displayed. For example, this is a React component that displays two tabs of information: one for *people* and one for /offices.

```
import {useState} from "react";
import People from "./People";
import Offices from "./Offices";

import "./About.css";

export default () => {
    const [tabId, setTabId] = useState("people")

    return <div className='About'>
        <div className='About-tabs'>
            <div onClick={() => setTabId("people")}
                 className={tabId === "people" ? "About-tab active" :
"About-tab"}
            >
                People
            </div>
            <div onClick={() => setTabId("offices")}
                 className={tabId === "offices" ? "About-tab active" :
"About-tab"}
            >
                Offices
            </div>
        </div>
        {tabId === "people" && <People/>}
        {tabId === "offices" && <Offices/>}
    </div>;
}
```

When a user clicks on a tab, an internal *tabId* variable is updated, and the *People* of *Offices* component is displayed. This is what it looks like:

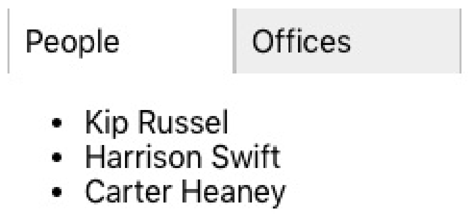


Figure 2-4. By default, the OldAbout component shows people details

What's the problem? The component works, but if we select the *Offices* tab and then refresh the page, the component resets to the *People* tab. Likewise, we can't bookmark the page when it's on the *Offices* tab. We can create a link anywhere else in the application which takes us directly to the *Offices*. Accessibility hardware is less likely to notice that the tabs are working as hyperlinks because they are not rendered in that way.

Solution

We are going to the *tabId* state from the component into the current browser location. So instead of rendering the component at */about* and then using *onClick* events to change internal state, we are instead going to have routes to */about/people* and */about/offices* which display one tab or the other. The tab selection will survive a browser refresh. We can bookmark the page on a given tab. We can jump to a given tab. And we make the tabs themselves real hyperlinks, which will be recognized as such by anyone navigating with a keyboard or screen reader.

What ingredients do we need? Just one: *react-router-dom*:

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

react-router-dom will allow us to synchronize the current browser URL with the components that we render on screen.

Our existing is already using *react-router-dom* to display the *OldAbout* component at path */oldabout* as you can see from this fragment of code from the *App.js* file:

```
<Switch>
    <Route path="/oldabout">
        <OldAbout/>
    </Route>
    <p>Choose an option</p>
</Switch>
```

You can see the full code for this file at the Github repository.

We’re going to create a new version of the *OldAbout* component, called *About* and we’re going to mount it at its own route:

```
<Switch>
    <Route path="/oldabout">
        <OldAbout/>
    </Route>
    <Route path="/about/:tabId?">
        <About/>
    </Route>
    <p>Choose an option</p>
</Switch>
```

This means that we will be able to open both versions of the code in the example application.

Our new version is going to appear to be virtually identical to the old component. We’ll extract the *tabId* from the component, and move it into the current path.

Setting the path of the *Route* to */about/:tabId?* means that both */about*, */about/offices*, */about/people* will all mount our component. The "*?*" indicates that the *tabId* parameter is optional.

We’ve now done the first part: we’ve put the state of the component into the path that displays it. Now we need to update the component to interact with the route, rather than an internal state variable.

In the *OldAbout* component, we had *onClick* listeners on each of the tabs:

```
<div onClick={() => setTabId("people")}
     className={tabId === "people" ? "About-tab active" : "About-tab"}
>
    People
</div>
<div onClick={() => setTabId("offices")}
     className={tabId === "offices" ? "About-tab active" : "About-
tab"}
>
    Offices
</div>
```

We’re going to convert these into *Link* components, going to */about/people* and */about/offices*. In fact, we’re going to convert them into *NavLink* components. A *NavLink* is like a link, except it has the ability to set an additional class-name, if the place it’s linking to is the current location. These means we don’t need the *className* logic in the original code:

```
<NavLink to="/about/people"
         className="About-tab"
```

```
          activeClassName="active">
    People
</NavLink>
<NavLink to="/about/offices"
          className="About-tab"
          activeClassName="active">
    Offices
</NavLink>
```

We no longer set the value of a *tabId* variable. We instead go to a new location with a new *tabId* value in the path.

But what do we do to read the *tabId* value? The *OldAbout* code displays the current tab contents like this:

```
{tabId === "people" && <People/>}
{tabId === "offices" && <Offices/>}
```

This code can be replaced with a *Switch* and a couple of *Route* components:

```
<Switch>
    <Route path='/about/people'>
        <People/>
    </Route>
    <Route path='/about/offices'>
        <Offices/>
    </Route>
</Switch>
```

We're now *almost* finished. There's just one step remaining: what to do if the path is simply */about* and contains no *tabId*.

The *OldAbout* sets a default value for *tabId* when it first creates the state:

```
const [tabId, setTabId] = useState("people")
```

We can achieve the same effect by adding a *Redirect* to the end of our *Switch*. If no *Route* matches the current path, we change the address to */about/people*. This will cause a re-render of the *About* component and the *People* tab will be selected by default:

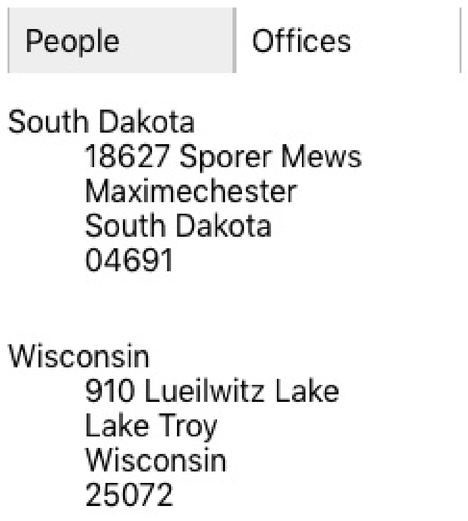
```
<Switch>
    <Route path='/about/people'>
```

```
        <People/>
    </Route>
    <Route path='/about/offices'>
        <Offices/>
    </Route>
    <Redirect to='/about/people'/>
</Switch>
```

This is our completed *About* component:

```
import {NavLink, Redirect, Route, Switch} from "react-router-dom";
import "./About.css";
import People from "./People";
import Offices from "./Offices";

export default () =>
    <div className='About'>
        <div className='About-tabs'>
            <NavLink to="/about/people"
                     className="About-tab"
                     activeClassName="active">
                People
            </NavLink>
            <NavLink to="/about/offices"
                     className="About-tab"
                     activeClassName="active">
                Offices
            </NavLink>
        </div>
        <Switch>
            <Route path='/about/people'>
                <People/>
            </Route>
            <Route path='/about/offices'>
                <Offices/>
            </Route>
            <Redirect to='/about/people'/>
        </Switch>
    </div>;
```

We no longer need an internal *tabId* variable, and we now have a purely declarative component.



South Dakota
18627 Sporer Mews
Maximechester
South Dakota
04691

Wisconsin
910 Lueilwitz Lake
Lake Troy
Wisconsin
25072

Figure 2-5. Going to http://localhost/about/offices with the new component

Discussion

Moving state out your components and into the address bar can simplify your code, but this is merely a fortunate side-effect. The real value is that your application starts to behave less like an application, and more like a web site. Pages can be bookmarked, and the browser's *Back* and *Forward* buttons work correctly. Managing more state in routes is not an abstract design decision, it's a way of making your application less surprising to users.

You can download the source for this recipe from the Github site.

2.3 Use MemoryRouter for Unit Testing

Problem

We use routes in React applications so that we make more of the facilities of the browser. We can bookmark pages, we can create deep links into an app, and we can go backward and forwards in history.

However, once we use routes, we make the component dependent upon something outside itself: the browser location. That might not seem like too big an issue, but it does have consequences.

Let’s say we want to unit test a route-aware component. As an example, let’s create a unit test for the *About* component we build in recipe 2 of this chapter. This is what a unit test might look like.[2]

```
describe('About component', () => {
    it('should show people', () => {
        render (<About/>);
        expect(screen.getByText('Kip Russel')).toBeInTheDocument();
    });
});
```

This unit test renders the component and then checks that it can find the name “Kip Russel” appearing in the output. When we run this test, we get the following error:

```
console.error node_modules/jsdom/lib/jsdom/virtual-console.js:29
    Error: Uncaught [Error: Invariant failed: You should not use
<NavLink> outside a <Router>]
```

This happened because a *NavLink* could not find a *Router* higher in the component tree. That means we need to wrap the component in a *Router* before we test it.

Also, we might want to write a unit test that checks that the *About* performs behaves correctly when it’s mounted on a specific route. Even if we provide some sort of *Router* component, how will we be able to fake a particular route?

It’s not just an issue with unit tests. If we’re using a library tool like Storybook[3], we might want to show an example of how a component appears when it is mounted on a particular.

What we need is something *like* a real browser router, but one which allows us to specify its behavior.

Solution

The *react-router-dom* library provides just such a router: *MemoryRouter*. The *MemoryRouter* appears to the outside world, just like *BrowserRouter*. The difference is that while the *BrowserRouter* is an interface to the underlying browser history API, the *MemoryRouter* has no dependency upon the browser at all. It can keep track of the current location, it can go backward and forwards in history, but it does it all through simple memory structures.

Let’s take another look at that failing unit test. Instead of just rendering the *About* component, let’s wrap it in a *MemoryRouter*:

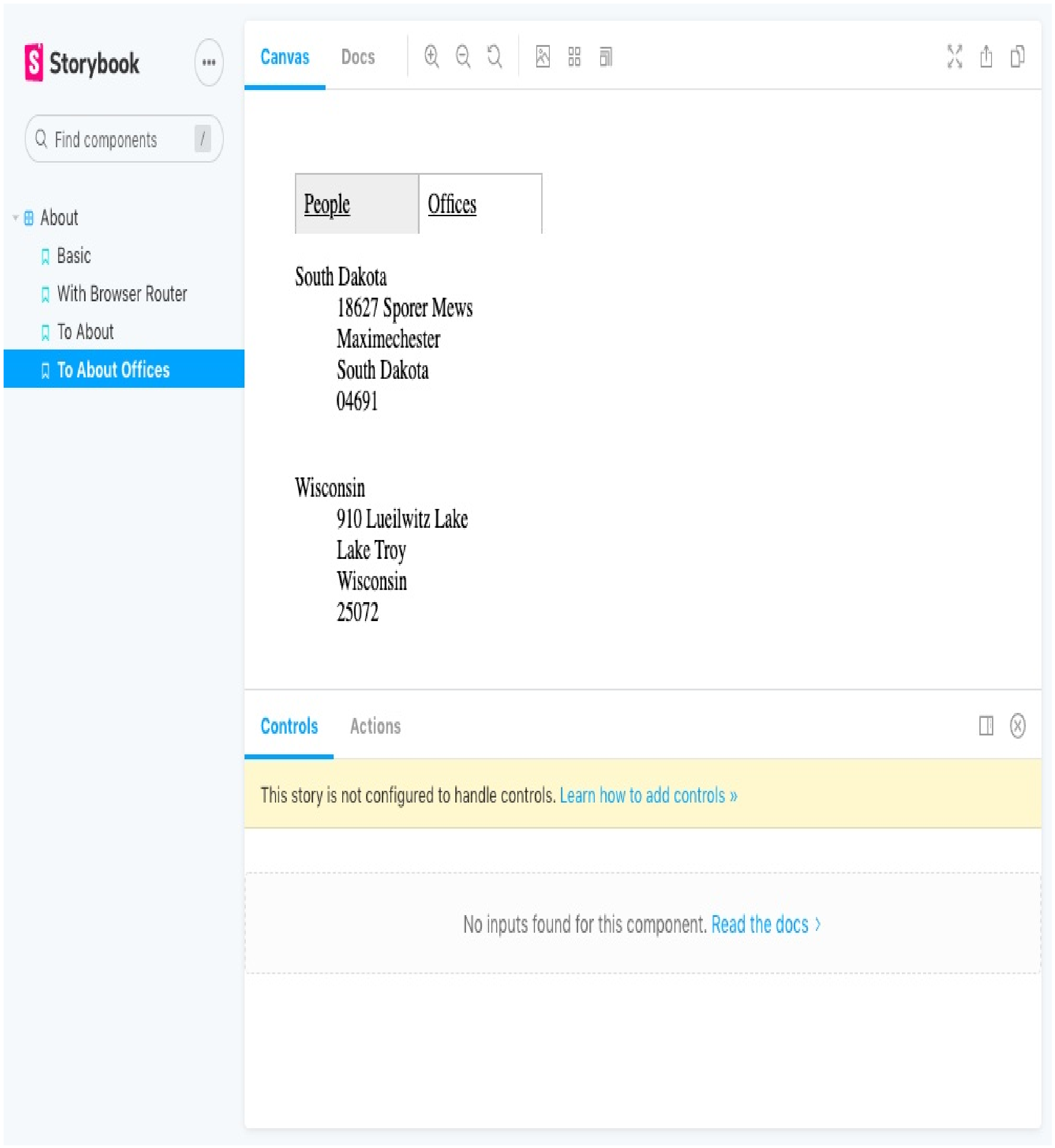
```
describe('About component', () => {
    it('should show people', () => {
        render (<MemoryRouter><About/></MemoryRouter>);

        expect(screen.getByText('Kip Russel')).toBeInTheDocument();
    });
});
```

Now, when we run the test, it works. That’s because the *MemoryRouter* injects a mocked-up version of the API into the context. That makes it available to all of its child components. When the *About* component wants to render a *Link* or a *Route*, it’s now able to because history is available in the context.

But the *MemoryRouter* has an additional advantage. Because it’s faking the browser history API, it can be given a completely fake history, using the *initialEntries* property. The *initialEntries* property should be set to an array of history entries. If you pass a single value array, it will be interpreted as the current location. That allows you to write unit tests that check for component behavior when it’s mounted on a given route:

```
describe('About component', () => {
    it('should show offices if in route', () => {
        render(<MemoryRouter initialEntries={[
            {pathname: '/about/offices'},
        ]}>
            <About/>
        </MemoryRouter>);

        expect(screen.getByText('South Dakota')).toBeInTheDocument();
    });
});
```

We can actually use a real *BrowserRouter* inside Storybook because we’re in a real browser. The *MemoryRouter* still has the advantage of being able to fake a current location, as we do in the *ToAboutOffices* story:



People Offices

South Dakota
18627 Sporer Mews
Maximechester
South Dakota
04691

Wisconsin
910 Lueilwitz Lake
Lake Troy
Wisconsin
25072

This story is not configured to handle controls. Learn how to add controls »

No inputs found for this component. Read the docs ›

Figure 2-6. Using MemoryRouter we can fake the /about/offices route

Discussion

Routers let you separate the details of *where* you want to go, from *how* you're going to get there. In this recipe, we see one advantage of this separation: we can create a fake browser location to examine component behavior on different routes. It is this separation that allows you to change the way that links are followed without breaking your application. If you convert your Single Page Application to a Server-Side Rendered application, you swap your *BrowserRouter* for a *StaticRouter*. The links that used to make calls into the browser’s history API will now become native hyperlinks, that cause the browser to make native page loads. Routers are an excellent example of the advantages of splitting policy (what you want to do) from mechanisms (how you’re going to do it).

You can download the source for this recipe from the Github site.

2.4 Use Prompt for Page Exit Confirmations

Problem

Sometimes you need to ask a user to confirm that they want to leave a page if they’re in the middle editing something. This seemingly simple task can be complicated because it relies on spotting when the user presses the back button and then finding a way to intercept the move back through history and potentially canceling it.

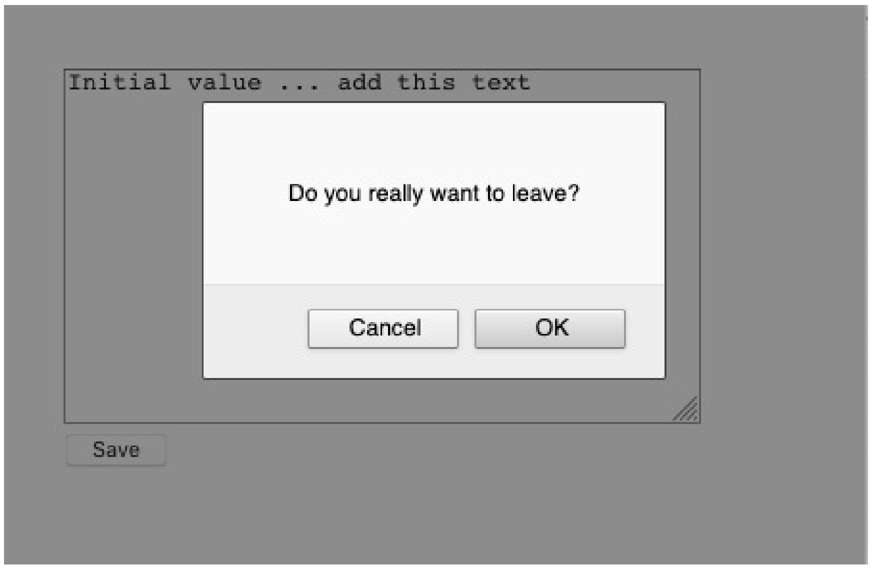


Figure 2-7. What if you want someone to confirm that they wish to leave a page?

What if there are several pages in the application which need the same feature?

Is there a simple way to create this feature across any component that needs it?

Solution

The *react-router-dom* library includes a component called *Prompt*, which is explicitly designed to get users to confirm that they wish to leave a page.

The only ingredient we really need for this recipe is the *react-router-dom* library itself:

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

Let’s say we are going to have a component called *Important* mounted at */important*, which allows a user to edit a piece of text.

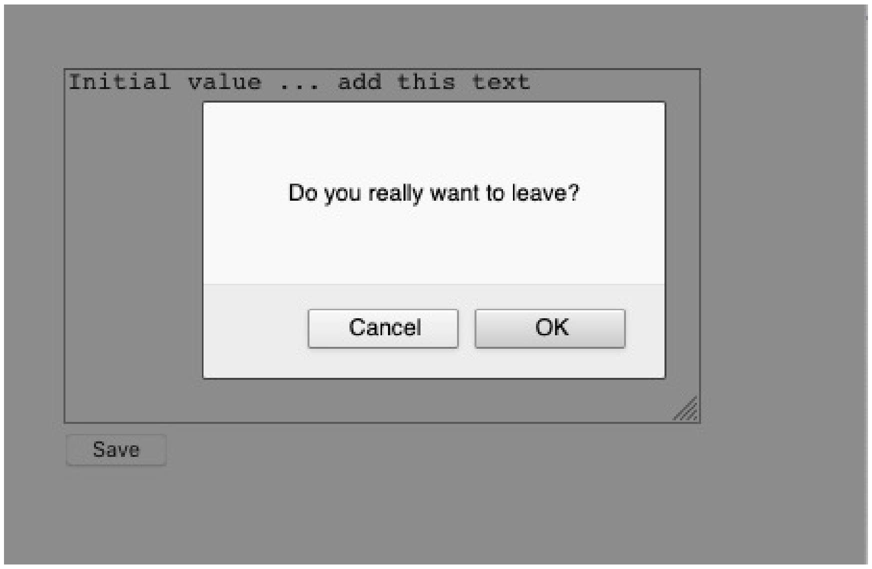
```
import React, {useEffect, useState} from "react";

export default () => {
    let initialValue = "Initial value";

    const [data, setData] = useState(initialValue);
    const [dirty, setDirty] = useState(false);

    useEffect(() => {
        if (data !== initialValue) {
            setDirty(true);
        }
    }, [data, initialValue]);

    return <div className='Important'>
        <textarea onChange={evt => setData(evt.target.value)}
                  cols={40} rows={12}>
            {data}
        </textarea>
        <br/>
        <button onClick={() => setDirty(false)}
                disabled={!dirty}>Save</button>
    </div>;
}
```

Important is already tracking whether the text in the *textarea* has changed from the original value. If the text is different, the value is *dirty* is *true*. How do we ask the user to confirm they want to leave the page if they press the *Back* button when *dirty* is *true*?

We add in a *Prompt* component:

```
return <div className='Important'>
    <textarea onChange={evt => setData(evt.target.value)}
              cols={40} rows={12}>
        {data}
    </textarea>
    <br/>
    <button onClick={() => setDirty(false)}
            disabled={!dirty}>Save</button>
    <Prompt
        when={dirty}
        message={() => "Do you really want to leave?"}
        />
</div>;
```

If the user edits the text and then hits the *Back* button, the *Prompt* appears:


Initial value ... add this text

Do you really want to leave?

Figure 2-8. The Prompt asks the user to confirm they want to leave

Adding the confirmation is very simple. But the default prompt interface is a simple JavaScript dialog. It would be useful if we could decide for ourselves how we want the user to confirm they're leaving.

To demonstrate how we can do this, let's add in the Material-UI component library to the application:

```
npm install '@material-ui/core' --save
```

The Material-UI library is a React implementation of Google’s Material Design standard. We’ll use it as an example of how to replace the standard *Prompt* interface with something more customized.

The *Prompt* component does not actually any UI itself. Instead, when *Prompt* is rendered, it asks the current *Router* object to show the confirmation. By default, *BrowserRouter* shows the default JavaScript dialog, but you can replace this with your own code.

When the *BrowserRouter* is added to the component tree, we can pass it a property called *getUserConfirmation*:

```
<div className="App">
    <BrowserRouter
        getUserConfirmation={(message, callback) => {
          // Custom code goes here
        }}
    >
        <Switch>
            <Route path='/important'>
                <Important/>
            </Route>
        </Switch>
    </BrowserRouter>
</div>
```

The *getUserConfirmation* property should be set to a function that accepts two parameters: the message that should be presented to the user, and a callback function.

When the user presses the *Back* button, the *Prompt* component will run *getUserConfirmation*, and then wait for the callback function to be called with the value *true* or *false*.

The callback function allows use to return the user’s response asynchronously. The *Prompt* component will wait while we ask the user what want to do. That allows use to create out own custom interface.

Let’s create a custom Material-UI dialog called *Alert*. We’ll show this instead of the default JavaScript modal:

```
import Button from '@material-ui/core/Button';
import Dialog from '@material-ui/core/Dialog';
import DialogActions from '@material-ui/core/DialogActions';
import DialogContent from '@material-ui/core/DialogContent';
import DialogContentText from '@material-ui/core/DialogContentText';
import DialogTitle from '@material-ui/core/DialogTitle';

export default ({open, title, message, onOK, onCancel}) => {
    return <Dialog
                open={open}
                onClose={onCancel}
                aria-labelledby="alert-dialog-title"
                aria-describedby="alert-dialog-description"
            >
                <DialogTitle id="alert-dialog-title">{title}
</DialogTitle>
                <DialogContent>
                    <DialogContentText id="alert-dialog-description">
                        {message}
                    </DialogContentText>
                </DialogContent>
                <DialogActions>
                    <Button onClick={onCancel} color="primary">
                        Cancel
                    </Button>
                    <Button onClick={onOK} color="primary" autoFocus>
                        OK
                    </Button>
                </DialogActions>
            </Dialog>;
}
```

Of course, there is no reason why we need to display a dialog. We could show a countdown timer or a snackbar message. We could even automatically save the user's changes for them. But we use the custom *Alert* dialog in this case.

How will use the *Alert* component in our interface? The first thing we'll need to do is create our own *getUserConfirmation* function. We'll store the message and the callback function, and then set a boolean value saying that we want to open the *Alert* dialog:

```
const [confirmOpen, setConfirmOpen] = useState(false);
const [confirmMessage, setConfirmMessage] = useState();
const [confirmCallback, setConfirmCallback] = useState();

return (
    <div className="App">
        <BrowserRouter
            getUserConfirmation={(message, callback) => {
```

```
                        setConfirmMessage(message);
                        // Use this setter form because callback is a function
                        setConfirmCallback(() => callback);
                        setConfirmOpen(true);
                    }}
                >
     .....
```

It’s worth noting that when we store the callback function, we use *setConfirmCallback(() ⇒ callback)* instead of simply writing *setConfirmCallback(callback)*. That’s because the setters returned by the *useState* hook will execute any function passed to them, rather than store them.

We can then use the values of *confirmMessage*, *confirmCallback*, and *confirmOpen* to render the *Alert* in the interface.

This is the complete *App.js* file:

```
import {useState} from 'react';
import './App.css';
import {BrowserRouter, Link, Route, Switch} from "react-router-dom";
import Important from "./Important";
import Alert from './Alert';

function App() {
    const [confirmOpen, setConfirmOpen] = useState(false);
    const [confirmMessage, setConfirmMessage] = useState();
    const [confirmCallback, setConfirmCallback] = useState();

    return (
        <div className="App">
            <BrowserRouter
                getUserConfirmation={(message, callback) => {
                    setConfirmMessage(message);
                    // Use this setter form because callback is a
function
                    setConfirmCallback(() => callback);
                    setConfirmOpen(true);
                }}
            >
                <Alert open={confirmOpen}
                       title='Leave page?'
                       message={confirmMessage}
                       onOK={() => {
                           confirmCallback(true);
                           setConfirmOpen(false);
                       }}
```

```
                        onCancel={() => {
                            confirmCallback(false);
                            setConfirmOpen(false);
                        }}
                    />
                    <Switch>
                        <Route path='/important'>
                            <Important/>
                        </Route>
                        <div>
                            <h1>Home page</h1>
                            <Link to='/important'>Go to important
page</Link>
                        </div>
                    </Switch>
                </BrowserRouter>
            </div>
    );
}

export default App;
```

Now when a user backs out of an edit, they see the custom dialog.

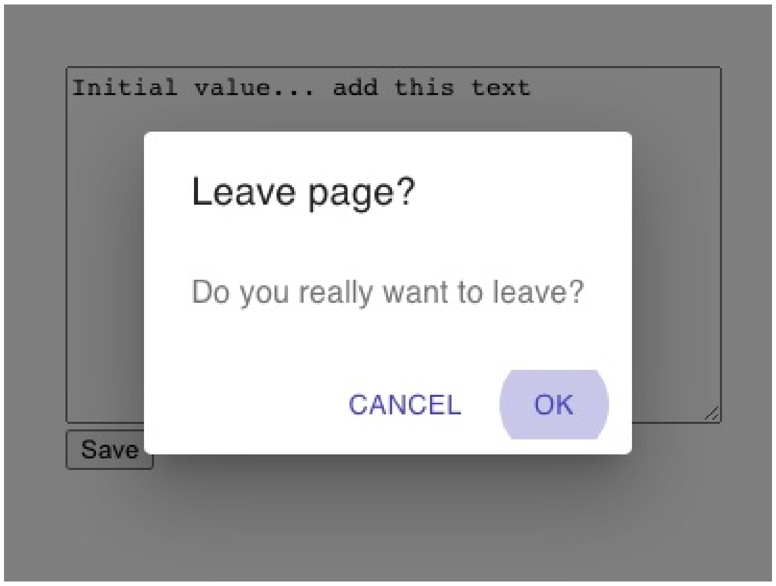


Figure 2-9. The custom Alert appears when the user presses the back button

Discussion

In this recipe, we have re-implemented the *Prompt* modal using a component library, but you don't need to be limited to just replacing one dialog box with another. There is no reason why, if someone leaves a page, that you couldn't do something else: such as store the work-in-progress somewhere so that they could return to it later. The asynchronous nature of the *getUserConfirmation* function allows this flexibility. It's another example of how *react-router-dom* abstracts away a cross-cutting concern.

You can download the source for this recipe from the Github site.

2.5 Add Page Transitions With react-transition-group

Problem

Native and desktop applications often use animation to visually connect different elements together. If you press an item in a list, it expands to show you the details. Swiping left or right can be used to indicate whether a user accepts or rejects an option.

Animations, therefore, are often used to indicate a change in location. They zoom in on the details. They take you to the next person on the list.

Changing locations in a React application. But how can we animate when we move from one location to another?

Solution

For this recipe, we're going to need the *react-router-dom* library and the *react-transition-group* library.

```
npm install react-router-dom --save
npm install react-transition-group --save
```

We're going to animate the *About* component that we've used previously[4]. The *About* component has two tabs called *People* and *Offices*, which are displayed for routes */about/people* and */about/offices*.

When someone clicks on one of the tabs, we're going to fade-out the content of the old tab and then fade in the content of the new tab. Although we're using a fade, there's no reason why we couldn't use a more complex animation, such as sliding the tab contents left or right[5]. However, a simple fade animation will more clearly demonstrate how it works.

Inside the *About* component, the tab contents are rendered by *People* and *Offices* components within distinct routes:

```
import {NavLink, Redirect, Route, Switch} from "react-router-dom";
import "./About.css";
import People from "./People";
import Offices from "./Offices";

export default () =>
    <div className='About'>
        <div className='About-tabs'>
            <NavLink to="/about/people"
                     className="About-tab"
                     activeClassName="active">
                People
            </NavLink>
            <NavLink to="/about/offices"
                     className="About-tab"
                     activeClassName="active">
                Offices
            </NavLink>
        </div>
        <Switch>
            <Route path='/about/people'>
                <People/>
            </Route>
            <Route path='/about/offices'>
                <Offices/>
            </Route>
            <Redirect to='/about/people'/>
        </Switch>
    </div>;
```

We need to animate the components inside the *Switch* component. We'll need two things to do this:

- Something to track when the location has changed
- Something to animate the tab contents when that happens

How do we know when the location has changed? We can get the current location from the *useLocation* hook from *react-router-dom*:

```
const location = useLocation();
```

Now onto the more complex task: the animation itself. What follows is actually quite a complex sequence of events, but it is worth taking the time to understand it.

When we are animating from one component to another, we need to keep both components on the page. As the *Offices* component fades out, the *People* component fades in.[6] This is done by keeping both components in a *transition group*. We can create a transition group by wrapping our animation in a *TransitionGroup* component. We also need a *CSSTransition* component to coordinate the details of the CSS animation.

Our updated code wraps the *Switch* in both a *TransitionGroup* and a *CSSTransition*:

```
import {NavLink, Redirect, Route, Switch, useLocation} from "react-
router-dom";
import People from "./People";
import Offices from "./Offices";
import {CSSTransition, TransitionGroup} from "react-transition-group";

import "./About.css";
import "./fade.css";

export default () => {
    const location = useLocation();

    return <div className='About'>
        <div className='About-tabs'>
            <NavLink to="/about/people"
                     className="About-tab"
                     activeClassName="active">
                People
            </NavLink>
            <NavLink to="/about/offices"
                     className="About-tab"
                     activeClassName="active">
                Offices
            </NavLink>
        </div>
        <TransitionGroup className='About-tabContent'>
            <CSSTransition
```

```
                    key={location.key}
                    classNames="fade"
                    timeout={500}
                >
                    <Switch location={location}>
                        <Route path='/about/people'>
                            <People/>
                        </Route>
                        <Route path='/about/offices'>
                            <Offices/>
                        </Route>
                        <Redirect to='/about/people'/>
                    </Switch>
                </CSSTransition>
            </TransitionGroup>
        </div>;
}
```

Notice that we pass the *location.key* to the *key* of the *CSSTransition* group, and we pass the *location* to the *Switch* component. When the user clicks on one of the tabs, the location changes, which refreshes the *About* component. The *TransitionGroup* will keep the existing *CSSTransition* in the tree of components until its timeout occurs: in 500 milliseconds. But it will now also have a second *CSSTransition* component.

Each of these *CSSTransition* components will keep their child components alive.

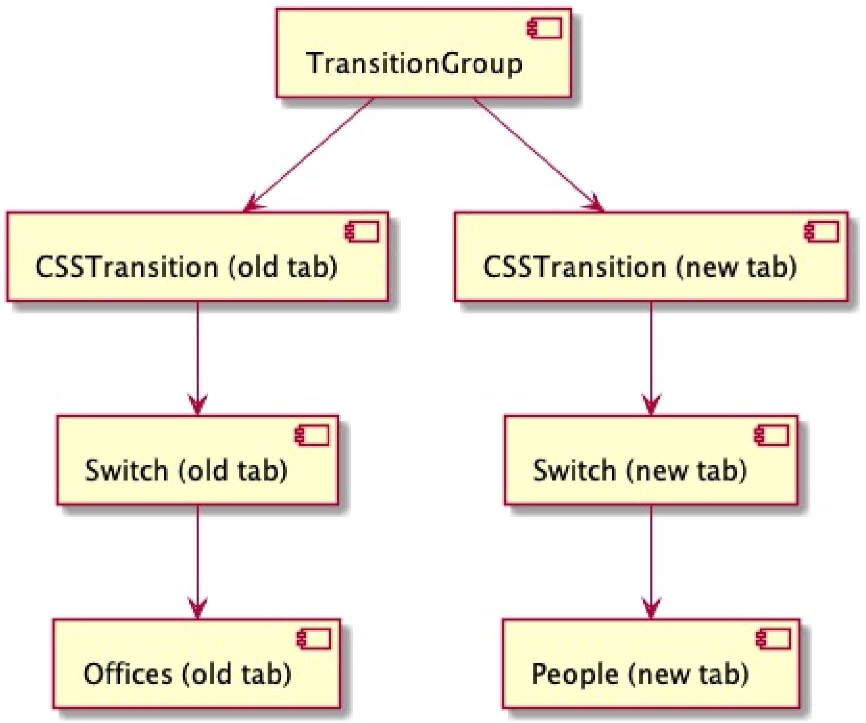


Figure 2-10. The TransitionGroup keeps both the old and new components in the virtual DOM.

This is why we pass the *location* value to the *Switch* components: we need the *Switch* for the old tab, and the *Switch* for the new tab to keep rendering their routes.

So now, onto the animation itself. The *CSSTransition* component accepts a property called *classNames*[7], which we have set to the value *fade*. *CSSTransition* will use to generate four distinct class-names:

- *fade-enter*
- *fade-enter-active*
- *fade-exit*
- *fade-exit-active*

The *fade-enter* class is for components that are about to start to animate into view. The *fade-enter-active* class is applied to components that are actually animating. *fade-exit* and *fade-exit-active* are for components that are beginning or animating their disappearance.

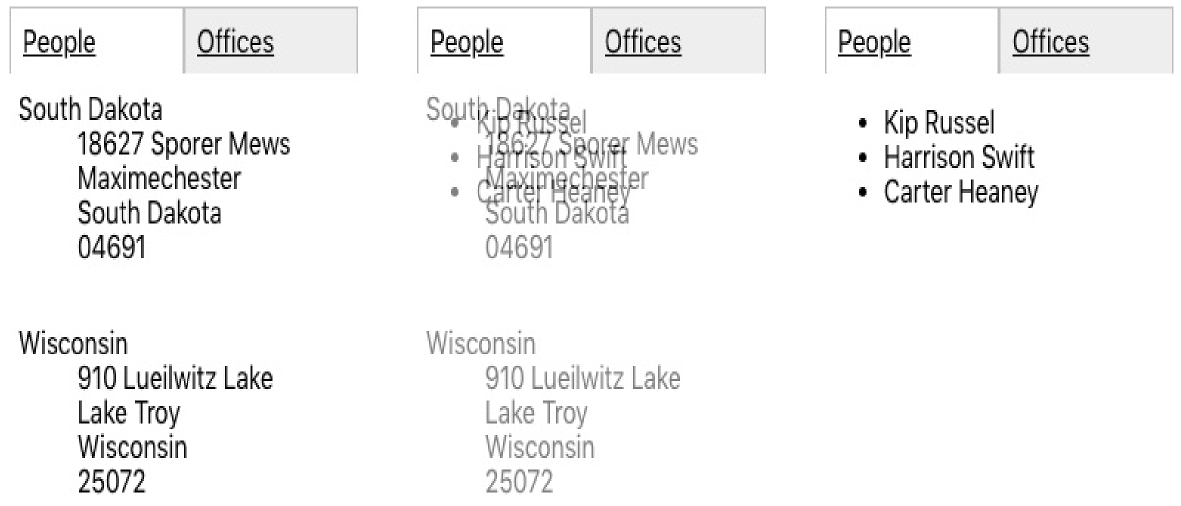
The *CSSTransition* component will add these class-names to their immediate children. If we are animating from the *Offices* tab to the *People* tab, then the old *CSSTransition* will add the *fade-enter-active* class to the *People* HTML, and add the *fade-exit-active* to the *Offices* HTML.

All that's left to do is define the CSS animations themselves:

```
.fade-enter {
    opacity: 0;
}
.fade-enter-active {
    opacity: 1;
    transition: opacity 250ms ease-in;
}
.fade-exit {
    opacity: 1;
}
.fade-exit-active {
    opacity: 0;
    transition: opacity 250ms ease-in;
}
```

The *fade-enter-* classes use CSS transitions to change the opacity of the component from 0 to 1. The *fade-exit-* classes animate the opacity from 1 back to 0. It's generally a good idea to keep the animation class definitions in their own CSS file. That way, they can be reused for other animations.

The animation is complete. When the user clicks on a tab, they see the contents cross-fade from the old data to the new data.



```
South Dakota
        18627 Sporer Mews
        Maximechester
        South Dakota
        04691

Wisconsin
        910 Lueilwitz Lake
        Lake Troy
        Wisconsin
        25072
```

- Kip Russel
- Harrison Swift
- Carter Heaney

Figure 2-11. The contents of the tab fade from offices to people

Discussion

Animations can be quite irritating when used poorly. Each animation you add should have some *intent*. If you find that you want to add an animation just because you think it will be attractive, then you will almost certainly find that the users will dislike it. Generally, it is best to ask a few questions before adding an animation:

- Will this animation clarify the relationship between two routes? Are you zooming-in to see more detail, or moving across to look at a related item?
- How short should the animation be? Any longer than half a second is probably too much.
- What is the impact on performance? CSS transitions usually have minimal effect if the browser hands the work off to the GPU. But what happens in an old browser on a mobile device?

You can download the source for this recipe from the Github site.

2.6 Create Secured Routes

Problem

Most applications need to prevent access to particular routes until a person logs in. But how do you secure some routes and not others? Is it possible to separate the security mechanisms from the user interface elements for logging in and logging out? And how do you do it without writing a vast amount of code?

Solution

Let’s look at one way to implement route-based security in a React application. This application contains a home page (*/*), a public page with no security (*/public*), and it also has two private pages (*/private1* and */private2*) that we need to secure:

```
import React from 'react';
import './App.css';
import {BrowserRouter, Route, Switch} from "react-router-dom";
import Public from "./Public";
import Private1 from "./Private1";
import Private2 from "./Private2";
import Home from "./Home";

function App() {
    return (
        <div className="App">
            <BrowserRouter>
                <Switch>
                    <Route exact path='/'>
                        <Home/>
                    </Route>
                    <Route path='/private1'>
                        <Private1/>
                    </Route>
                    <Route path='/private2'>
                        <Private2/>
                    </Route>
                    <Route exact path='/public'>
                        <Public/>
                    </Route>
                </Switch>
            </BrowserRouter>
        </div>
    );
}

export default App;
```

We're going to build the security system using a *context*. A context is a place where data can be stored by a component and made available to the component's children. A *BrowserRouter* uses a context to pass routing information to the *Route* components within it.

We're going to create a custom context called *SecurityContext*

```
import React from "react";

export default React.createContext({});
```

The default value of our context is an empty object. We need something that will places function into the context for logging in and logging out. We'll do that by creating a *SecurityProvider*.

```
import {useState} from "react";
import SecurityContext from "./SecurityContext";

export default (props) => {
    const [loggedIn, setLoggedIn] = useState(false);

    return <SecurityContext.Provider
        value={{
            login: (username, password) => {
                // Note to engineering team:
                // Maybe make this more secure...
                if (username === 'fred' && password === 'password') {
                    setLoggedIn(true);
                }
            },
            logout: () => setLoggedIn(false),
            loggedIn
        }}>
        {props.children}
    </SecurityContext.Provider>
};
```

This is obviously **not** what you would use in a real system. You would probably create a component that logged in and logged out using some sort of web service or third party security system. But in our example, the *SecurityProvider* keeps track of whether we are logged in using a simple *loggedIn* boolean value. The *SecurityProvider* puts three things into the context:

- A function for logging (*login()*)
- A function for logging out (*logout()*)
- A boolean value saying whether we are logged in or out (*loggedIn*)

These three things will be available to any components placed inside a *SecurityProvider* component. To allow any component inside a *SecurityProvider* to access these functions, we'll add a custom hook called *useSecurity*:

```
import SecurityContext from "./SecurityContext";
import {useContext} from "react";

export default () => useContext(SecurityContext);
```

Now that we have a *SecurityProvider* we need a way to use it to secure a sub-set of the routes. We'll create another component, called *SecureRoute*:

```
import Login from "./Login";
import {Route} from "react-router-dom";
import useSecurity from "./useSecurity";

export default (props) => {
    const {loggedIn} = useSecurity();

    return <Route {...props}>
        {loggedIn ? props.children : <Login/>}
    </Route>;
}
```

The *SecureRoute* component gets the current *loggedIn* status from the *SecurityContext* (using the *useSecurity()* hook), and if the user is logged-in, it renders the contents of the route. If the user is not logged in, it displays a log-in form.[8]

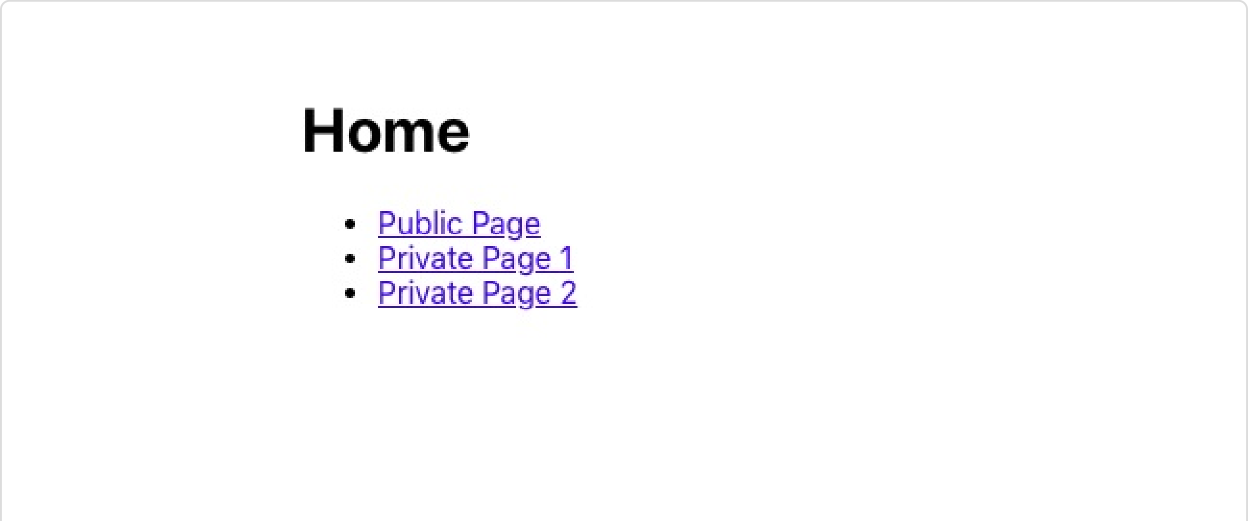
The *LoginForm* calls the *login()* function, which–if successful–will re-render the *SecureRoute* and then show the secured data.

How do we use all of these new components? This is an updated version of the *App.js* file:

```
import './App.css';
import {BrowserRouter, Route, Switch} from "react-router-dom";
import Public from "./Public";
import Private1 from "./Private1";
import Private2 from "./Private2";
import Home from "./Home";
import SecurityProvider from "./SecurityProvider";
import SecureRoute from "./SecureRoute";

function App() {
    return (
        <div className="App">
            <BrowserRouter>
                <SecurityProvider>
                    <Switch>
                        <Route exact path='/'>
                            <Home/>
                        </Route>
                        <SecureRoute path='/private1'>
                            <Private1/>
                        </SecureRoute>
                        <SecureRoute path='/private2'>
                            <Private2/>
                        </SecureRoute>
                        <Route exact path='/public'>
                            <Public/>
                        </Route>
                    </Switch>
                </SecurityProvider>
            </BrowserRouter>
        </div>
    );
}

export default App;
```

The *SecurityProvider* wraps our whole routing system, making *login()*, *logout()*, and *loggedIn* available to each *SecureRoute*.

What does the application look like when we run it?

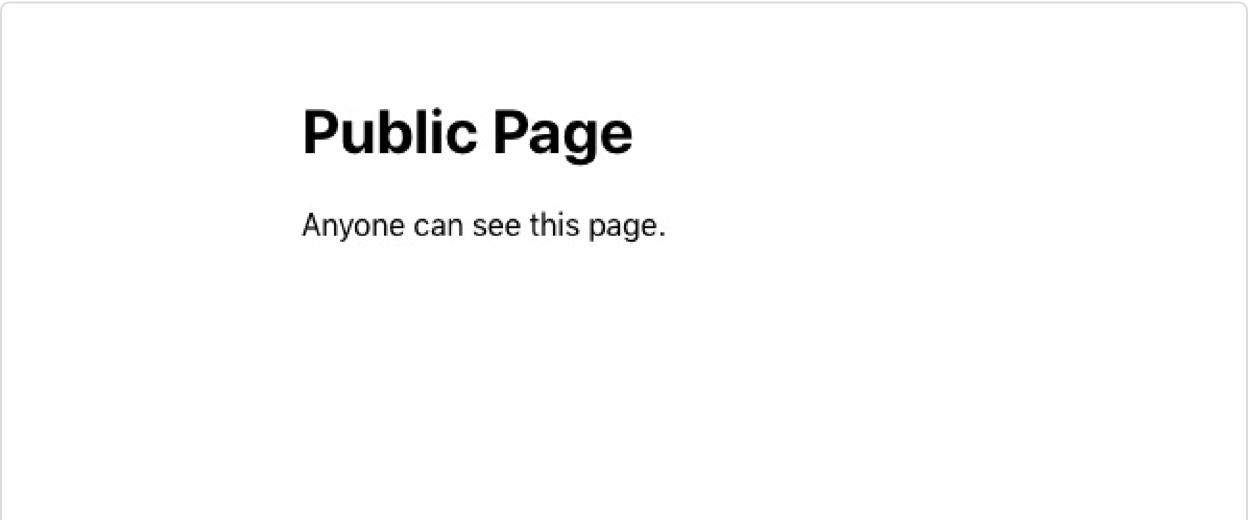


Home

- Public Page
- Private Page 1
- Private Page 2

Figure 2-12. The home page has links to the other pages

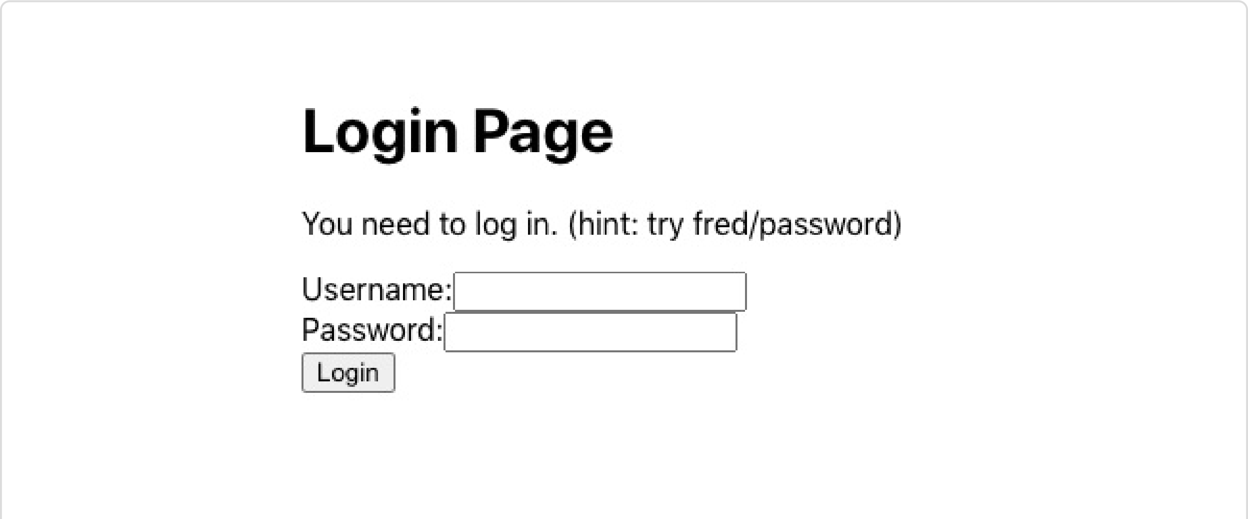
If we click on the link for the *Public Page*, we see it contents, no problem.


Public Page

Anyone can see this page.

Figure 2-13. The public page is available without logging in

But if we click on *Private Page 1*, we're presented with the log-in screen:


Login Page

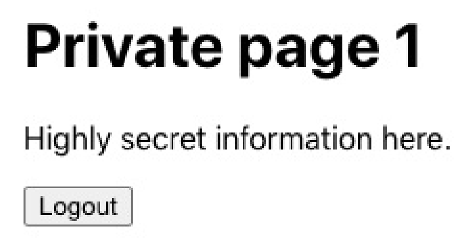
You need to log in. (hint: try fred/password)

Username:

Password:

Figure 2-14. You need to log in before you can see Private Page 1

If you log in with the username *fred*, and password *password*, you will then see the private content


Private page 1

Highly secret information here.

Figure 2-15. The content of Private Page 1 after log-in

Discussion

Real security is only ever provided by secured back-end services. However, secured routes prevent a user from stumbling into a page that will be unable to read data from the server.

A better implementation of the *SecurityProvider* would defer to some third-party OAuth tool or other security services. But by splitting the *SecurityProvider* from the security UI (*Login* and *Logout*) and the main application, you should be able to modify the security mechanisms over time without having to change a lot of code in your application.

If you want to see how your components behave when people are logged in and out, you can always create a mocked version of the *SecurityProvider* for use in unit tests.

You can download the source for this recipe from the Github site.

---

1 We're won't show the code for the *PeopleList* here, but it is available on Github

2 We are using the React testing-library in this example.

3 See recipe 1-9 in chapter 1.

4 See recipes 2 and 3 in this chapter

5 This is a common feature of third-party tabbed components. The animation reinforces in the user's mind that they are moving left and right through the tabs. This is particularly true if we allow the user to change the tab by swiping left or right.

6

The code uses relative positioning to place both components in the same position during the fade.

[7] Please note, this is plural, to distinguish it from the standard *className* attribute.

[8] We’ll omit the contents of the Login component here, but the code is available on the Github repository.

Chapter 3. Managing State

When we manage state in React, we have to not only store data, but we also need to record data dependencies. Dependencies are intrinsic to the way that React works. They allow React to update the page efficiently and only when necessary.

Managing data dependencies, then, are the key to managing state in React. You will see through this chapter that most of the tools and techniques we use are to ensure that we manage dependencies efficiently.

A key concept in the following recipes is a data *reducer*. A reducer is simply a function which receives a single object or an array, and then returns a modified copy. This simple concept is what lies behind much of the state management in React. We'll look at how React uses reducer functions natively, and how we can use the Redux library to manage data application-wide with reducers.

We'll also look at *selector functions*. These allow us to drill in to the state returned by reducers. Selectors help us ignore the irrelevant data, and in doing so, they greatly improve the performance of our code.

Along the way we'll look at simple ways of checking if you're connected to the network, how to manage form data and various other tips and tricks to keep your application ticking along.

3.1 Use reducers to manage complex state

Problem

Many React components are straightforward. They do little more than render a section of HTML and perhaps show a few properties.

However, some components can be more complicated. They might need to manage several pieces of internal state. For example, consider this simple number game.

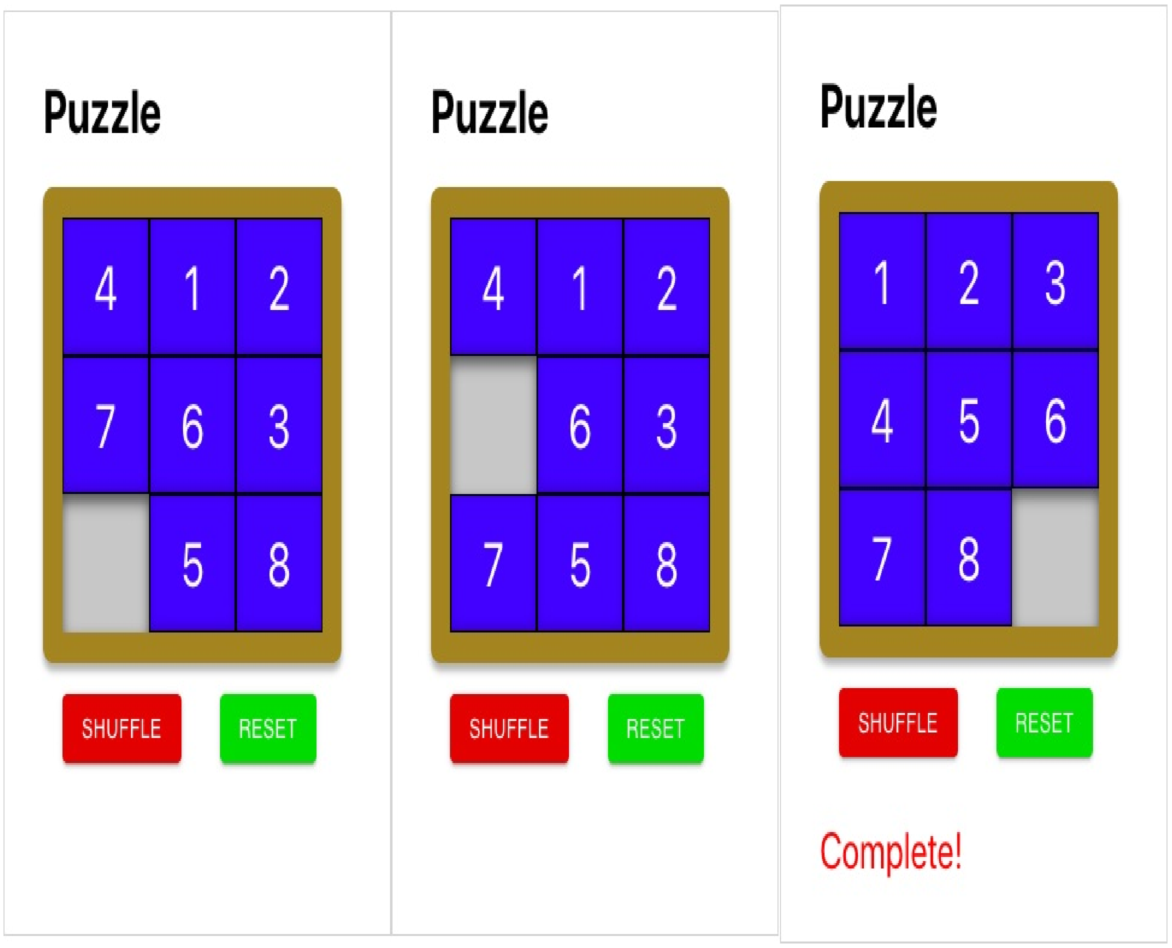


Figure 3-1. A simple number puzzle

The component displays a series of numeric tiles, in a grid, with a single space. If the user clicks a tile next to the space, they can move it. In this way, the user can rearrange the tiles until they are in the correct order from 1 - 8.

This component renders a small amount of HTML, but it will require some fairly complex logic and data. It will record the positions of the tiles. It will need to know whether a user can move a given tile. It will need to know how to move the tile. It will need to know whether the game is complete. It will also need to do other things, such as reset the game by shuffling the tiles.

It’s entirely possible to write all of this code inside the component, but it will be harder to test it. You could use the react testing library, but that is probably overkill, given that most of the code will have very little to do with rendering HTML.

Solution

If you have a component with some complex internal state, or which needs to manipulate its state in complex ways, consider using a reducer.

A reducer is a function that accepts two parameters:

- an object or array that represents a given state, and
- an **action**, which describes how you want to modify the state

The function returns a new copy of the state we pass to it.

The action parameter can be whatever you want, but typically it is an object with a string `type` attribute and a `payload` with additional information. You can think of the `type` as a command name and the `payload` as parameters to the command.

For example, if we number our tile positions from 0 (top-left) to 8 (bottom-right), we might tell the reducer to move whatever tile is in the top-left corner with:

```
{type: 'move', payload: 0}
```

We need an object or array that completely defines our game’s internal state. We could use a simple array of strings:

```
['1', '2', '3', null, '5', '6', '7', '8', '4']
```

Which would represent the tiles laid out like this:

1	2	3

	5	6
7	8	4

However, a slightly more flexible approach uses an object for our state and gives it an `items` attribute containing the current tile layout.

```
{
    items: ['1', '2', '3', null, '5', '6', '7', '8', '4']
}
```

Why would we do this? Because it will allow our reducer to return other state values, such as whether or not the game is complete, and so on.

Now we’ve thought of an action and decided what the state is like, we can create a test:

```
import reducer from "./reducer";

describe('reducer', () => {
    it('should be able to move 1 down if gap below', () => {
        let state = {
            items: ['1', '2', '3', null, '5', '6', '7', '8', '4']
        };

        state = reducer(state, {type: 'move', payload: 0});

        expect(state.items).toEqual([null, '2', '3', '1', '5', '6',
'7', '8', '4'])
    });

    it('should say when it is complete', () => {
        let state = {
            items: ['1', '2', '3', '4', '5', '6', '7', null, '8'],
        };

        state = reducer(state, {type: 'move', payload: 8});

        expect(state.complete).toBe(true);

        state = reducer(state, {type: 'move', payload: 5});

        expect(state.complete).toBe(false);
    });
});
```

In our first test scenario, we pass in the tiles' locations in one state. Then we check that the reducer returns the tiles in a new state.

In our second test, we perform two tile moves then look for a `complete` attribute to tell us the game has ended.

OK, we've delayed looking at the actual reducer code long enough.

```
function trySwap(newItems, position, t) {
    if (newItems[t] === null) {
        const temp = newItems[position];
        newItems[position] = newItems[t];
        newItems[t] = temp;
    }
}

function arraysEqual(a, b) {
    for (let i = 0; i < a.length; i++) {
        if (a[i] !== b[i]) {
            return false;
        }
    }
    return true;
}

const CORRECT = ['1', '2', '3', '4', '5', '6', '7', '8', null];

function reducer(state, action) {
    switch (action.type) {
        case "move": {
            const position = action.payload;
            const newItems = [...state.items];
            const col = position % 3;

            if (position < 6) {
                trySwap(newItems, position, position + 3);
            }
            if (position > 2) {
                trySwap(newItems, position, position - 3);
            }
            if (col < 2) {
                trySwap(newItems, position, position + 1);
            }
            if (col > 0) {
                trySwap(newItems, position, position - 1);
            }

            return {
                ...state,
                items: newItems,
                complete: arraysEqual(
                    newItems,
                    CORRECT),
            }
        }
        default: {
            throw new Error("Unknown action: " + action.type);
        }
    }
}

export default reducer;
```

Our reducer currently recognizes a single action: *move*. The code in our Github repository also includes actions for *shuffle* and *reset*. The repository also has a more exhaustive set of tests that we used to create the code you can see above.

But **none** of this code includes any React components. It's pure JavaScript and so can be created and tested in isolation from the outside world.

WARNING

Be careful to generate a new object in the reducer to represent the new state. Doing so ensures each new state completely independent of those that came before it.

Now it's time to wire up our reducer into a React component, with the `useReducer` hook:

```
    import {useReducer} from "react";
    import reducer from "./reducer";

    import "./Puzzle.css";

    export default () => {
        const [state, dispatch] = useReducer(reducer, {
            items: ['4', '1', '2', '7', '6', '3', null, '5', '8']
        });

        return <div className='Puzzle'>
            <div className='Puzzle-squares'>
                {
```

```
                state.items.map((s, i) => <div
                    className={`Puzzle-square ${s ? '' : 'Puzzle-
square-empty'}`}
                    key={`square-${i}`}
                    onClick={() => dispatch({type: 'move', payload:
i})}
                >{s}</div>)
            }
        </div>
        <div className='Puzzle-controls'>
            <button className='Puzzle-shuffle'
                    onClick={() => dispatch({type:
'shuffle'})}>Shuffle</button>
            <button className='Puzzle-reset'
                    onClick={() => dispatch({type:
'reset'})}>Reset</button>
        </div>
        {
            state.complete &&
            <div className='Puzzle-complete'>Complete!</div>
        }
    </div>;
};
```

Even though our puzzle component is doing something quite complicated, that actual React code is relatively short.

The `useReducer` accepts a reducer function and a starting state, and it returns a two-element array:

- The first element in the array is the current state from the reducer,
- The second element is a `dispatch` function that allows us to send actions to the reducer.

We display the tiles by looping through the strings in the array given by `state.items`.

If someone clicks on a tile at position `i`, we send a *move* command to the reducer:

```
onClick={() => dispatch({type: 'move', payload: i})}
```

The React component has no idea what it takes to move the tile. It doesn't even know if it can move the tile at all. The component sends the action to the reducer.

If the *move* action moves a tile, the component will automatically re-render the component with the tiles in their new positions. If the game is complete, the component will know by the value of `state.complete`:

```
state.complete && <div className='Puzzle-complete'>Complete!</div>
```

We also added two buttons to run the *shuffle* and *reset* actions, which we omitted above, but which you can find in the code on the Github repository.

OK, now that we've created our component, let's try it out. When we first load the component, we see it in its initial state:

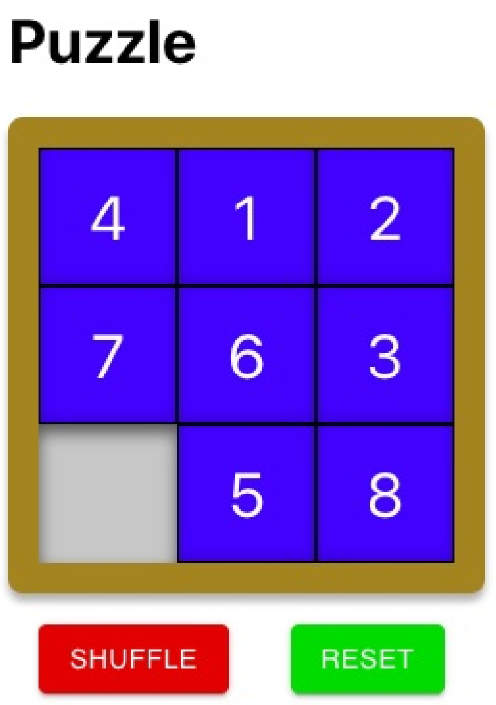


Figure 3-2. The starting state of the game

If we click the tile labeled "7", it moves into the gap:

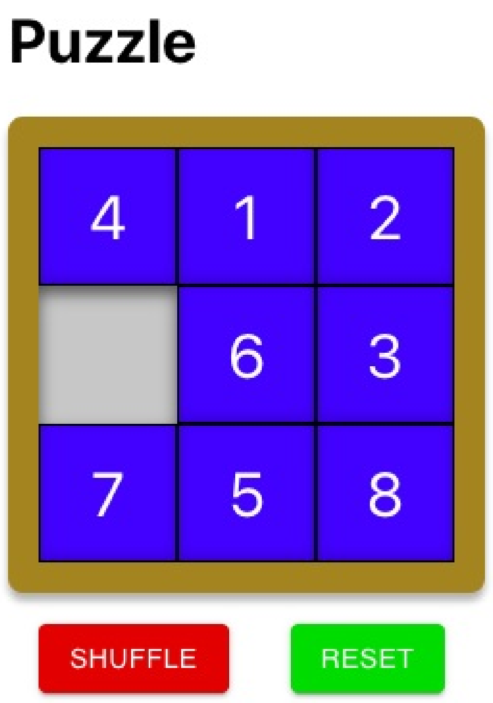


Figure 3-3. After moving tile 7

If we click the “Shuffle” button, the reducer rearranges tiles randomly:

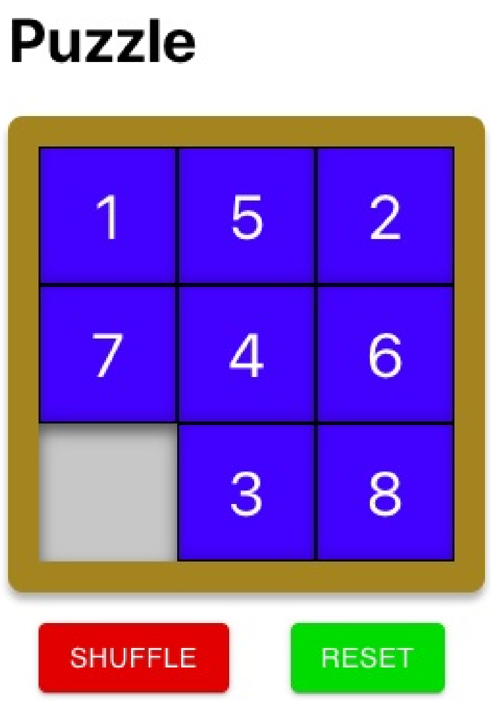


Figure 3-4. The shuffle button moves tiles to random positions

And if we click “Reset”, the puzzle changes to the complete position, and the “Complete” text appears:

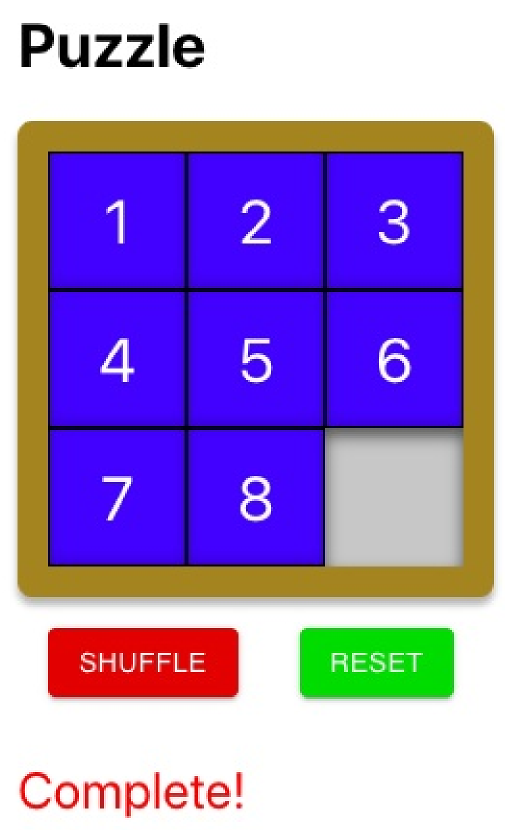


Figure 3-5. The reset button moves the tiles to their correct positions

We bury all of the complexity inside the reducer function, where it can be easily tested, and the component is simple and easy to maintain.

Discussion

Reducers are a way of managing complexity. You will typically use a reducer if either:

- You have a large amount of internal state to manage, or
- You need complex logic to manage the internal state of your component

If either of these things is correct, then a reducer can make your code significantly easier to manage.

However, be wary of using reducers for very simple components. If your component has simple state and very little logic, you probably don't need the overhead of a reducer.

Sometimes, even if you do have complex state, there are alternative approaches. For example, if you are capturing and validating data in a form, it might be better to create a validating form component (see the recipe elsewhere in this chapter).

You need to ensure that your reducer does not have any side effects. Avoid, say, making network calls that update a server. If your reducer has side-effects, there is every chance that it might break. In development mode, React (very sneakily) might sometimes make additional calls to your reducer to make sure that no side effects are happening. If you're using a reducing and noticed that React calls your code twice when rendering a component, this is React checking to see that you are doing anything untoward.

With all of those provisos, reducers are an excellent tool at fighting complexity. They are integral to libraries such as Redux, can easily be reused and combined, simplify components, and make your React code significantly easier to test.

You can download the source for this recipe from the Github site.

3.2 Creating an Undo Feature

Problem

Part of the promise of JavaScript rich frameworks like React is that web applications can achieve a closer resemblance to desktop applications. One common feature in desktop applications is the ability to undo an action. Some native components within React applications automatically support and undo function. If you edit some text in a text area, then press CMD/CTRL-Z, it will undo your edit. But what about extending undo into custom components? How is it possible to track state-changes without a large amount of code?

Solution

If a reducer function manages the state in your component (see recipe earlier in this chapter), you can implement a quite general undo function using an undo-reducer.

Consider this piece of code from the `Puzzle` example in the first recipe of this chapter:

```
const [state, dispatch] = useReducer(reducer, {
    items: ['4', '1', '2', '7', '6', '3', null, '5', '8']
});
```

This code uses a reducer function (called `reducer`) and an initial state to manage the tiles in a number-puzzle game.

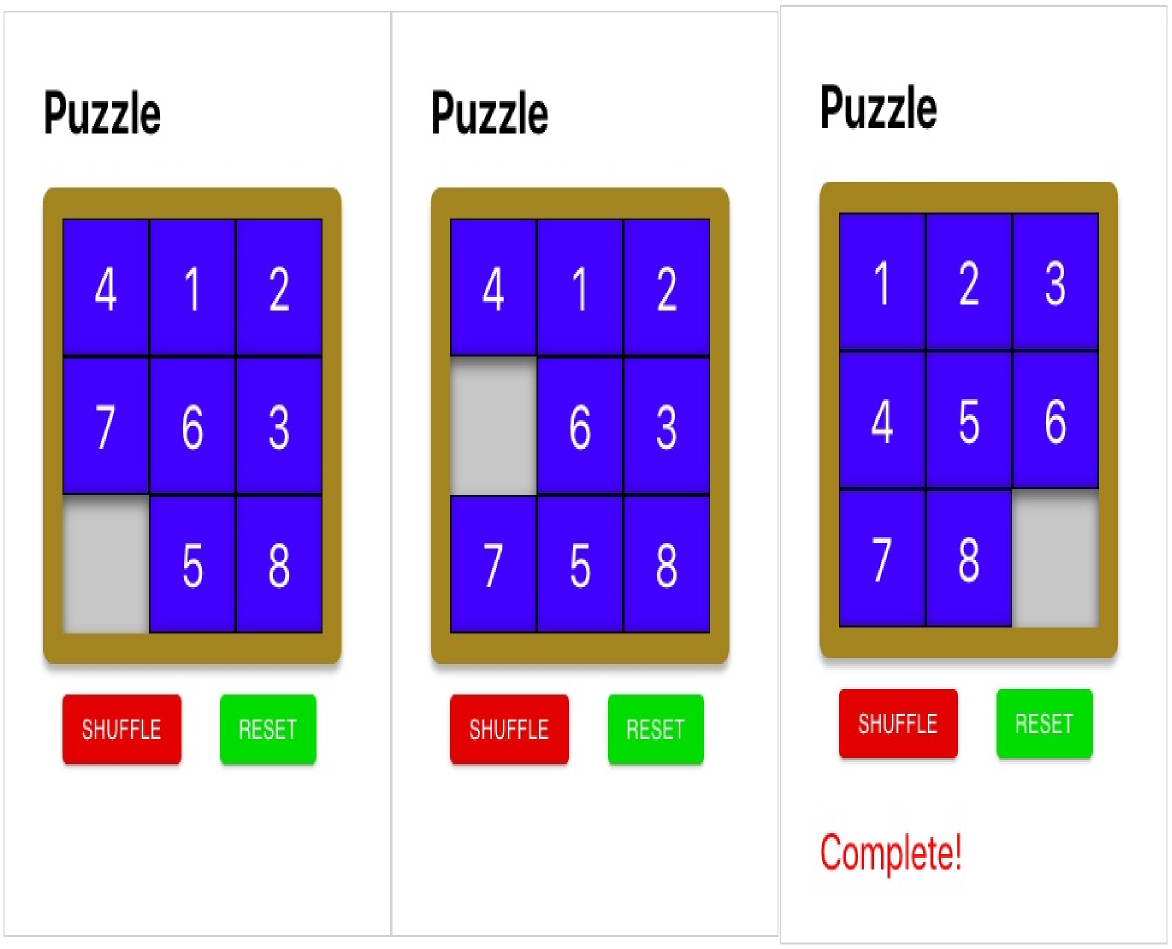


Figure 3-6. A simple number puzzle game

If the user pressed the `Shuffle` button, the component updates the tile state by sending a “shuffle” action to the reducer:

```
<button className='Puzzle-shuffle'
        onClick={() => dispatch({type: 'shuffle'})}>Shuffle</button>
```

(for more details on what reducers are and when you should use them, see the first recipe in this chapter)

What we’re going to do in this recipe, is create a new hook called `userUndoReducer`, which is a drop-in replacement for `userReducer`.

```
const [state, dispatch] = useUndoReducer(reducer, {
    items: ['4', '1', '2', '7', '6', '3', null, '5', '8']
});
```

The `useUndoReducer` hook will magically give out component the ability to go back in time:

```
<button className='Puzzle-undo'
        onClick={() => dispatch({type: 'undo'})}>Undo</button>
```

If we add this button to the component, it undoes the last action the user performed.

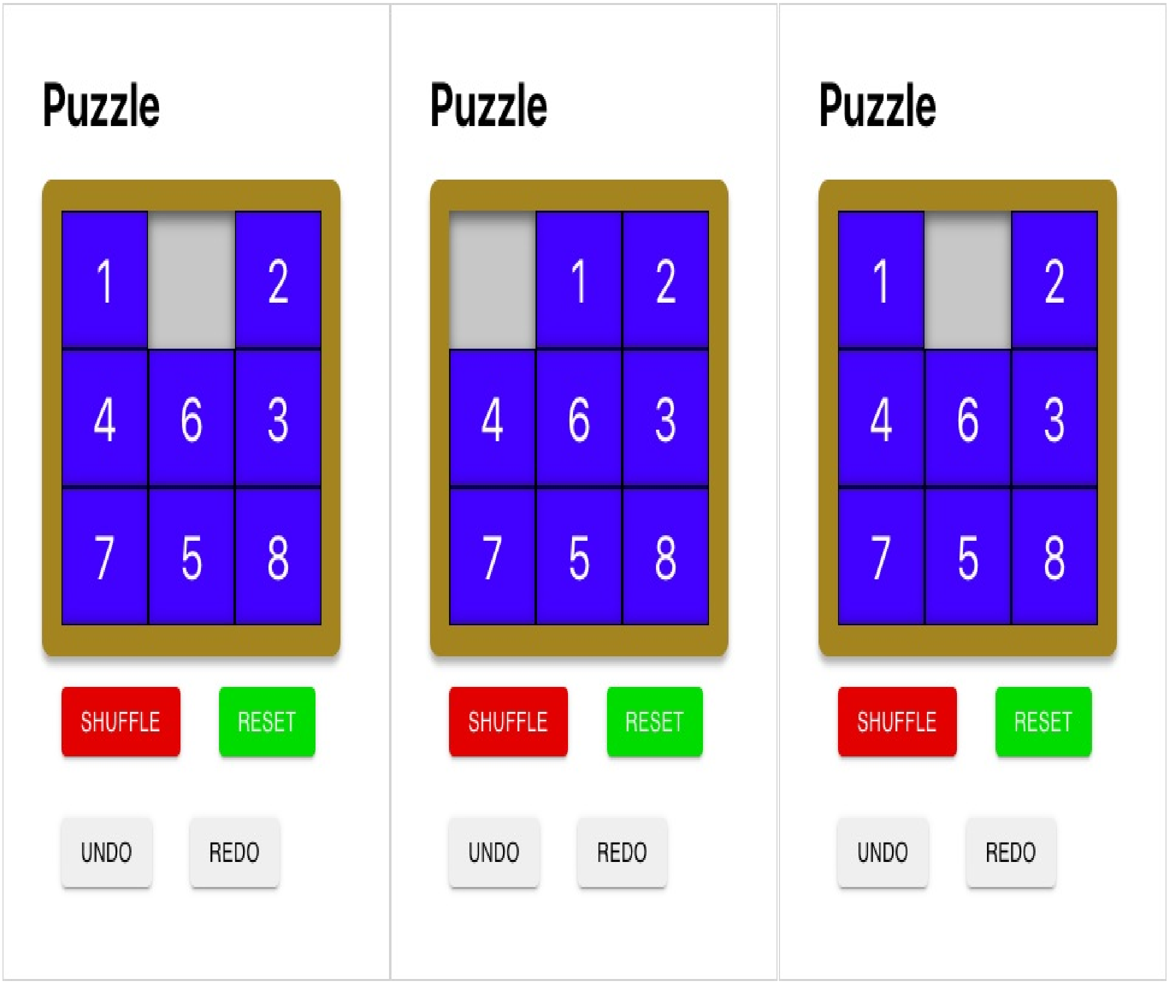


Figure 3-7. 1. Game in progress. 2. Make a move 3. Click Undo to undo move

But how do we perform this magic? Although `useUndoReducer` is relatively easy to use, it's somewhat harder to understand. But it's worth doing so that you can adjust the recipe to your requirements.

We can take advantage of the fact that reducers are work in a very similar way:

- The action defines what you want to do
- The reducer returns a fresh state after each action
- No side effects are allowed when calling the reducer

Also, reducers are just simple JavaScript functions, which always accept a state object and an action object as parameters.

Because reducers work in such a well-defined way, we can create a new reducer–an undo-reducer–that wraps around another reducer function. Our undo-reducer will work as an intermediary. It will pass most actions through to the underlying reducer while keeping a history of all previous states. If someone wants to undo an action, it will find the last state from its history, and then return that without calling the underlying reducer.

We’ll begin by creating a higher-order function that accepts one reducer and returns another:

```
import lodash from 'lodash';

export default (reducer) =>
    (state, action) => {
        let {undoHistory = [],
            undoActions = [],
            ...innerState} = lodash.cloneDeep(state);
        switch (action.type) {
            case 'undo': {
                if (undoActions.length > 0) {
                    undoActions.pop();
                    innerState = undoHistory.pop();
                }
                break;
            }

            case 'redo': {
                if (undoActions.length > 0) {
                    undoHistory = [
                        ...undoHistory,
                        {...innerState}
                    ];
                    undoActions = [
                        ...undoActions,
                        undoActions[undoActions.length - 1]
                    ];
                    innerState = reducer(
                        innerState,
                        undoActions[undoActions.length - 1]
                    );
                }
                break;
            }

            default: {
                undoHistory = [
                    ...undoHistory,
                    {...innerState}
                ];
                undoActions = [
                    ...undoActions,
                    action
                ];
                innerState = reducer(innerState, action);
            }
        }
        return {...innerState, undoHistory, undoActions};
    };
```

This reducer is quite a complex function, so it's worth taking some time to understand what it does.

It creates a reducer function that keeps track of the actions and states we pass to it. Let's say our game component sends an action to shuffle the tiles in the game. Our reducer will first check if the action has type "undo" or "redo." It doesn't. So it passes the "shuffle" action to the underlying reducer that manages the tiles in our game.

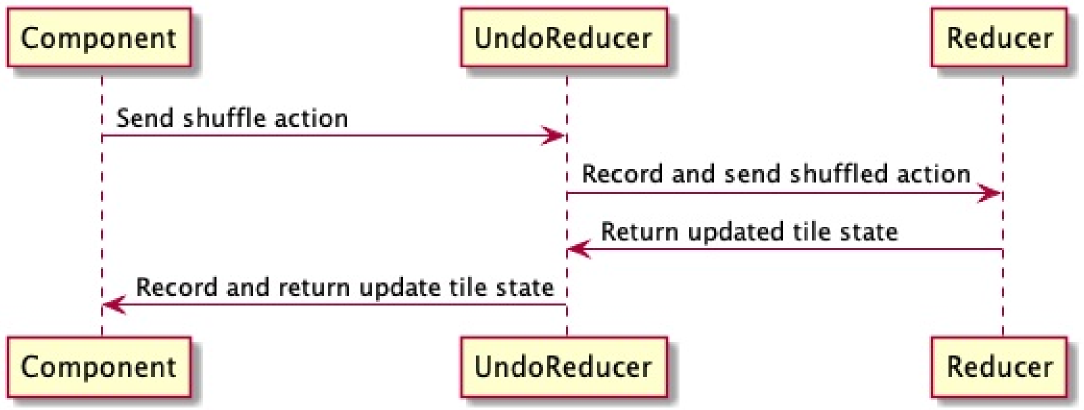

Component
UndoReducer
Reducer
Send shuffle action
Record and send shuffled action
Return updated tile state
Record and return update tile state
Component
UndoReducer
Reducer

Figure 3-8. The undo-reducer passes most actions to the underlying reducer

As it passes the "shuffle" action through to the underlying reducer, the `undo` code keeps track of the existing state and the "shuffle" action by adding them to the `undoHistory` and `undoActions`. It then returns the state of the underlying game reducer, as well as the `undoHistory` and `undoActions`.

If our puzzle component sends in an "undo" action, the undo-reducer returns the previous state from the `undoHistory`, completely bypassing the game's own reducer function.

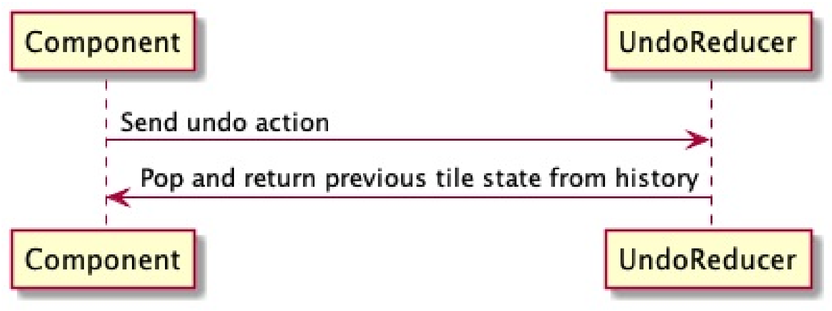


Figure 3-9. For undo actions, the undo-reducer returns the latest historic state

Now let's look at the `useUndoReducer` hook itself.

```
import {useReducer} from "react";
import undo from "./undo";

export default (reducer, initialState) => useReducer(undo(reducer),
initialState);
```

This `useUndoReducer` hook is a concise piece of code. It's simply a call to the built-in `useReducer` hook, but instead of passing the reducer straight through, it passes `undo(reducer)`. The result is that your component uses an enhanced version of the reducer you provide: one that can undo and redo actions.

This is our updated `Puzzle` component (see recipe 1 in this chapter for the original version):

```
import reducer from "./reducer";
import useUndoReducer from "./useUndoReducer";

import "./Puzzle.css";

export default () => {
    const [state, dispatch] = useUndoReducer(reducer, {
        items: ['4', '1', '2', '7', '6', '3', null, '5', '8']
    });

    return <div className='Puzzle'>
        <div className='Puzzle-squares'>
            {
                state.items.map((s, i) => <div
                    className={`Puzzle-square ${s ? '' : 'Puzzle-
square-empty'}`}
                    key={`square-${i}`}
                    onClick={() => dispatch({type: 'move', payload:
i})}
                >{s}</div>)
            }
        </div>
        <div className='Puzzle-controls'>
            <button className='Puzzle-shuffle'
                    onClick={() => dispatch({type:
'shuffle'})}>Shuffle</button>
            <button className='Puzzle-reset'
                    onClick={() => dispatch({type:
'reset'})}>Reset</button>
        </div>
        <div className='Puzzle-controls'>
            <button className='Puzzle-undo'
                    onClick={() => dispatch({type:
'undo'})}>Undo</button>
            <button className='Puzzle-redo'
                    onClick={() => dispatch({type:
'redo'})}>Redo</button>
        </div>
        {
            state.complete &&
            <div className='Puzzle-complete'>Complete!</div>
        }
    </div>;
};
```

The only changes are that we use `useUndoReducer` instead of `useReducer`, and we've added a couple of buttons to call the "undo" and "redo" actions.

If you now load the component and makes some changes, you can undo the changes one at a time:

actions image::images/ch03-reducer-1-overview.png["After replacing useReducer with useUndoReducer, you can now send "undo" and "redo" actions"]

Discussion

The undo-reducer shown here will work with reducers that accept and return state-objects. If your reducer manages state using arrays, you will have to modify the `undo` function.

Because it keeps a history of all previous states, you probably want to avoid using it if your state data is extensive, or if you're using it in circumstances where might make large (thousands?) or changes. Also, bear in mind that it maintains its history in memory. If a user reloads the entire page, then the history will disappear. It should be possible to get around this issue by persisting the global state in local storage whenever it changes.

You can download the source for this recipe from the Github site.

3.3 Creating and Validating Forms

Problem

Most React applications use forms to some degree, and most applications take an ad-hoc approach to creating them. If a team is building your application, you might find that some developers manage the state of individual fields in separate state variables. Others will choose to record form-state in a single value object, which is simpler to pass into and out of the form but can be quite tricky for each form-field to update. Field validation often leads to spaghetti code, with some forms validating at submit time, others validating dynamically as the user types. Some forms might show validation messages when the form first loads. In other forms, the messages might appear only after the user has touched the fields.

These variations in design can lead to poor user experience and an inconsistent approach to writing code. In our experience working with React teams, forms, and form validation, are a frequent stumbling block for developers.

Solution

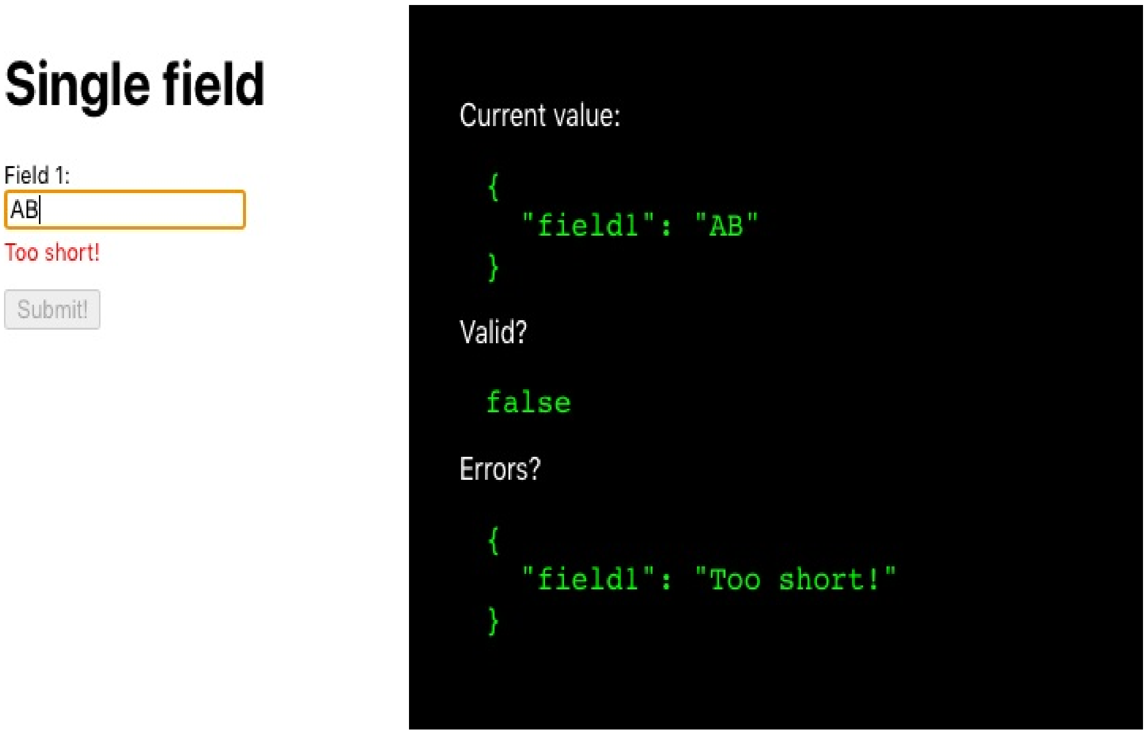
To apply some consistency to form development, we will create a `SimpleForm` component that we will wrap around one or more `InputField` components. This is an example of the use of `SimpleForm` and `InputField`:

```
import {useEffect, useState} from 'react';
import './App.css';
import SimpleForm from "./SimpleForm";
import InputField from "./InputField";
```

```
export default ({onSubmit, onChange, initialValue = {}}) => {
    const [formFields, setFormFields] = useState(initialValue);

    const [valid, setValid] = useState(true);
    const [errors, setErrors] = useState({});

    useEffect(() => {
        if (onChange) {
            onChange(formFields, valid, errors);
        }
    }, [onChange, formFields, valid, errors]);

    return <div className='TheForm'>
        <h1>Single field</h1>

        <SimpleForm value={formFields}
                    onChange={setFormFields}
                    onValid={(v, errs) => {
                        setValid(v);
                        setErrors(errs)
                    }}>
            <InputField name='field1'
                        onValidate={v => (!v || v.length < 3)
                            ? "Too short!" : null}/>

            <button
                onClick={() => onSubmit && onSubmit(formFields)}
                disabled={!valid}>Submit!
            </button>
        </SimpleForm>
    </div>;
}
```

We track the state of the form in a single object `formFields`. Whenever we change a field in the form, the field will call `onChange` on the `SimpleForm`. The `field1` field is validated using the `onValidate` method, and whenever the validation state changes, the field calls the `onValid` method on the `SimpleForm`. Validation will only occur if the user has interacted with a field: making it *dirty*.

This is the form running:

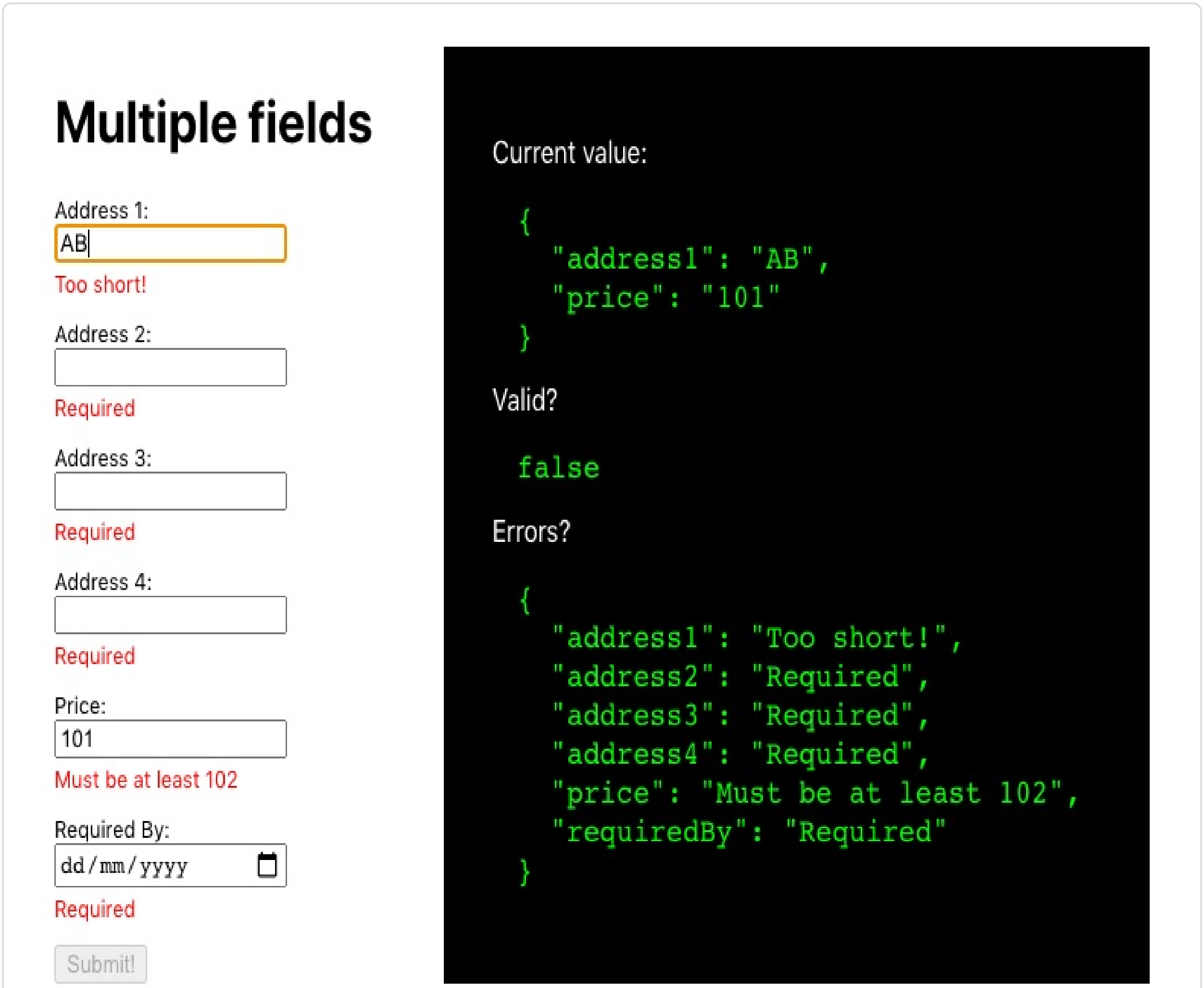


```
Current value:

  {
    "field1": "AB"
  }

Valid?

  false

Errors?

  {
    "field1": "Too short!"
  }
```

Figure 3-10. A simple form with field validation

There is no need to track individual field values. The form value object records individual field values with attributes derived from the name of the field. The `InputField` handles all of the details of when to run the validation: it will update the form-value and decide when to display errors.

Here is a slightly more complex example which uses the `SimpleForm` with several fields:


Multiple fields

Address 1:
AB
Too short!

Address 2:
Required

Address 3:
Required

Address 4:
Required

Price:
101
Must be at least 102

Required By:
dd/mm/yyyy
Required

Submit!

```
Current value:

  {
    "address1": "AB",
    "price": "101"
  }

Valid?

  false

Errors?

  {
    "address1": "Too short!",
    "address2": "Required",
    "address3": "Required",
    "address4": "Required",
    "price": "Must be at least 102",
    "requiredBy": "Required"
  }
```

Figure 3-11. A more complex form

To create the `SimpleForm` and `InputField` components, we must first look at they will communicate. An `InputField` component will need to tell the `SimpleForm` when it's value has changed, and whether or not the new value is valid. It will do this with a *context*.

A context is a storage scope. When a component stores values in a context, that value is visible to its sub-components. The `SimpleForm` will create a context called `FormContext` and use it to store a set of callback functions that any child component can use to communicate with the form.

```
import {createContext} from "react";

export default createContext({});
```

To see how `SimpleForm` works, let’s begin with a simplified version, which only tracks its sub-components’ values, without worrying about validation just yet:

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

import './SimpleForm.css';
import FormContext from "./FormContext";

function updateWith(oldValue, field, value) {
    const newValue = {...oldValue};
    newValue[field] = value;
    return newValue;
}

export default ({children, value, onChange, onValid}) => {
    const [values, setValues] = useState(value || {});

    useEffect(() => {
        setValues(value || {});
    }, [value]);

    useEffect(() => {
        if (onChange) {
            onChange(values);
        }
    }, [onChange, values]);

    let setValue = useCallback(
        (field, v) => setValues(vs => updateWith(vs, field, v)),
[setValues]);
    let getValue = useCallback(
        field => values[field], [values]);
    let form = {
        setValue: setValue,
        value: getValue,
    };

    return <div className='SimpleForm-container'>
        <FormContext.Provider value={form}>
            {children}
        </FormContext.Provider>
    </div>;
}
```

The final version of `SimpleForm` will have additional code for tracking validation and errors, but this cut-down form is easier to understand.

The form is going to track all of its field values in the `values` object. The form creates two callback functions called `getValue` and `setValue` and puts them into the context (as the `form` object), where sub-components will find them. We put the `form` into the context by wrapping a `<FormContext.Provider>` around the child components.

Notice that we have wrapped the `getValue` and `setValue` callbacks in `useCallback`, which prevents the component from creating a new version of each function every time we render the `SimpleForm`.

Whenever a child component calls the `form.getValue` function, they will be given the current value of the specified field. If a child component calls `form.setValue`, it will update that value.

Now let’s look at a simplified version of the `InputField` component, again with any validation code removed to make it easier to understand:

```
import React, {useContext} from "react";
import FormContext from "./FormContext";

import "./InputField.css";

export default (props) => {
    const form = useContext(FormContext);

    if (!form.value) {
        return "InputField should be wrapped in a form"
    }

    const {name, label, ...otherProps} = props;

    let value = form.value(name);

    return <div className='InputField'>
        <label htmlFor={name}>
            {label || name}:
        </label>
        <input
            id={name}
            value={value || ''}
            onChange={event => {
                form.setValue(name, event.target.value);
            }}
            {...otherProps}
        /> {
        }
        </div>;
    };
```

The `InputField` extracts the `form` object from the `FormContext`. If it cannot find a `form` object, it knows that we have not wrapped it in a `SimpleForm` component. The `InputField` then renders a `<input/>` field, setting its value to whatever is returned by `form.value(name)`. If the user changes the field's value, the `InputField` component sends the new value to `form.setValue(name, event.target.value)`.

If you need a form field other than an `input`, you can wrap it in some component similar to the `InputField` shown here.

The validation code is just more of the same. In the same way that the form tracks its current value in the `values` state, it also needs to track which fields are dirty and which are invalid. It then needs to pass callbacks for `setDirty`, `isDirty`, and `setInvalid`. These callbacks are used by the child fields when running their `onValidate` code.

Here is the final version of the `SimpleForm` component, including validation.

```
import {useCallback, useEffect, useState} from "react";

import './SimpleForm.css';
import FormContext from "./FormContext";

function updateWith(oldValue, field, value) {
    const newValue = {...oldValue};
    newValue[field] = value;
    return newValue;
}

export default ({children, value, onChange, onValid}) => {
    const [values, setValues] = useState(value || {});
    const [dirtyFields, setDirtyFields] = useState({});
    const [invalidFields, setInvalidFields] = useState({});

    useEffect(() => {
        setValues(value || {});
    }, [value]);

    useEffect(() => {
        if (onChange) {
            onChange(values);
        }
    }, [onChange, values]);

    useEffect(() => {
        if (onValid) {
            onValid(Object.keys(invalidFields)
                .every(i => !invalidFields[i]), invalidFields);
        }
    }, [onValid, invalidFields]);

    let setValue = useCallback(
        (field, v) => setValues(vs => updateWith(vs, field, v)),
[setValues]);
    let getValue = useCallback(
        field => values[field], [values]);
    let setDirty = useCallback(
        field => setDirtyFields(df => updateWith(df, field, true)),
[setDirtyFields]);
    let getDirty = useCallback(
        field => Object.keys(dirtyFields).includes(field),
[dirtyFields]);
    let setInvalid = useCallback((field, error) => {
        setInvalidFields(i => updateWith(i, field, error ? error :
undefined));
    }, [setInvalidFields]);
    let form = {
        setValue: setValue,
        value: getValue,

        setDirty: setDirty,
        isDirty: getDirty,

        setInvalid: setInvalid,
    };

    return <div className='SimpleForm-container'>
        <FormContext.Provider value={form}>
            {children}
        </FormContext.Provider>
    </div>;
}
```

And this is the final version of the `InputField` component. Notice that the field is counted as *dirty* once it loses focus.

```
import {useContext, useEffect, useState} from "react";
import FormContext from "./FormContext";
import "./InputField.css";

const splitCamelCase = s => s
    .replace(/([a-z0-9])([A-Z0-9])/g, '$1 $2')
    .replace(/^([a-z])/, x => x.toUpperCase());

export default (props) => {
    const form = useContext(FormContext);

    if (!form.value) {
        return "InputField should be wrapped in a form"
    }

    const [error, setError] = useState('');

    const {onValidate, name, label, ...otherProps} = props;

    let value = form.value(name);

    useEffect(() => {
        if (onValidate) {
            setError(onValidate(value));
        }
    }, [onValidate, value]);

    const setInvalid = form.setInvalid;

    useEffect(() => {
        setInvalid(name, error);
    }, [setInvalid, name, error]);

    return <div className='InputField'>
        <label htmlFor={name}>
            {label || splitCamelCase(name)}:
        </label>
        <input
            id={name}
            onBlur={() => form.setDirty(name)}
            value={value || ''}
            onChange={event => {
                form.setDirty(name);
                form.setValue(name, event.target.value);
            }}
            {...otherProps}
        /> {
        <div className='InputField-error'>
            {form.isDirty(name) && error ? error : <>&nbsp;</>}
        </div>
    }
    </div>;
};
```

Discussion

You can use this recipe to create most simple forms, and you can extend it for use with any React component. For example, if you are using a third-party calendar or date picker, you would only need to wrap it in component similar to `InputField` to use it inside a `SimpleForm`.

This recipe doesn’t support forms within forms or arrays of forms. It should be possible to modify the `SimpleForm` component to behave like an `InputField` so that we could place one form inside another.

You can download the source for this recipe from the Github site.

3.4 Measuring Time with a Clock

Problem

Sometimes a React application needs to respond to the time of day. It might only need to display the current time, or it might need to poll a server at regular intervals or change its interface as day turns to night. But how do you cause your code to re-render as the result of a time change? How to avoid rendering too often? And how do you all that without littering your code with a lot of `setTimeout` calls and a large number of extraneous state variables?

Solution

We’re going to create a `useClock` hook. The `useClock` hook will give us access to a formatted version of the current date and time, and automatically update the interface when the time changes. Here’s an example of it in use:

```
import {useEffect, useState} from 'react';
import useClock from "./useClock";
import ClockFace from "./ClockFace";

import "./Ticker.css";
```

```
export default () => {
    const [isTick, setTick] = useState(false);

    const time = useClock("HH:mm:ss");

    useEffect(() => {
        setTick(t => !t);
    }, [time]);

    return (
        <div className="Ticker">
            <div className='Ticker-clock'>
                <h1>Time {isTick ? 'Tick!' : 'Tock!'}</h1>
                {time}
                <br/>
                <ClockFace time={time}/>
            </div>
        </div>
    );
};
```

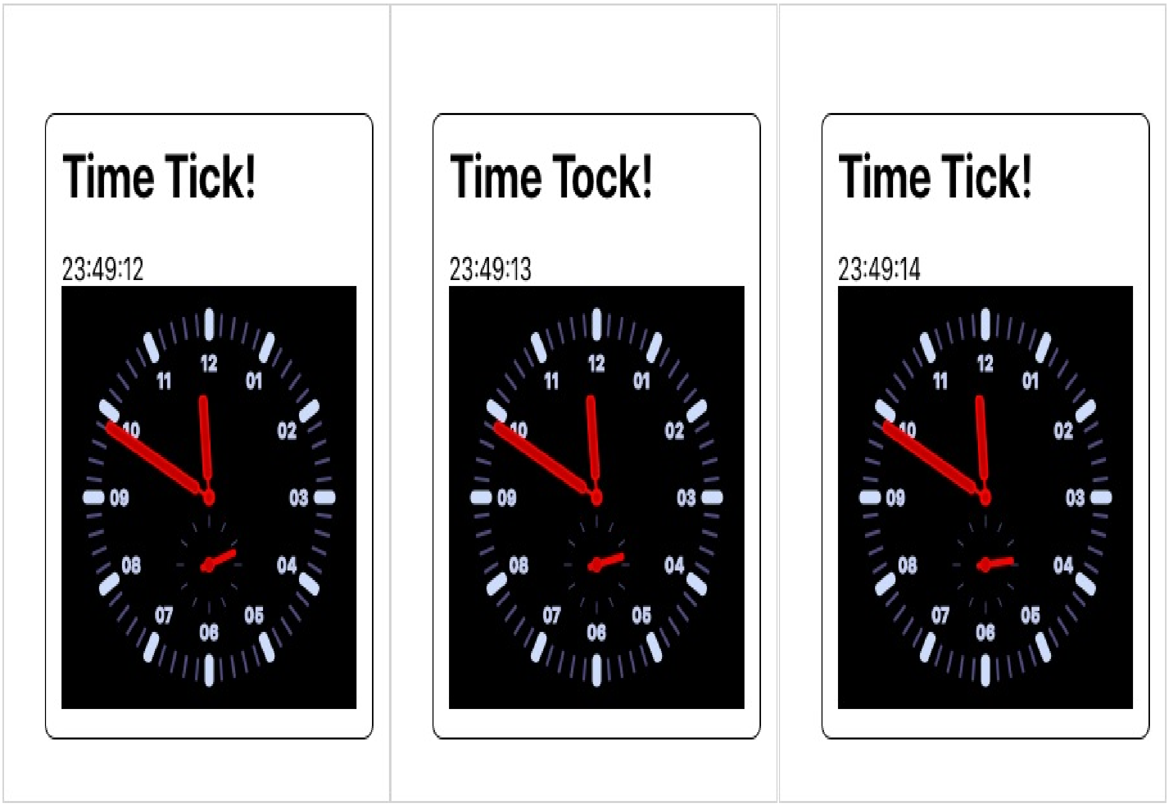
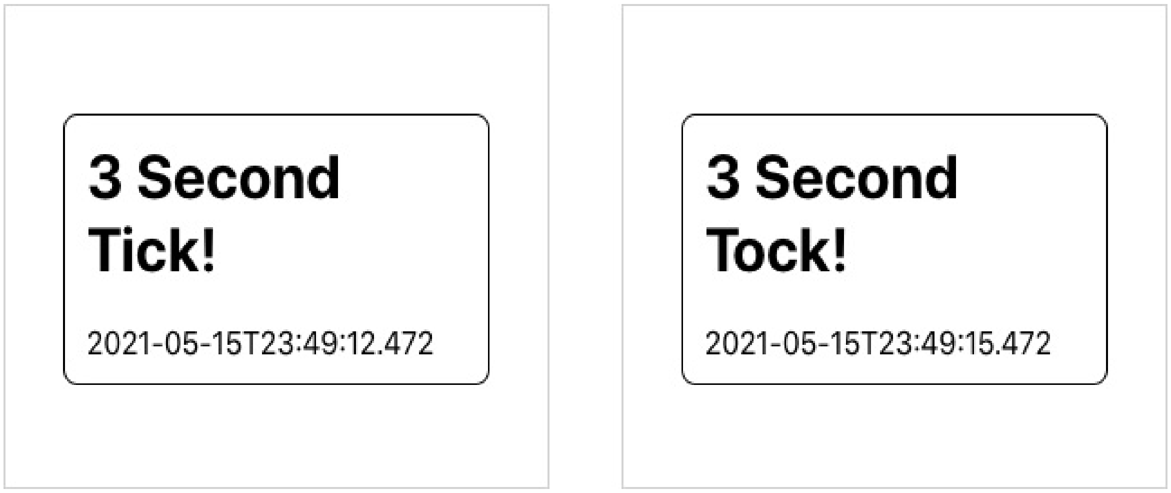


Figure 3-12. The SimpleTicker over three seconds

The `time` variable contains the current time in the format `HH:mm:ss`. When the time changes, the value of `isTick` state is toggled between true/false and then used to display the word “Tick!” or “Tock!”. We show the current time, and then also display the time with a `ClockFace` component.

As well as accepting a date/time format, `useClock` can also be given a number that represents the number of milliseconds between updates.

```
import {useEffect, useState} from 'react';
import useClock from "./useClock";

import "./Ticker.css";

export default () => {
    const [isTick3, setTick3] = useState(false);

    const tickThreeSeconds = useClock(3000);

    useEffect(() => {
        setTick3(t => !t);
    }, [tickThreeSeconds]);

    return (
        <div className="Ticker">
            <div className='Ticker-clock'>
                <h1>{isTick3 ? '3 Second Tick!' : '3 Second Tock!'}
</h1>
                {tickThreeSeconds}
            </div>
        </div>
    );
};
```


3 Second Tick!

2021-05-15T23:49:12.472

3 Second Tock!

2021-05-15T23:49:15.472

Figure 3-13. The IntervalTicker re-renders the component every three seconds

This version is more useful if you want to perform some task at regular intervals. If you pass a numeric parameter to `useClock`, it will return a time string in a format like `2021-06-11T14:50:34.706`

To build this hook, we're going to use a third-party library called *MomentJS* to handle date and time formatting. If you would prefer to use another library, such as *Datejs*, it should be straightforward to convert.

```
npm install moment --save
```

This is the code for `useClock`:

```
import {useEffect, useState} from "react";
import moment from "moment";

export default (formatOrInterval) => {
    const format = (typeof formatOrInterval === 'string')
        ? formatOrInterval : 'YYYY-MM-DDTHH:mm:ss.SSS';
    const interval = (typeof formatOrInterval === 'number')
        ? formatOrInterval : 500;
    const [response, setResponse] = useState(
        moment(new Date()).format(format));

    useEffect(() => {
        const newTimer = setInterval(() => {
            setResponse(moment(new Date()).format(format));
        }, interval);
        return () => clearInterval(newTimer);
    }, [format, interval]);

    return response;
};
```

We derive the date/time `format`, and the required *ticking* `interval` from the `formatOrInterval` parameter passed to the hook. Then we create a timer with `setInterval`. This time will set the `response` value every `interval` milliseconds. If the `response` is the same as previously, there will be no update of the interface. But if we set the `response` string to a new time, any component that relies on `useClock` will re-render.

We need to make sure that we cancel any timers that are no longer in use. We can do this using a feature of the `useEffect` hook. If we return a function at the end of our `useEffect` code, then that function will be called the next time `useEffect` needs to run. So we can use it to clear the old timer before creating a new one.

So if we pass a new format or interval to `useClock`, it will cancel its old timer and respond using a new timer.

Discussion

This recipe is an example of how you can use hooks to solve a simple problem simply. React code (the clue is in the name) is designed to react to dependency changes. Rather than thinking "How can I run this piece of code every second," the `useClock` hook allows you to write code that depends on the current time and hides away all of the gnarly details of creating timers, updating state, and clearing timers.

If you use the `useClock` hook several times in a component, then a time change can result in multiple renders. For example, if you have two clocks that format the current time in 12-hour format ("04:45") and 24-hour format ("16:45"), then your component will render twice when the minute changes. This is unlikely to cause much of a performance impact.

You can also use the `useClock` hook inside other hooks. If you create a `useMessages` hook to retrieve messages from a server, you can call `useClock` inside it to poll the server at regular intervals.

You can download the source for this recipe from the Github site.

3.5 Monitoring Online Status

Problem

Let’s say someone is using your application on their cell phone, and then they head into a subway with no data connection. How can you check that the network connection has disappeared? What’s a React-friendly way of updating your interface to either tell the user that there’s a problem or to disable some features that require network access?

Solution

We will create a hook called `useOnline` that will tell us whether or not we have a connection to a network. We need code that runs when the browser loses or regains a connection to the network. Fortunately, there are window/body-level events called “online” and “offline” that do exactly that:

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

export default () => {
    const [online, setOnline] = useState(navigator.onLine);

    useEffect(() => {
        if (window.addEventListener) {
            window.addEventListener("online", () => setOnline(true),
false);
            window.addEventListener("offline", () => setOnline(false),
false);
        } else {
            document.body.ononline = () => setOnline(true);
            document.body.onoffline = () => setOnline(false);
        }
    }, []);

    return online;
}
```

This hook manages its connection state in the `online` variable. When the hook is first run (notice the empty dependency array), we register listeners to the browser's online/offline events. When either of these events occurs, we can set the value of `online` to `true` or `false`. If this is a change to the current value, then any component using this hook will re-render.

Here's an example of the hook in action:

```
import useOnline from './useOnline';
import './App.css';

function App() {
    const online = useOnline();

    return <div className="App">
        <h1>Network Checker</h1>
        <span>
            You are now....
            {online
                ? <div className='App-indicator-online'>ONLINE</div>
                : <div className='App-indicator-offline'>OFFLINE</div>
            }
        </span>
    </div>;
}

export default App;
```

If you run the app, the page will currently show as online. If you disconnect/reconnect your network, the message will switch to "OFFLINE" then back to "ONLINE"

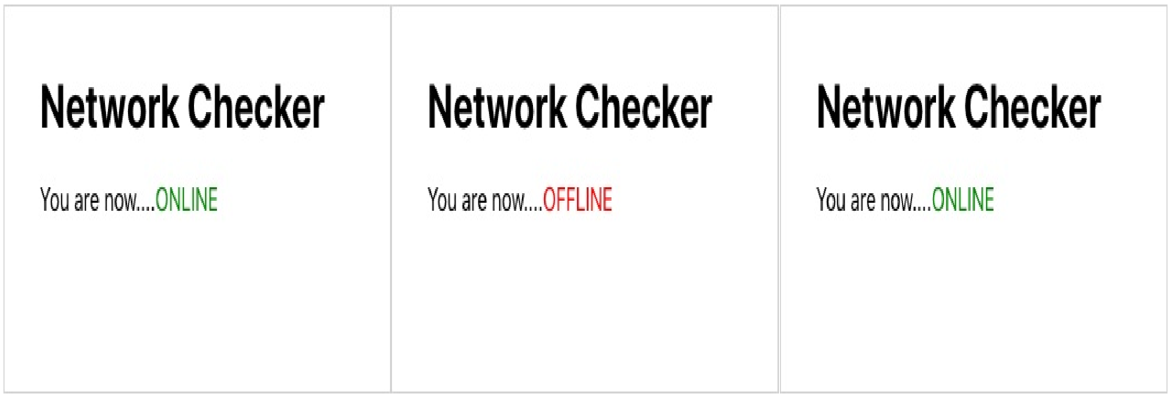


Figure 3-14. The code re-renders when the wifi is switched off and back on again

Discussion

It’s important to note that this hook checks your browser’s connection to a network, not whether it connects to the broader Internet or your server. If you would like to check that your server is running and available, you would have to write additional code.

You can download the source for this recipe from the Github site.

3.6 Manage Global Application State with Redux

Problem

In other recipes in this chapter, we’ve seen that you can manage complex component state with a pure JavaScript function called a *reducer*. Reducers simplify components and make business logic more testable.

But what if you have some data, such as the contents of a shopping basket, which needs to be accessed globally in the application?

Solution

We will use the *Redux* library to manage the global application state. Redux uses the same reducers we can give to the React `useReducer` function, but they are used to manage a single state object for the entire application. Plus, there are many extensions to Redux that solve common programming problems and develop and manage your application more quickly.

First, we need to install the Redux library:

```
npm install redux --save
```

We will also install the React-Redux library, which will make Redux far easier to use with React:

```
npm install react-redux --save
```

We’re going to use Redux to build a simple shopping application, which contains a shopping basket.

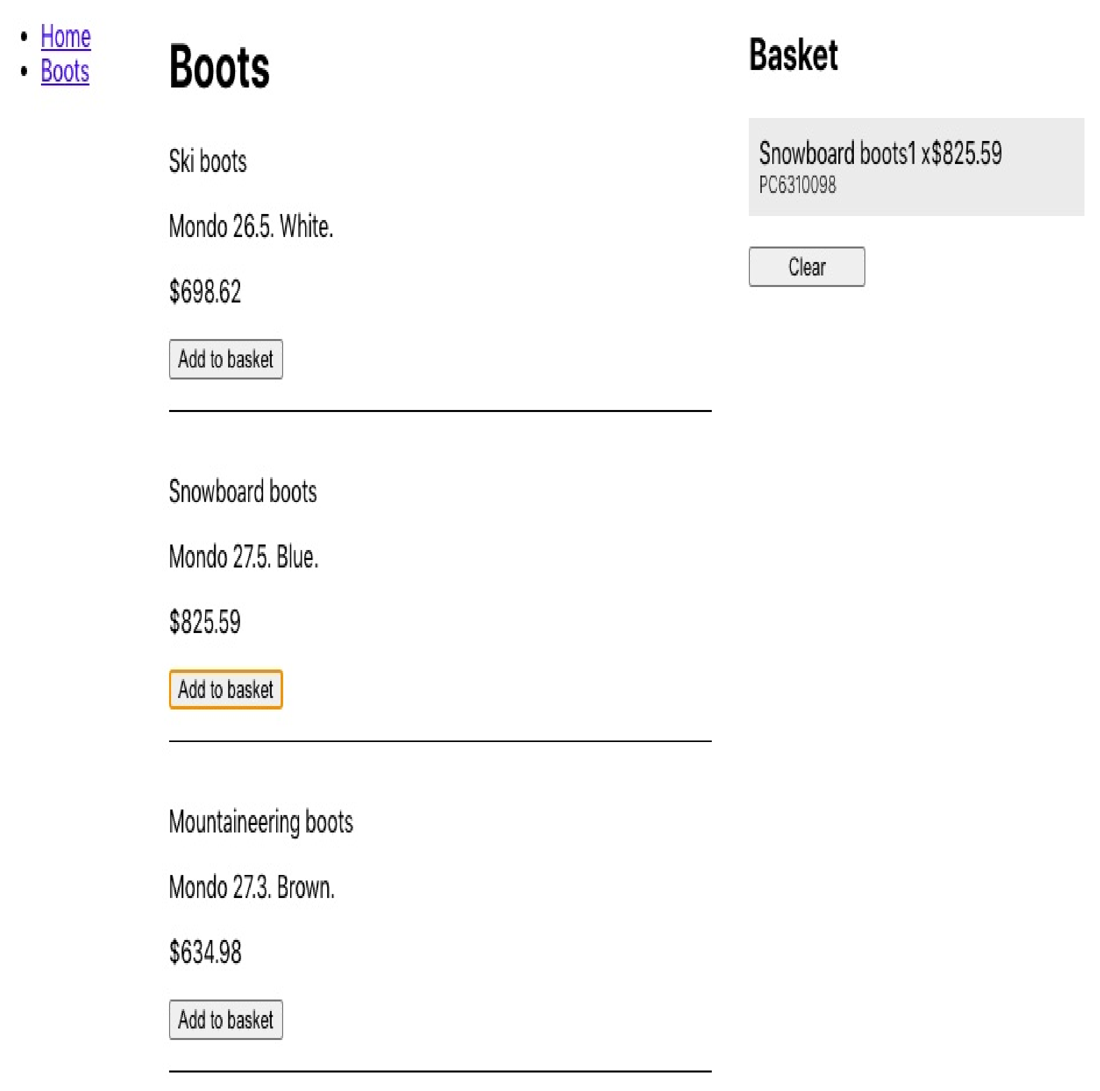


Figure 3-15. When a customer buys a product, the application adds it to the basket

If a customer clicks on a *Buy* button, the application adds the product to the basket. If they click the *Buy* button again, the quantity in the basket is updated. The basket will appear in several places across the application, so it’s a good candidate for moving to Redux. This is the reducer function that we will use to manage the basket:

```
export default (state = {}, action = {}) => {
    switch (action.type) {
        case 'buy': {
            const basket = state.basket ? [...state.basket] : [];
            const existing = basket.findIndex(
                item => item.productId === action.payload.productId);
            if (existing !== -1) {
                basket[existing].quantity = basket[existing].quantity
+ 1;
            } else {
                basket.push({quantity: 1, ...action.payload})
            }
            return {
                ...state,
                basket
            };
        }
        case 'clearBasket': {
            return {
                ...state,
                basket: []
            }
        }
        default:
            return {...state};
    }
};
```

TIP

We are creating a single reducer here. Once your application grows in size, you will probably want to split your reducer into smaller sub-reducers, which you can combine with the Redux `combineReducers` function.

The reducer function responds to `buy` and `clearBasket` actions. The `buy` action will either add a new item to the basket or update the quantity of an existing item if one has a matching `productId`. The `clearBasket` action will set the basket back to an empty array.

Now that we have a reducer function, we will use it to create a Redux *store*. The store is going to be our central repository for shared application state. To create a store, add these two lines to some top-level component such as `App.js`:

```
import reducer from "./reducer";

const store = createStore(reducer);
```

The store needs to be available globally in the app and to do that we need to inject it into the context of the components which might need it. The React-Redux library provides a component to inject the store in a component context called `Provider`:

```
<Provider store={store}>
  All the components inside here can access the store
</Provider>
```

This is what the `App.js` looks like in the example application you can find on the Github repository for this book:

```
export default (state = {}, action = {}) => {
    switch (action.type) {
        case 'buy': {
            const basket = state.basket ? [...state.basket] : [];
            const existing = basket.findIndex(
                item => item.productId === action.payload.productId);
            if (existing !== -1) {
                basket[existing].quantity = basket[existing].quantity
+ 1;
            } else {
                basket.push({quantity: 1, ...action.payload})
            }
            return {
                ...state,
                basket
            };
        }
        case 'clearBasket': {
            return {
                ...state,
                basket: []
            }
        }
        default:
            return {...state};
    }
};
```

Now that the store is available to our components, how do we use it? Redux-React allows you to access the store through hooks. If you want to read the contents of the global state, you can use `useSelector`:

```
const basket = useSelector(state => state.basket);
```

The `useSelector` hook accepts a function to extract part of the central state. Selectors are quite efficient and will only cause your component to re-render if the particular part of the state you are interested in changes.

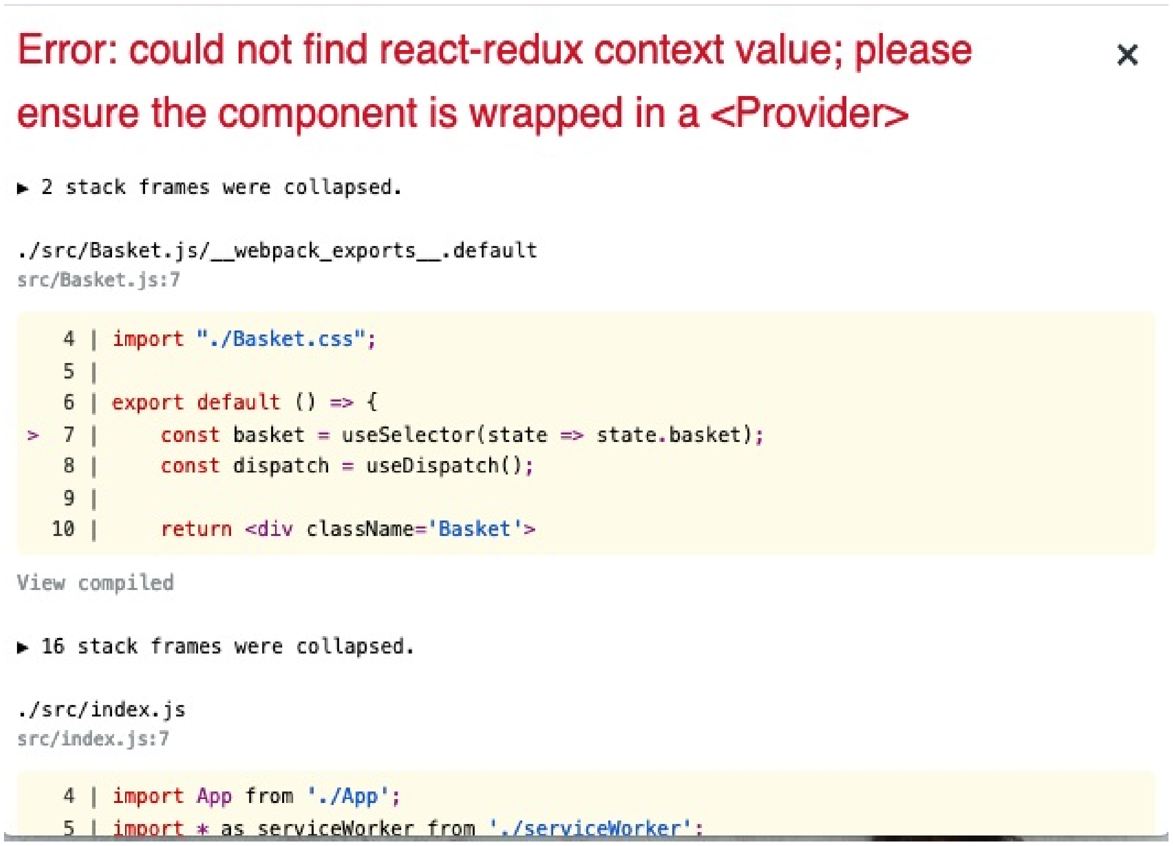
If you need to submit an action to the central store, you can do it with the `useDispatch` hook:

```
const dispatch = useDispatch();
```

This returns a `dispatch` function which you can use to send actions to the store:

```
dispatch({type: 'clearBasket'})}
```

These hooks work by extracting the store from the current context. If you forget to add a `Provider` to your application or try to run `useSelector` or `useDispatch` outside of a `Provider` context, you will get an error:


Error: could not find react-redux context value; please ensure the component is wrapped in a <Provider>

```
▶ 2 stack frames were collapsed.

./src/Basket.js/__webpack_exports__.default
src/Basket.js:7

import "./Basket.css";

export default () => {
    const basket = useSelector(state => state.basket);
    const dispatch = useDispatch();

    return <div className='Basket'>

View compiled

▶ 16 stack frames were collapsed.

./src/index.js
src/index.js:7

import App from './App';
import * as serviceWorker from './serviceWorker';
```

Figure 3-16. If you forget to include a Provider, you will get this error

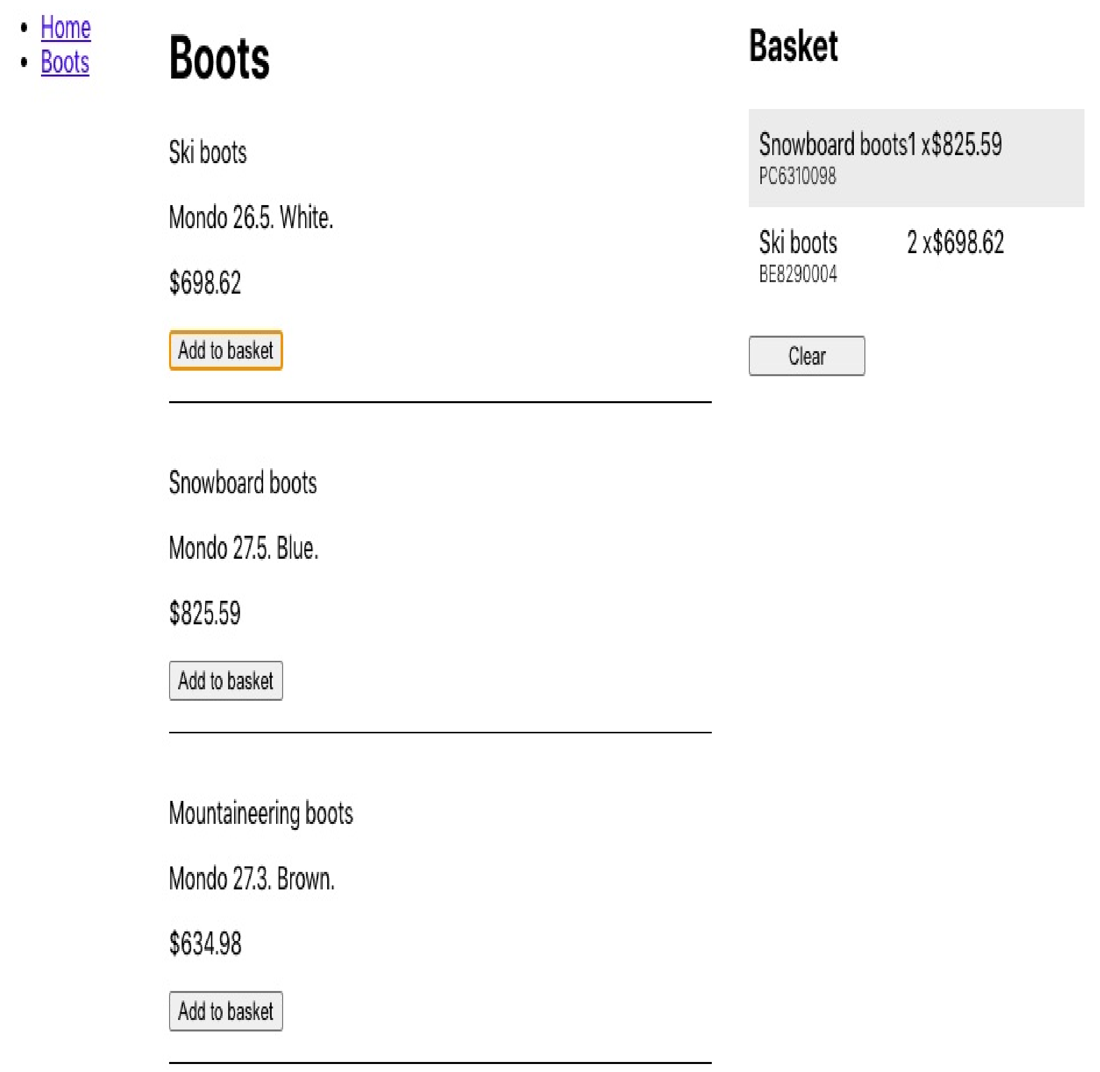
This is the completed `Basket` component that reads and clears the app-wide shopping basket:

```
import {useDispatch, useSelector} from "react-redux";

import "./Basket.css";

export default () => {
    const basket = useSelector(state => state.basket);
    const dispatch = useDispatch();

    return <div className='Basket'>
        <h2>Basket</h2>
        {
            (basket && basket.length) ?
                <>
                    {basket.map(item => <div className='Basket-item'>
                        <div className='Basket-itemName'>
                            {item.name}
                        </div>
                        <div className='Basket-itemProductId'>
                            {item.productId}
                        </div>
                        <div className='Basket-itemPricing'>
                            <div className='Basket-itemQuantity'>
                                {item.quantity}
                            </div>
                            <div className='Basket-itemPrice'>
                                {item.price}
                            </div>
                        </div>
                    </div>)}
                    <button
                        onClick={() => dispatch({type:
'clearBasket'})}>
                        Clear
                    </button>
                </>
                : "Empty"
        }
    </div>;
};
```

To demonstrate some code adding items to the basket, here’s a `Boots` component that allows a customer to buy a selection of products:

```
import {useDispatch} from "react-redux";

import "./Boots.css";

const products = [
    {productId: "BE8290004", name: 'Ski boots',
        description: 'Mondo 26.5. White.', price: 698.62},
    {productId: "PC6310098", name: 'Snowboard boots',
        description: 'Mondo 27.5. Blue.', price: 825.59},
    {productId: "RR5430103", name: 'Mountaineering boots',
        description: 'Mondo 27.3. Brown.', price: 634.98},
];

export default () => {
    const dispatch = useDispatch();

    return <div className='Boots'>
        <h1>Boots</h1>

        <dl className='Boots-products'>
            {
                products.map(product => <>
                    <dt>{product.name}</dt>
                    <dd>
                        <p>{product.description}</p>
                        <p>${product.price}</p>
                        <button
                            onClick={() => dispatch({type: 'buy',
payload: product})}>
                            Add to basket
                        </button>
                    </dd>
                </>)
            }
        </dl>
    </div>;
};
```

These two components may appear at very different locations in the component tree they share the same Redux store. As soon as a customer adds a product to the basket, the `Basket` component will automatically update with the change:



Boots

Ski boots

Mondo 26.5. White.

$698.62

Snowboard boots

Mondo 27.5. Blue.

$825.59

Mountaineering boots

Mondo 27.3. Brown.

$634.98

Basket

Snowboard boots1 x$825.59
PC6310098

Ski boots 2 x$698.62
BE8290004

Figure 3-17. The Redux-React hooks make sure that when a user buys a product, the Basket is re-rendered

Discussion

The Redux library has long been used with the React framework. For a long time, it seemed, almost every React application included Redux by default. It's probably true that Redux was often overused or used inappropriately. We have seen projects that have even banned local state properties in favor of using

Redux for **all** state. This is a mistake. Redux is intended for central application state management, not for simple component state. If you are storing data that is only of concern to a component, or its sub-components, you probably don’t need to use Redux.

However, if your application manages some global application state, then Redux is still the tool of choice.

You can download the source for this recipe from the Github site.

3.7 Survive Page Reloads with redux-persist

Problem

Redux is an excellent way of managing the application state centrally. However, it does have a small problem: when you reload the page, the entire state disappears:

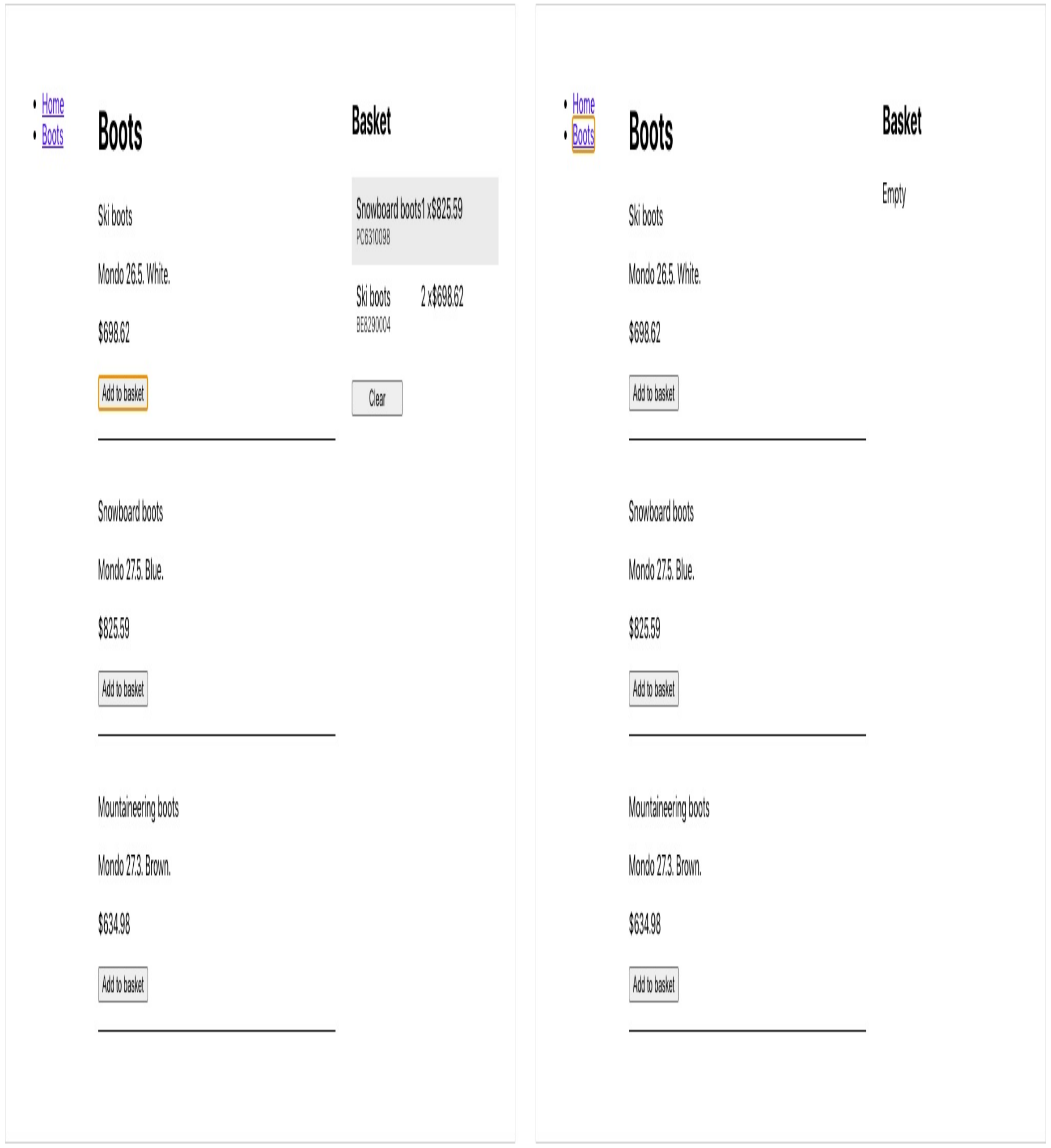


Figure 3-18. Redux state (left) is lost if the page is reloaded (right)

The state disappears because Redux keeps its state in memory. How do we prevent the state from disappearing?

Solution

We will use the `redux-persist` library to keep a copy of the Redux state in local-storage. To install `redux-persist` type:

```
npm install redux-persist --save
```

The first thing we need to do is create a *persisted reducer*, wrapped around our existing reducer:

```
import storage from 'redux-persist/lib/storage';

const persistConfig = {
    key: 'root',
    storage,
};

const persistedReducer = persistReducer(persistConfig, reducer);
```

The `storage` specifies where we will persist the Redux state: it will be in `localStorage` by default. The `persistConfig` says that we want to keep our state in a `localStorage` item called `persist:root`. When the Redux state changes, the `persistedReducer` will write a copy with `localStorage.setItem('persist:root', ...)`. We now need to create our Redux store with `persistedReducer`:

```
const store = createStore(persistedReducer);
```

We need to interject the `redux-persist` code between the Redux store and the code that's accessing the Redux store. We do that with a component called `PersistGate`:

```
import { PersistGate } from 'redux-persist/integration/react'
import {persistStore} from 'redux-persist';

const persistor = persistStore(store);
...
<Provider store={store}>
    <PersistGate loading={<div>Loading...</div>} persistor=
{persistor}>
        Components live in here
    </PersistGate>
</Provider>
```

The `PersistGate` must be *inside* the Redux `Provider` and *outside* the components that are going to use Redux. The `PersistGate` will watch for when the Redux state is lost and will then reload it from `localStorage`. It might take a moment to reload the data, and if you want to show that the UI is briefly busy, you can pass a `loading` component to the `PersistGate`. The loading component will be displayed in place of its child components when Redux is reloading. If you don't want a loading component, you can set it to `null`.

This is the final version of the modified `App.js` from the example app:

```
import {BrowserRouter, Route, Switch} from "react-router-dom";
import {Provider} from 'react-redux'
import {createStore} from 'redux';

import Menu from "./Menu";
import Home from "./Home";
import Boots from "./Boots";
import Basket from "./Basket";

import './App.css';
import reducer from "./reducer";

import {persistStore, persistReducer} from 'redux-persist';
import { PersistGate } from 'redux-persist/integration/react'
import storage from 'redux-persist/lib/storage';

const persistConfig = {
    key: 'root',
    storage,
};

const persistedReducer = persistReducer(persistConfig, reducer);

const store = createStore(persistedReducer);

const persistor = persistStore(store);

function App() {
    return (
        <div className="App">
            <Provider store={store}>
                <PersistGate loading={<div>Loading...</div>}
persistor={persistor}>
                    <BrowserRouter>
                        <Menu/>
                        <Switch>
                            <Route exact path='/'>
                                <Home/>
                            </Route>
                            <Route path='/boots'>
                                <Boots/>
                            </Route>
                        </Switch>
                        <Basket/>
                    </BrowserRouter>
                </PersistGate>
            </Provider>
        </div>
    );
}

export default App;
```

Now, when the user reloads the page, the Redux state survives:

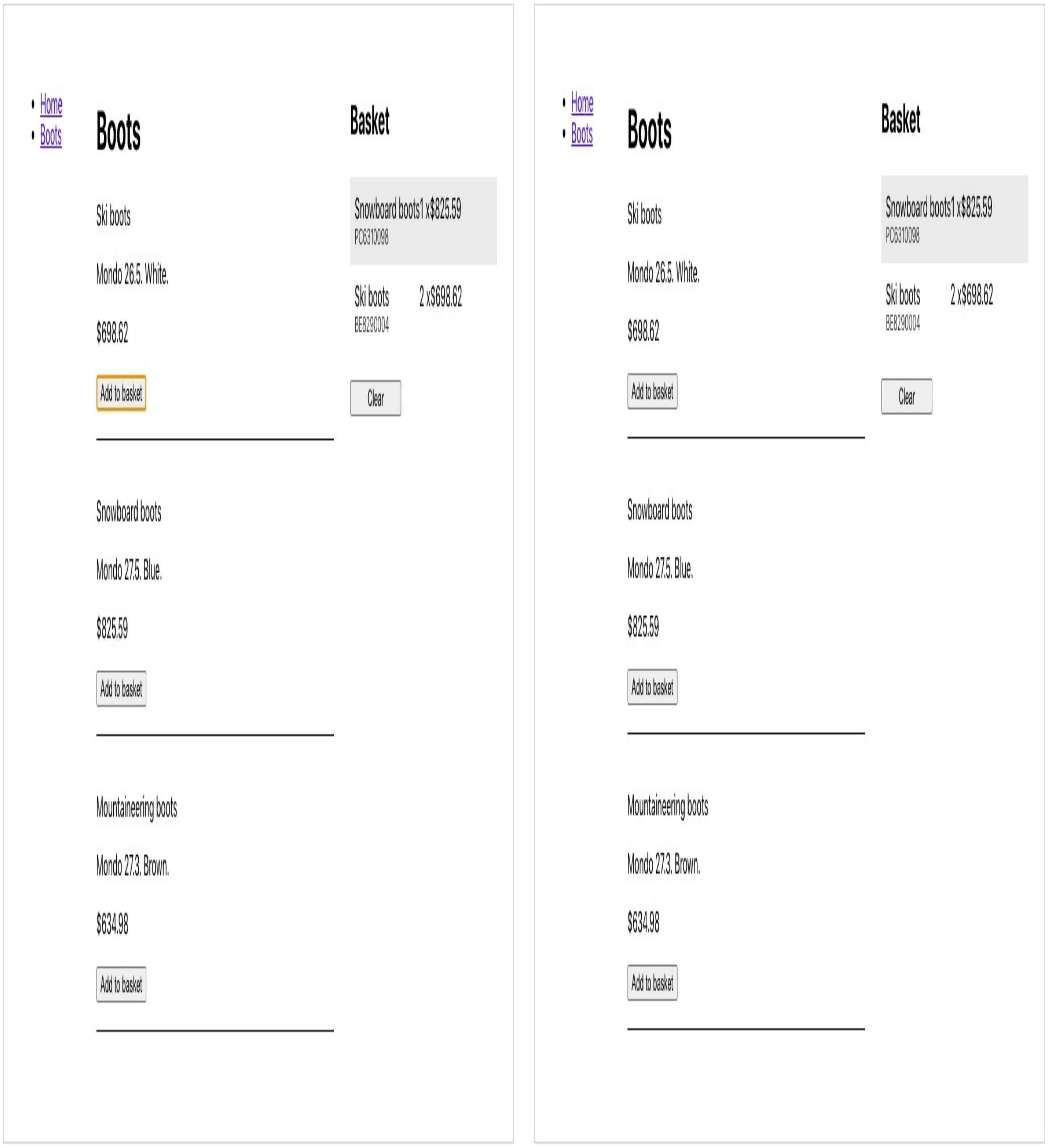


Figure 3-19. Redux state before the reload (left) and after (right)

Discussion

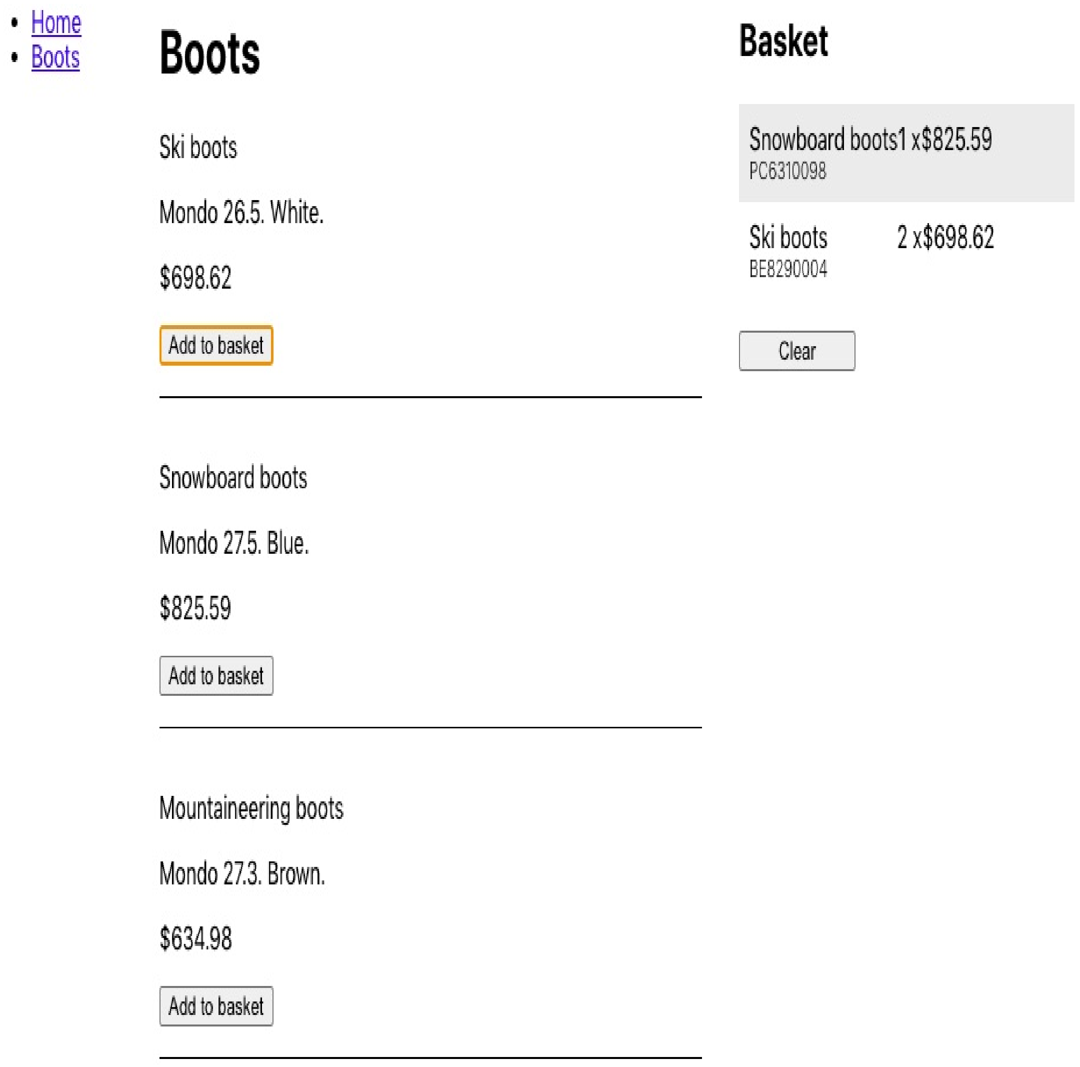
The `redux-persist` library is a simple way of persisting Redux state through page reloads. If you have a substantial amount of Redux data, you will need to be careful not to break the `localStorage` limit in the browser, which is typically 10MB. However, if your Redux data is that size, you should consider offloading some of it to a server.

You can download the source for this recipe from the Github site.

3.8 Calculate Derived State with reselect

Problem

When you extract your application state into an external object with a tool like Redux, you often need to process the data in some way before displaying it. For example, this is an application that we have used in a few recipes in this chapter:


- Home
- Boots

Boots

Ski boots

Mondo 26.5. White.

$698.62

Add to basket

Snowboard boots

Mondo 27.5. Blue.

$825.59

Add to basket

Mountaineering boots

Mondo 27.3. Brown.

$634.98

Add to basket

Basket

Snowboard boots1 x$825.59
PC6310098

Ski boots 2 x$698.62
BE8290004

Clear

Figure 3-20. What's the best method for calculating the total cost and tax of the basket?

What if we want to calculate the total cost of the items in the basket, and then calculate the amount of sales tax to pay? We could create a JavaScript function that reads through the basket items calculates both, but that function would have to recalculate the values every time the basket renders. Is there a way of calculating derived values from state that only needs to be performed when the state changes?

Solution

The Redux developers have created a library specifically designed to derive values efficiently from state objects, called `reselect`.

The `reselect` library creates *selector functions*. A selector function takes a single parameter–a state object–and returns a processed version.

We’ve already seen one selector in the Redux recipe in this chapter. We used it to return the current basket from the central Redux state:

```
const basket = useSelector(state => state.basket);
```

The `state => state.basket` is a selector function; it derives some value from a state object. The `reselect` library creates highly efficient selector functions that can cache their results if the state they depend upon has not changed.

To install `reselect`, enter this command:

```
npm install reselect --save
```

Let’s begin by creating a selector function that will:

- Count the total number of items in a basket, and
- Calculate the total cost of all of the items

We’ll call this function `summarizer`. Before we go into the details of how we’ll write, we’ll begin by writing a test that will show what it will need to do:

```
it('should be able to handle multiple products', () => {
    const actual = summarizer({
        basket: [
            {productId: '1234', quantity: 2, price: 1.23},
            {productId: '5678', quantity: 1, price: 1.50},
        ]
    });
    expect(actual).toEqual({itemCount: 3, cost: 3.96});
});
```

So if we pass it a state object, it will add up the quantities and costs and return an object containing the `itemCount` and `cost`.

We can create a selector function called `summarizer.js` with the `reselect` library like this:

```
import {createSelector} from "reselect";

export default createSelector(
    state => (state.basket || []),
    basket => ({
            itemCount: basket.reduce((i, j) => i + j.quantity, 0),
            cost: basket.reduce((i, j) => i + (j.quantity * j.price),
0),
        })
);
```

The `createSelector` function creates a selector function *based* on other selector functions. Each of the parameters passed to it–except the last parameter–should be selector functions. We are passing just one:

```
state => (state.basket || [])
```

Which extracts the basket from the state.

The final parameter passed to `createSelector` (the *combiner*) is a function which derives a new value, based of the results of the preceding selectors:

```
basket => ({
            itemCount: basket.reduce((i, j) => i + j.quantity, 0),
            cost: basket.reduce((i, j) => i + (j.quantity * j.price),
0),
        })
```

The `basket` value is the result of running the state through the first selector.

Why on Earth would anyone create functions this way? Isn't it **way** more complicated than just creating a JavaScript function manually, without the need to pass all of these functions to functions?

The answer is *efficiency*. State objects can be complex and might have dozens of attributes. But we are only interested in the contents of the `basket` attribute, and we don't want to have to recalculate our costs if anything else changes.

What `reselect` does is work out when the value it returns is likely to have changed. Let's say we call it one time, and it calculates the `itemCount` and `value` like this:

```
{itemCount: 3, cost: 3.96}
```

Then the user runs a bunch of commands that update personal preferences, posts a message to somebody, adds several things to their wish-list, etc.

Each of the events might update the global application state. But the next time we run the `summarizer` function, it will return the cached value that it produced before:

```
{itemCount: 3, cost: 3.96}
```

Why? Because it knows that this value is **only** dependent upon the `basket` value in the global state. And if that hasn't changed, then it doesn't need to recalculate the return value.

Because `reselect` allows us to build selector functions from other selector functions, we could build another selector called `taxer` to calculate the basket's sales tax:

```
import {createSelector} from "reselect";

export default createSelector(
    state => (state.basket || []),
    basket => ({
            itemCount: basket.reduce((i, j) => i + j.quantity, 0),
            cost: basket.reduce((i, j) => i + (j.quantity * j.price),
0),
        })
);
```

The `taxer` selector uses the value returned by the `summarizer` function. It takes the `cost` of the `summarizer` result and multiplies it by 25%. If the basket's summarized total doesn't change, then the `taxer` function will not need to update its result.

Now we have the `summarizer` and `taxer` selectors, we can use them inside a component, just as we would any other selector function:

```
import {useDispatch, useSelector} from "react-redux";

import "./Basket.css";
import summarizer from "./summarizer";
import taxer from "./taxer";

export default () => {
    const basket = useSelector(state => state.basket);
    const {itemCount, cost} = useSelector(summarizer);
    const tax = useSelector(taxer);
    const dispatch = useDispatch();

    return <div className='Basket'>
        <h2>Basket</h2>
        {
            (basket && basket.length) ?
                <>
                    {basket.map(item => <div className='Basket-item'>
                        <div className='Basket-itemName'>
                            {item.name}
                        </div>
                        <div className='Basket-itemProductId'>
                            {item.productId}
                        </div>
                        <div className='Basket-itemPricing'>
                            <div className='Basket-itemQuantity'>
                                {item.quantity}
                            </div>
                            <div className='Basket-itemPrice'>
                                {item.price}
                            </div>
                        </div>
                    </div>)}
                    <p>{itemCount} items</p>
                    <p>Total: ${cost.toFixed(2)}</p>
                    <p>Sales tax: ${tax.toFixed(2)}</p>
                    <button
                        onClick={() => dispatch({type:
'clearBasket'})}>
                        Clear
                    </button>
                </>
                : "Empty"
        }
    </div>;
};
```

When we run the code now, we see a summary at the bottom of the shopping basket, which will update whenever we buy a new product.

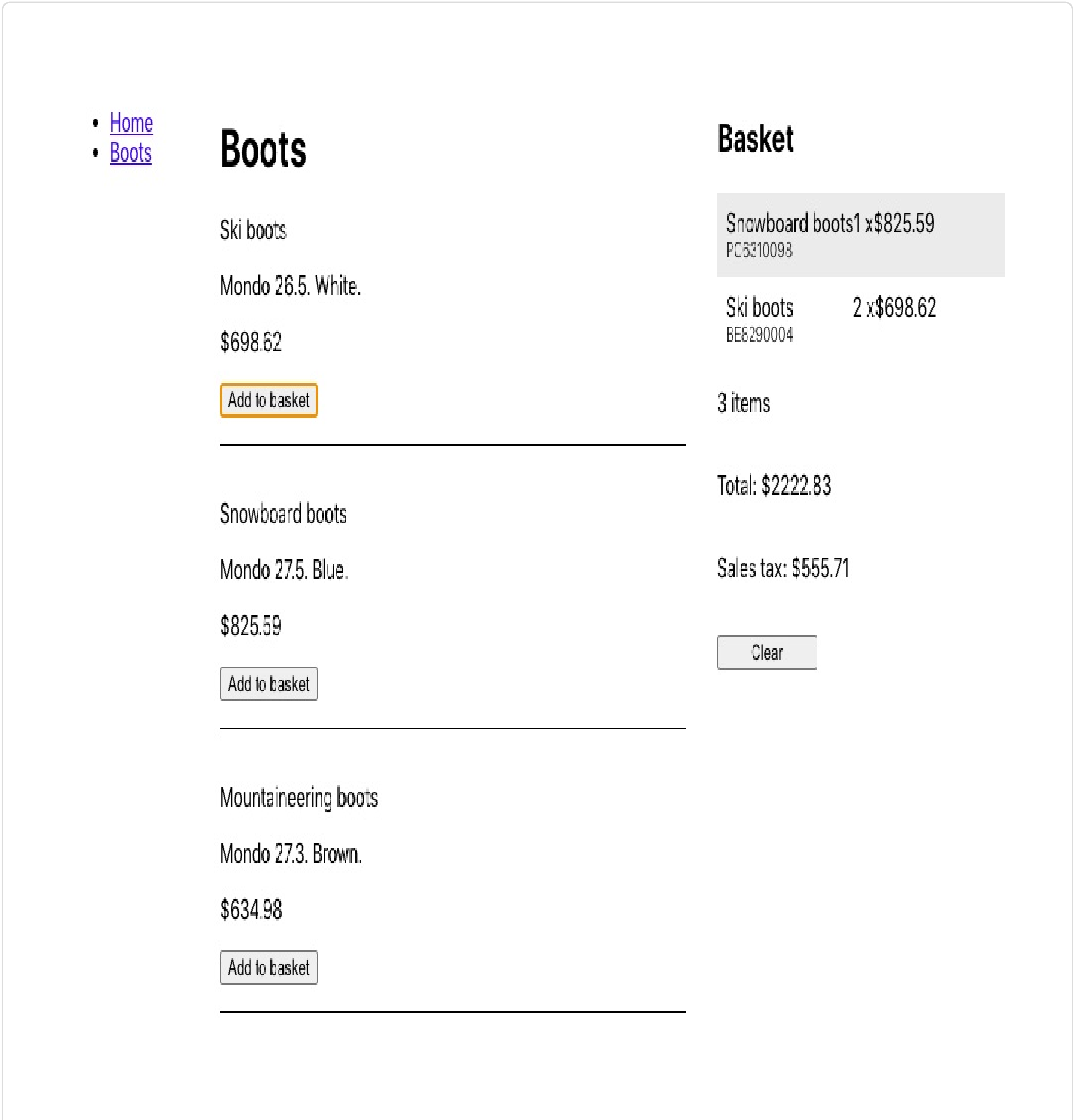


Figure 3-21. The selectors recalculate the total cost and sales tax only when the basket changes

Discussion

The first time you meet selector functions, they can seem complicated and hard to understand. But it is worth taking the time to understand them. There is nothing Redux-specific about them. There is no reason why you can’t also use them with non-Redux reducers. Because they have no dependencies beyond the `reselect` library itself, they are very easy to unit test. Example tests are included in the code for this chapter.

You can download the source for this recipe from the Github site.

Chapter 4. Interaction Design

In this chapter, we look at some recipes that address a bunch of typical interface problems. How do you deal with errors? How do you help people use your system? How do you create complex input sequences without writing a bunch of spaghetti code?

The recipes you'll find in this chapter we've used in some React applications over the years. It's an assortment: a collection of useful tips that we've seen come in useful, time and again. At the end of the chapter, we look at various ways of adding animation to your application. We take a low-tech approach where possible, and hopefully, the recipes we include will add meaning to your interface designs with a minimum of fuss.

4.1 Centralized Error Handler

Problem

It's hard to define precisely what makes good software good. But one thing that most excellent has in common is how it responds to errors and exceptions. There will always be exceptional, unexpected situations that occur when people are running your code: the network can disappear, the server can crash, the storage can become corrupted. It's important to consider how you should deal with these situations when they occur.

One approach that is almost certain to fail is to ignore the fact that error conditions occur and to hide the gory details of what went wrong. Somewhere, somehow, you need to leave a trail of evidence that you can use to prevent that error from happening again.

When we're writing server code, we might log the error details and return an appropriate message to a request. But if we're writing client code, we need a plan for how we'll deal with local errors. We might choose to display the crash's details to the user and ask them to file an error report. We might use a third-party service like Sentry.io[1] to remotely log the details.

Whatever our code does, it should be consistent. But how can we handle exceptions consistently in a React application?

Solution

In this recipe, we're going to look at one way of creating a centralized error handler. To be clear: this code won't automatically capture all exceptions. It still needs to be added explicitly to JavaScript `catch` blocks. It's also not a replacement for dealing with any error from which we can otherwise recover. If an order fails because the server is down for maintenance, it is much better to ask the user to try again later.

But this technique is useful for catching any errors for which we have not previously planned.

As a general principle, when something goes wrong, there are three things that you should tell the user:

- What happened
- Why it happened
- What they should do about it

In the example we show here, we're going to handle errors by display a dialog box that shows the details of a JavaScript `Error` object and asks the user to email the contents to systems support. We want a simple error-handler function that we can call when an error happens:

```
setVisibleError('Cannot do that thing', errorObject);
```

If we want to make the function readily available across the entire application, the usual way is by using a *context*. A context is a kind of scope that we can wrap around a set of React components. Anything we put into that context is available to all the child components. We will use our context to store the error-handler function that we can run when an error occurs.

We'll call our context `ErrorHandlerContext`:

```
import React from "react";

export default React.createContext(
    () => {}
);
```

To allow us to make the context available to a set of components, let's create an `ErrorHandlerProvider` component that will create an instance of the context and make it available to any child components we pass to it:

```
import ErrorHandlerContext from "./ErrorHandlerContext";

let setError = () => {};

export default (props) => {
    if (props.callback) {
        setError = props.callback;
    }

    return (
        <ErrorHandlerContext.Provider value={setError}>
            {props.children}
        </ErrorHandlerContext.Provider>
    );
};
```

Now we need some code that says what to do when we call the error-handler function. In our case, we need some code that will respond to an error report by displaying a dialog box containing all of the error details. If you want to handle errors differently, this is the code you need to modify:

```
import {useCallback, useState} from "react";
import ErrorHandlerProvider from "./ErrorHandlerProvider";
import ErrorDialog from "./ErrorDialog";
```

```
export default (props) => {
    const [error, setError] = useState();
    const [errorTitle, setErrorTitle] = useState();
    const [action, setAction] = useState();

    if (error) {
        console.error(
            "An error has been thrown",
            errorTitle,
            JSON.stringify(error)
        );
    }

    const callback = useCallback((title, err, action) => {
        console.error("ERROR RAISED ");
        console.error("Error title: ", title);
        console.error("Error content", JSON.stringify(err));
        setError(err);
        setErrorTitle(title);
        setAction(action);
    }, []);
    return (
        <ErrorHandlerProvider
            callback={callback}
        >
            {props.children}

            {error && (
                <ErrorDialog
                    title={errorTitle}
                    onClose={() => {
                        setError(null);
                        setErrorTitle("");
                    }}
                    action={action}
                    error={error}
                />
            )}
        </ErrorHandlerProvider>
    );
};
```

The `ErrorContainer` displays the details using an `ErrorDialog`. We won't go into the details of the code for `ErrorDialog` here [2] as this is the code that you are most likely to replace with your implementation.

We need to wrap the bulk of our application in an `ErrorContainer`. Any components inside the `ErrorContainer` will be able to call the error-handler:

```
import './App.css';
import ErrorContainer from "./ErrorContainer";
import ClockIn from "./ClockIn";

function App() {
  return (
    <div className="App">
      <ErrorContainer>
        <ClockIn/>
      </ErrorContainer>
    </div>
  );
}

export default App;
```

How does a component use the error-handler? We’ll create a custom hook called `useErrorHandler()` which will get the error-handler function out of the context and return it:

```
import ErrorHandlerContext from "./ErrorHandlerContext";
import { useContext } from "react";

const useErrorHandler = () => useContext(ErrorHandlerContext);

export default useErrorHandler;
```

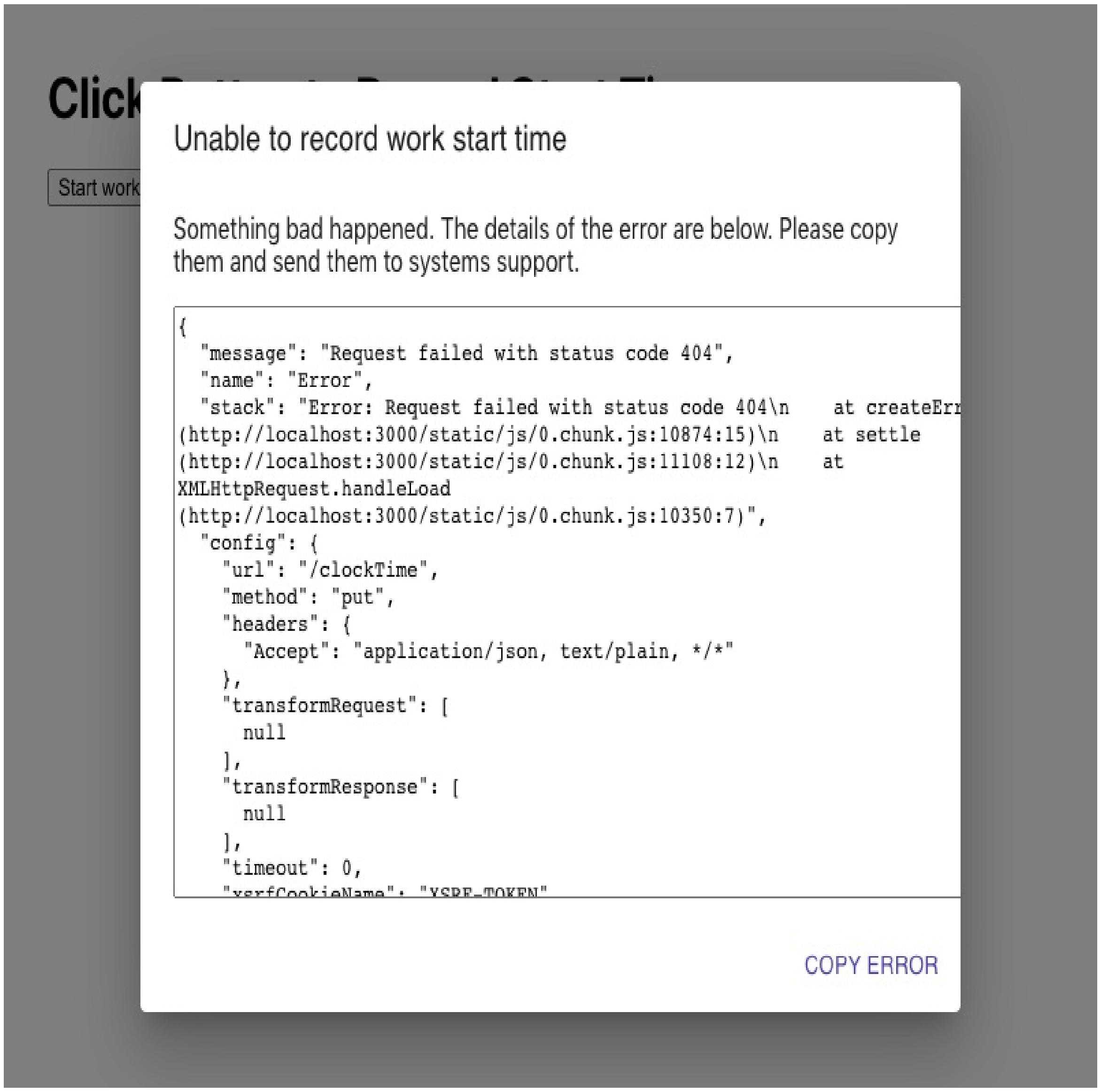
That’s quite a complex set of code, but now we come to use the error-handler, it’s very, very simple. This example piece of code tries to make a network request when a user presses a button. If the network request fails, then the details of the error are passed to the error-handler:

```
import useErrorHandler from "./useErrorHandler";
import axios from "axios";

export default () => {
    const setVisibleError = useErrorHandler();

    const doClockIn = async () => {
        try {
            await axios.put('/clockTime');
        } catch(err) {
            setVisibleError('Unable to record work start time', err);
        }
    };

    return <>
        <h1>Click Button to Record Start Time</h1>
        <button onClick={doClockIn}>Start work</button>
        </>;
}
```

You can see what the app looks like when it start in *figure 4-1*.

Figure 4-1. The time-recording app

When you click the button, the network request fails because the server code doesn't exist. *Figure 4-2* shows the error dialog that appears. Notice that it shows: what went wrong, why it went wrong and what the user should do about it.



Unable to record work start time

Something bad happened. The details of the error are below. Please copy them and send them to systems support.

```
{
  "message": "Request failed with status code 404",
  "name": "Error",
  "stack": "Error: Request failed with status code 404\n    at createErr
(http://localhost:3000/static/js/0.chunk.js:10874:15)\n    at settle
(http://localhost:3000/static/js/0.chunk.js:11108:12)\n    at
XMLHttpRequest.handleLoad
(http://localhost:3000/static/js/0.chunk.js:10350:7)",
  "config": {
    "url": "/clockTime",
    "method": "put",
    "headers": {
      "Accept": "application/json, text/plain, */*"
    },
    "transformRequest": [
      null
    ],
    "transformResponse": [
      null
    ],
    "timeout": 0,
    "xsrfCookieName": "XSRF-TOKEN"
```

Figure 4-2. When the network request throws an exception, we pass it to the error handler

Discussion

Of all of the recipes that we’ve created over the years, this one has saved the most time. During development, code often breaks, and if the only evidence of a failure is a stack trace hidden away inside the JavaScript console, you are likely to miss it.

Significantly when some piece of infrastructure (networks, gateways, servers, databases) fails, this small amount of code can save you untold hours tracking down the cause.

You can download the source for this recipe from the Github site.

4.2 Create an Interactive Help Guide

Problem

Tim Berners-Lee deliberately designed the web to have very few features. It has a simple protocol (HTTP), and it originally had a straightforward markup language (HTML). The lack of complexity meant that new users of web sites immediately knew how to use them. If you saw something that looked like a hyperlink, you could click on it and go to another page.

But rich JavaScript applications have changed all that. No longer are web applications a collection of hyperlinked web pages. Instead, they resemble old desktop applications; they are more powerful and feature-rich, but the down-side is that they are now far more complex to use.

How can your application help people to use it?

Solution

We're going to build a simple help-system that you can overlay onto an existing application. When the user opens the help, they will see a series of popup notes that describe how to use the various features they can see on the page, as show in *figure 4-3*.

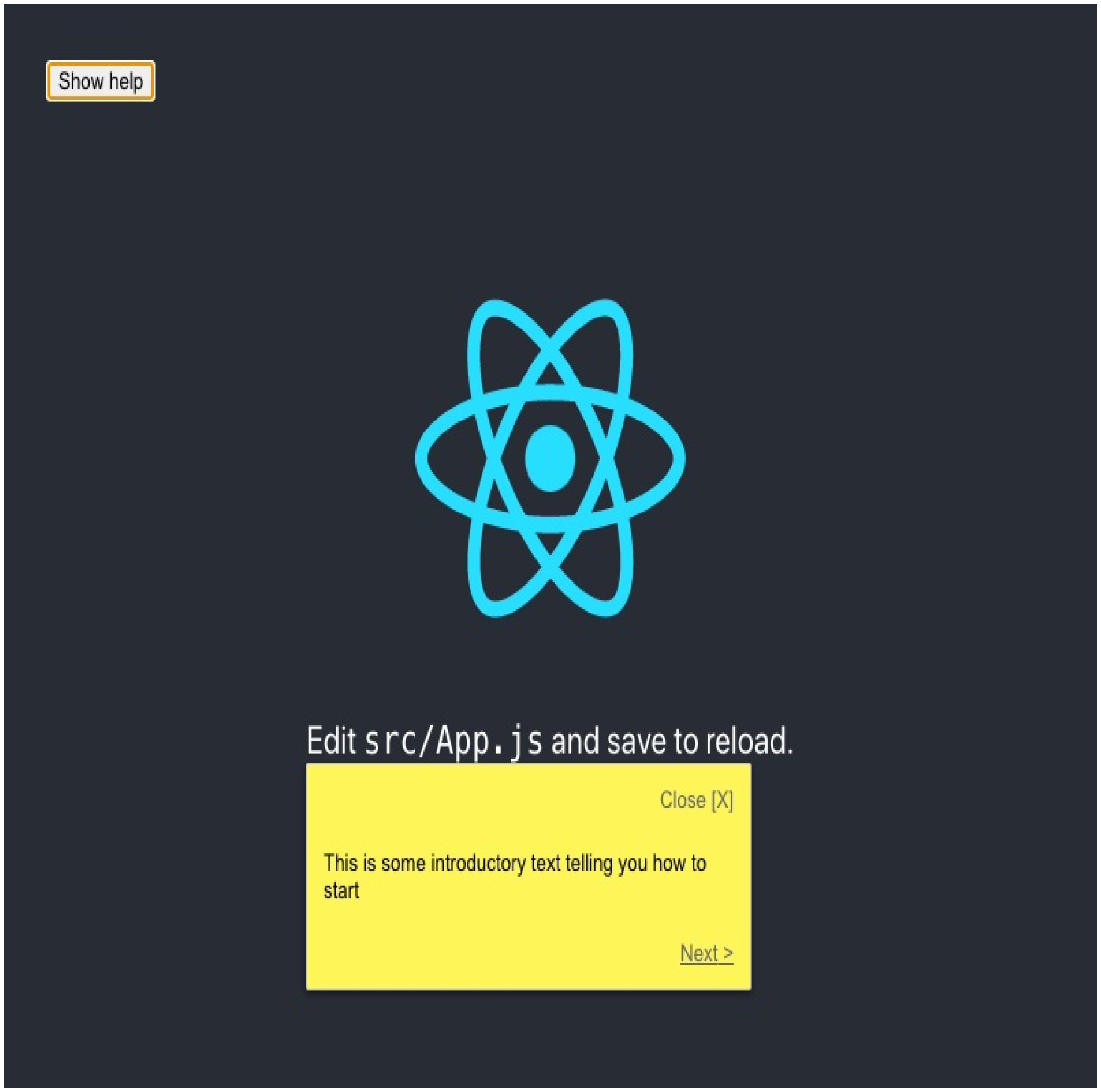

Show help
Edit src/App.js and save to reload.
Close [X]
This is some introductory text telling you how to start
Next >

Figure 4-3. Show a sequence of help messages when the user asks

We want something that will be easy to maintain and will only provide help for visible components. That sounds like quite a big task, so let's begin by first constructing a component that will display a popup-help message:

```
import {Popper} from "@material-ui/core";
import './HelpBubble.css';

export default (props) => {
    const element = props.forElement ?
document.querySelector(props.forElement) : null;
```

```
    return element ?
        <Popper className='HelpBubble-container'
                open={props.open}
                anchorEl={element}
                placement={props.placement || 'bottom-start'}>
            <div className='HelpBubble-close'
                 onClick={props.onClose}
            >Close [X]
            </div>
            {props.content}
            <div className='HelpBubble-controls'>
                {
                    props.previousLabel ?
                        <div className='HelpBubble-control HelpBubble-
previous'
                             onClick={props.onPrevious}
                        >&lt; {props.previousLabel}
                        </div>
                        : <div>&nbsp;</div>
                }
                {
                    props.nextLabel ?
                        <div className='HelpBubble-control HelpBubble-
next'
                             onClick={props.onNext}
                        >{props.nextLabel} &gt;</div>
                        : <div>&nbsp;</div>
                }
            </div>
        </Popper>
        : null;
}
```

We're using the `Popper` component from the `@material-ui` library. The `Popper` component that can be anchored on the page, next to some other component. Our `HelpBubble` takes a `forElement` string, which will represent a CSS selector such as `".class-name"` or `"#some-id"`. Will use selectors to associate things on the screen with popup messages.

Now that we have a popup message component, we'll need something that coordinates a sequence of `HelpBubbles`. We'll call this the `HelpSequence`:

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

import HelpBubble from "./HelpBubble";
```

```
function isVisible(e) {
    return !!( e.offsetWidth || e.offsetHeight ||
e.getClientRects().length );
}

export default (props) => {
    const [position, setPosition] = useState(0);
    const [sequence, setSequence] = useState();

    useEffect(() => {
        if (props.sequence) {
            const filter = props.sequence.filter(i => {
                if (!i.forElement) {
                    return false;
                }
                const element = document.querySelector(i.forElement);
                if (!element) {
                    return false;
                }
                return isVisible(element);
            });
            setSequence(filter);
        } else {
            setSequence(null);
        }
    }, [props.sequence, props.open]);

    const data = sequence && sequence[position];

    useEffect(() => {
        setPosition(0);
    }, [props.open]);

    const onNext = () => setPosition(p => {
        if (p === sequence.length - 1) {
            props.onClose && props.onClose();
        }
        return p + 1;
    });

    const onPrevious = () => setPosition(p => {
        if (p === 0) {
            props.onClose && props.onClose();
        }
        return p - 1;
    });

    return <div className='HelpSequence-container'>
        {
            data &&
            <HelpBubble open={props.open} forElement={data.forElement}
                        placement={data.placement}
                        onClose={props.onClose}
                        previousLabel={(position > 0) && 'Previous'}
                        nextLabel={(position < sequence.length - 1) ?
'Next' : 'Finish'}
                        onPrevious={onPrevious}
                        onNext={onNext}
                        content={data.text}
            />
        }
    </div>;
}
```

The `HelpSequence` takes an array of JavaScript objects like this:

```
[
    {forElement: "p", text: "This is some introductory text telling
you how to start"},
    {forElement: ".App-link", text: "This will show you how to use
React"},
    {forElement: ".App-nowhere", text: "This help text will never
appear"},
]
```

and converts it into a dynamic sequence of `HelpBubbles`. It will only show a `HelpBubble` if it can find an element that matches the `forElement` selector. It then places the `HelpBubble` next to element and shows the help text.

Let's add a `HelpSequence` to the default `App.js` code generated by `create-react-app`:

```
import {useState} from 'react';
import logo from './logo.svg';
import HelpSequence from "./HelpSequence";
import './App.css';

function App() {
  const [showHelp, setShowHelp] = useState(false);

  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
```

```
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
      <button onClick={() => setShowHelp(true)}>Show help</button>
      <HelpSequence
        sequence={[
          {forElement: "p", text: "This is some introductory text
telling you how to start"},
          {forElement: ".App-link", text: "This will show you how to
use React"},
          {forElement: ".App-nowhere", text: "This help text will
never appear"},
        ]}
        open={showHelp}
        onClose={() => setShowHelp(false)}
      />
    </div>
  );
}

export default App;
```

To begin with, we cannot see anything different, other than a help-button (see *figure 4-4*).

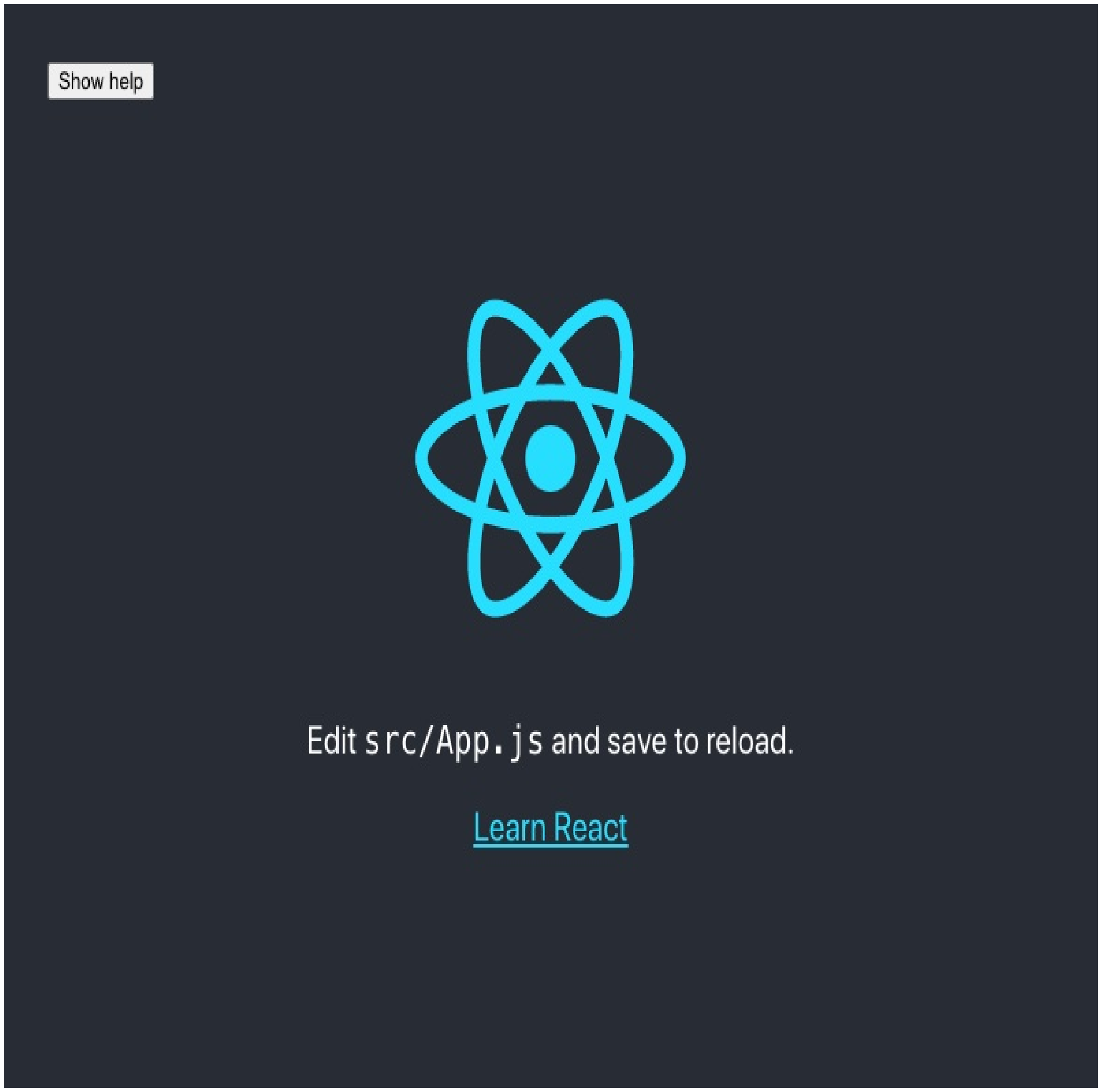
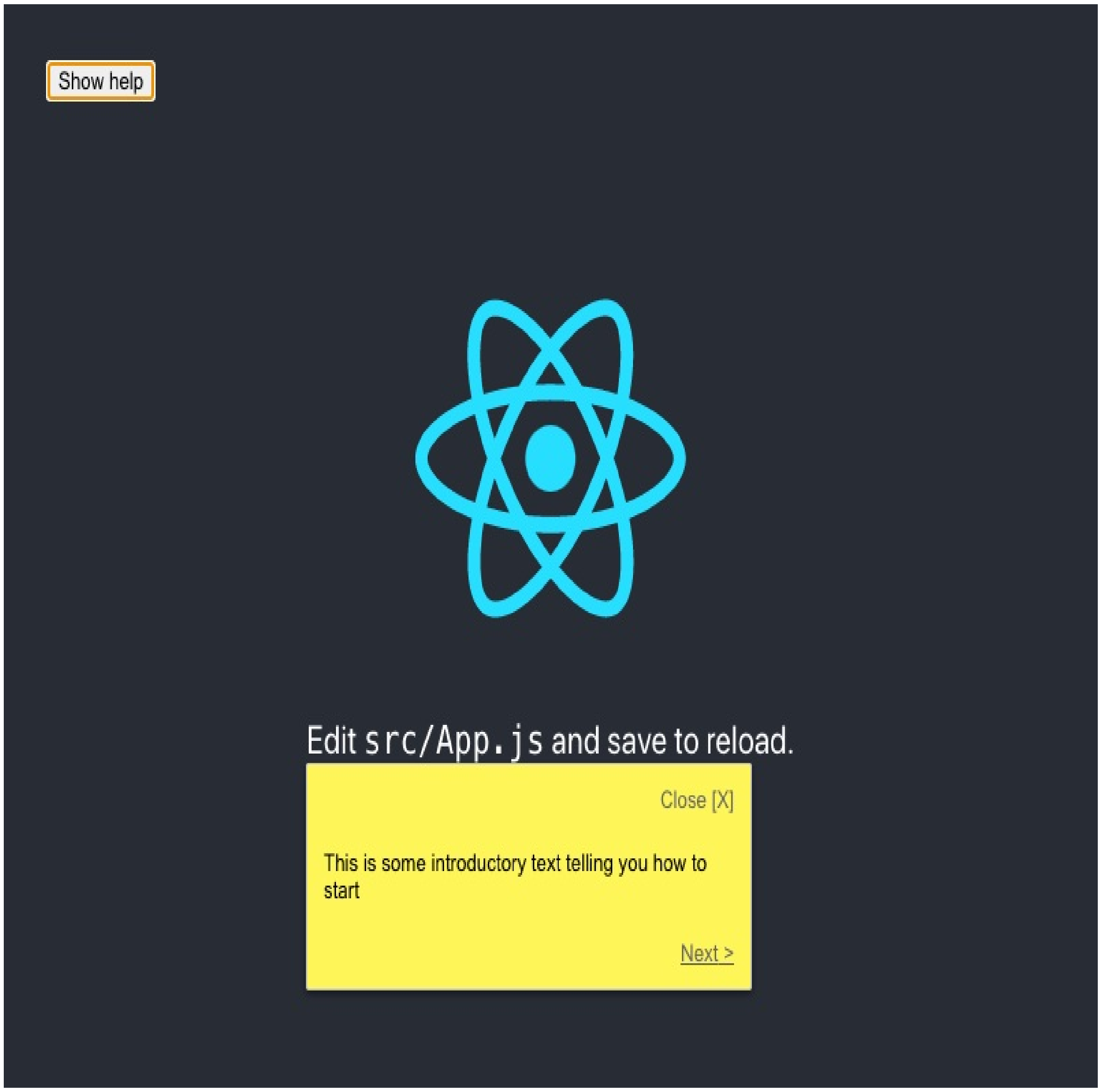


Figure 4-4. The application when it first loads

When the user clicks the help button, the first help topic appears, as shown in *figure 4-5*.



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

This is some introductory text telling you how to start

Figure 4-5. When they click the help button, the help bubble appears for the first match

Figure 4-6 shows that when the user clicks next, the help will move to the next element. The user can continue moving to the next help item until there are no more matching elements visible.

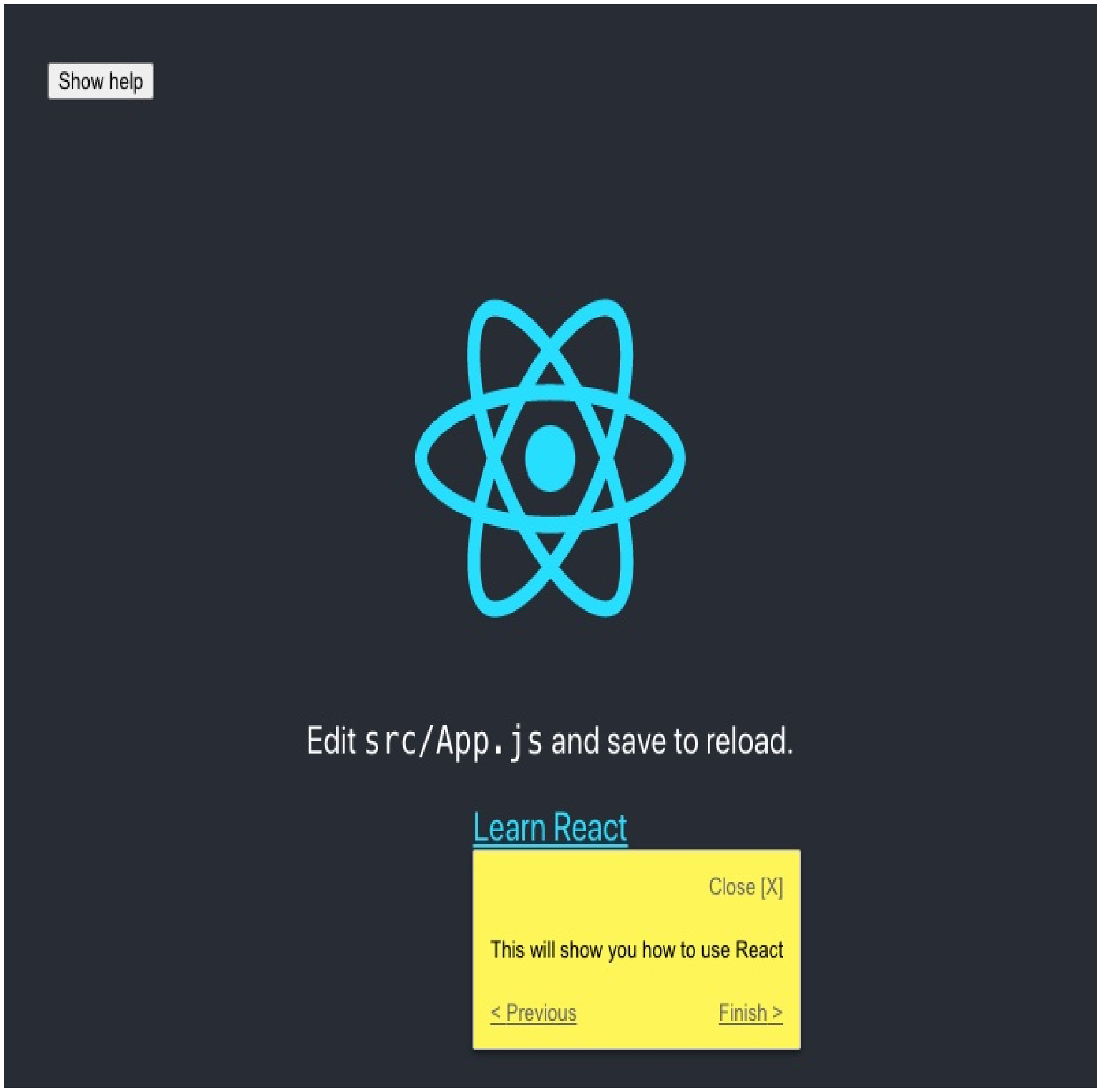

Show help
Edit src/App.js and save to reload.
Learn React
Close [X]
This will show you how to use React
< Previous
Finish >

Figure 4-6. The final element has a Finish button

Discussion

Adding interactive help to your application makes your user-interface *discoverable*. Developers spend a lot of their time adding functionality to applications that people might never use, simply because they don't know that it's there.

The implementation you can see this in this recipe displays the help as simple plain text. You might consider using Markdown, as this will allow for a richerexperience, and help topics can then include links to other more expansive help pages.[3]

The help-topics are automatically limited to just those elements which are visible on the page. You could choose to create either a separate help-sequence for each page or choose to create a single large help-sequence that will automatically adapt to the user’s current view of the interface.

Finally, a help system like this is perfectly suited for storage in a headless CRM, which will allow you to update help dynamically, without the need to create a new deployment each time.

You can download the source for this recipe from the Github site.

4.3 Use Reducers for Complex Sequences

Problem

Applications frequently need users to follow a sequence of actions. They might be completing the steps in a wizard, or (see *figure 4-7*) they might need to log in and confirm some dangerous operation.

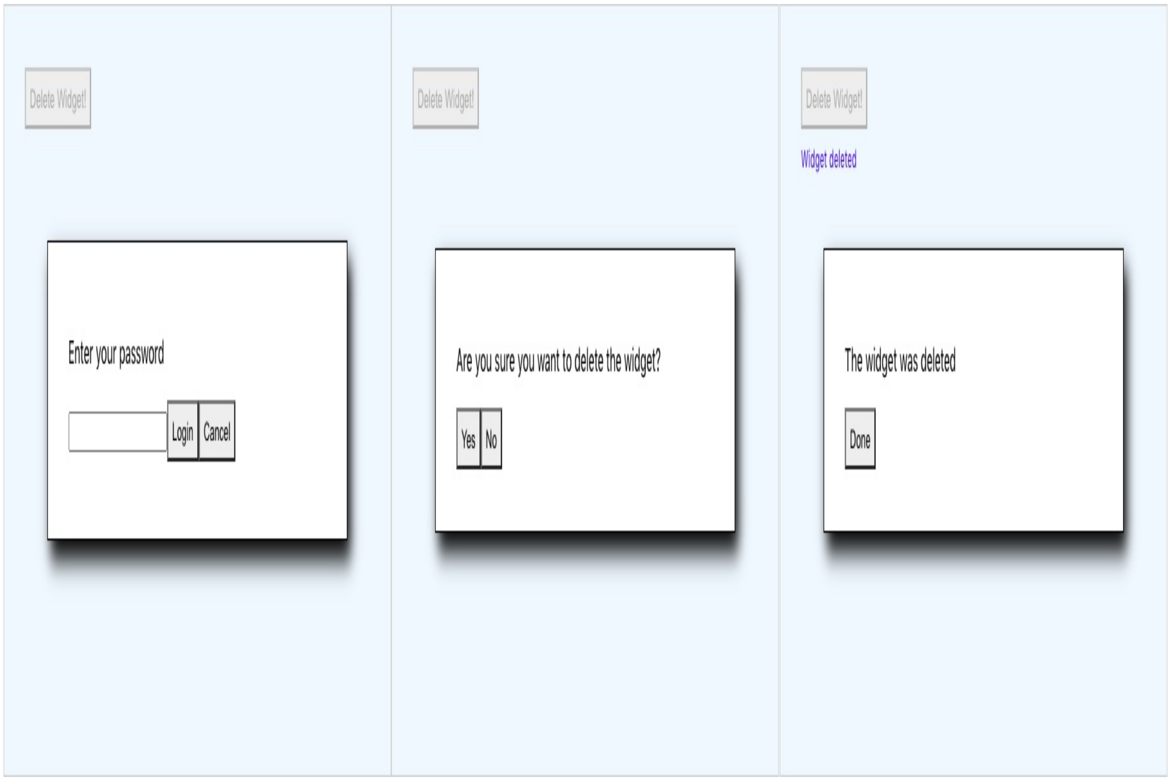


Figure 4-7. This deletion process requires logging in and then confirming the deletion

Not only will the user need to perform a sequence of steps, but the steps might be conditional. If the user has logged in recently, they perhaps don't need to log in again. They might want to cancel some-way through the sequence.

If you model the complex sequences inside your components, you can soon find your application is full of spaghetti code.

Solution

We are going to use a reducer to manage a complex sequence of operations. We introduced reducers for managing state in chapter 3. A reducer is a function that accepts a state object and an action. The reducer uses the action to decide how to change the state, and it must have no side-effects.

Because reducers have no user-interface code, they are perfect for managing gnarly pieces of inter-related state without worrying about how it will appear on the screen. They are particularly amenable to unit testing.

For example, let us say we will implement the deletion sequence mentioned at the start of this recipe. We can begin in classic test-driven style by writing a unit test:

```
import deletionReducer from "./deletionReducer";

describe('deletionReducer', () => {
    it('should show the login dialog if we are not logged in', () => {
        const actual = deletionReducer({}, {type: 'START_DELETION'});
        expect(actual.showLogin).toBe(true);
        expect(actual.message).toBe('');
        expect(actual.deleteButtonDisabled).toBe(true);
        expect(actual.loginError).toBe('');
        expect(actual.showConfirmation).toBe(false);
    });
});
```

Here our reducer function is going to be called `deletionReducer`. We pass it an empty object (`{}`) and an action that says we want to start the deletion process (`{type: 'START_DELETION'}`); We then say that we expect the new version of the state to have a `showLogin` value of true, a `showConfirmation` value to be false and so on.

We can then implement the code for a reducer to do just that:

```
function deletionReducer(state, action) {
    switch (action.type) {
        case 'START_DELETION':
            return {
                ...state,
                showLogin: true,
                message: '',
                deleteButtonDisabled: true,
                loginError: '',
                showConfirmation: false,
            };
        default:
            return null; // Or anything
    }
};
```

At first, we are merely setting the state attributes to values that pass the test. As we add more and more tests, our reducer improves as it handles more situations.

Eventually, we get something that looks like this:[4]

```
function deletionReducer(state, action) {
    switch (action.type) {
        case 'START_DELETION':
            return {
                ...state,
                showLogin: !state.loggedIn,
                message: '',
                deleteButtonDisabled: true,
                loginError: '',
                showConfirmation: !!state.loggedIn,
            };
        case 'CANCEL_DELETION':
            return {
                ...state,
                showLogin: false,
                showConfirmation: false,
                showResult: false,
                message: 'Deletion canceled',
                deleteButtonDisabled: false,
            };
        case 'LOGIN':
            const passwordCorrect = action.payload === 'swordfish'
            return {
                ...state,
                showLogin: !passwordCorrect,
                showConfirmation: passwordCorrect,
                loginError: passwordCorrect ? '' : 'Invalid password',
                loggedIn: true,
            };
        case 'CONFIRM_DELETION':
            return {
                ...state,
                showConfirmation: false,
                showResult: true,
                message: 'Widget deleted',
            };
        case 'FINISH':
            return {
                ...state,
                showLogin: false,
                showConfirmation: false,
                showResult: false,
                deleteButtonDisabled: false,
            };
        default:
            throw new Error('Unknown action: ' + action.type);
    }
}

export default deletionReducer;
```

Although this code is complicated, you can write it quickly if you create the tests first.

Now that we have the reducer, we just need to use it in our application.

```
import {useReducer, useState} from 'react';
import './App.css';
import deletionReducer from "./deletionReducer";

function App() {
    const [state, dispatch] = useReducer(deletionReducer, {});
    const [password, setPassword] = useState();

    return <div className="App">
        <button
            onClick={() => {
                dispatch({type: 'START_DELETION'});
            }}
            disabled={state.deleteButtonDisabled}
        >Delete Widget!
        </button>
        <div className='App-message'>{state.message}</div>
        {
            state.showLogin &&
                <div className='App-dialog'>
                    <p>Enter your password</p>
                    <input type='password' value={password}
                        onChange={evt =>
setPassword(evt.target.value)}/>
                    <button onClick={() => dispatch({type: 'LOGIN',
                        payload: password})}>Login</button>
                    <button onClick={() => dispatch({type:
'CANCEL_DELETION'})}
                        >Cancel</button>
                    <div className='App-error'>{state.loginError}
</div>
                </div>
        }
        {
            state.showConfirmation &&
                <div className='App-dialog'>
                    <p>Are you sure you want to delete the widget?</p>
                    <button onClick={() => dispatch({
                        type: 'CONFIRM_DELETION'})}>Yes</button>
                    <button onClick={() => dispatch({
                        type: 'CANCEL_DELETION'})}>No</button>
                </div>
        }
        {
            state.showResult &&
            <div className='App-dialog'>
                <p>The widget was deleted</p>
                <button onClick={() => dispatch({
                    type: 'FINISH'})}>Done</button>
            </div>
        }
    </div>;
}

export default App;
```

Most of this code is purely creating the user-interface for each of the dialogs that appear in the sequence. There is virtually no logic in this component. It just does what the reducer tells it. It will take the user through the *happy path* of logging in and confirming the deletion (see *figure 4-8*).

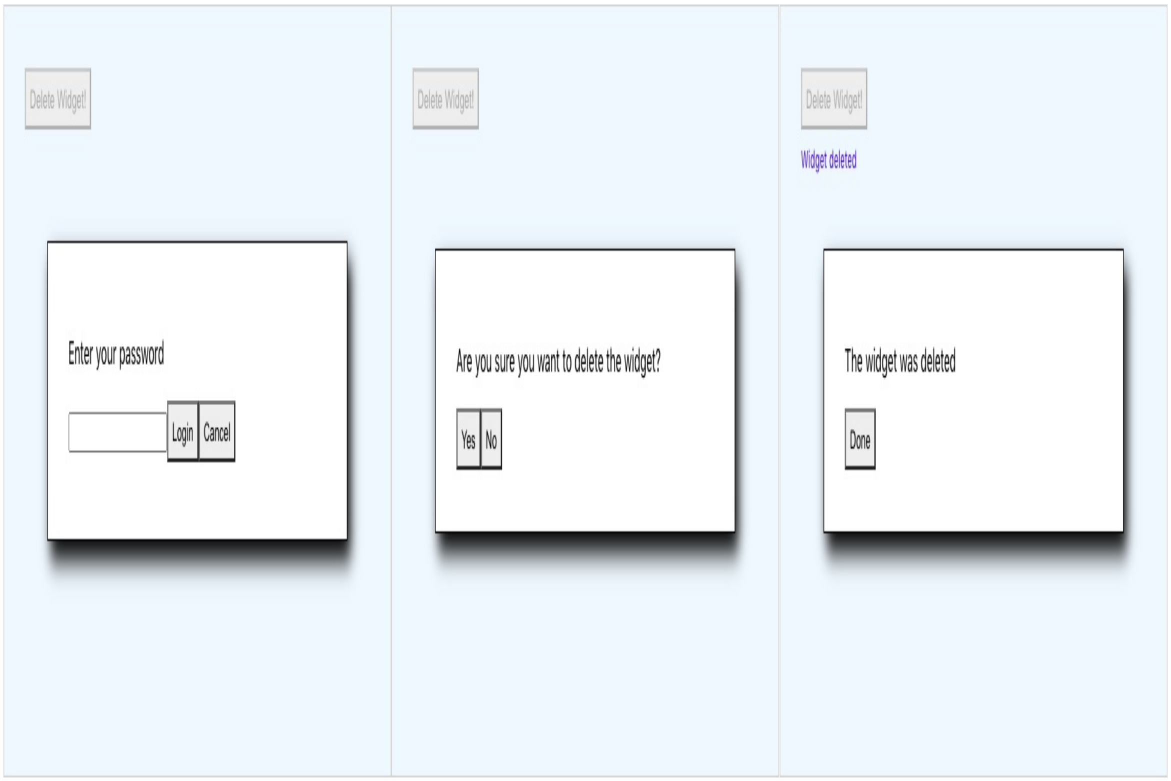


Figure 4-8. The final result

But *figure 4-9* shows it also handles all of the edge cases, such as invalid passwords and cancellation.

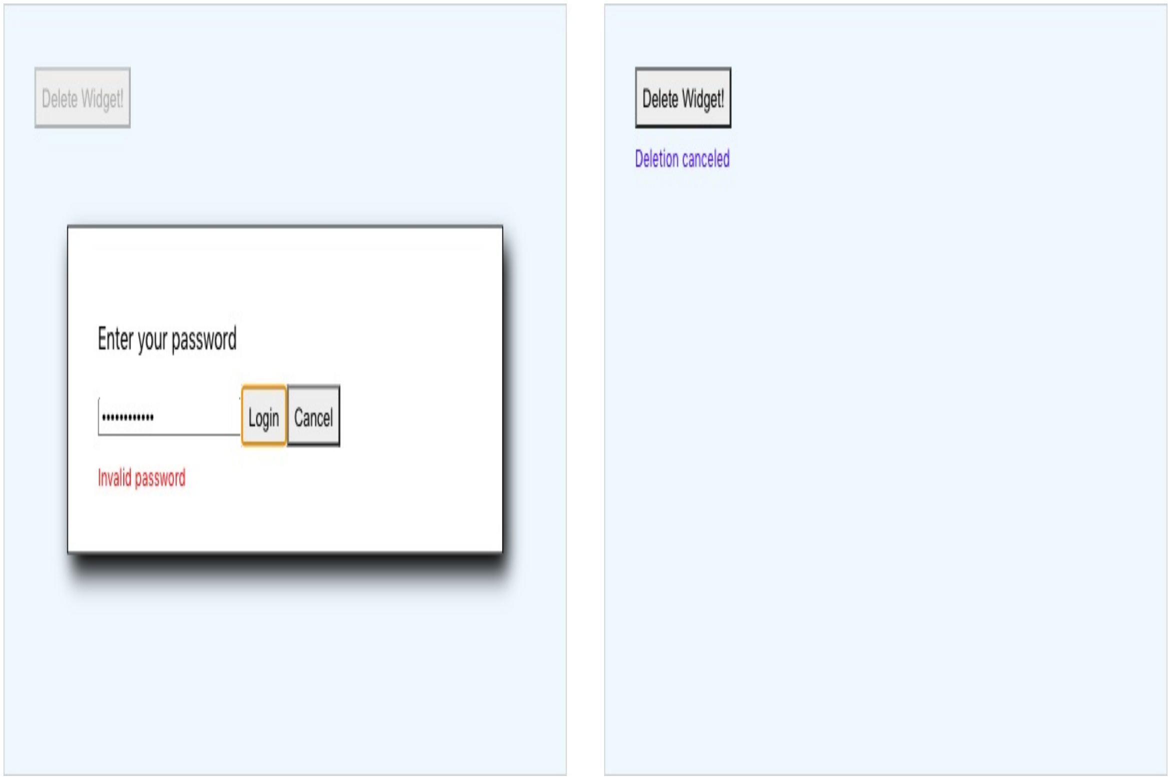


Figure 4-9. The edge cases are all handled by the reducer

Discussion

There are times when reducers can make your code convoluted; if you have very few pieces of state with very few interactions between them, you probably don't need a reducer. But if you find yourself drawing a flowchart or a state diagram to describe a sequence of user interactions, that's a sign that might need a reducer.

You can download the source for this recipe from the Github site.

4.4 Keyboard Interaction

Problem

Power users like to use keyboards for common operations. React components can respond to keyboard events, but only when (or their children) have focus.

What do you do if you want your component to respond to events at the document level?

Solution

We're going to create a key-listener hook to listen for `keydown` events at the `document` level. Still, it could be easily modified to listen for any other JavaScript event in the DOM. This is the hook:

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

export default (callback) => {
    useEffect(() => {
        const listener = (e) => {
            e = e || window.event;
            const tagName = e.target.localName || e.target.tagName;
            // Only accept key-events that originated at the body
level
            // to avoid key-strokes in e.g. text-fields being included
            if (tagName.toUpperCase() === 'BODY') {
                callback(e);
            }
        };
        document.addEventListener('keydown', listener, true);
        return () => {
            document.removeEventListener('keydown', listener, true);
        }
    }, [callback]);
};
```

The hook accepts a callback function and registers it for `keydown` events on the `document` object. At the end of the `useEffect`, it returns a function that will un-register the callback. If the callback function we pass in changes, we will first un-register the old function before registering the new one.

How do we use the hook? This is an example. See if you notice the little coding-wrinkle we have to deal with:

```
import {useCallback, useState} from 'react';
import './App.css';
import useKeyListener from "./useKeyListener";

const RIGHT_ARROW = 39;
const LEFT_ARROW = 37;
const ESCAPE = 27;

function App() {
    const [angle, setAngle] = useState(0);
    const [lastKey, setLastKey] = useState('');

    let onKeyDown = useCallback(evt => {
        if (evt.keyCode === LEFT_ARROW) {
            setAngle(c => Math.max(-360, c - 10));
            setLastKey('Left');
        } else if (evt.keyCode === RIGHT_ARROW) {
            setAngle(c => Math.min(360, c + 10));
            setLastKey('Right');
        } else if (evt.keyCode === ESCAPE) {
            setAngle(0);
            setLastKey('Escape');
        }
    }, [setAngle]);
    useKeyListener(onKeyDown);

    return (
        <div className="App">
            <p>Angle: {angle} Last key: {lastKey}</p>
            <svg width="400px" height="400px" title='arrow'
fill='none'
                 strokeWidth="10" stroke='black' style={{
                transform: `rotate(${angle}deg)`,
            }}>
                <polyline points="100,200 200,0 300,200"/>
                <polyline points="200,0 200,400"/>
            </svg>
        </div>
    );
}

export default App;
```

This code listens for the user pressing the left/right cursor keys. Our `onKeyDown` function says what should when those key-presses occur, but notice that we've wrapped it in a `useCallback`. If we **didn't** do that, the browser would re-create the `onKeyDown` function each time it rendered the `App` component. The new function would do the same as the old `onKeyDown` function, but it would live in a different place in memory, and the `useKeyListener` would keep un-registering and re-registering it.

WARNING

If you forget to wrap your callback function in a useCallback, it may result in a blizzard of render calls, which might slow your application down.

By using `useCallback`, we can ensure that we only create the function if `setAngle` changes.

If you run the application, you will see an arrow on the screen. If you press the left/right cursor keys (*figure 4-10*), you can rotate it. If you press the *escape* key, you can reset it to vertical.

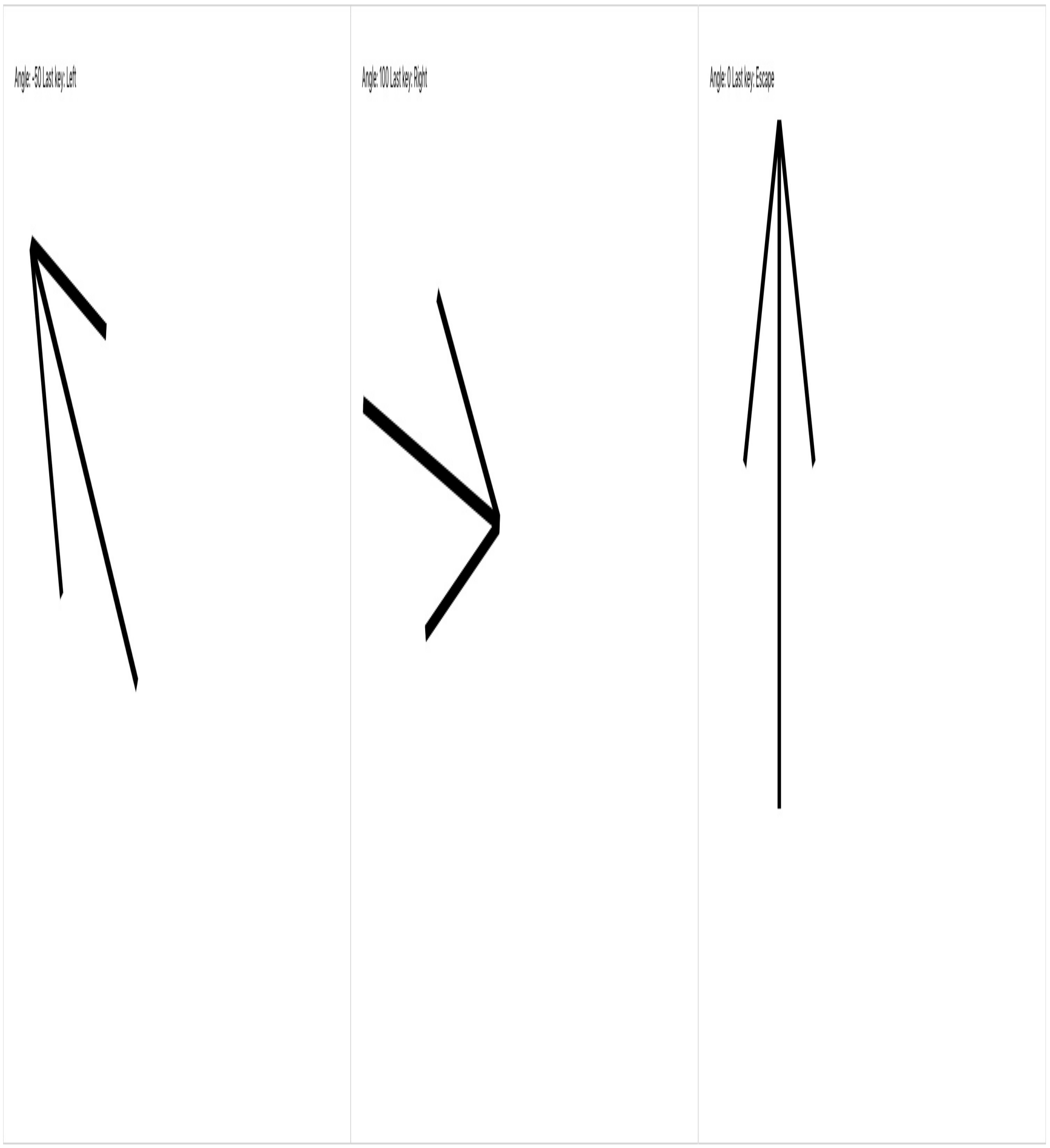

Angle: -50 Last key: Left
Angle: 100 Last key: Right
Angle: 0 Last key: Escape

Figure 4-10. Pressing the left/right/escape keys causes the arrow to rotate

Discussion

We are careful in the `useKeyListener` function to only listen to events that originated at the `body` level. If the user presses the arrow keys in a text field, the browser won't send those events to your code.

You can download the source for this recipe from the Github site.

4.5 Use Markdown for Rich Content

Problem

If your application allows users to provide large blocks of text content, it would be useful if that content could also include formatted text, links, and so forth. However, allowing users to pass in such horrors as raw HTML can lead to security flaws and untold misery for developers.

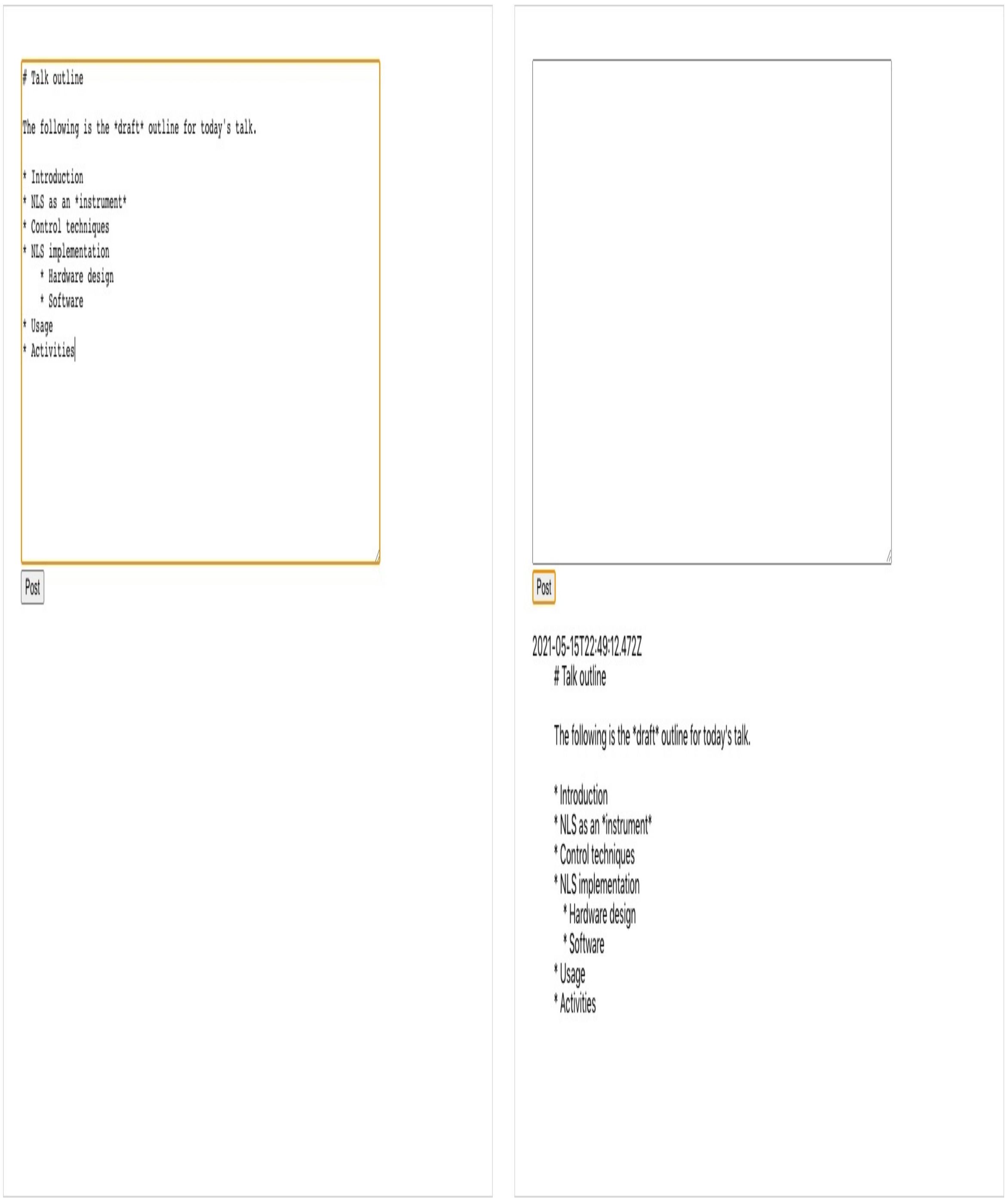
How do you allow users to post rich content without undermining the security of your application?

Solution

Markdown is a wonderful way of allowing users to post rich content into your application safely. To see how to use Markdown in your application, let us consider this simple application which allows a user to post a timestamped series of messages into a list:

```
import {useState} from "react";
import './Forum.css';

export default () => {
    const [text, setText] = useState('');
    const [messages, setMessages] = useState([])

    return <section className='Forum'>
        <textarea cols={80} rows={20} value={text}
            onChange={evt => setText(evt.target.value)}/>
        <button onClick={() => {
            setMessages(msgs => [{
                body: text,
                timestamp: new Date().toISOString()}, ...msgs]);
            setText('');
        }}>Post</button>
        {messages.map(msg => {
            return <dl>
                <dt>{msg.timestamp}</dt>
                <dd>{msg.body}</dd>
            </dl>;
        })}
    </section>;
}
```

When you run the application (*figure 4-11*), you see a large text-area. When you post a plain-text message, the app preserves white-space and line-breaks.



```
# Talk outline

The following is the *draft* outline for today's talk.

* Introduction
* NLS as an *instrument*
* Control techniques
* NLS implementation
  * Hardware design
  * Software
* Usage
* Activities
```

```
2021-05-15T22:49:12.472Z
  # Talk outline

  The following is the *draft* outline for today's talk.

  * Introduction
  * NLS as an *instrument*
  * Control techniques
  * NLS implementation
    * Hardware design
    * Software
  * Usage
  * Activities
```

Figure 4-11. A user enters text into a textarea, and it gets posted as a plain-text message

Any time your application contains a text-area, it's worth considering allowing the user to enter Markdown content.

There are many, many Markdown libraries available, but most of them are wrappers for `react-markdown`[5] or a syntax highlighter like PrismJS or codemirror.

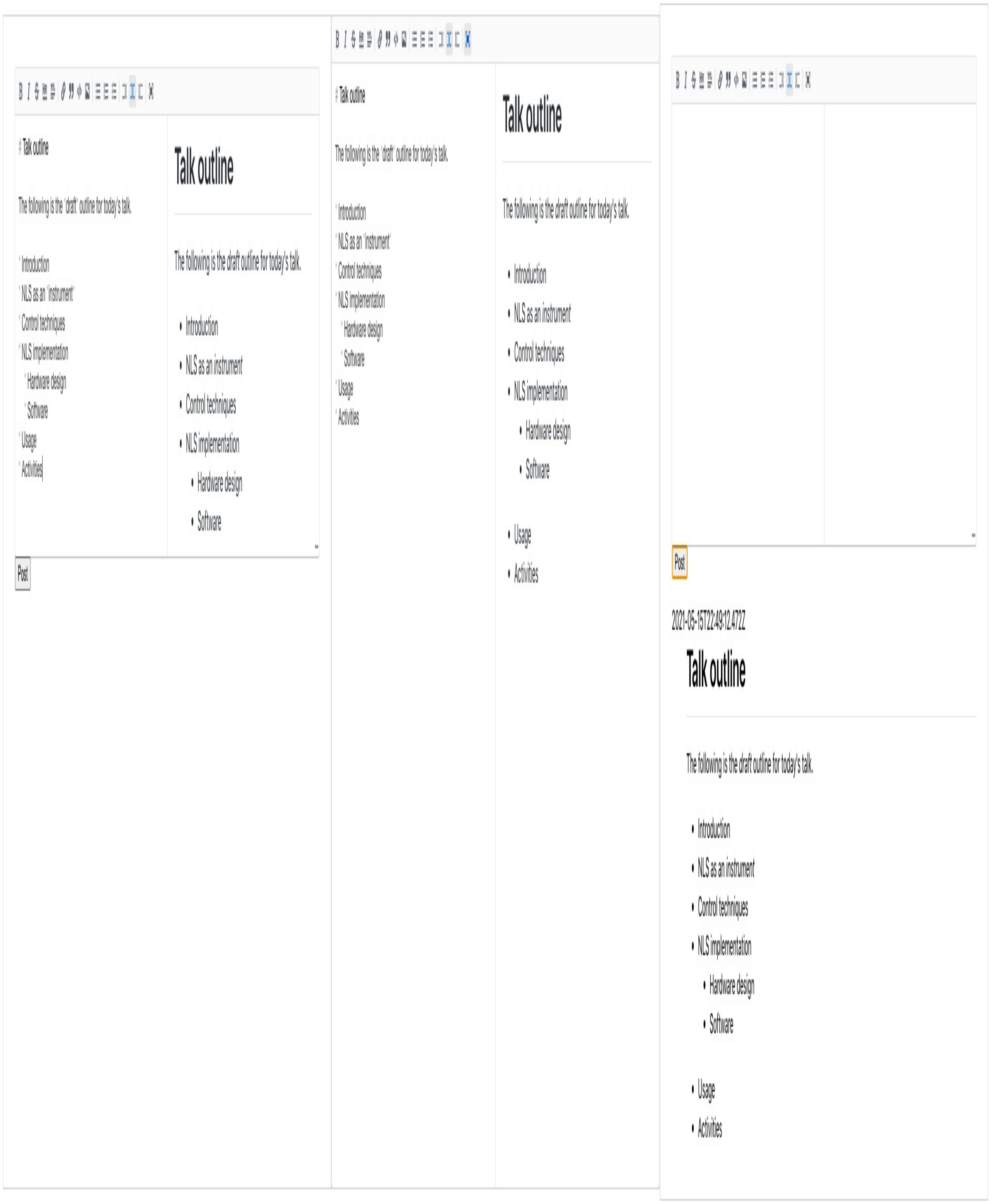
We'll look at a library that allows you to both display Markdown and edit, called `react-md-editor`. Begin by installing the library:

```
npm install @uiw/react-md-editor
```

We'll now convert our plain text-area to a Markdown editor and convert the posted messages from Markdown to HTML:

```
import {useState} from "react";
import MDEditor from '@uiw/react-md-editor';

export default () => {
    const [text, setText] = useState('');
    const [messages, setMessages] = useState([])

    return <section className='Forum'>
        <MDEditor height={300} value={text} onChange={setText}/>
        <button onClick={() => {
            setMessages(msgs => [{
                body: text,
                timestamp: new Date().toISOString()}, ...msgs]);
            setText('');
        }}>Post
        </button>
        {messages.map(msg => {
            return <dl>
                <dt>{msg.timestamp}</dt>
                <dd><MDEditor.Markdown source={msg.body}/></dd>
            </dl>;
        })}
    </section>;
}
```

Converting plain-text to Markdown is a small change with a large return. As you can see in *figure 4-12*, the user can apply rich formatting to message and choose to edit it fullscreen before posting it.



```
# Talk outline

The following is the 'draft' outline for today's talk.

* Introduction
* NLS as an 'instrument'
* Control techniques
* NLS implementation
  * Hardware design
  * Software
* Usage
* Activities
```

Talk outline

The following is the draft outline for today's talk.

- Introduction
- NLS as an instrument
- Control techniques
- NLS implementation
  - Hardware design
  - Software
- Usage
- Activities

Post

2021-05-15T22:49:12.472Z

Talk outline

The following is the draft outline for today's talk.

- Introduction
- NLS as an instrument
- Control techniques
- NLS implementation
  - Hardware design
  - Software
- Usage
- Activities

Figure 4-12. The markdown editor shows a preview as you type and also allows you to work fullscreen

Discussion

Adding Markdown to an application is quick and improves the user's experience with minimal effort. For more details on Markdown, see John Gruber’s original guide.

You can download the source for this recipe from the Github site.

4.6 Animations with CSS Classes

Problem

You want to add a small amount of simple animation to your application, but you don’t want to increase your application size by installing a third-party library.

Solution

Most of the animation you are ever likely to need in a React application will probably not require a third party animation library. That’s because CSS animation now gives browsers the native ability to animate CSS properties with minimal effort. It takes very little code, and the animation is smooth because the graphics hardware will generate it. GPU animation uses less power, making it more appropriate for mobile devices.

TIP

If you are looking add animation to your React application, begin with CSS animation before looking elsewhere

How does CSS animation work? It uses a CSS property called `transition`. Let’s say we want to create an expandable information-panel. When the user clicks on the button, the panel smoothly opens. When they click it again, it closes smoothly, as show in *figure 4-13*.

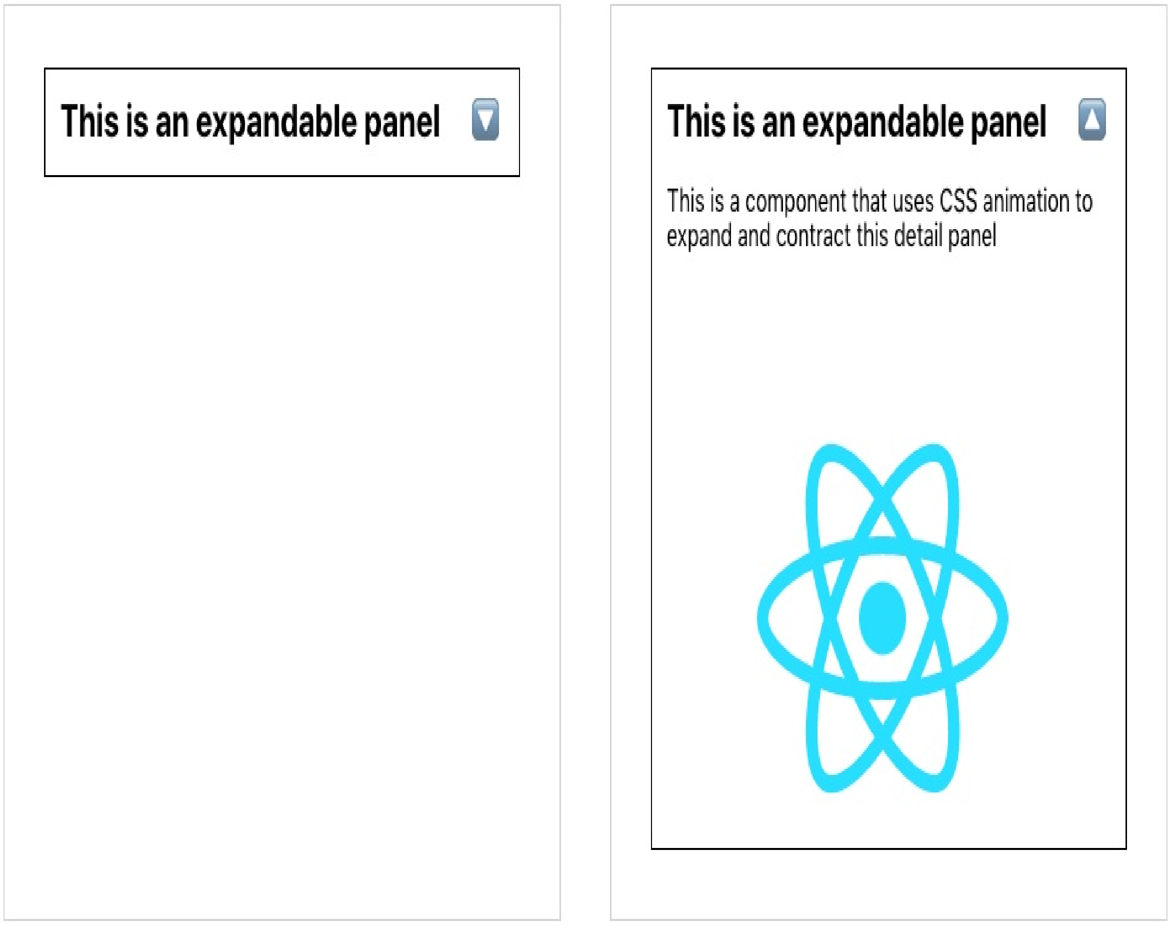

This is an expandable panel
This is an expandable panel
This is a component that uses CSS animation to expand and contract this detail panel

Figure 4-13. Simple CSS animation will smoothly expand and contract the panel

We can create this effect using the CSS `transition` property:

```
.InfoPanel-details {
    height: 350px;
    transition: height 0.5s;
}
```

This CSS specifies a `height`, as well as a `transition` property. This combination translates to *"Whatever your current height, animate to my preferred height during the next half-second."*

The animation will occur whenever the `height` of the element changes, such as when an additional CSS rule becomes valid. For example, if we have an extra CSS class-name with a different height:

```
.InfoPanel-details {
    height: 350px;
    transition: height 0.5s;
}
.InfoPanel-details.InfoPanel-details-closed {
    height: 0;
}
```

If an InfoPanel-details element suddenly acquires an additional .InfoPanel-details-closed`footnote:[This class name structure is an example of Block-Element-Modifier (BEM) naming. The _block_ is the component (`InfoPanel), the *element* is a thing inside the block (Details), and the *modifier* says something about the element's current state (closed).] class, the height will change from 350px to 0, and transition property will smoothly shrink the element. Conversely, if the component *loses* the .InfoPanel-details-closed class, the element will expand again.

That means that we can defer the hard work to CSS, and all we need to do in our React code is add or remove the class to an element:

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

import './InfoPanel.css';

export default ({title, children}) => {
    const [open, setOpen] = useState(false);

    return <section className='InfoPanel'>
        <h1>
            {title}
            <button onClick={() => setOpen(v => !v)}>
                {open ? ''  : '' }
            </button>
        </h1>
        <div className={`InfoPanel-details ${(open ? ''
                : 'InfoPanel-details-closed')}`}>
            {children}
        </div>
    </section>
}
```

Discussion

We have frequently seen many projects bundle in third party component libraries to use some small widget that expands or contracts its contents. As you can see above, such animation is trivial to include.

You can download the source for this recipe from the Github site.

4.7 Animations with react-animation

Problem

CSS animations are very low-tech and will be appropriate for most animations that you are likely to need.

However, they require you to understand a lot about the various CSS properties and the effects of animating them. If you want to illustrate an item being deleted by it rapidly expanding and becoming transparent, how do you do that?

Libraries such as animate.css contain a whole host of pre-canned CSS animations, but they often require more advanced CSS animation concepts like keyframes and are not particularly tuned for React. How can we add CSS library animations to a React application?

Solution

The `react-animations` library is a React-wrapper for the `animate.css` library. It will efficiently add animated-styling to your components without generating unnecessary renders or significantly increasing the size of the generated DOM.

It’s able to work so efficiently because `react-animations` works with a CSS-in-JS library. CSS-in-JS is a technique for coding your style information directly in your JavaScript code. React will let you add your style attributes as React components, but CSS-in-JS does this more efficiently, dynamically creating shared style elements in the `head` of the page.

There are several CSS-in-JS libraries to choose from, but in this recipe, we’re going to use one called Radium

Let’s begin by installing Radium and `react-animations`:

```
npm install radium
npm install react-animations
```

Our example application (*figure 4-14*) is going to run an animation each time we add an image item to the collection.

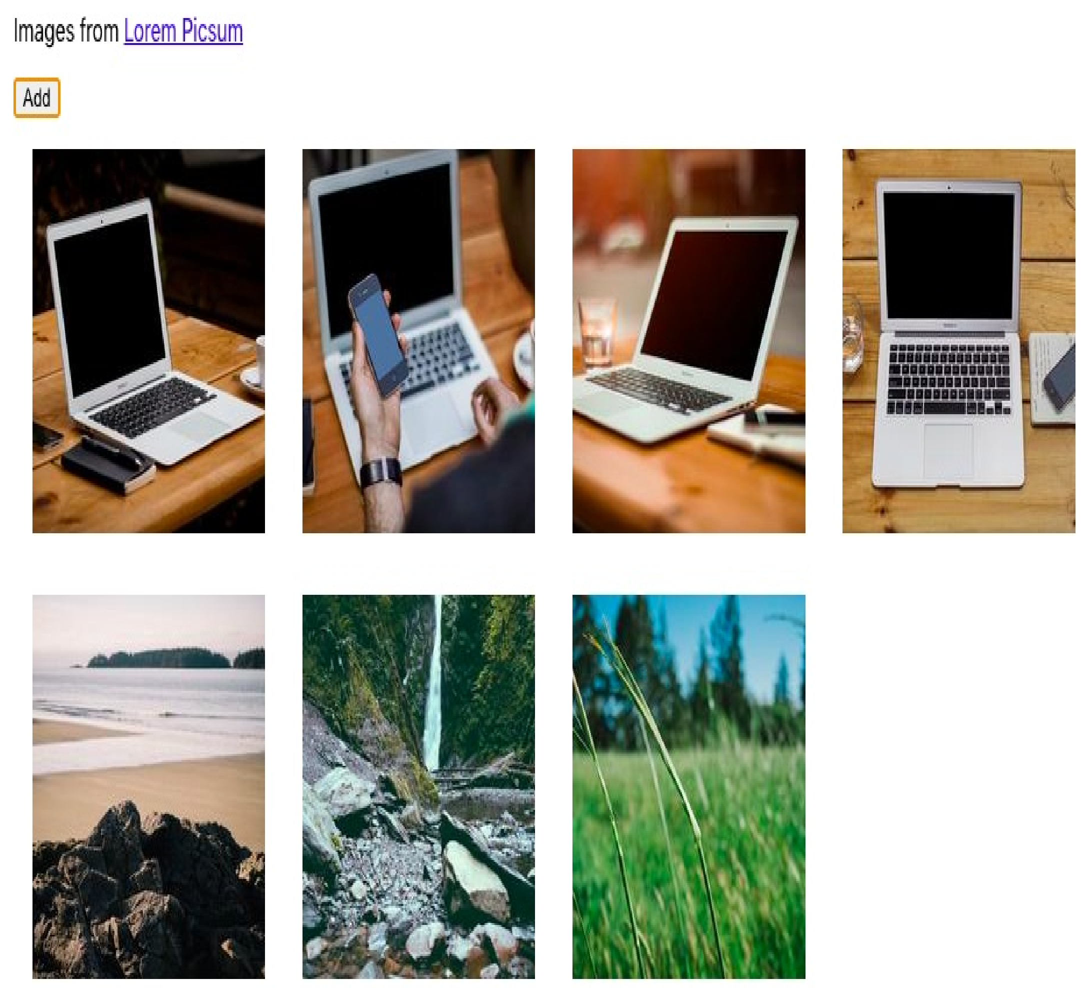


Figure 4-14. Clicking the add button will load a new image from picsum.images

Likewise, when a user clicks on an image, it shows a fade-out animation before removing the images from the list[6], as shown in *figure 4-15*.

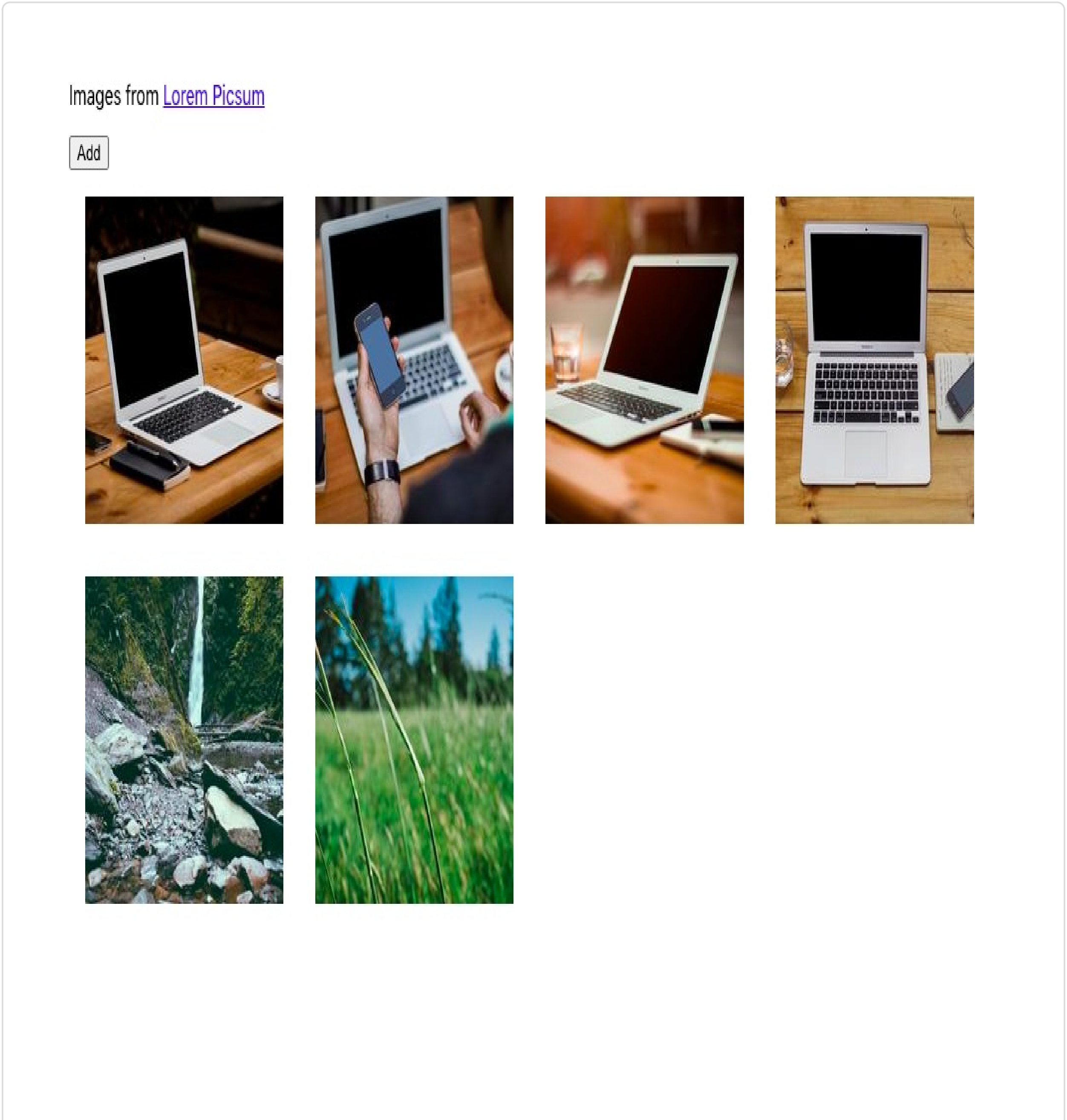


Figure 4-15. If we click the 5th image, it will fade out from the list and disappear

We'll begin by importing some animations and helper code from Radium:

```
import {pulse, zoomOut, shake, merge} from 'react-animations';
import Radium, {StyleRoot} from 'radium';

const styles = {
    created: {
        animation: 'x 0.5s',
        animationName: Radium.keyframes(pulse, 'pulse')
    },
    deleted: {
        animation: 'x 0.5s',
        animationName: Radium.keyframes(merge(zoomOut, shake),
'zoomOut')
    },
};
```

From `react-animations` we are getting the details for `pulse`, `zoomOut`, and `shake` animations. We are going to use the `pulse` animation when we add an image. We'll use a *combined* animation of `zoomOut` and `shake` when we remove an image. We can combine animations using `react-animations` `merge` function.

The `styles` generate all of the CSS-styles needed to run each of these animations in half a second. The call to `Radium.keyframes` handles all of the animation details for us.

We must know when an animation has completely ended. If we delete an image before the deletion-animation completes, there would be no image to animate.

We can keep track of CSS animations by passing an `onAnimationEnd` callback to any element we are going to animate. For each item in our image collection, we are going to track three things:

- The URL of the image it represents
- A boolean value which will be true while the "created" animation is running
- A boolean value which will be true while the "deleted" animation is running

This is the example code to animate images into an out of the collection:

```
import {useState} from 'react';
import {pulse, zoomOut, shake, merge} from 'react-animations';
import Radium, {StyleRoot} from 'radium';

import './App.css';

const styles = {
    created: {
        animation: 'x 0.5s',
        animationName: Radium.keyframes(pulse, 'pulse')
    },
    deleted: {
        animation: 'x 0.5s',
        animationName: Radium.keyframes(merge(zoomOut, shake),
'zoomOut')
    },
};

function getStyleForItem(item) {
    return item.deleting ? styles.deleted : item.creating ?
styles.created : null;
}

function App() {
    const [data, setData] = useState([]);

    let deleteItem = (i) => setData(d => {
        const result = [...d];
        result[i].deleting = true;
        return result;
    });
    let createItem = () => {
        setData(d => [...d, {
            url: `https://picsum.photos/id/${d.length * 3}/200`,
            creating: true
        }]);
    };
    let completeAnimation = (d, i) => {
        if (d.deleting) {
            setData(d => {
                const result = [...d];
                result.splice(i, 1);
                return result;
            });
        } else if (d.creating) {
            setData(d => {
                const result = [...d];
                result[i].creating = false;
                return result;
            });
        }
    };
    return (
        <div className="App">
            <StyleRoot>
                <p>
                    Images from&nbsp;
                    <a href='https://picsum.photos/'>Lorem Picsum</a>
                </p>
                <button onClick={createItem}>Add</button>
                {
                    data.map((d, i) => <div
                        style={getStyleForItem(d)}
                        onAnimationEnd={() => completeAnimation(d,
i)}>
                        <img
                            id={`image${i}`}
                            src={d.url}
                            width={200}
                            height={200}
                            alt='Random'
                            title='Click to delete'
                            onClick={() => deleteItem(i)}
                        />
                    </div>)
                }
            </StyleRoot>
        </div>
    );
}

export default App;
```

Discussion

When choosing which animation to use, it's important to first ask: what will this animation *mean*?

All animation should have meaning. It can show something existential[7]. It might indicate a change of state[8]. It might zoom in to show detail, or zoom out to reveal a broader context. Or it might illustrate a limit or boundary[9], or allow a user to express a preference [10]

Animation should also be short. Most animations should probably be over in less than a second so that the user can experience the meaning of the animation without being consciously aware of its appearance.

An animation should never be merely *attractive*.

You can download the source for this recipe from the Github site.

4.8 Create Animated Infographics with TweenOne

Problem

CSS animations are smooth and extremely efficient. Browsers might defer CSS animations to the graphics hardware at the compositing stage, which means that not only are the animations running at machine-code speeds, the machine-code itself is not running on the CPU.

However, the downside to CSS animations being handed-off in this way is that your application code won't know what's happening *during* an animation. You can track when an animation has started, ended, or has been repeated[11] but everything that happens in between is a mystery.

If you are animating an infographic, you may want to animate the numbers on a bar chart as the bars change height. Or, if you are writing an application to track cyclists, you might want to show the current altitude as the bicycle animates its way up and down the terrain.

But how do you create animations that you can **listen** to while they are happening?

Solution

The TweenOne library creates animations with JavaScript, which means you can track them as they happen, frame-by-frame.

Let's begin by installing the TweenOne library.

```
npm install rc-tween-one
```

TweenOne works with CSS, but it doesn't use CSS animations. Instead, it generates CSS transforms, which it updates many times each second.

You need to wrap the thing you want to animate in a `<TweenOne/>` element. For example, let's say we want to animate a `rect` inside an SVG:

```
<TweenOne component='g' animation={...details here}>
    <rect width="2" height="6" x="3" y="-3" fill="white">
</TweenOne>
```

`TweenOne` takes an element name and an object that will describe the animation to perform. We'll come to what that animation object looks like shortly.

TweenOne will use the element name (`g` in this case) to generate a wrapper around the animated thing. This wrapper will have a style attribute that will include a set of CSS transforms to move and rotate the contents somewhere.

So in our example, at some point in the animation, the DOM might look like this:

```
<g style="transform: translate(881.555px, 489.614px)
rotate(136.174deg);">
  <rect width="2" height="6" x="3" y="-3" fill="white">
</g>
```

This means that although you can create very similar effects to CSS animations, the TweenOne library works differently. Instead of handing the animation to the hardware, the TweenOne library uses JavaScript to create each frame, which has two consequences. First, we’ll use more CPU power, and second, we can track the animation while it’s happening.

If we pass `TweenOne` an `onUpdate` callback, we will be sent information about the animation on every single frame:

```
<TweenOne component='g' animation={...details here} onUpdate={info=>
{...}>
    <rect width="2" height="6" x="3" y="-3" fill="white">
</TweenOne>
```

The `information` object passed to `onUpdate` has a `ratio` value between 0 and 1, representing the proportion of the way the TweenOne element is through an animation. We can use the `ratio` to animate text that is associated with the graphics.

For example, if we build an animated dashboard that shows vehicles on a race track, we can use `onUpdate` to show each car’s speed and distance as it animates.

We’ll create the visuals for this example in SVG. First, let’s create a string containing an SVG path, which represents the track:

```
export default 'm 723.72379,404.71306 ...  -8.30851,-3.00521 z';
```

This is a greatly truncated version of the actual path that we'll use. We can import the path string from `track.js` like this:

```
import path from './track';
```

To display the track inside a React component, we can render an `svg` element:

```
<svg height="600" width="1000" viewBox="0 0 1000 600"
     style={{backgroundColor: 'black'}}>
  <path stroke='#444' strokeWidth={10}
        fill='none' d={path}/>
</svg>
```

We can add a couple of rectangles for the vehicle - a red one for the body and a white one for the windshield:

```
<svg height="600" width="1000" viewBox="0 0 1000 600"
     style={{backgroundColor: 'black'}}>
  <path stroke='#444' strokeWidth={10}
        fill='none' d={path}/>
  <rect width={24} height={16} x={-12} y={-8} fill='red'/>
  <rect width={2} height={6} x={3} y={-3} fill='white'/>
</svg>
```

Figure 4-16 shows the track with the vehicle at the top-left hand corner.

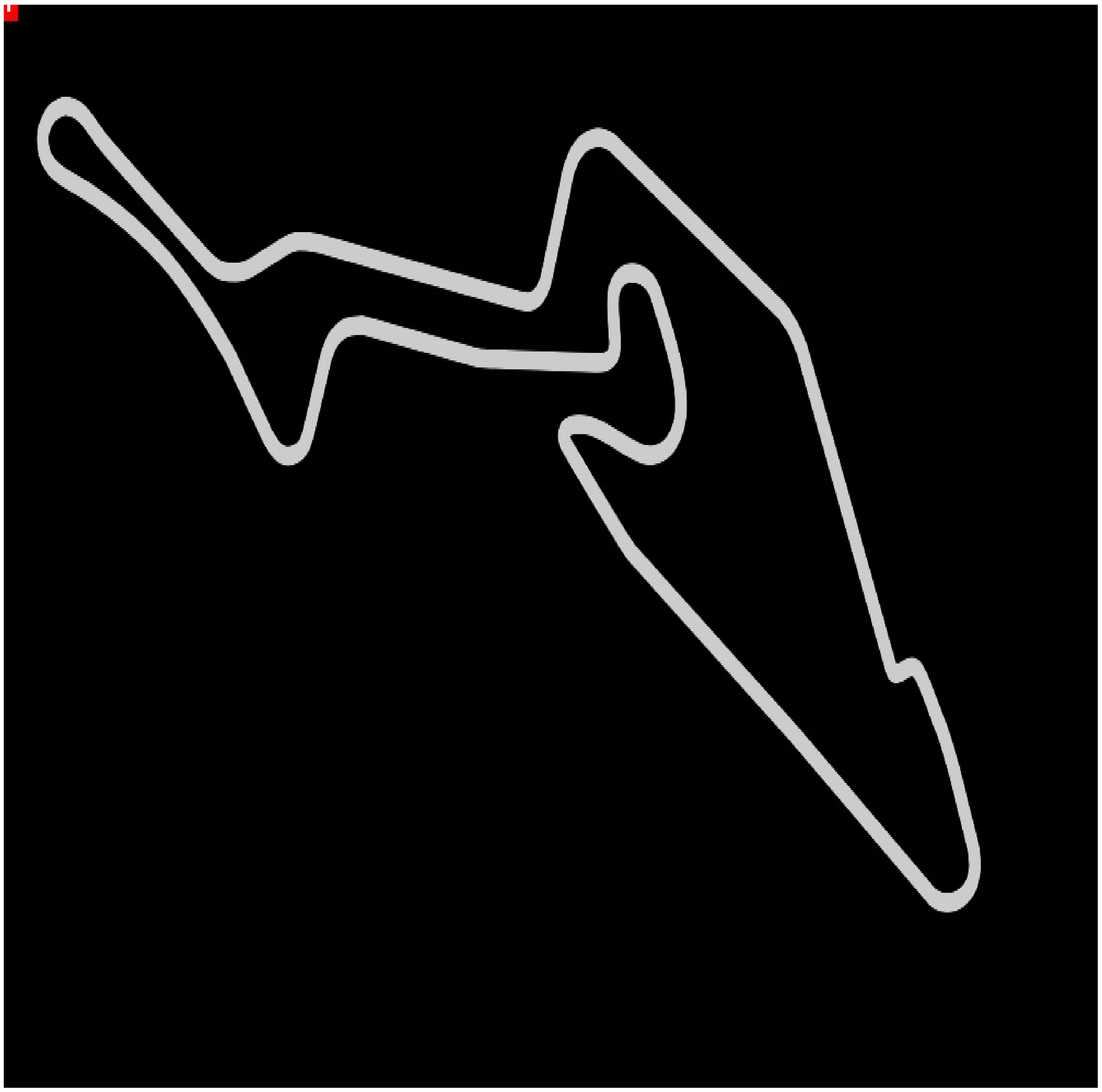


Figure 4-16. The static image with the vehicle at the top-left

But how do we animate the vehicle around the track? TweenOne makes this very easy because it contains a plug-in to generate animations that follow SVG path strings.

```
import PathPlugin from 'rc-tween-one/lib/plugin/PathPlugin';

TweenOne.plugins.push(PathPlugin);
```

We've configured TweenOne for use with SVG path animations. That means we can look at how to describe an animation for TweenOne. We do it with a simple JavaScript object:

```
import path from './track';

const followAnimation = {
    path: {x: path, y: path, rotate: path},
    repeat: -1,
};
```

We tell TweenOne two things with this object: firstly, we're telling it to generate translates and rotations that follow the `path` string that we've imported from `track.js`. Secondly, we're saying that we want the animation to loop infinitely by setting the `repeat` count to -1.

We can use this as the basis of animation for our car:

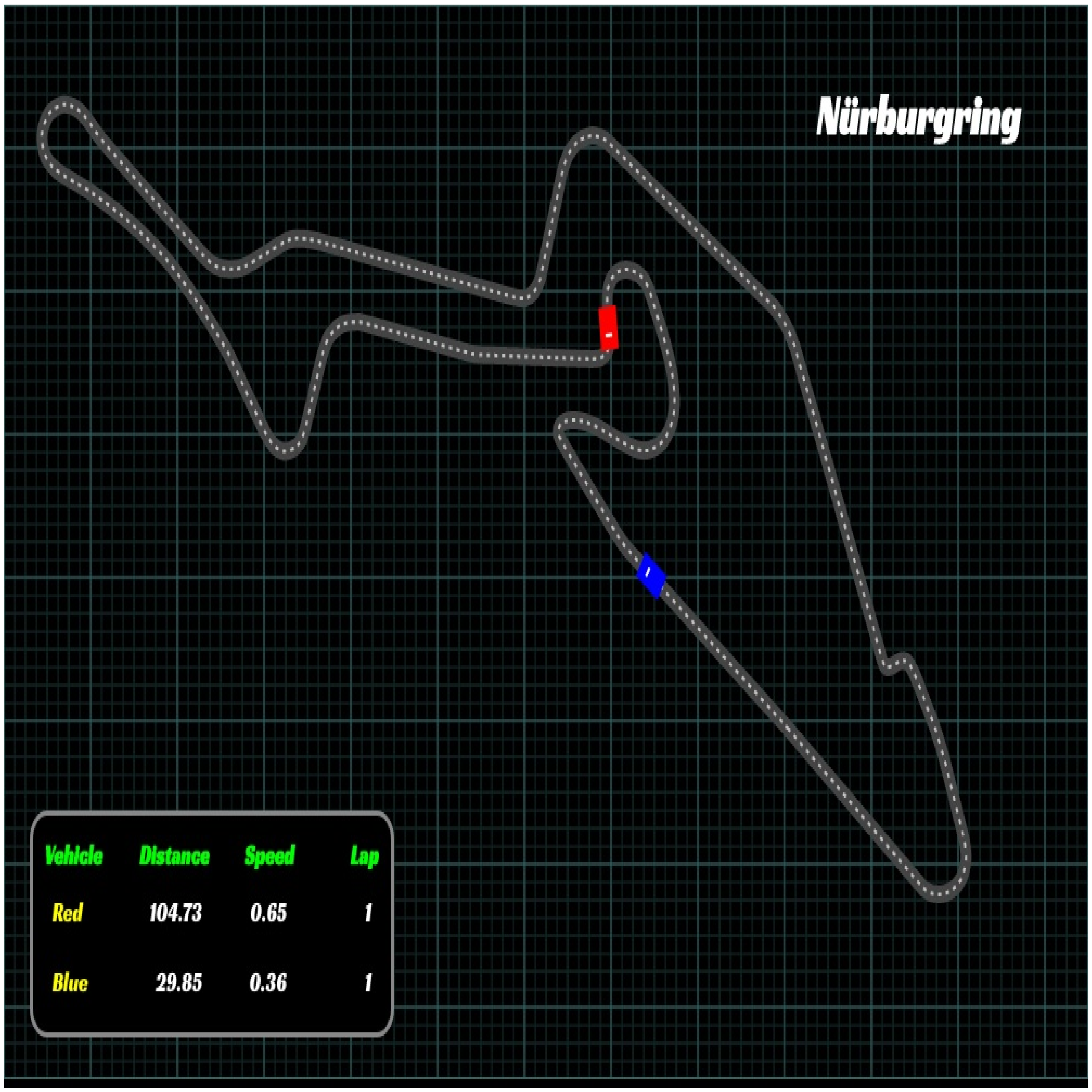
```
<svg height="600" width="1000" viewBox="0 0 1000 600"
     style={{backgroundColor: 'black'}}>
  <path stroke='#444' strokeWidth={10}
        fill='none' d={path}/>
  <TweenOne component='g' animation={{...followAnimation, duration:
16000}}>
    <rect width={24} height={16} x={-12} y={-8} fill='red'/>
    <rect width={2} height={6} x={3} y={-3} fill='white'/>
  </TweenOne>
</svg>
```

Notice that we're using the spread operator to provide an additional animation parameter: `duration`. A value of 16000 means we want the animation to take 16 seconds.

We can add a second vehicle and use the `onUpdate` callback method to create a very rudimentary set of faked telemetry statistics for each one as they move around the track. This is the completed code:

```
import {useState} from 'react';
import TweenOne from 'rc-tween-one';
import Details from "./Details";
import path from './track';
import PathPlugin from 'rc-tween-one/lib/plugin/PathPlugin';
import grid from './grid.svg';
import './App.css';

TweenOne.plugins.push(PathPlugin);

const followAnimation = {
    path: {x: path, y: path, rotate: path},
    repeat: -1,
};

function App() {
    const [redTelemetry, setRedTelemetry] = useState({
        dist: 0, speed: 0, lap: 0});
    const [blueTelemetry, setBlueTelemetry] = useState({
        dist: 0, speed: 0, lap: 0});

    const trackVehicle = (info, telemetry) => ({
        dist: info.ratio,
        speed: info.ratio - telemetry.dist,
        lap: info.ratio < telemetry.dist ?
            telemetry.lap + 1 : telemetry.lap
    });

    return <div className="App">
        <h1>Nürburgring</h1>
        <Details redTelemetry={redTelemetry}
                 blueTelemetry={blueTelemetry}/>
        <svg height="600" width="1000" viewBox="0 0 1000 600"
             style={{backgroundColor: 'black'}}>
            <image href={grid}
                   width={1000} height={600}/>
            <path stroke='#444' strokeWidth={10}
                  fill='none' d={path}/>
            <path stroke='#c0c0c0' strokeWidth={2}
                  strokeDasharray='3 4' fill='none' d={path}/>

            <TweenOne component='g' animation={{
                ...followAnimation,
                duration: 16000,
                onUpdate: (info) => setRedTelemetry(
                    (telemetry) => trackVehicle(info, telemetry)),
            }}>
                <rect width={24} height={16} x={-12} y={-8}
fill='red'/>
                <rect width={2} height={6} x={3} y={-3} fill='white'/>
            </TweenOne>

            <TweenOne component='g' animation={{
                ...followAnimation,
                delay: 3000,
                duration: 15500,
                onUpdate: (info) => setBlueTelemetry(
                    (telemetry) => trackVehicle(info, telemetry)),
            }}>
                <rect width={24} height={16} x={-12} y={-8}
fill='blue'/>
                <rect width={2} height={6} x={3} y={-3} fill='white'/>
            </TweenOne>
        </svg>
    </div>;
}

export default App;
```

Figure 4-17 shows the animation. The vehicles follow the path of the race track, rotating to face the direction of travel.



Vehicle	Distance	Speed	Lap
Red	104.73	0.65	1
Blue	29.85	0.36	1

Figure 4-17. Our final animation with telemetry generated from the current animation state

Discussion

CSS animations are what you should use for most UI animation. However, in the case of infographics, you often need to synchronize the text and the graphics. TweenOne makes that possible.

You can download the source for this recipe from the Github site.

---

1 *https://sentry.io/*

[2] You can download all source for this recipe on the Github repository

[3] See the 5th recipe in this chapter for details on how to use Markdown in your application.

[4] See the Github repository for the tests we used to drive out this code

[5] A Markdown rendering library which (safely) converts Markdown text to HTML

[6] Paper books are wonderful things, but to fully experience the animation effect, see the full code on Github

[7] Creation or deletion

[8] Becoming enabled or disabled

[9] A spring-back operation at the end of a long list

[10] Swipe left or swipe right

[11] onAnimationStart, onAnimationEnd, onAnimationIteration

Chapter 5. Connecting to services

React, unlike frameworks like Angular, does not include everything you might need for an application. In particular, it does not provide a standard way to get data from network services into your application. That freedom is excellent because it means that React applications can use whatever the latest technology. The downside is that developers just starting with React are left to struggle on their own.

In this chapter, we will look at a few ways to attach network services to your application. We will see some common themes through each of these recipes. We will try to keep the network code separate from the components which use it. That way, when the next web service technology sweeps past, we will be able to switch over to it, with the minimum of impact to the rest of our application.

5.1 Convert Network Calls to Hooks

Problem

One of the advantages of component-based development is that it breaks the code down into small management chunks, each of which performs a distinct, identifiable action. In some ways, the best kind of component is one that you can see on a large screen without scrolling. One of the great features of React is that it has, in many ways, gotten simpler over time. React hooks and the move away from class-based components have removed boilerplate and reduced the amount of code.

However, one way to inflate the size of a component is by filling it with networking code. If you aim to create simple code, you should try to strip out networking code from your components. The components will become smaller, and the network code will be more reusable.

But how should we split-out the networking code?

Solution

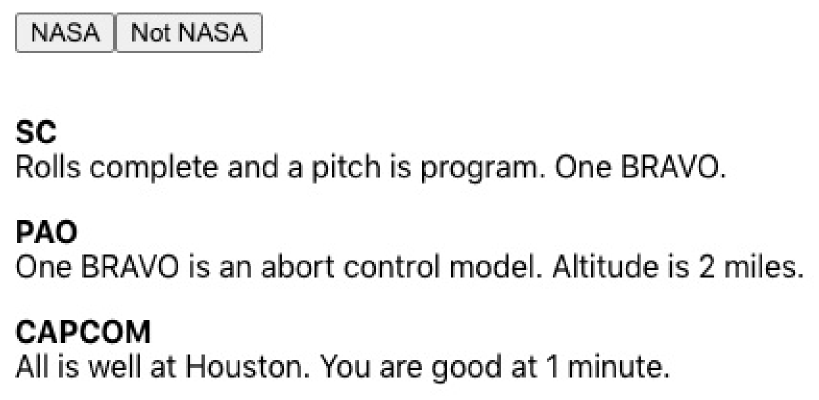
In this recipe, we will look at a way of moving your network requests into React hooks to track whether a network request is still underway or if there has been some error that prevented it from succeeding.

Before we look at the details, we need to think about what things are important to us when making an asynchronous network request. There are three things that we need to track:

- The data returned by the request,
- Whether the request is still loading the data from the server, and
- Any errors which might have occurred when running the request

You will see these three things appearing in each of the recipes in this chapter. It doesn't matter whether we are making the requests with `fetch` or `axios` commands, via Redux middleware or through a querying layer like *GraphQL*; our component will always care about data, loading-state, and errors.

As an example, let's build a simple message board that contains several forums. The messages on each forum contain an `author` and a `text` field. *Figure 5-1* shows a screenshot of the example application, which you can download from the Github site.



SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

Figure 5-1. The buttons select the NASA or Not NASA forums

The buttons at the top of the page select the “NASA” or “Not NASA” forums. A small node server provides the back-end for our example application, which has pre-populated some messages into the NASA forum. Once you have downloaded the source code, you can run the back-end server by running the `server.js` script in the application’s main directory:

```
$ node ./server.js
```

The back-end server runs at *http://localhost:5000*. We can start the React application itself in the usual way:

```
npm run start
```

The React application will run on port 3000.

TIP

When in development mode, we proxy all back-end requests through the React server. If you're using `create-react-app`, you can do this by adding a `proxy` property to the `package.json` and setting it to `"http://localhost:5000"` The React server will pass API calls to our `server.js` back-end. For example, *http://localhost:3000/messages/nasa* (which returns an array of messages for the NASA forum) will be proxied to *http://localhost:5000/messages/nasa*.

We'll make the network request to read the messages using a simple `fetch` command.

```
const response = await fetch(`/messages/${forum}`);
if (!response.ok) {
    const text = await response.text();
    throw new Error(
        `Unable to read messages for ${forum}: ${text}`
    );
}
const body = await response.json();
```

Here, the `forum` value will contain the string id of the forum. The `fetch` command is asynchronous and returns a promise so that we will `await` it. Then we can check if the call failed with any bad HTTP status, and if so, we will throw an error. We will extract the JSON object out of the response and store it in the `body` variable. If the response body is not a correctly formatted JSON object, we will also throw an error.

We need to keep track of three things in this call: the data, the loading-state, and any errors. We're going to bundle this whole thing up inside a custom hook, so let's have three states called `data`, `loading`, and `error`:

```
const useMessages = (forum) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    ....

    return {data, loading, error};
};
```

We'll pass in the forum name as a parameter to the `useMessages` hook, which will return an object containing the `data`, `loading`, and `error` states. We can use the spread operator to extract and rename the values in any component that uses the hook, like this:

```
const {data: messages, loading: messagesLoading, error: messagesError}
= useMessages('nasa');
```

Renaming the variables helps avoid naming conflicts if, for example, you wanted to read the messages from more than one forum.

Let’s get back to the `useMessages` hook. The network request depends upon the `forum` value that we pass in, so we need to make sure that we run the `fetch` request inside a `useEffect`:

```
useEffect(() => {
    setError(null);
    if (forum) {
        ....
    } else {
        setData([]);
        setLoading(false);
    }
}, [forum]);
```

We’re omitting for the moment the code that makes the actual request. The code inside the `useEffect` will run the first time hook is called. If the client-component is re-rendered and passes in the same value for `forum`, the `useEffect` will not run because the `[forum]` dependency will not have changed. It will only run again if the `forum` value changes.

Now let’s look at how we can drop in the `fetch` request to this hook:

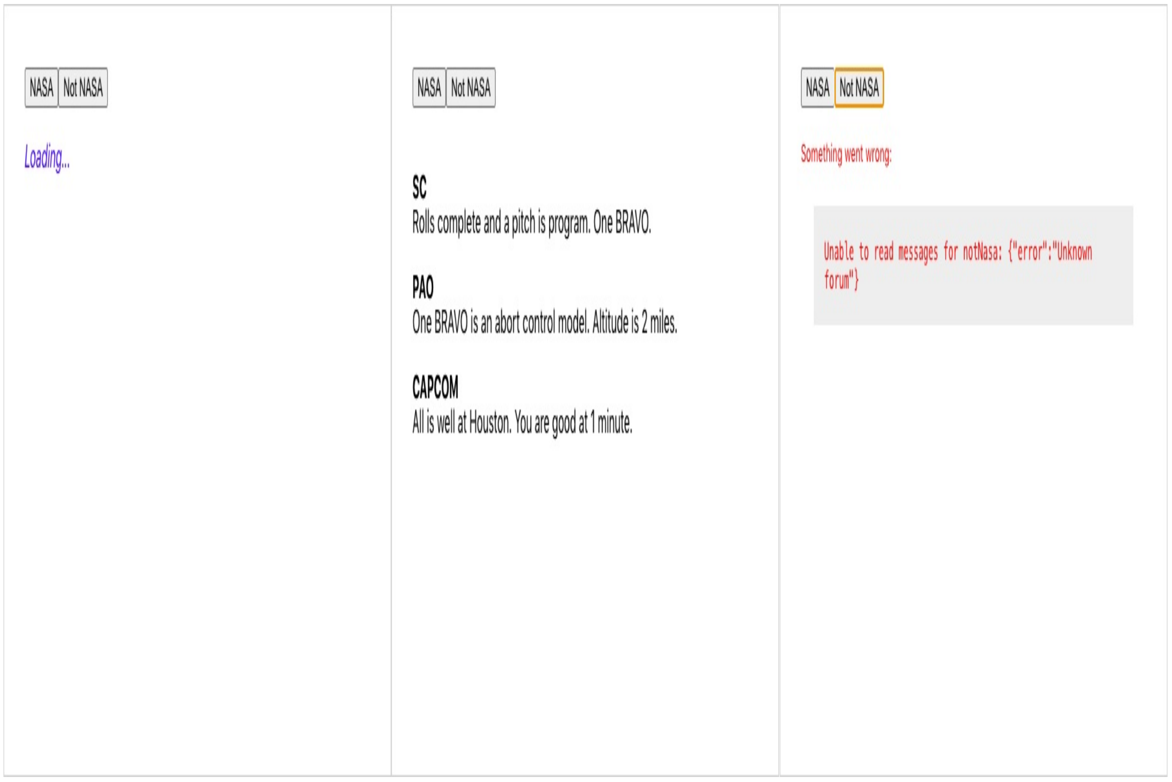
```
import {useEffect, useState} from "react";

const useMessages = (forum) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    useEffect(() => {
        setError(null);
        if (forum) {
            (async () => {
                try {
                    setLoading(true);
                    const response = await
fetch(`/messages/${forum}`);
                    if (!response.ok) {
                        const text = await response.text();
                        throw new Error(
                            `Unable to read messages for ${forum}:
${text}`
                        );
                    }
                    const body = await response.json();
                    setData(body);
                } catch(err) {
                    setError(err);
                } finally {
                    setLoading(false);
                }
            })();
        } else {
            setData([]);
            setLoading(false);
        }
    }, [forum]);

    return {data, loading, error};
};

export default useMessages;
```

Because we're using `await` to handle the promises correctly, we need to wrap the code in a rather ugly `(async () => {...})()` call. Inside there, we're able to set values for `data`, `loading`, and `error` as the request runs, finishes, and (possibly) fails. All of this will happen asynchronously after the call to the hook has been completed. When the `data`, `loading`, and `error` states change, the hook will cause the component to be re-rendered with the new values.

Let's take a look at the `App.js` in the example application to see what it looks like to use this hook:

```
import './App.css';
import {useState} from "react";
import useMessages from "./useMessages";

function App() {
    const [forum, setForum] = useState('nasa');
    const {data: messages, loading: messagesLoading, error:
messagesError} = useMessages(forum);

    return (
        <div className="App">
            <button onClick={() => setForum('nasa')}>NASA</button>
            <button onClick={() => setForum('notNasa')}>Not
NASA</button>
            {
                messagesError ?
                    <div className='error'>
                        Something went wrong:
                        <div className='error-contents'>
                            {messagesError.message}
                        </div>
                    </div>
                    : messagesLoading ?
                    <div className='loading'>Loading...</div>
                    :
                    (messages && messages.length) ? <dl>
{messages.map(m => <>
                            <dt>{m.author}</dt>
                            <dd>{m.text}</dd>
                        </>)}</dl>
                        : 'No messages'
            }
        </div>
    );
}

export default App;
```

Our example application changes which forum is loaded when you click either the “NASA” or “Not NASA” buttons. The example server returns a 404-status for the “Not NASA” forum, which causes an error to appear on-screen. In *figure 5-2*, we can see the example application showing the loading state, the messages from the NASA forum, and an error when we try to load data from the missing “Not NASA” forum.



SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

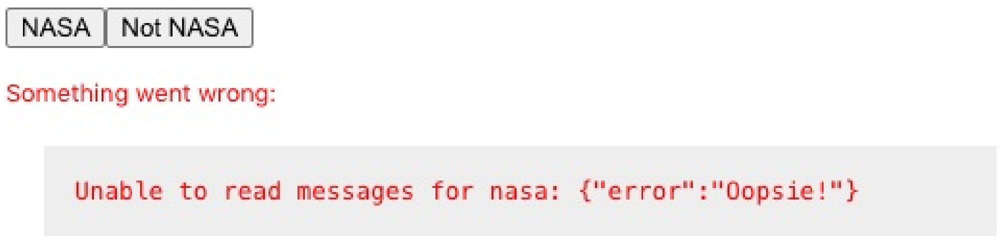
CAPCOM
All is well at Houston. You are good at 1 minute.

Something went wrong:

```
Unable to read messages for notNasa: {"error":"Unknown
forum"}
```

Figure 5-2. The application showing loading, messages, and errors

The `useMessages` hook will also cope if the server throws an error, as shown in *figure 5-3*.



```
Something went wrong:

Unable to read messages for nasa: {"error":"Oopsie!"}
```

Figure 5-3. The component can display errors that are thrown by the server.

Discussion

When you’re creating an application, it’s tempting to spend your time building features that assume everything works. But it is worth investing the time to handle errors and make an effort to show when data is still loading. Your application will be pleasant to use, and you will have an easier time tracking down slow services and errors.

You might also consider combining this recipe with the error-handling recipe in chapter 4, which will make it easier for users to describe what happened.

You can download the source for this recipe from the Github site.

5.2 Generate Automatic Refreshes with State Counters

Problem

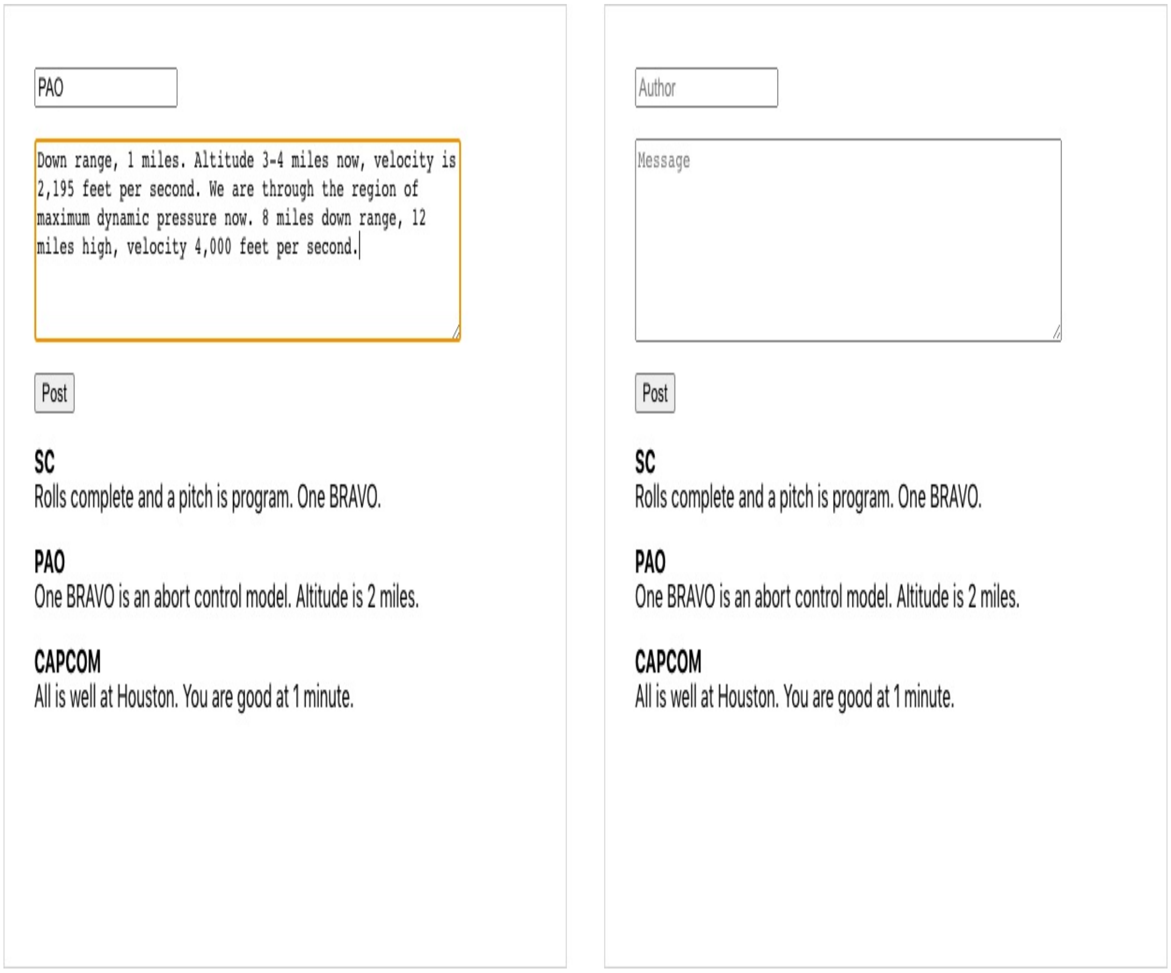
Network services often need to interact with each other. Take, for example, the forum application we used in the previous recipe. If we add a form to post a new message, we want the message-list to update automatically every time a person posts something.

In the previous version of this application, we created a custom hook called `useMessages`, which contained all of the code needed to read a forum's messages.

We'll add a form to the application to post new messages to the server:

```
const {
  data: messages,
  loading: messagesLoading,
  error: messagesError,
} = useMessages('nasa');
const [text, setText] = useState();
const [author, setAuthor] = useState();
const [createMessageError, setCreateMessageError] = useState();
// Other code here...
<input type='text' value={author} placeholder='Author'
       onChange={evt => setAuthor(evt.target.value)}/>
<textarea value={text} placeholder='Message'
          onChange={evt => setText(evt.target.value)}/>
<button onClick={async () => {
    try {
      await [code to post message here]
      setText('');
      setAuthor('');
    } catch(err) {
      setCreateMessageError(err);
    }
  }}
  >Post
</button>
```

Here's the problem: when you post a new message, it doesn't appear on the list unless you refresh the page manually (see *figure 5-4*).


PAO

Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

Figure 5-4. Posting a message does not refresh the message list.

How do we automatically reload the messages from the server each time we post a new one?

Solution

We’re going to trigger data refreshes by using a thing called a *state counter*. A state counter is just an increasing number. It doesn’t matter what the counter’s current value is; it just matters that we change it every time we want to reload the data.

```
const [stateVersion, setStateVersion] = useState(0);
```

You can think of a state counter as representing our perceived version of the data on the server. When we do something that we suspect will change the server state, we update the state counter to reflect the change:

```
// code to post a new message here
setStateVersion(v => v + 1);
```

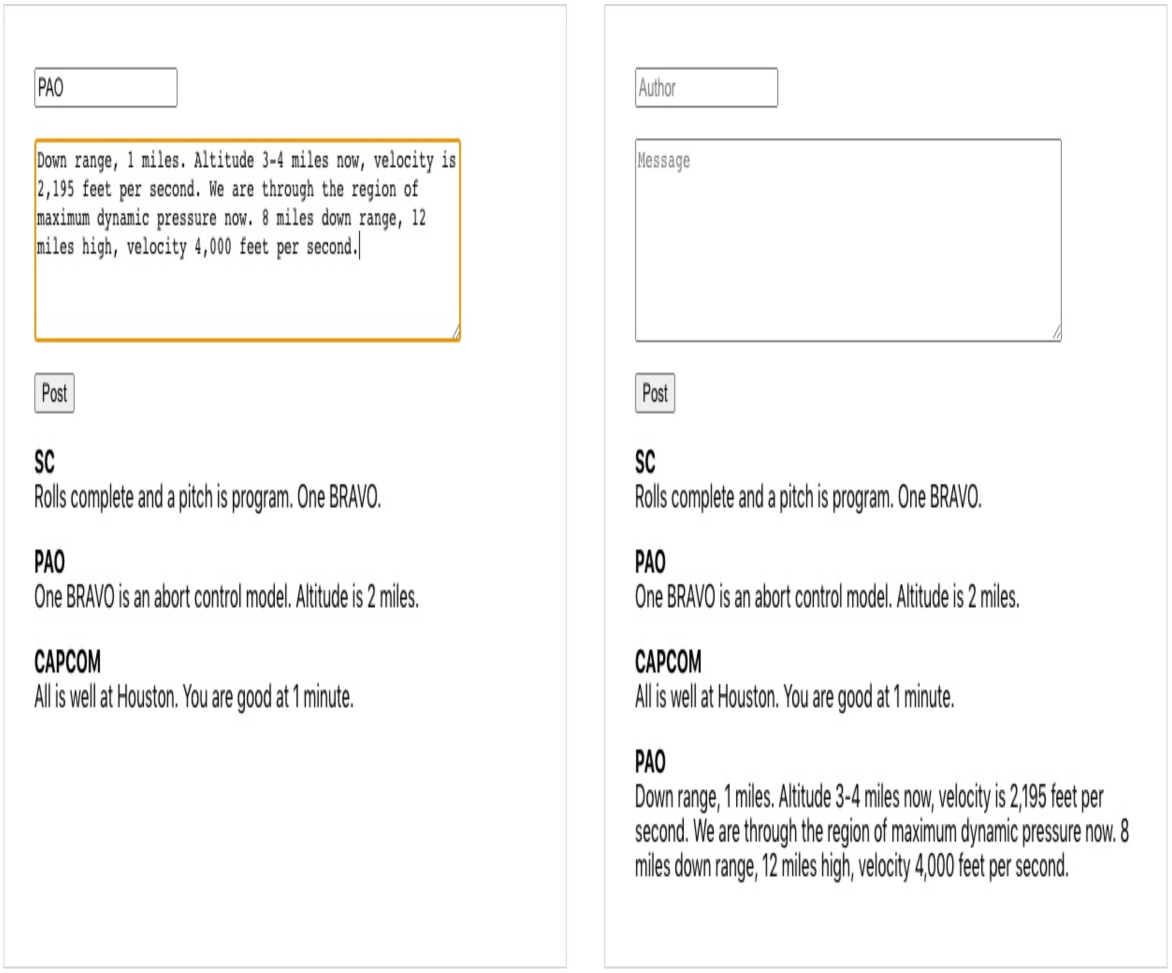
WARNING

Notice that we're increasing the `stateVersion` value using a function, rather than saying `setStateVersion(stateVersion + 1)`. You should always use a function to update a state value if the new value depends upon the old value. That's because React sets states asynchronously. If we ran `setStateVersion(stateVersion + 1)` twice in rapid succession, the value of `stateVersion` might not change in between the two calls, and we would miss an increment.

The code which reads the current set of messages is wrapped inside a `useEffect` which we can force to re-run by making it dependent upon the `stateVersion` value:

```
useEffect(() => {
    setError(null);
    if (forum) {
        // Code to read /messages/:forum
    } else {
        setData([]);
        setLoading(false);
    }
}, [forum, stateVersion]);
```

If the value of the `forum` variable changes, or if the `stateVersion` changes, it will automatically reload the messages (see *figure 5-5*).


PAO

Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

PAO
Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

Figure 5-5. Posting a new message causes the message-list to reload.

So that’s our approach. Now we need to look at where we’re going to put the code. This is the previous version of the component, which is only reading messages:

```
import './App.css';
import {useState} from "react";
import useMessages from "./useMessages";

function App() {
    const [forum, setForum] = useState('nasa');
    const {data: messages, loading: messagesLoading, error:
messagesError} = useMessages(forum);

    return (
        <div className="App">
            <button onClick={() => setForum('nasa')}>NASA</button>
            <button onClick={() => setForum('notNasa')}>Not
NASA</button>
            {
                messagesError ?
                    <div className='error'>
                        Something went wrong:
                        <div className='error-contents'>
                            {messagesError.message}
                        </div>
                    </div>
                    : messagesLoading ?
                    <div className='loading'>Loading...</div>
                    :
                    (messages && messages.length) ? <dl>
{messages.map(m => <>
                            <dt>{m.author}</dt>
                            <dd>{m.text}</dd>
                        </>)}</dl>
                        : 'No messages'
            }
        </div>
    );
}

export default App;
```

We're going to add the new form into this component. We could also include the networking code, and the state-counter code right here, inside the component. However that would put the posting-code in the component, and the reading-code in the `useMessages` hook. It's better to keep all the networking code together in the hook. Not only will the component be cleaner but the networking code will be more re-usable.

This is code we'll use for a new version of the `useMessages` hook, which we will rename `useForum`. [1]

```
import {useCallback, useEffect, useState} from "react";

const useForum = (forum) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();
    const [creating, setCreating] = useState(false);
    const [stateVersion, setStateVersion] = useState(0);

    const create = useCallback(async (message) => {
        try {
            setCreating(true);
            const response = await fetch(`/messages/${forum}`, {
                method: 'POST',
                body: JSON.stringify(message),
                headers: {
                    "Content-type": "application/json; charset=UTF-8"
                }
            });
            if (!response.ok) {
                const text = await response.text();
                throw new Error(
                    `Unable to create a ${forum} message: ${text}`
                );
            }
            setStateVersion(v => v + 1);
        } finally {
            setCreating(false);
        }
    }, [forum]);

    useEffect(() => {
        setError(null);
        if (forum) {
            (async () => {
                try {
                    setLoading(true);
                    const response = await
fetch(`/messages/${forum}`);
                    if (!response.ok) {
                        const text = await response.text();
                        throw new Error(
                            `Unable to read messages for ${forum}:
${text}`
                        );
                    }
                    const body = await response.json();
                    setData(body);
                } catch (err) {
                    setError(err);
                } finally {
                    setLoading(false);
                }
            })();
        } else {
            setData([]);
            setLoading(false);
        }
    }, [forum, stateVersion]);
    return {data, loading, error, create, creating};
};

export default useForum;
```

We now construct a `create` function inside the `useForum` hook and then return it with various other pieces of state to the component. Notice that we are wrapping the `create` function inside a `useCallback`. This means that we won't create a new version of the function, unless we need to for a different `forum` value.

WARNING

Be careful when creating functions inside hooks and components. React will often trigger a re-render if a new function object is created, even if that function does exactly the same thing as the previous version.

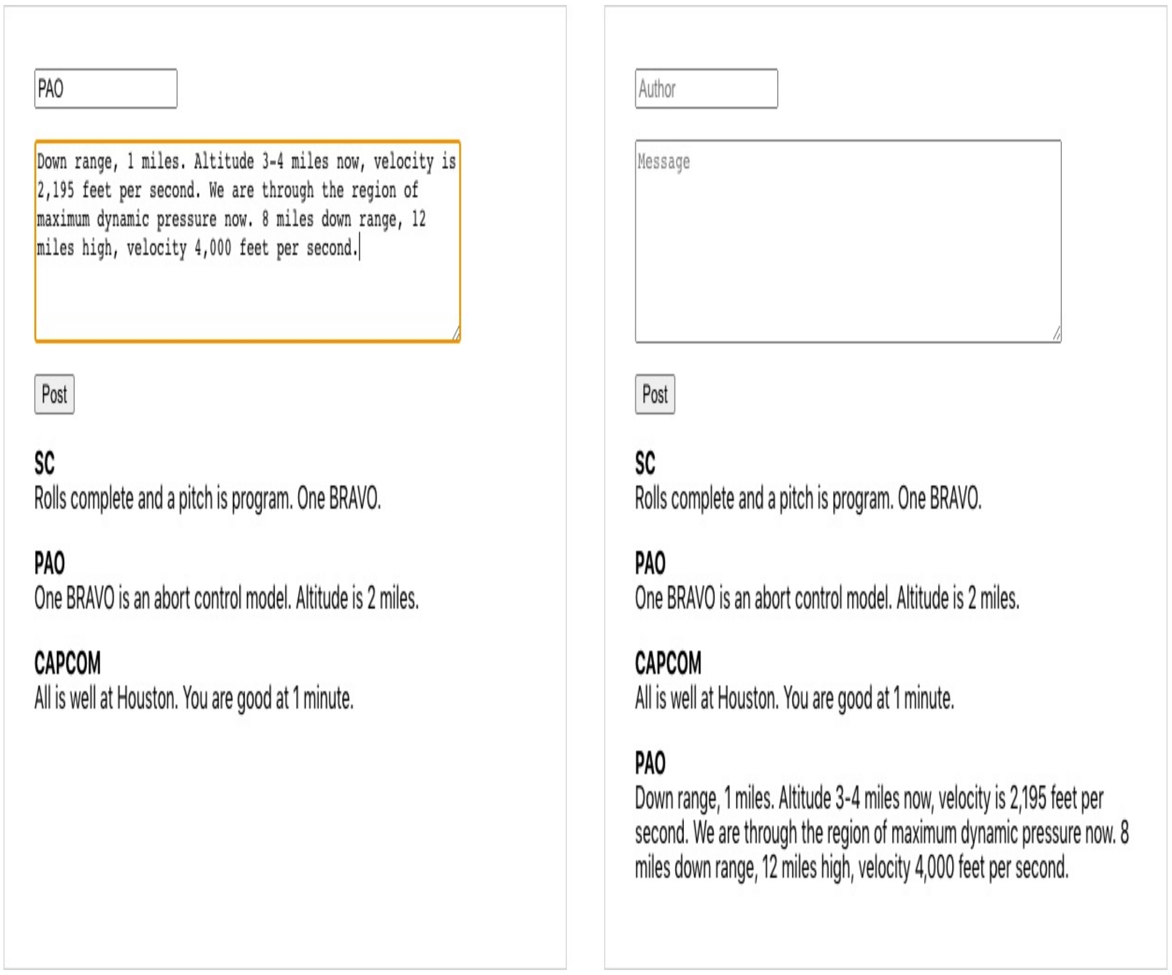
When the `create` function is called, it posts a new message to the forum, and then updates the `stateVersion` value, which will automatically cause the hook to re-read the messages from the server. Notice, that we also have a `creating` value which is set to `true` when the network code is in the process of sending the message to the server. This will allow us to disable the *POST* button while the message is being sent.

However, we **don't** track any errors inside the `create`. Why don't we? After all we do when we're **reading** data from the server. It's because you often want more control over exception handling when changing data on the server, than you do when you are simply reading it. In the example application, we clear out the message form if a message is correctly sent to the server. If there's an error, we want to leave the text in the message form.

OK, so that's what our hook looks like. What about the client code which calls it?

```
import './App.css';
import {useState} from "react";
import useForum from "./useForum";

function App() {
    const {
        data: messages,
        loading: messagesLoading,
        error: messagesError,
        create: createMessage,
        creating: creatingMessage,
    } = useForum('nasa');
    const [text, setText] = useState();
    const [author, setAuthor] = useState();
    const [createMessageError, setCreateMessageError] = useState();

    return (
        <div className="App">
            <input type='text' value={author} placeholder='Author'
                   onChange={evt => setAuthor(evt.target.value)}/>
            <textarea value={text} placeholder='Message'
                      onChange={evt => setText(evt.target.value)}/>
            <button onClick={async () => {
                try {
                    await createMessage({author, text});
                    setText('');
                    setAuthor('');
                } catch(err) {
                    setCreateMessageError(err);
                }
            }}
                    disabled={creatingMessage}
            >Post
            </button>
            {
                createMessageError ?
                    <div className='error'>
                        Unable to create message
                        <div className='error-contents'>
                            {createMessageError.message}
                        </div>
                    </div> : null
            }
            {
                messagesError ?
                    <div className='error'>
                        Something went wrong:
                        <div className='error-contents'>
                            {messagesError.message}
                        </div>
                    </div>
                    : messagesLoading ?
                    <div className='loading'>Loading...</div>
                    :
                    (messages && messages.length) ? <dl>
{messages.map(m => <>
                            <dt>{m.author}</dt>
                            <dd>{m.text}</dd>
                        </>)}</dl>
                        : 'No messages'
            }
        </div>
    );
}

export default App;
```

All of details of how we read and write messages are now hidden inside the `useForum` hook. We use the spread operator to assign the `create` function to the `createMessage` variable, and if we call `createMessage` then it will not only post the message, it will also automatically re-read the new messages from the forum, and update the screen.



PAO
Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.
Post
Author
Message
Post

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

PAO
Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

Figure 5-6. Posting a new message and automatically reloading

Our hook if no longer just a way to read data from the server. It’s becoming a *service* for managing the forum itself.

Discussion

Be careful using this approach if you intend to post data to the server in one component, and then read data in a *different* component; separate hook instances will have separate state counters, and posting data from one component, will not automatically re-read the data in another component. If you want to split post and read across separate components, then call the custom hook in some common parent component and pass the data and the posting functions to the child components that need them.

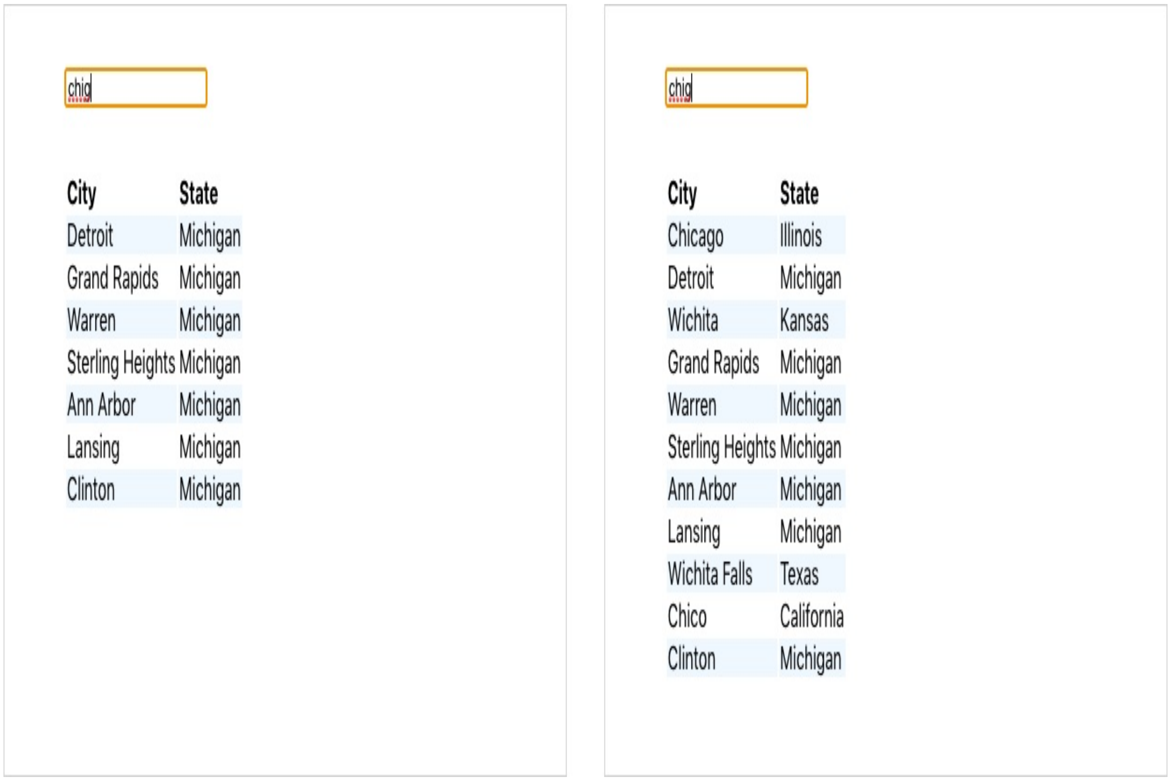
If you want to make your code poll a network service at a regular interval, then consider creating a clock[2] and making your network code depend upon the current clock value, much as the above code depends upon the state counter.

You can download the source for this recipe from the Github site.

5.3 Prevent Late Responses With Cancel Tokens

Problem

Let’s consider a buggy application that can be used to search for cities. When a user starts to type a name in the search field, a list of matching cities appears. As the user type “C… H… I… G…” then the matching cities appears in the table of results. But then, after a moment, a longer list of cities appears which includes erroneous results, such as *Wichita Falls* (see *figure 5-7*).



chig

City	State
Detroit	Michigan
Grand Rapids	Michigan
Warren	Michigan
Sterling Heights	Michigan
Ann Arbor	Michigan
Lansing	Michigan
Clinton	Michigan

chig

City	State
Chicago	Illinois
Detroit	Michigan
Wichita	Kansas
Grand Rapids	Michigan
Warren	Michigan
Sterling Heights	Michigan
Ann Arbor	Michigan
Lansing	Michigan
Wichita Falls	Texas
Chico	California
Clinton	Michigan

Figure 5-7. The search works initially, then the wrong cities appear

The problem is that the application is sending a new network request each time the user types a character. But not all network requests take the same amount of time. In the example you can see here, the network request that was search for “CHI” took a couple of seconds longer than the search for “CHIG”. That meant that the “CHI” results returned *after* the results for “CHIG”.

How can you prevent a series of asynchronous network calls returning out of sequence?

Solution

If you are making multiple GET calls to a network server, you can cancel old calls before sending new ones. This means that you will never get results back out of order because you will only have one network request calling the service at a time.

For this recipe we are going to use the *axios* network library. That means that we have to install it:

```
$ npm install axios
```

The *axios* library is a wrapper for the native *fetch* command. There is nothing you can do with *axios* that you can't do with *fetch*, but *axios* makes it a little easier.

Let's begin by looking at our problem code. The network code is wrapped in a custom hook[3]

```
import {useEffect, useState} from "react";
import axios from "axios";

export default (terms) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    useEffect(() => {
        setError(null);
        if (terms) {
            (async () => {
                try {
                    setLoading(true);
                    const response = await axios.get("/search",
                        {
                            params: {terms},
                        },
                    );
                    setData(response.data);
                } catch (err) {
                    setError(err);
                } finally {
                    setLoading(false);
                }
            })();
        } else {
            setData([]);
            setLoading(false);
        }
    }, [terms]);

    return {data, loading, error};
};
```

The `terms` parameter contains the string that we are searching for. The problem occurred because the code made a network request to `/search` for the string

`"CHI"`.

While that was in progress another call was made with the string `"CHIG"`. This earlier request took longer, which caused the bug.

We're going to avoid this problem using an axios *cancel-token*. If we attach a token to a request, we can then later use the token to cancel the request. The browser will terminate the request, and we'll never hear back from it.

To use the token, we need to first create a source for it:

```
const source = axios.CancelToken.source();
```

The `source` is like a remote control for the network request. Once a network request is connected to a source, we can tell the source to cancel it. We associate a source with a request using `source.token`:

```
const response = await axios.get("/search",{
    params: {terms},
    cancelToken: source.token,
});
```

axios will remember which token is attached to which network request. It we want to cancel the request, we just call:

```
source.cancel("axios request canceled")
```

We need to make sure that we only cancel a request if a new request is called. Fortunately, out network call is inside a `useEffect`, which has a very useful feature. If we return a function that cancels the current request, then this function will be run *just before* the `useEffect` runs again. So if we return a function that cancels the current network request, we will automatically cancel the old network request, each time we run a new one[4]. This is the updated version of the custom hook:

```
import {useEffect, useState} from "react";
import axios from "axios";

export default (terms) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    useEffect(() => {
        setError(null);
        if (terms) {
            const source = axios.CancelToken.source();
            (async () => {
                try {
                    setLoading(true);
                    const response = await axios.get("/search",
                        {
                            params: {terms},
                            cancelToken: source.token,
                        },
                    );
                    setData(response.data);
                } catch (err) {
                    setError(err);
                } finally {
                    setLoading(false);
                }
            })();

            return () => {
                source.cancel("axios request cancelled");
            };
        } else {
            setData([]);
            setLoading(false);
        }
    }, [terms]);

    return {data, loading, error};
};
```

Discussion

You should only use this approach if you are accessing services that are idempotent. In practice, this means that you should use it for *GET* requests where you are only interested in the latest results.

You can download the source for this recipe from the Github site.

5.4 Make Network Calls with Redux Middleware

Problem

Redux is a library that allows you to manage application state centrally[5]. When you want to change the application state, you do it by dispatching commands (called *actions*), which are captured and processed by JavaScript functions called *reducers*. Redux is popular with React developers because it provides a way to separate state-management logic from component code. Actions are performed asynchronously, but in a strict order. This means you can create large, complex applications that are both efficient and stable.

It would be great if we could leverage the power of Redux to orchestrate all of our network requests. We could dispatch actions that say things like “Go and read the latest search results”, and Redux could make the network request, and then update the central state.

However, to ensure that Redux code is stable, reducer functions have to meet a number of quite strict criteria: and one of them is that *no reducer function can have side effects*. That means that you should never make network requests inside a reducer.

But if we cannot make network requests inside reducer functions, how can we configure Redux to talk to the network for us?

Solution

In a React-Redux application, components publish (*dispatch*) actions, and reducers respond to actions by updating the central state (see *figure 5-8*).

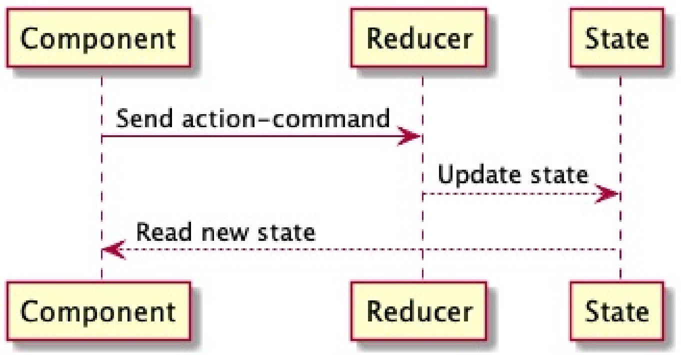


Figure 5-8. Using Redux reducers to update central state

If we want create actions that have side-effects, we will have to use Redux *middleware*. Middleware receives actions before they are sent to the reducers, and they can transform actions, cancel them or create new actions. Most importantly, Redux middleware code is allowed to have side-effects. That means that if have a component that dispatches an action that says “Go and search for this string…”, we can write middleware that receives that action, generates a network call, and then convert the response into a new “Store these search results” action. You can see how Redux middleware works in *figure 5-9*.

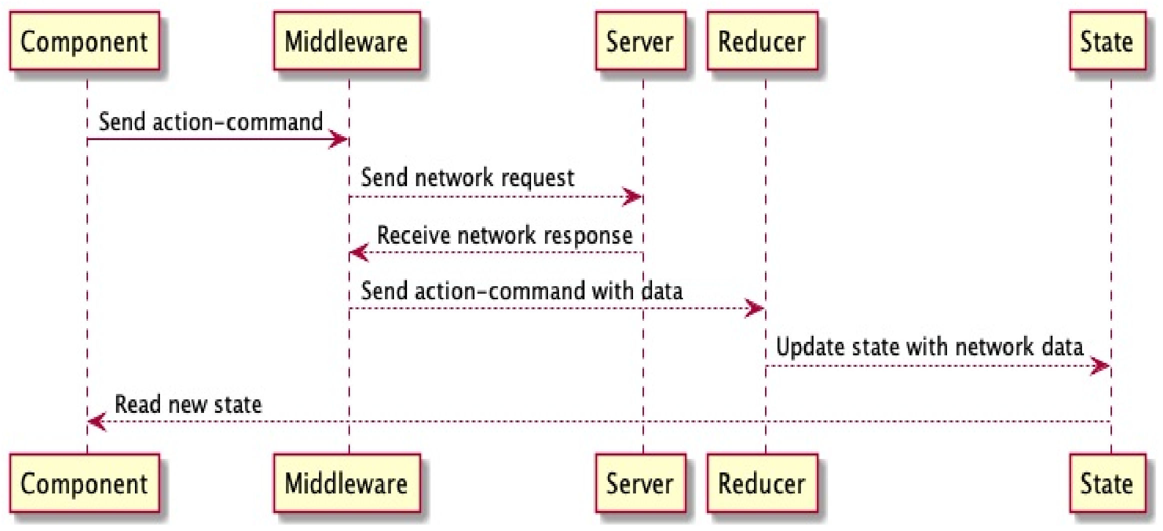


Figure 5-9. Middleware can be used to make network calls.

Let’s create some middleware which intercepts an action with type `"SEARCH"` and uses it to generate a network service.

When we get the results back from the network, we will then create a new action with type `"SEARCH_RESULTS"`, which we can then use to store the search results in the central Redux state. Our action object will look something like this:

```
{
  "type": "SEARCH",
  "payload": "Some search text"
}
```

This is the `aciosMiddleware.js` code that we’ll use to intercept `SEARCH` actions:

```
import axios from 'axios';

let axiosMiddleware = store => next => action => {
    if (action.type === 'SEARCH') {
        const terms = action.payload;
        if (terms) {
            (async () => {
                try {
                    store.dispatch({
                        type: 'SEARCH_RESULTS',
                        payload: {
                            loading: true,
                            data: null,
                            error: null,
                        }
                    });
                    const response = await axios.get("/search",
                        {
                            params: {terms},
                        },
                    );
                    store.dispatch({
                        type: 'SEARCH_RESULTS',
                        payload: {
                            loading: false,
                            error: null,
                            data: response.data
                        }
                    });
                } catch(err) {
                    store.dispatch({
                        type: 'SEARCH_RESULTS',
                        payload: {
                            loading: false,
                            error: err,
                            data: null
                        }
                    });
                }
            })();
        }
    }
    return next(action);
};
export default axiosMiddleware;
```

The function signature for axios middleware can be confusing. You can think of it as a function which is given a store, an action and another function called next, which can be used to forward actions on to the rest of Redux.

In the code above, we check to see if the action is of type SEARCH. If it is, we will make a network call. If it isn't, we run next(action), which will pass it on to any other code which might be interested in it.

When we start the network call, when we receive data, and if we capture any errors, then we can generate a new SEARCH_RESULTS action:

```
store.dispatch({
    type: 'SEARCH_RESULTS',
    payload: {
        loading: ...,
        error: ...,
        data: ...
    }
});
```

The payload for our new action, has:

- A boolean flag called loading, which is initially set to false, until the network request is complete
- A data object, which contains the response from the server, and
- An error object containing the details of any error which might occur[6]

We can then create reducer-code which will store SEARCH_RESULTS in the central state:

```
let reducer = (state, action) => {
    if (action.type === 'SEARCH_RESULTS') {
        return {
            ...state,
            searchResults: {...action.payload},
        };
    }
    return {...state};
};
export default reducer;
```

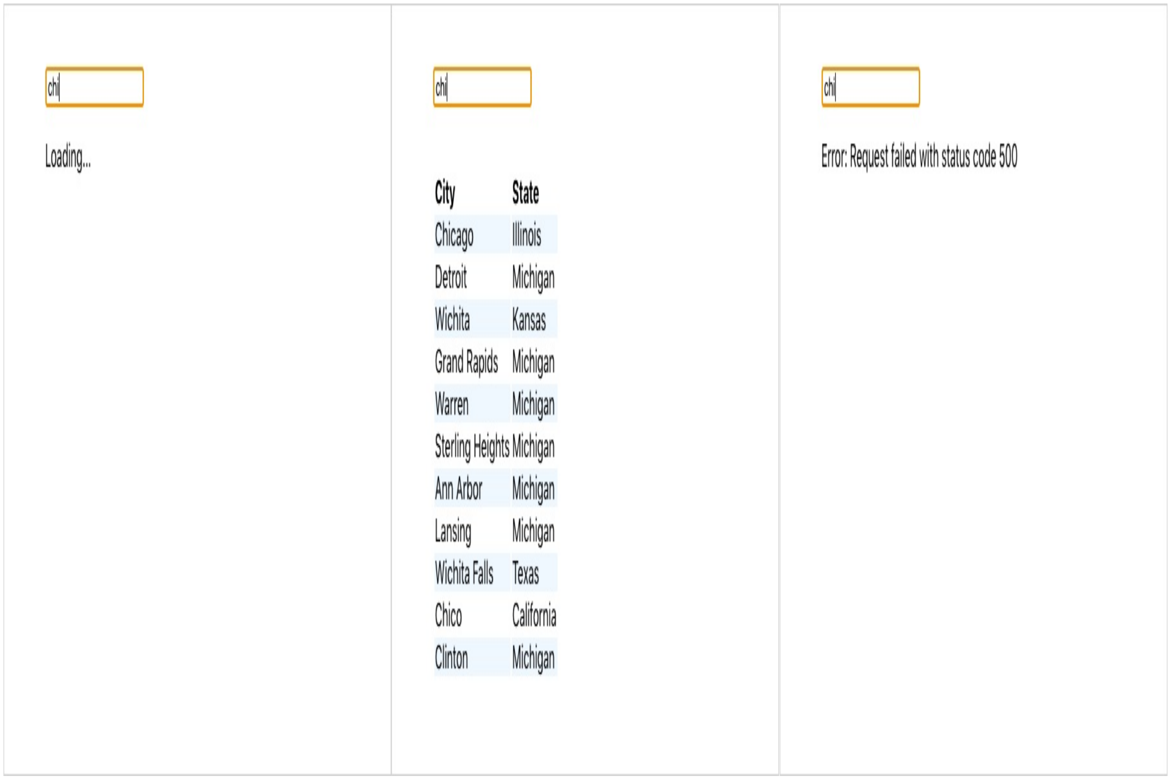
We also need to register out middleware using the Redux applyMiddleware function, when we create the Redux store. In the example code, we do it in the App.js file:

```
import {Provider} from 'react-redux'
import {createStore, applyMiddleware} from 'redux';
import './App.css';

import reducer from "./reducer";
import Search from "./Search";
import axiosMiddleware from "./axiosMiddleware";

const store = createStore(reducer, applyMiddleware(axiosMiddleware));

function App() {
  return (
    <div className="App">
        <Provider store={store}>
            <Search/>
        </Provider>
    </div>
  );
}

export default App;
```

Finally, we can wire everything up in a `Search` component, which will dispatch a search request, and then read the results through a Redux selector:

```
import {Provider} from 'react-redux'
import {createStore, applyMiddleware} from 'redux';
import './App.css';

import reducer from "./reducer";
import Search from "./Search";
import axiosMiddleware from "./axiosMiddleware";

const store = createStore(reducer, applyMiddleware(axiosMiddleware));

function App() {
  return (
    <div className="App">
        <Provider store={store}>
            <Search/>
        </Provider>
    </div>
  );
}

export default App;
```

You can see the demo application running in *figure 5-10*.


chi

Loading...

chi

City	State
Chicago	Illinois
Detroit	Michigan
Wichita	Kansas
Grand Rapids	Michigan
Warren	Michigan
Sterling Heights	Michigan
Ann Arbor	Michigan
Lansing	Michigan
Wichita Falls	Texas
Chico	California
Clinton	Michigan

chi

Error: Request failed with status code 500

Figure 5-10. The application when data is loading, loaded or errored.

Discussion

Redux reducers always process actions in strict dispatch-order. The same is not true for network requests generated by middleware. If you are making of a lot network requests in quick succession, you might find that responses return in a different order. If this is likely to lead to bugs, then consider using cancellation tokens[7]

You might also consider moving all Redux `useDispatch()/useSelector()` code out of components and into custom hooks. This will give you a more flexible architecture by separating your service later from your component code.

You can download the source for this recipe from the Github site.

5.5 Connecting to GraphQL

Problem

GraphQL is a great way of creating APIs. If you’ve used REST services for a while, then some features of GraphQL will seem odd (or even heretical), but having worked on a few GraphQL projects we would certainly recommend that you consider it for your next development project.

When people refer to GraphQL, they can mean several things. They might be referring to the GraphQL language, which is managed and maintained by GraphQL Foundation. The GraphQL allows you to specify APIs, and to create queries to access and mutate the data stored behind those APIs. They might be referring to a GraphQL server, which stitches together multiple low-level data access methods into a rich web service. Or they might be referring to a GraphQL client, which allows you rapidly create new client requests with the minimal of coding and to transfer just the data you need across the network.

But you do integrate GraphQL with your React application?

Solution

Before we look at how to use GraphQL from React, will begin by creating a small GraphQL server. The first thing we need is a GraphQL *schema*. The schema is a formal definition of the data and services that our GraphQL server will provide.

This is the `schema.graphql` schema we’ll use. It’s a GraphQL specification of the forum message example we’ve used elsewhere in this chapter.

```
type Query {
    messages: [Message]
}

type Message {
    id: ID!
    author: String!
    text: String!
}

type Mutation {
    addMessage(
        author: String!
        text: String!
    ): Message
}
```

This schema defines a single *query* (method for reading data) called `messages`, which returns an array of `Message` objects. Each `Message` has an `id`, a string called `author` and a string called `text`. We also also have a single *mutation* (method for changing data) called `addMessage`, which will store a message based on an `author` string and a `text` string.

Before we create our sample server, we'll install a few libraries:

```
$ npm install apollo-server
$ npm install graphql
$ npm install require-text
```

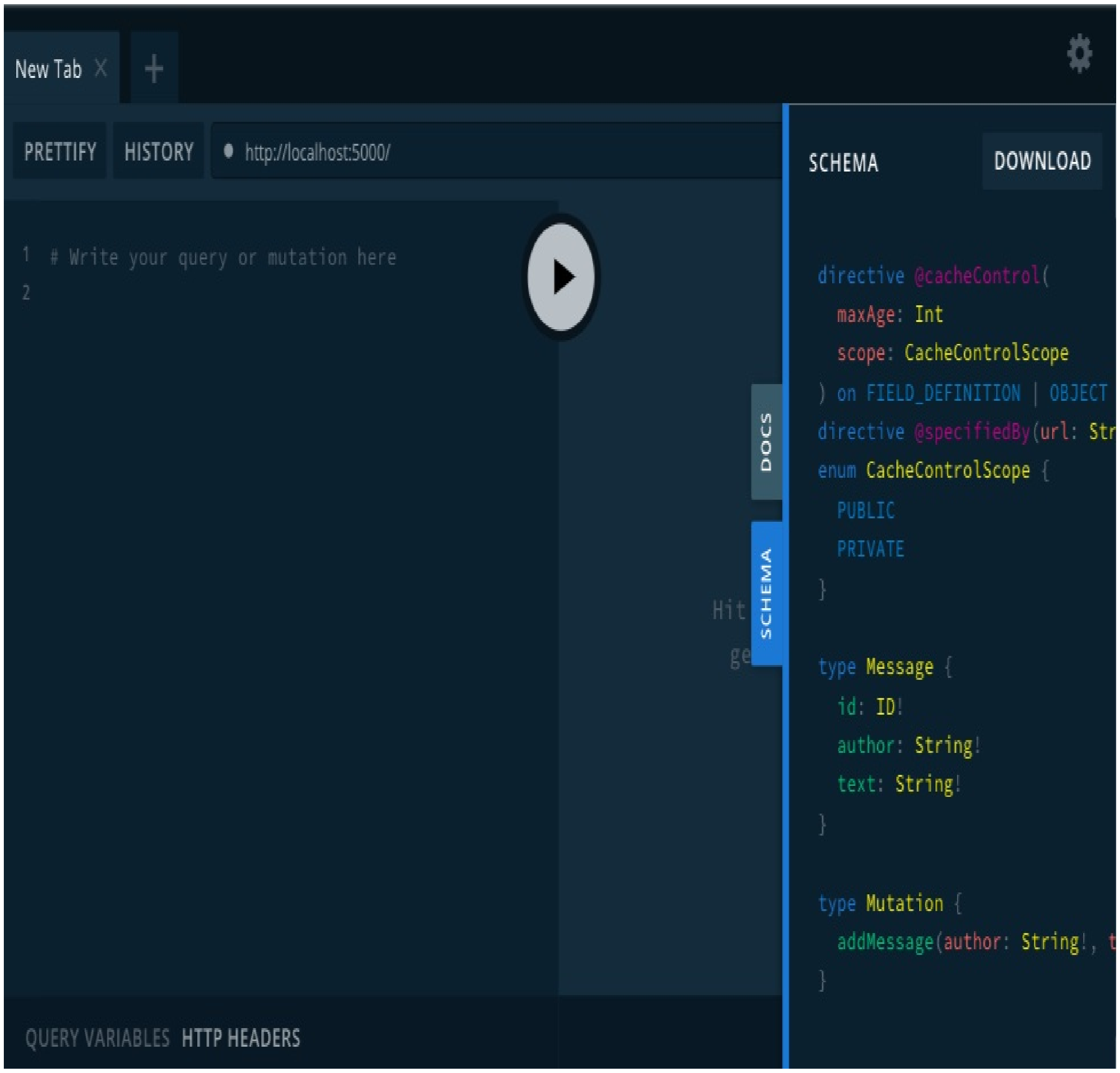
The `apollo-server` is a framework for creating GraphQL servers. The `require-text` library will allows us to read the `schema.graphql` file. This is our `server.js`, our example server:

```
type Query {
    messages: [Message]
}

type Message {
    id: ID!
    author: String!
    text: String!
}

type Mutation {
    addMessage(
        author: String!
        text: String!
    ): Message
}
```

All of the message data in the server is stored in memory in the `messages` array, which is pre-populated with a few messages. You can start the server with:

```
$ node ./server.js
```

This should start the server on port 5000. If you open a browser to *http://localhost:5000* you should see the GraphQL Playground client. This is a web tool which allows you try out queries and mutations interactively, before adding them to your code (see *figure 5-11*).



```
# Write your query or mutation here
```

```
directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
) on FIELD_DEFINITION | OBJECT
directive @specifiedBy(url: Str
enum CacheControlScope {
  PUBLIC
  PRIVATE
}

type Message {
  id: ID!
  author: String!
  text: String!
}

type Mutation {
  addMessage(author: String!, t
}
```

Figure 5-11. The GraphQL Playground should be running at http://localhost:5000

Now we can start to look at the React client code. We’ll install the Apollo client:

```
$ npm install @apollo/client
```

All of the requests to a GraphQL server are done with *POST* requests. This avoids any cross-domain issues, which means you can connect to a third-party GraphQL server, without having to proxy. As a consequence, it means that a GraphQL client has to handle its own caching, so we will need to provide a cache and the address of the server when we configure the client in `App.js`:

```
import './App.css';
import {ApolloClient, ApolloProvider, InMemoryCache} from
'@apollo/client';
import Forum from "./Forum";

const client = new ApolloClient({
    uri: 'http://localhost:5000',
    cache: new InMemoryCache()
});

function App() {
  return (
    <div className="App">
        <ApolloProvider client={client}>
            <Forum/>
        </ApolloProvider>
    </div>
  );
}

export default App;
```

The `ApolloProvider` makes the client available to any child component. If you forget to add the `ApolloProvider`, you will find that all of your GraphQL client code will fail.

We’re going to make the calls to GraphQL from inside the `Forum` component. We’ll be performing two action:

- A *query* called `Messages` which reads all of the messages, and
- A *mutation* called `AddMessage` which will post a new message

The query and the mutation of written in the GraphQL language. Here’s the `Messages` query:

```
query Messages {
  messages {
    author text
  }
}
```

This query means that we want to read all of the messages, but we only want to return the `author` adnd `text` strings. Because we're not asking for the message `id`, it won't be returned. This is part of the flexibility of GraphQL: you specify what you want at query time, rather than by crafting a special API call for each variation.

The `AddMessage` mutation is a little more complex, because it needs to be parameterized, so that we can specify the `author` and `text` values each time we call it:

```
mutation AddMessage(
  $author: String!
  $text: String!
) {
  addMessage(
    author: $author
    text: $text
  ) {
    author
    text
  }
}
```

We're going to use the `useQuery` and `useMutation` hooks provided by the Apollo GraphQL client. The `useQuery` hook returns an object with `data`, `loading` and `error` attributes[8]. The `useMutation` hook returns an array with two values: a function and an object representing the result.

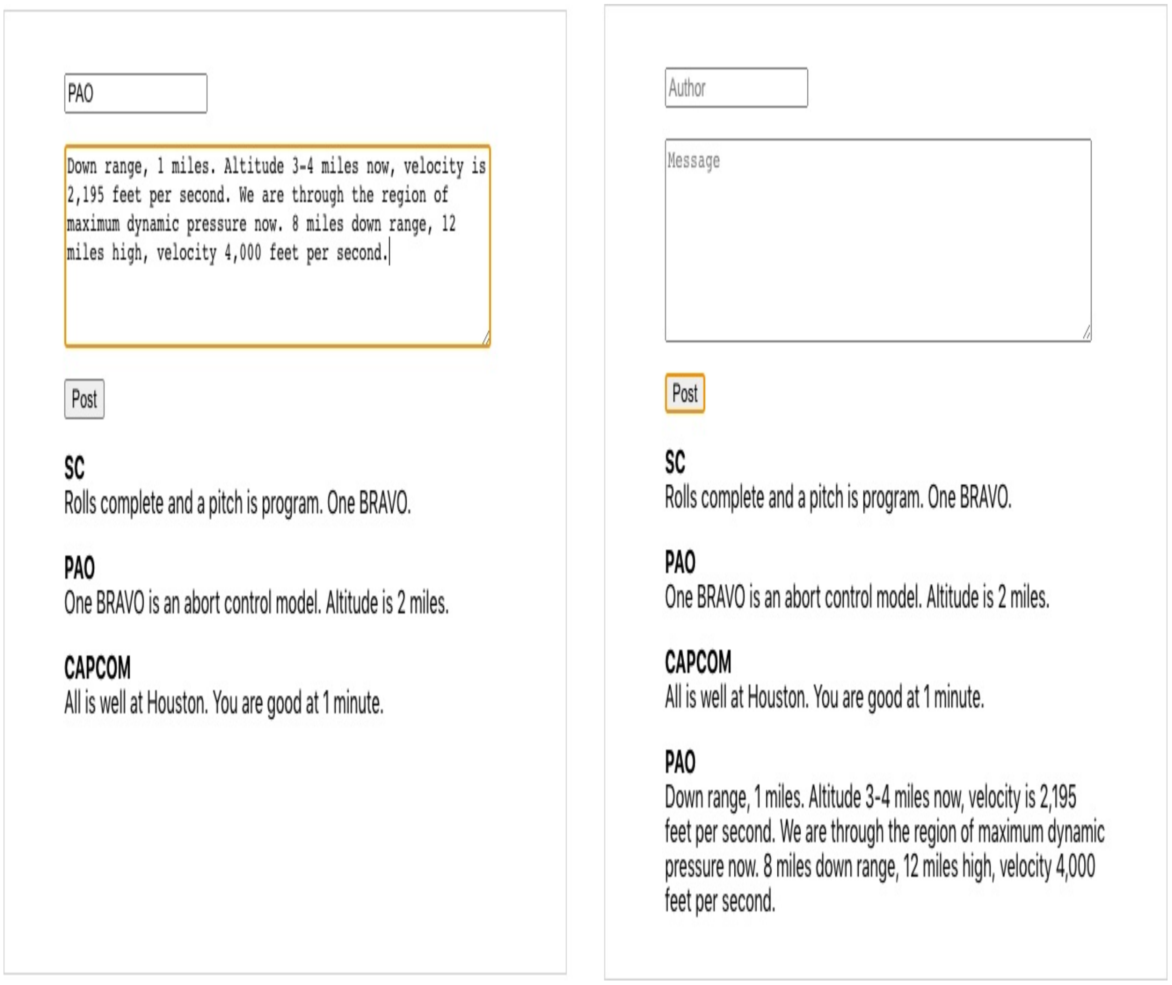
Previously[9] we have looked at the problem of how to how to automatically re-load data after some mutation has changed it on the server. Thankfully, the Apollo client has a ready-made solution. When you call a mutation, you can specify an array of other queries which should be re-run if the mutation is successful:

```
await addMessage({variables: {author, text}, refetchQueries:
['Messages']});
```

The `'Messages'` string refers to the name word `query` in the GraphQL query. This means we can be running multiple queries against the GraphQL service, and specify which of them are likely to need refreshing after a change. Finally, here is the complete `Forum` component:

```
import {gql, useMutation, useQuery} from '@apollo/client';
import {useState} from "react";

const MESSAGES = gql`
    query Messages {
        messages {
            author text
        }
    }
`;

const ADD_MESSAGE = gql`
    mutation AddMessage(
        $author: String!
        $text: String!
    ) {
        addMessage(
            author: $author
            text: $text
        ) {
            author
            text
        }
    }
`;

let Forum = () => {
    const {loading: messagesLoading, error: messagesError, data} =
useQuery(MESSAGES);
    const [addMessage] = useMutation(ADD_MESSAGE);
    const [text, setText] = useState();
    const [author, setAuthor] = useState();

    const messages = data && data.messages;

    return (
        <div className="App">
            <input type='text' value={author} placeholder='Author'
                   onChange={evt => setAuthor(evt.target.value)}/>
            <textarea value={text} placeholder='Message'
                      onChange={evt => setText(evt.target.value)}/>
            <button onClick={async () => {
                try {
                    await addMessage({variables: {author, text},
                        refetchQueries: ['Messages']});
                    setText('');
                    setAuthor('');
                } catch (err) {
                }
            }}
            >Post
            </button>
            {
                messagesError ?
                    <div className='error'>
                        Something went wrong:
                        <div className='error-contents'>
                            {messagesError.message}
                        </div>
                    </div>
                    : messagesLoading ?
                    <div className='loading'>Loading...</div>
                    :
                    (messages && messages.length) ? <dl>
{messages.map(m => <>
                            <dt>{m.author}</dt>
                            <dd>{m.text}</dd>
                        </>)}</dl>
                        : 'No messages'
            }
        </div>
    );
};
export default Forum;
```

When you run the application, and post a new message, the messages list is automatically updated with the new message added to the end, as you can see in *figure 5-12*.



Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

PAO
Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

Figure 5-12. After the message is posted, it appears on the list

Discussion

GraphQL is particularly useful if you have a large team, split between front- and back-end developers. Unlike REST, a GraphQL system does not require the back-end developers to hand-craft every API call made by the client. Instead, the back-end team can focus on providing a solid and consistent API structure, and leave it to the front-end team to decide exactly *how* they will use it.

If you are creating a React application using GraphQL, you might consider extracting all of the `useQuery` and `useMutation` calls into a custom hooks[10]. In this way you will create a more flexible architecture in which the components are less bound to the details of the service layer.

You can download the source for this recipe from the Github site.

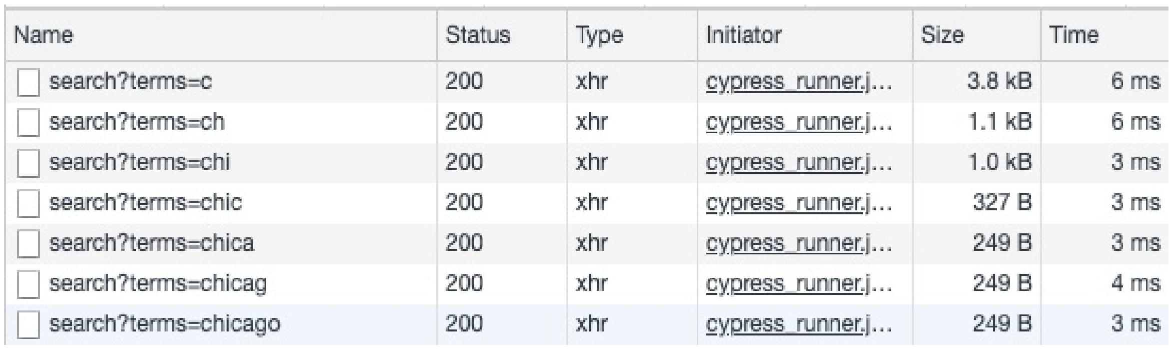
5.6 Reduce Network Load With Debounced Requests

Problem

It is very easy when you're working with in a development system to pnot worry too much about performance. That's probably a good thing, because it's more important that code does the right thing, rather than do the wrong thing quickly.

But when your application gets deployed to it's first real environment–such as one used for user acceptance testing–then performance will then become more important. The kind of dynamic interfaces associated with React often make a **lot** of network calls, and the cost of these calls will only really be noticeable once the server has to cope with a lot of concurrent clients.

We've used an example search application a few times in this chapter. In the search app, a user can look for a city by name or state. The search happens immediately–while they are typing. If you open the developer tools and look at the network requests (see *figure 5-13*) you will see that it generates as many network requests as there are characters typed.



Name	Status	Type	Initiator	Size	Time
search?terms=c	200	xhr	cypress_runner.j...	3.8 kB	6 ms
search?terms=ch	200	xhr	cypress_runner.j...	1.1 kB	6 ms
search?terms=chi	200	xhr	cypress_runner.j...	1.0 kB	3 ms
search?terms=chic	200	xhr	cypress_runner.j...	327 B	3 ms
search?terms=chica	200	xhr	cypress_runner.j...	249 B	3 ms
search?terms=chicag	200	xhr	cypress_runner.j...	249 B	4 ms
search?terms=chicago	200	xhr	cypress_runner.j...	249 B	3 ms

Figure 5-13. The demo search application runs a network request for each character.

Most of these network requests will provide almost no value. The average typist will probably hit a key every half second, and if they looking at their keyboard they probably won't even see the results for each of those searches. Of the 7 requests they send to the server, they will likely only read the results from one of them: the last. That means the server is doing 7 times more work than was needed.

What can we do avoid to avoid sending so many wasted requests?

Solution

We’re going to **debounce** the network requests for the search calls. Debouncing means that we will delay sending a network request for a very short period of time, say a half-second. If another request comes in while we’re waiting, we’ll forget about the first request and then create another delayed request. In this way we defer sending any request until the we receive no new requests for half a second.

To see how to do this, look at our example search code, `useSearch.js`:

```
import {useEffect, useState} from "react";
import axios from "axios";

const useSearch = (terms) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    useEffect(() => {
        setError(null);
        if (terms) {
            (async () => {
                try {
                    setLoading(true);
                    const response = await axios.get("/search",
                        {
                            params: {terms},
                        },
                    );
                    setData(response.data);
                } catch (err) {
                    setError(err);
                } finally {
                    setLoading(false);
                }
            })();
        } else {
            setData([]);
            setLoading(false);
        }
    }, [terms]);

    return {data, loading, error};
};
export default useSearch;
```

The code that sends the network request is wrapped up inside the `(async ()....)()` block of code. This is what we need to delay until we get a half second to spare.

The JavaScript function `setTimeout` allows to run code after a delay. This will be key to how we implement the debounce feature:

```
const newTimer = setTimeout(SOMEFUNCTION, 500)
```

The `newTimer` value it returns can be used to clear the timeout, which if we do it quickly enough, might mean that our function never gets called. To see how we can use this to debounce the network requests, look at `useDebouncedSearch.js`, and a debounced version of `useSearch.js`:

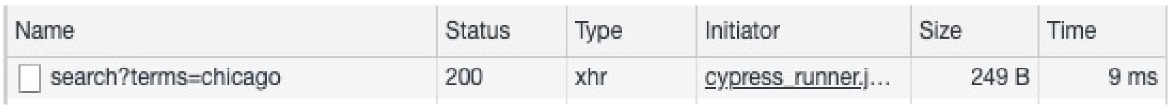
```
import {useEffect, useState} from "react";
import axios from "axios";

const useDebouncedSearch = (terms) => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState();

    useEffect(() => {
        setError(null);
        if (terms) {
            const newTimer = setTimeout(() => {
                (async () => {
                    try {
                        setLoading(true);
                        const response = await axios.get("/search",
                            {
                                params: {terms},
                            },
                        );
                        setData(response.data);
                    } catch (err) {
                        setError(err);
                    } finally {
                        setLoading(false);
                    }
                })();
            }, 500);
            return () => clearTimeout(newTimer);
        } else {
            setData([]);
            setLoading(false);
        }
    }, [terms]);

    return {data, loading, error};
};

export default useDebouncedSearch;
```

We pass the network code into the `setTimeout` function, and then return:

```
() => clearTimeout(newTimer)
```

If you return a function from `useEffect`, then this code is called just before the next time `useEffect` triggers. This means if network keep coming in within half a second, we keep deferring the network requests. Only if the code if left in peace for half a second, will it actually submit any network requests.

The original version of the `useSearch` ran a network request for every single character. With the debounced version of the hook, then typing at a normal speed will result in just a single network request (see *figure 5-14*).



Name	Status	Type	Initiator	Size	Time
search?terms=chicago	200	xhr	cypress_runner.j…	249 B	9 ms

Figure 5-14. The debounced search hook will send fewer requests.

Discussion

It’s important to remember that debouncing reduces the number of unnecessary network requests. It does **not** avoid the problem of network responses returning in a different order. For more details on how to avoid the response order problem, see recipe 3 in this chapter.

You can download the source for this recipe from the Github site.

[1] We’re renaming it because it is no longer just a way to read a list of messages but the forum as a whole. We could eventually add functions to delete, edit, or flag messages.

[2] See recipe 4 in chapter 3.

[3] Compare this code with recipe 1 in this chapter, which uses fetch.

[4] If the previous network request has completed, cancelling it will have no effect.

[5] It can also be quite confusing when you first use it. See chapter 3 for more Redux recipes.

[6] To simplify things, we are simply storing the entire object. In reality, you would want to ensure that the error contained only serializable text.

[7] See the third recipe in this chapter.

[8] This is a common set of values for an asynchronous service. We’ve used them in each of the non-GraphQL recipes in this chapter.

[9] In the second recipe of this chapter.

[10] Much as we do with HTTP network calls in recipe 2 of this chapter.

Chapter 6. Component Libraries

If you are building an application of any size, you are likely to need a component library. The number of data types that native HTML support are somewhat limited, and the implementations can vary from browser to browser. For example a date input field looks very different on Chrome, Firefox and Edge browsers.

Component libraries allow you to create a consistent feel for your application, regardless of the client that’s used. They will often adapt well when switching between desktop and mobile clients. Most importantly, component libraries often give your application a usability-boost. They have either been generated from design standards that have been thoroughly tested in UX labs (such as Material Design) or else have been developed over several years, and any rough UX corners have generally been smoothed out.

Be aware: there is no such thing as the *perfect* component library. They all have strengths and weaknesses, and you need to choose a library that best meets your needs. If you have a large UX team, and a strong set of pre-existing design standards, you are likely to want a library that allows for a lot of *tinkering* to adapt the library to match your corporate themes. An example would be Material-UI, which allows you to modify its components quite significantly. If you have a very small UX team, or no UX team at all, you would probably want to consider something like Semantic UI, which is clean and functional and get you up-and-running quickly.

Whichever library you choose, always remember that the most important thing in UX is not how your applications looks, but how it behaves. Users will soon ignore whatever flashy graphics you add to the interface, but they will never forget (or forgive) some part of the interface that irritates them each time they use it.

6.1 Use Material Design with Material-UI

Problem

Many applications are now available on both the web and as native applications on mobile devices. Google's Material Design is intended to provide a seamless experience across all platforms. Material Design is just a specification, and there are several implementations available. One such is the Material UI library for React. But what are the steps involved in using Material UI, and how do you install it?

Solution

Let's begin by installing the core Material UI library.

```
$ npm install @material-ui/core
```

The core library includes the main components, but there is one notable feature it omits: the standard typeface. In order to make Material UI feel the same as it does in a native mobile application, you should also install Google's Roboto type-face:

```
$ npm install fontsource-roboto
```

Material Design also specifies a large set of standard icons. These provide a common visual language for standard tasks so as editing task, creating new items, sharing content, and so on. In order to use high quality versions of these icons, you should also install the Material-UI icon library:

```
$ npm install @material-ui/icons
```

Now that we have Material-UI up and running, what can we do with it? We can't look in detail at all of the available components here[1], but we swill take a look at some of the more popular features.

We’ll begin by looking at the basics of styling within Material-UI. In order to ensure that Material-UI components appear the same across different browsers, they have included a `CssBaseline` component. This will normalize the basic styling of your application. It will remove margins and apply standard background colors. You should add a `CssBaseline` component somewhere near the start of your application. For example, if you are using `create-react-app`, you should probably add it to your `App.js`:

```
import CssBaseline from "@material-ui/core/CssBaseline";
'''

function App() {
    // ...

    return (
        <div className="App">
            <CssBaseline/>
            ...
        </div>
    );
}

export default App;
```

Next, we’ll take a look at the Material Design `AppBar` and `Toolbar` components. These provide the standard heading you see in most Material Design application, and are where other features such as hamburger-menus and drawer-panels will appear.

We’ll place an `AppBar` at the top of the screen, and put a `Toolbar` inside. This will give us a chance to look at the way that typography is handled inside Material-UI:

```
<div className="App">
    <CssBaseline/>
    <AppBar position='relative'>
        <Toolbar>
            <Typography component='h1' variant='h6' color='inherit'
noWrap>
                Material-UI Gallery
            </Typography>
        </Toolbar>
    </AppBar>
    <main>
      // Main content goes here....
    </main>
</div>
```

Although you can insert ordinary textual content inside Material-UI applications, it is generally bettwe to display it inside `Typography`. A `Typography` component will ensure that the text is rendered in accordance with the Material Design standards. It can also be used to display text inside the appropriate markup elements. In this case, we're going to display the text in the `Toolbar` as a `h1` element. That's what the `Typography component` attribute specifies: the HTML element that should be used to wrap the text. However, we can also tell Material-UI to *style* the text as if it's a `h6` heading. That will make it a little smaller, and less overpowering as a page heading.

Next, let's look at how Material-UI styles the output. It uses *themes*. A theme is a JavaScript object which is used to define a hierarchy of CSS styles. Themes can be defined centrally, and this allows you to control the overall appearance of your application.

Themes are extensible. We'll import a function called `makeStyles`, which will allow us to create a modified version of the default theme.

```
import {makeStyles} from "@material-ui/core/styles";
```

We're going to make our example application display a gallery of images, so we will want to create styles for gallery items, descriptions, and so on. We can create styles for these different screen elements with `makeStyles`:

```
const useStyles = makeStyles((theme) => ({
    galleryGrid: {
        paddingTop: theme.spacing(4),
    },
    galleryItemDescription: {
        overflow: 'hidden',
        textOverflow: 'ellipsis',
        whiteSpace: 'nowrap',
    },
}));
```

In this simplified example, we are extending the base theme to also include styles for classes called `galleryGrid` and `galleryItemDescription`. We can either add in additional CSS attributes literally, or (in the case of `paddingTop` in the `galleryGrid`) we can use a reference to some value if the current theme: in this case `theme.spacing(4)`. This allows us to defer parts of the styling to a centralized theme, where it can modified at a later date.

The `useStyles` returned by `makeStyles` is actually a hook which will generate a set of CSS classes, and then return their names that they can be used inside our component.

For example, we will want to display a grid of images, using `Container` and `Grid` components[2]. We can attached the styles to them from the theme like this:

```
const classes = useStyles();

return (
    <div className="App">
        ...
        <main>
            <Container className={classes.galleryGrid}>
                <Grid container spacing='4'>
                    <Grid item>...</Grid>
                    <Grid item>...</Grid>
                    ...
                </Grid>
            </Container>
        </main>
    </div>
);
```

Each `Grid` component is either a *container* or an *item*. We will display a gallery image within each container.

In Material Design, important **things** are generally displayed as `Cards`. A `Card` is a rectangular panel that appears to float slight above the background. If you’ve ever used the Google Play Store, you will have seen cards used to display applications, music tracks or other things that you might want to download. We will place a card inside each `Grid` item, and used it to display a preview of the image, a text description, and a button which can be used to show a more detailed version of the image. You can see the cards in the example application in *figure 6-1*.

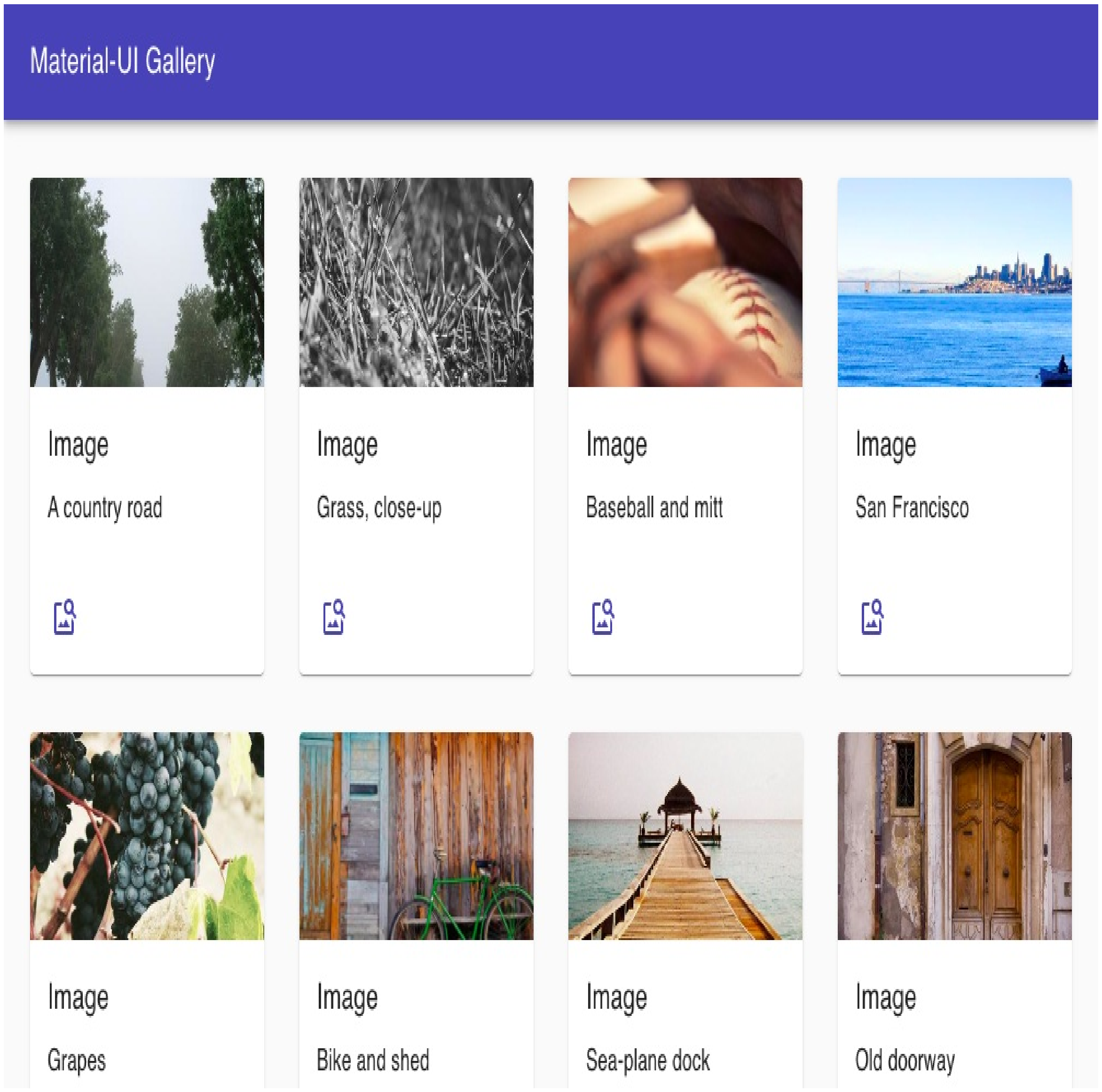


Figure 6-1. Cards are inside grid items, which are inside a container

Material UI also has extensive support for dialog windows. This is an example of a custom dialog:

```
import Dialog from "@material-ui/core/Dialog";
import DialogTitle from "@material-ui/core/DialogTitle";
import Typography from "@material-ui/core/Typography";
import DialogContent from "@material-ui/core/DialogContent";
import DialogActions from "@material-ui/core/DialogActions";
import Button from "@material-ui/core/Button";
import CloseIcon from '@material-ui/icons/Close';
```

```
export const MyDialog = ({onClose, open, title, children}) => {
    return <Dialog
        open={open}
        onClose={onClose}
    >
        <DialogTitle>
            <Typography component='h1' variant='h5' color='inherit'
noWrap>
                {title}
            </Typography>
        </DialogTitle>
        <DialogContent>
            {children}
        </DialogContent>
        <DialogActions>
            <Button variant='outlined'
                    startIcon={<CloseIcon />}
                    onClick={onClose}>
                Close
            </Button>
        </DialogActions>
    </Dialog>;
};
```

Notice that we are importing an SVG icon from the Material-UI SVG library that we installed earlier. The `DialogTitle` appears at the top of the dialog. The `DialogActions` are the buttons that appear at the base of the dialog. The main body of the dialog is given in the `DialogContent`.

Here is the complete code for `App.js`

```
import './App.css';
import CssBaseline from "@material-ui/core/CssBaseline";
import AppBar from "@material-ui/core/AppBar";
import {Toolbar} from "@material-ui/core";
import Container from "@material-ui/core/Container";
import Grid from "@material-ui/core/Grid";
import Card from "@material-ui/core/Card";
import CardMedia from "@material-ui/core/CardMedia";
import CardContent from "@material-ui/core/CardContent";
import CardActions from "@material-ui/core/CardActions";
import Typography from "@material-ui/core/Typography";
import {makeStyles} from "@material-ui/core/styles";
import {useState} from "react";
import {MyDialog} from "./MyDialog";
import ImageSearchIcon from '@material-ui/icons/ImageSearch';
import gallery from "./gallery.json";
import IconButton from "@material-ui/core/IconButton";

const useStyles = makeStyles((theme) => ({
    galleryGrid: {
        paddingTop: theme.spacing(4),
    },
    galleryItem: {
        height: '100%',
        display: 'flex',
        flexDirection: 'column',
        // maxWidth: '200px'
    },
    galleryImage: {
        paddingTop: '54%',
    },
    galleryItemDescription: {
        overflow: 'hidden',
        textOverflow: 'ellipsis',
        whiteSpace: 'nowrap',
    },
}));

function App() {
    const [showDetails, setShowDetails] = useState(false);
    const [selectedImage, setSelectedImage] = useState();
    const classes = useStyles();

    return (
        <div className="App">
            <CssBaseline/>
            <AppBar position='relative'>
                <Toolbar>
                    <Typography component='h1' variant='h6'
color='inherit' noWrap>
                        Material-UI Gallery
                    </Typography>
                </Toolbar>
            </AppBar>
            <main>
                <Container className={classes.galleryGrid}>
                    <Grid container spacing='4'>
                        {
                            gallery.map((item, i) => {
                                return <Grid item key={`photo-${i}`}
xs={12} sm={3} lg={2}>
                                    <Card className=
{classes.galleryItem}>
                                        <CardMedia image={item.image}
                                                   className=
{classes.galleryImage}
                                                   title='A photo'
                                        />
                                        <CardContent>
                                            <Typography gutterBottom
variant="h6" component="h2">
                                                Image
                                            </Typography>
                                            <Typography className=
{classes.galleryItemDescription}>
                                                {item.description}
                                            </Typography>
                                        </CardContent>
                                        <CardActions>
                                            <IconButton aria-
label="delete"
                                                    onClick={() =>
{
setSelectedImage(item);
setShowDetails(true);
                                                    }}
color="primary">
                                                <ImageSearchIcon />
                                            </IconButton>
                                        </CardActions>
                                    </Card>
                                </Grid>
                            })
                        }
                    </Grid>
                </Container>
            </main>
            <MyDialog
                open={showDetails}
                title='Details'
                onClose={() => setShowDetails(false)}>
                <img src={selectedImage && selectedImage.image}
alt='From PicSum'/>
                <Typography>
                    {selectedImage && selectedImage.description}
                </Typography>
            </MyDialog>
        </div>
    );
}

export default App;
```

Discussion

Material-UI is a great library to use and is one of the most popular libraries currently used with React. Users coming to your application will almost certainly have used it elsewhere, and that will increase the usability of your application. Before launching into using Material-UI on your application, it is worth spending some time understanding the Material Design principles. In that way, you will not only create an application is nice to look at, but one that is easy to use, is full of meaning, and which is accessible to users.

You can download the source for this recipe from the Github site.

6.2 Create Simple Effective UI with React Bootstrap

Problem

The most popular CSS library of the last 10 years is probably Twitter's Bootstrap library. It is incredibly popular, and it's a good choice if you are creating a new application and have very little time to worry about creating a custom UI, and just want something that is easy-to-use, easy-to-maintain and familiar to the vast number of users.

But Bootstrap was created at a time before large-scale application like React existed. It's made up of a large number of CSS resources, and a set of JavaScript libraries which are intended to be used with web pages containing a small amount of hand-crafted client code. The base Bootstrap library doesn't really play well with a framework like React.

How do you use Bootstrap when you're creating a React application?

Solution

There are actually several ports of the Bootstrap library for use with React. In this recipe we are going to look at *React Bootstrap*. React Bootstrap works alongside the ordinary Bootstrap CSS libraries, but it extends the Bootstrap JavaScript to make it more React-friendly.

Let’s begin by first install the React Bootstrap components, and the Bootstrap JavaScript libraries:

```
$ npm install react-bootstrap bootstrap
```

The React Bootstrap library does not include any CSS styling of its own. You will need to include a copy of that yourself. The most common way of doing this is by downloading it from a Content Distribution Network in your HTML. For example, if you are using `create-react-app`, you should include something like this in your `public/index.html` file:

```
<link
  rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.mi
n.css"
  integrity="sha384-
9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk"
  crossorigin="anonymous"
/>
```

You should replace this with whatever if the latest stable version of Bootstrap that’s available.

Bootstrap is a good, general purpose library, but it’s support for forms is particularly strong. Good form layout can take time, and can be tedious. Bootstrap handles all of the hard work for you and allows you to focus on the functionality of your form. For example, the `react-boostrap` `Form` component contains almost everything you need to create a form. The `Form.Control` component will generate an `input` by default. The `Form.Label` will generate a `label`, and a `Form.Group` will associate the two together, and lay them out appropriately:

```
<Form.Group controlId="startupName">
    <Form.Label>Startup name</Form.Label>
    <Form.Control placeholder="No names ending in ...ly, please"/>
</Form.Group>
```

Form fields are normally displayed on a single line and take up the available width. If you want more than one field to appear on a line, then you can use a

`Form.Row`:

```
<Form.Row>
    <Form.Group as={Col} controlId="startupName">
        <Form.Label>Startup name</Form.Label>
        <Form.Control placeholder="No names ending in ...ly, please"/>
    </Form.Group>
    <Form.Group as={Col} controlId="market">
        <Form.Label>Market</Form.Label>
        <Form.Control placeholder="e.g. seniors on Tik-Tok"/>
    </Form.Group>
</Form.Row>
```

The `Col` component ensures that the labels and fields are sized appropriately. If you want a form field that's other than an `input`, you can use the `as` attribute:

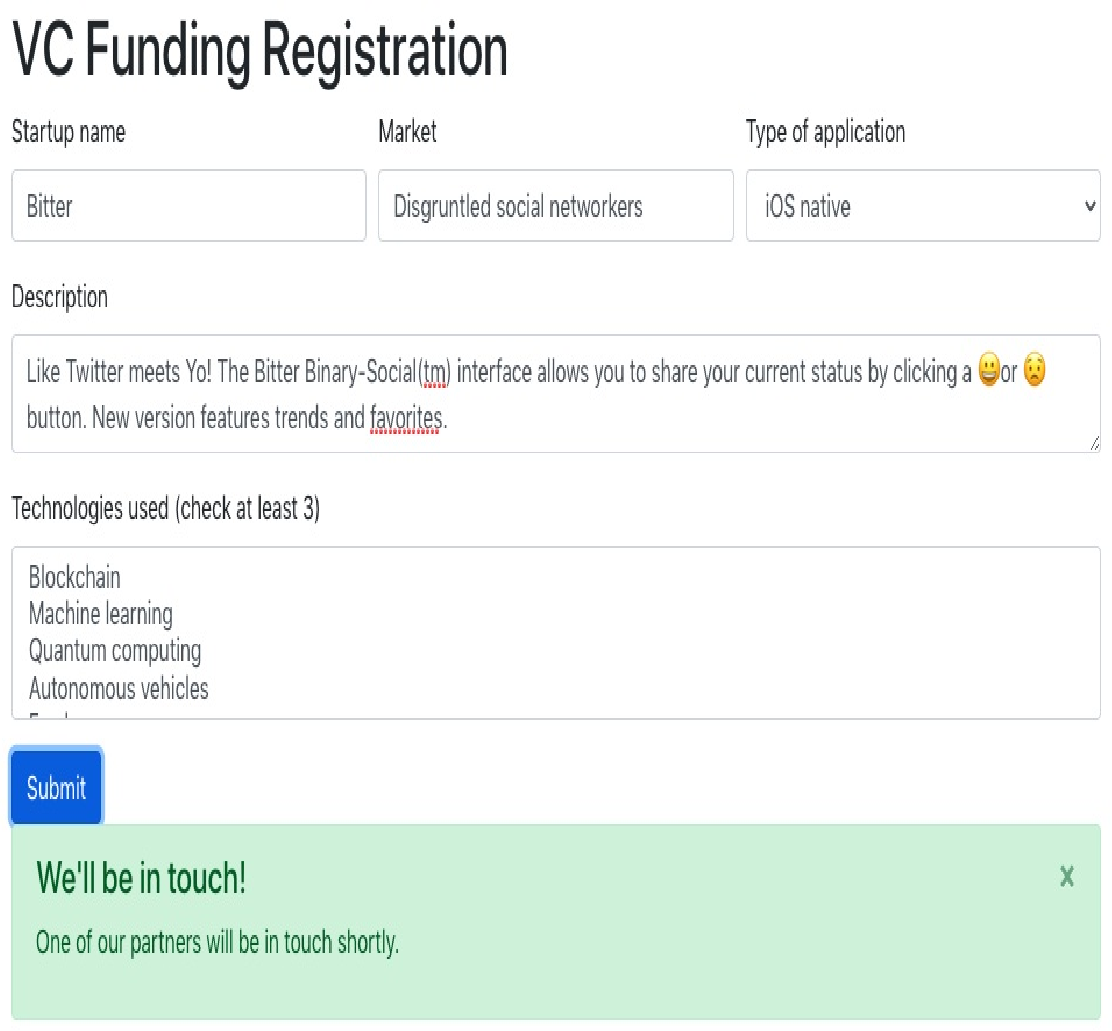
```
<Form.Control as="select" defaultValue="Choose...">
    <option>Progressive web application</option>
    <option>Conservative web application</option>
    <option>Android native</option>
    <option>iOS native</option>
    <option>New Jersey native</option>
    <option>VT220</option>
</Form.Control>
```

This will generate a bootstrap styled `select` element.

Putting the whole thing together leads to the form you can see in *figure 6-2*:

```
import Form from 'react-bootstrap/Form';
import Col from 'react-bootstrap/Col';
import Button from 'react-bootstrap/Button';
import Alert from 'react-bootstrap/Alert';
import {useState} from "react";
import './App.css';

function App() {
    const [submitted, setSubmitted] = useState(false);

    return (
        <div className="App">
            <h1>VC Funding Registration</h1>
            <Form>
                <Form.Row>
                    <Form.Group as={Col} controlId="startupName">
                        <Form.Label>Startup name</Form.Label>
                        <Form.Control placeholder="No names ending in
...ly, please"/>
                    </Form.Group>
                    <Form.Group as={Col} controlId="market">
                        <Form.Label>Market</Form.Label>
                        <Form.Control placeholder="e.g. seniors on
Tik-Tok"/>
                    </Form.Group>
                    <Form.Group as={Col} controlId="appType">
                        <Form.Label>Type of application</Form.Label>
                        <Form.Control as="select"
defaultValue="Choose...">
                            <option>Progressive web
application</option>
                            <option>Conservative web
application</option>
                            <option>Android native</option>
                            <option>iOS native</option>
                            <option>New Jersey native</option>
                            <option>VT220</option>
                        </Form.Control>
                    </Form.Group>
                </Form.Row>

                <Form.Row>
                    <Form.Group as={Col} controlId="description">
                        <Form.Label>Description</Form.Label>
                        <Form.Control as='textarea'/>
                    </Form.Group>
                </Form.Row>

                <Form.Group id="technologiesUsed">
                    <Form.Label>Technologies used (check at least 3)
</Form.Label>
                    <Form.Control as="select" multiple>
                        <option>Blockchain</option>
                        <option>Machine learning</option>
                        <option>Quantum computing</option>
                        <option>Autonomous vehicles</option>
                        <option>For-loops</option>
                    </Form.Control>
                </Form.Group>

                <Button variant="primary"
                        onClick={() => setSubmitted(true)}
                >
                    Submit
                </Button>
            </Form>
            <Alert show={submitted}
```

```
                        variant="success"
                        onClose={() => setSubmitted(false)} dismissible>
                     <Alert.Heading>We'll be in touch!</Alert.Heading>
                     <p>
                         One of our partners will be in touch shortly.
                     </p>
                 </Alert>
             </div>
         );
     }

     export default App;
```


VC Funding Registration

Startup name: Bitter

Market: Disgruntled social networkers

Type of application: iOS native

Description: Like Twitter meets Yo! The Bitter Binary-Social(tm) interface allows you to share your current status by clicking a 😀or 😟 button. New version features trends and favorites.

Technologies used (check at least 3)

Blockchain
Machine learning
Quantum computing
Autonomous vehicles

We'll be in touch!

One of our partners will be in touch shortly.

Figure 6-2. A React bootstrap form and alert box.

Discussion

Bootstrap is a much older UI toolkit than Material Design, but there are still markets where it feels more appropriate. If you’re building an application that has to feel more like a traditional web site, then Bootstrap will give it that more traditional feel. If you want to build something like feels more like a crossplatform **application**, then you should consider Material-UI[3].

You can download the source for this recipe from the Github site.

6.3 View Large Volumes of Data with React Window

Problem

Some applications are designed to display a seamingly endless quantity of data. If you are writing an application like Twitter, you don't want to download all of the tweets that are in the user timeline, because it would probably take several hours, days or months. The solution is to *window* the data. When you display a list of items, you only keep the items in memory that you are currently displaying. As you scroll up or down, the application downloads the data needed for the current view.

But creating this windowing logic is quite complex. Not only does it involve very careful tracking of what's currently visible[4] but if you're not careful you can easily run into memory issues if you fail to cache the windowing data efficiently.

How do you implement windowing code inside a React application?

Solution

The *React Window* library is a set of components specifically intended for applications that need to scroll a large amount of data. We'll look at how to create a large, fixed-size list[5].

To start, we need to create a component which will show the details for a single item. In our example application we're going to create a set of 10,000 date strings. We will render each individual date with a component called `DateRow`. This will be our *item-renderer*. `react-window` works by only rendering the items that are currently visible in the current viewport. As the user scrolls up or down the list, items will be created as they come into view and be removed as they disappear.

When `react-window` renders an item-renderer, it passes it two properties: an item number, which begins at 0, and a style object.

This is our `DateRow` item-renderer:

```
import moment from "moment";

const DateRow = ({index, style}) => (
    <div className={`aDate ${index % 2 && 'aDate-odd'}`} style=
{style}>
        {moment().add(index, 'd')
            .format('dddd, MMMM Do YYYY')}
    </div>
);

export default DateRow;
```

This component calculates a date `index` days in the future. In a real application, this component would probably download an item of data from a back-end server.

To generate the list itself, we will use a `FixedSizeList`. We need to give the list a fixed width and height. The list needs to calculate how many items are currently visible in the view-port, and it does this using the height of the list, and the height of each individual item, which is given in the `itemSize` attribute. If the `height` is `400` and the `itemHeight` is `40`, then the list will only need to display `10` or `11` `DateRow` (see *figure 6-3*.)

This is the final version of the code. Notice that the `FixedSizeList` does not include an instance of the `DateRow` component. That's because it wants to use the `DateRow` function to create multiple items dynamically as the list is scrolled. So instead of using `<DateRow/>`, the list uses the `{DateRow}` function itself.

```
import {FixedSizeList} from "react-window";
import DateRow from "./DateRow";
import './App.css';

function App() {
    return (
        <div className="App">
            <FixedSizeList
                height={400}
                itemCount={10000}
                itemSize={40}
                width={300}
            >
                {DateRow}
            </FixedSizeList>
        </div>
    );
}

export default App;
```

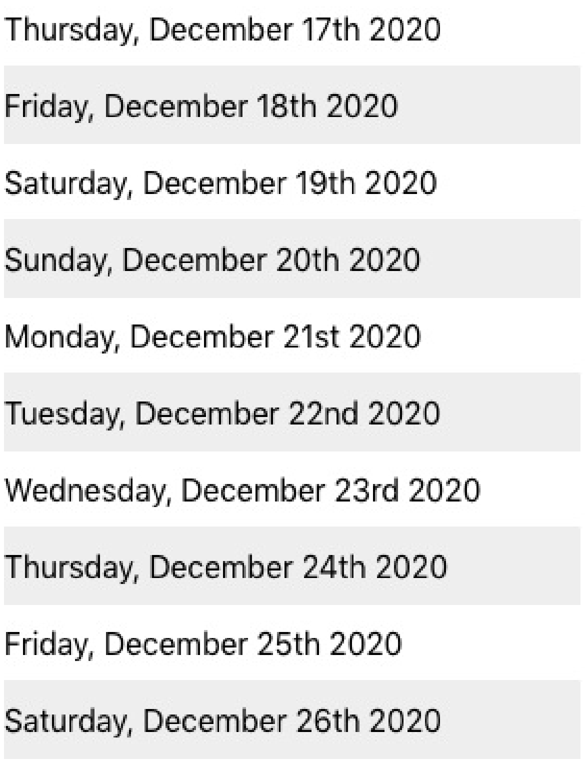

Thursday, December 17th 2020
Friday, December 18th 2020
Saturday, December 19th 2020
Sunday, December 20th 2020
Monday, December 21st 2020
Tuesday, December 22nd 2020
Wednesday, December 23rd 2020
Thursday, December 24th 2020
Friday, December 25th 2020
Saturday, December 26th 2020

Figure 6-3. The list contains only visible items.

One final point to note is that because the items are dynamically added and removed the to the list, you have to be very careful using the `nth-child` selector in CSS:

```
.aDate:nth-child(even) { /* This won't work */
```

```
    background-color: #eee;
}
```

Instead, you need to dynamically check the current index for an item, and check if it’s odd using a little modulo-2 math, as we do in the example:

```
<div className={`aDate ${index % 2 && 'aDate-odd'}`} ...>
```

Discussion

`react-window` is a very narrowly-focused component library, but an extremely useful one if you need to present a very large data set. You would still be responsible for downloading and caching the data that appears in the list but this is a relatively simple task, compared to the kind of virtual-list-magic that `react-window` performs.

You can download the source for this recipe from the Github site.

6.4 Create Responsive Dialogs with Material-UI

Problem

If you’re using a component library, there’s a good chance that at some point you will display a dialog window. Dialogs allow you to add additional UI detail without making the user feel that they are travelling to another page. They are very good for content creation, or as a quick way of displaying more detail about an item.

However, dialogs don’t plkay very well with mobile devices. Mobiles have a small display screen, and dialogs frequently waste a lot of space around the edges to display the background page.

How can you create responsive dialogs, which act just like floating windows when you are using a desktop machine, but look like separate fullscreen pages when you are on a mobile device?

Solution

The Material-UI includes a higher-order function which can tell you when you are on a mobile device, and should run display dialogs as full-screen windows.

```
import {withMobileDialog} from "@material-ui/core";
...

export const ResponsiveDialog = withMobileDialog()(
    ({... fullScreen, ...}) => {
        return // Some component using the fullScreen (true/false)
property;
    }
);
```

The `withMobileDialog` gives any component it wraps an extra property called `fullScreen`, which is set to `true` or `false`. A `Dialog` component can use this property to change its behavior. If you pass `fullScreen` to a `Dialog` like this:

```
import {withMobileDialog} from "@material-ui/core";
import Dialog from "@material-ui/core/Dialog";
import DialogTitle from "@material-ui/core/DialogTitle";
import Typography from "@material-ui/core/Typography";
import DialogContent from "@material-ui/core/DialogContent";
import DialogActions from "@material-ui/core/DialogActions";
import Button from "@material-ui/core/Button";
import CloseIcon from '@material-ui/icons/Close';

export const ResponsiveDialog = withMobileDialog()(
    ({onClose, open, title, fullScreen, children}) => {
        return <Dialog
            open={open}
            fullScreen={fullScreen}
            onClose={onClose}
        >
            <DialogTitle>
                <Typography component='h1' variant='h5'
color='inherit' noWrap>
                    {title}
                </Typography>
            </DialogTitle>
            <DialogContent>
                {children}
            </DialogContent>
            <DialogActions>
                <Button variant='outlined'
                        startIcon={<CloseIcon />}
                        onClick={onClose}>
                    Close
                </Button>
            </DialogActions>
        </Dialog>;
    }
);
```

The dialog will change it’s behavior when running on a mobile or desktop device.

Let’s say we modify the application we created in the first recipe of this chapter. In our original application, a dialog was displayed when the user clicked on an image in a gallery. When on a mobile device, the dialog looked like you can see in *figure 6-4*.

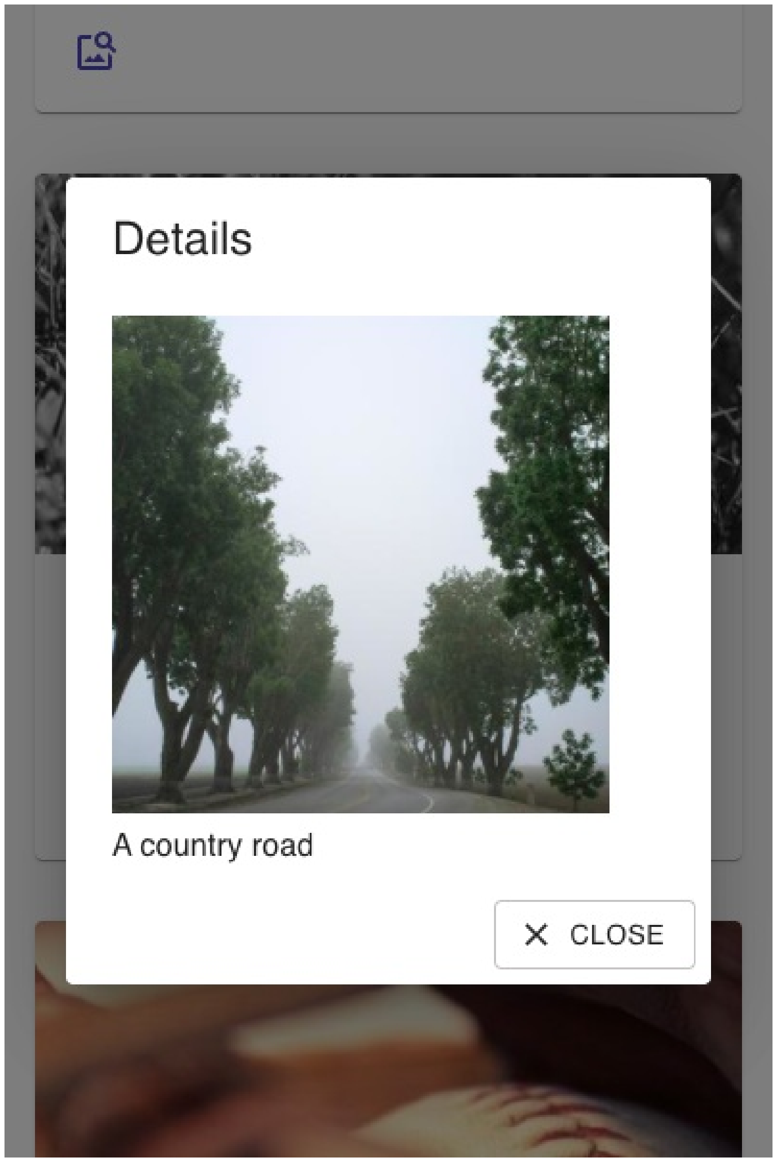


Figure 6-4. By default, a dialog on a mobile device has space around the edge.

If you replace this dialog with a `ResponsiveDialog`, it will look exactly the same on a large screen. But on a small screen, the dialog will completely fill the display, as you can see in *figure 6-5*. This not only gives you more space for the contents of the dialog, but it will also give the application a simpler appearance for mobile users. Instead of it working like a popup window, it will feel more like a separate page.

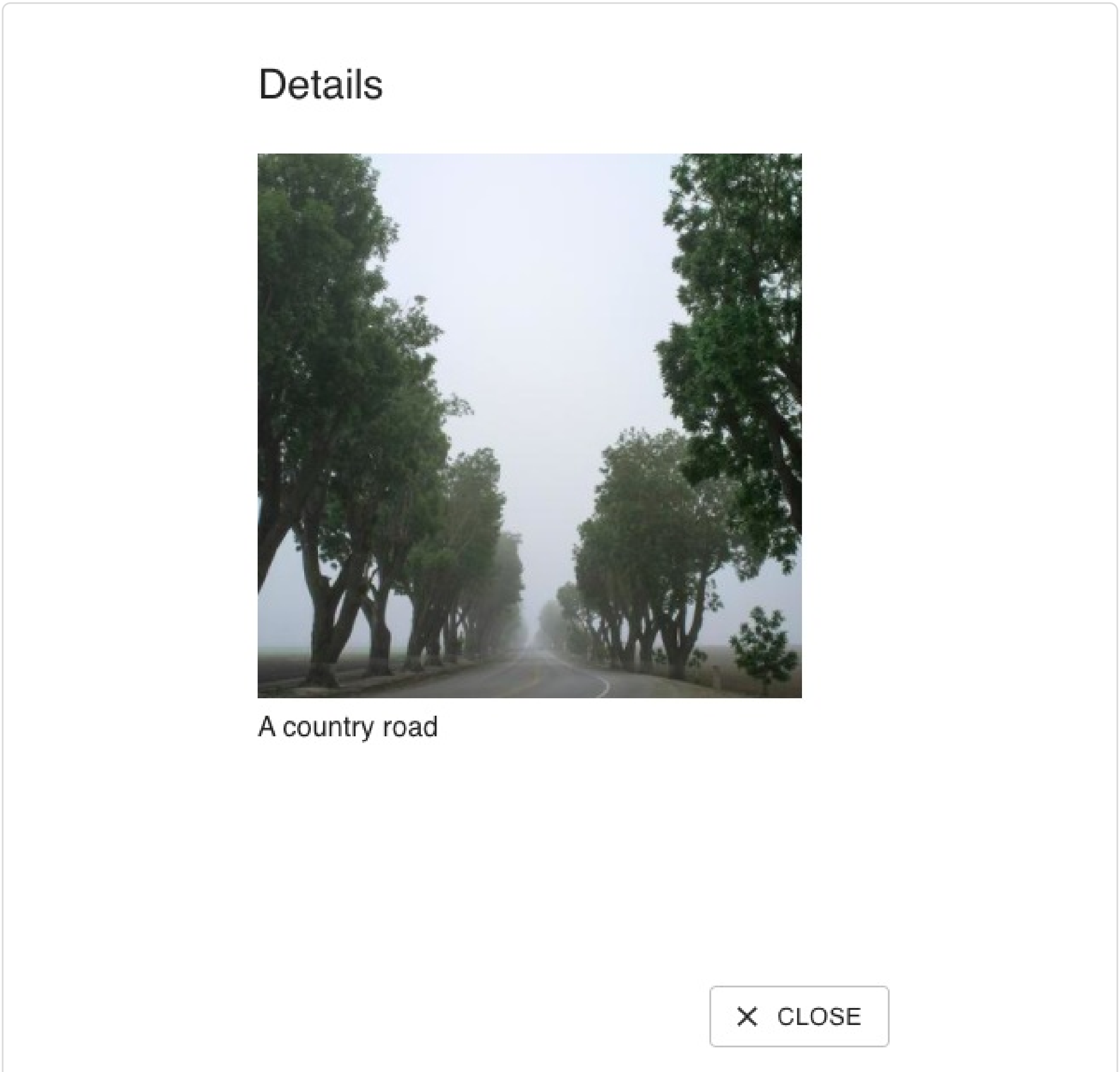


Figure 6-5. On a mobile device, the responsive dialog fills the screen.

Discussion

For more ideas on how to deal with responsive interfaces, see also the recipe for responsive routes in chapter 2 of this book.

You can download the source for this recipe from the Github site.

6.5 Build an Admin Console with React Admin

Problem

Developers can spend so long creating and maintaining end-user applications, that one important task is often left neglected: admin consoles. Admin consoles are not used by customers; they are used by backoffice staff and administrators to look at the current data set and to investigate and resolve data issue in an application. Some data storage systems like Firebase have quite advanced admin consoles built in. But that’s not the case for most back-end services. Instead, developers often have to dig into data problems by directly accessing the databases, which themselves are often hidden several walls of cloud infrastructure.

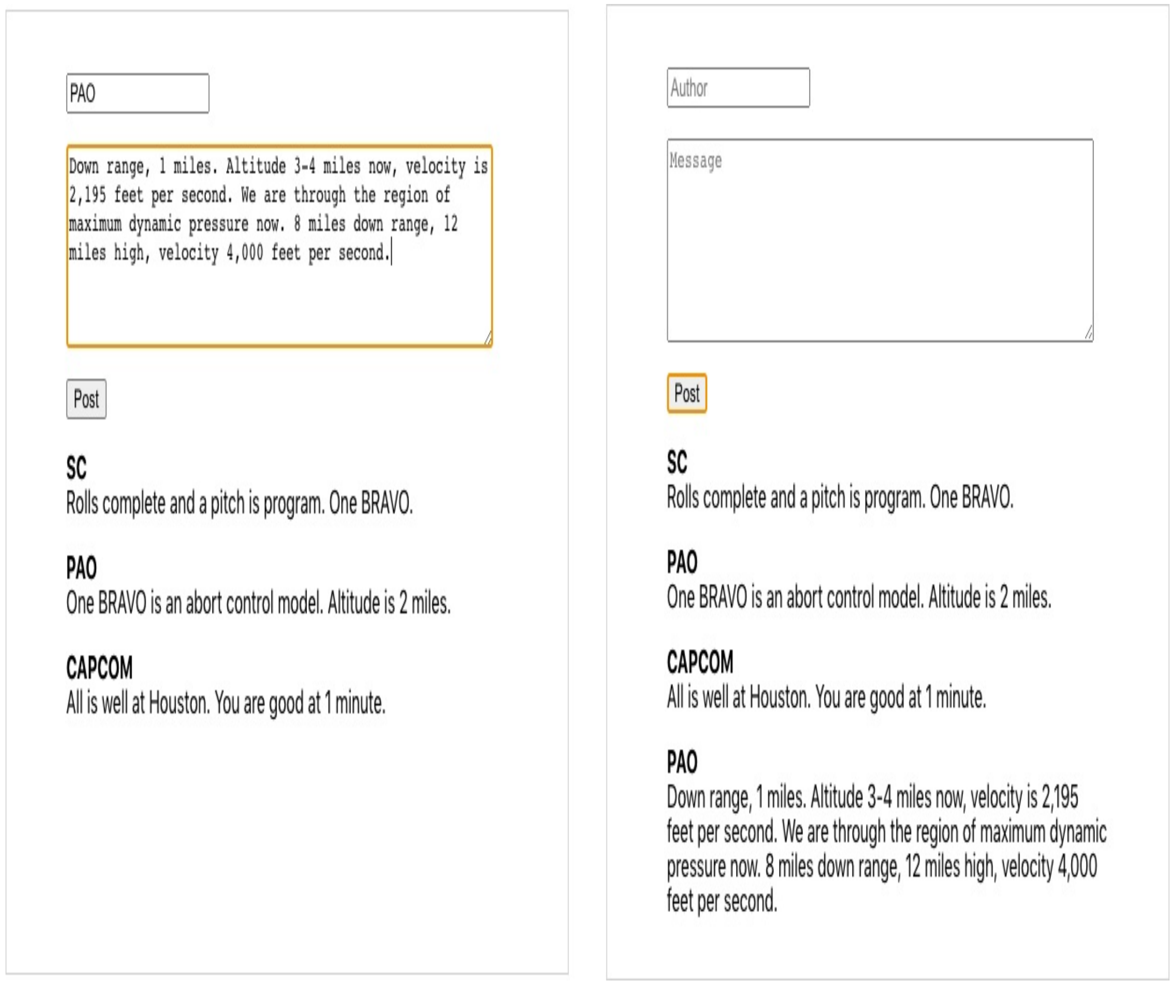
What is the fatest and simplest way to create an admin console for almost application using React?

Solution

We’re going to look at `react-admin`, and although this chapter is about component libraries, `react-admin` is actually far more. It’s really an application framework that makes it easy to build interfaces to allow administrators to examine and maintain the data in your application.

Different applications will use different network service layers. They might REST, or GraphQL, or one of many other systems. But in most cases data will accessed as a set of *resources* held on the server. `react-admin` has most of the pieces in place for creating an admin application that will allow you to browse through each resource. To search and filter the data within it. To admin, update, or delete data. Even to export it to an external application.

To show how `react-admin` works, we’re going to create an admin console for a message-board application we created in chapter 5 (see /figure 6-6).


PAO

Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

Post

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

Author

Message

Post

SC
Rolls complete and a pitch is program. One BRAVO.

PAO
One BRAVO is an abort control model. Altitude is 2 miles.

CAPCOM
All is well at Houston. You are good at 1 minute.

PAO
Down range, 1 miles. Altitude 3-4 miles now, velocity is 2,195 feet per second. We are through the region of maximum dynamic pressure now. 8 miles down range, 12 miles high, velocity 4,000 feet per second.

Figure 6-6. The original message board application.

The back-end for the application was a simple GraphQL server. The GraphQL server had a fairly simple scheme. Messages were defined in the schema language like this:

```
type Message {
    id: ID!
    author: String!
    text: String!
}
```

Each message had a unique `id`. The text of the message and the name of the author were stored as strings.

There was only one type of change that a user could make to the data: they could add a message. There was one type of query they could run: they could read all of the messages.

To create a `react-admin` application, you first need to create a new React application, and then install the `react-admin` library:

```
$ npm install react-admin
```

The main component of the library is called `Admin`. This will form the shell of our entire application:

```
<Admin dataProvider={...}>
  ...UI for separate resources goes here...
</Admin>
```

An `Admin` component needs a *data provider*. A data provider is an adapter which will connect the application to the back end service. Our back-end service is written in GraphQL, so we will need a GraphQL data provider:

```
$ npm install ra-data-graphql-simple
```

There are data providers available for most back-end services. See the React Admin web site for more details. We'll need to initialize our data provider before we can use it. The GraphQL is configured with a `buildGraphQLProvider` function that is asynchronous, so we need to be careful that it's ready before we use it:

```
import {Admin} from "react-admin";
import buildGraphQLProvider from 'ra-data-graphql-simple';
import {useEffect, useState} from "react";

function App() {
    const [dataProvider, setDataProvider] = useState();

    useEffect(() => {
        (async () => {
            const dp = await buildGraphQLProvider({
                clientOptions: {uri: 'http://localhost:5000'}
            });
            setDataProvider(() => dp);
        })();
    }, [])
    return (
        <div className="App">
            {
                dataProvider &&
                <Admin dataProvider={dataProvider}>
                   ...resource UI here...
                </Admin>
            }
        </div>
    );
}

export default App;
```

The data provider connects to our GraphQL server running on port 5000[6]. The data provider will first download the scheme for the application. This will tell it what resources (just a single resource `Messages` in our case) are available, and what things can be done to them.

If we try to run the application now, it won't do anything. That's because even though it knows that there's a `Messages` resources on the server, it doesn't know that we want it to do anything with it. SO let's add the `Messages` resource to the app.

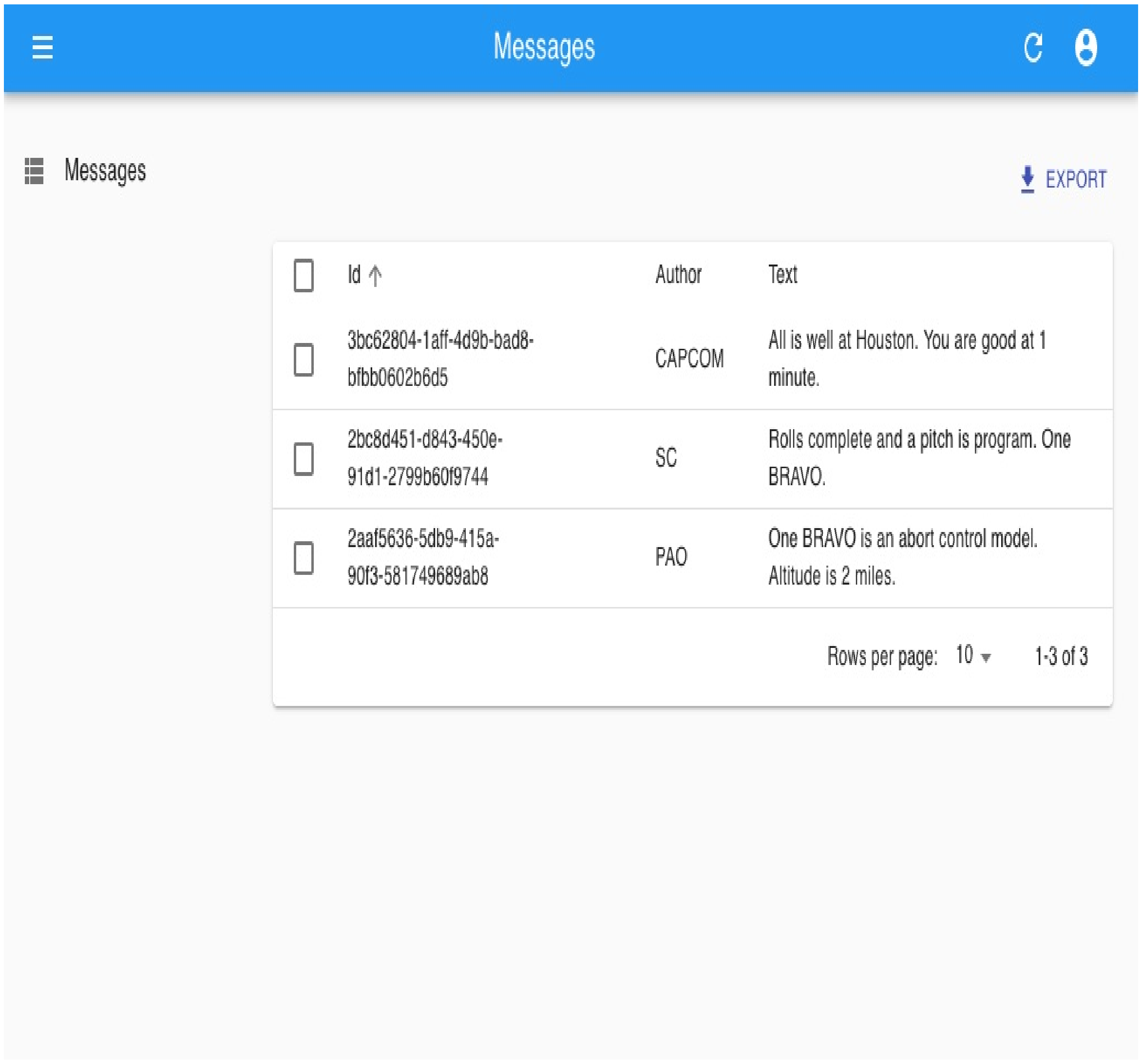
If we want the application to list all of the messages on the server, we will need to create a simple component called `ListMessages`. This will use some of the ready-components in `react-admin` to build its interface:

```
const ListMessages = (props) => {
    return <List {...props}>
        <Datagrid>
            <TextField source="id"/>
            <TextField source="author"/>
            <TextField source="text"/>
        </Datagrid>
    </List>;
};
```

This will create a table with columns for message `id`, `author` and `text`. We can now pass tell the admin system about the new component by passing a `Resource` to the `Admin` component:

```
<Admin dataProvider={dataProvider}>
    <Resource name="Message" list={ListMessages}/>
</Admin>
```

The `Admin` component will see the new `Resource`, and will contact the server to read the messages from the server, and then render them with a `ListMessages` component (see *figure 6-7*.)



Id	Author	Text
3bc62804-1aff-4d9b-bad8-bfbb0602b6d5	CAPCOM	All is well at Houston. You are good at 1 minute.
2bc8d451-d843-450e-91d1-2799b60f9744	SC	Rolls complete and a pitch is program. One BRAVO.
2aaf5636-5db9-415a-90f3-581749689ab8	PAO	One BRAVO is an abort control model. Altitude is 2 miles.

Figure 6-7. Displaying the messages from the server.

This appears to work by magic. Actually, it’s because the server has to follow certain conventions so that the GraphQL adapter knows which service to call. In this case, it will find a query called `allMessages` which returns messaages:

```
type Query {
    Message(id: ID!): Message
    allMessages(page: Int, perPage: Int,
        sortField: String, sortOrder: String,
        filter: MessageFilter): [Message]
}
```

This might mean that you have to slightly change your back-end API to meet the requirements of your data provider, the services that you add, will probably be useful to your main application.

Notice that the `allMessages` query accepts a set of parameters which allow the `Admin` interface to page through the data from you server. It also can pass a property called `filter` which can be used to search or narrow-down the data to download. The `MessageFilter` in the example scheme will allow the admin console to find messages containing strings for `author` and `text`. It will also allow the admin console to send a general search string (`q`) which can be used to find messages that contain a string in any field. This is the GraphQL schema definition of the `MessageFilter` object. You would obviously need to create something similar for each resource in your application:

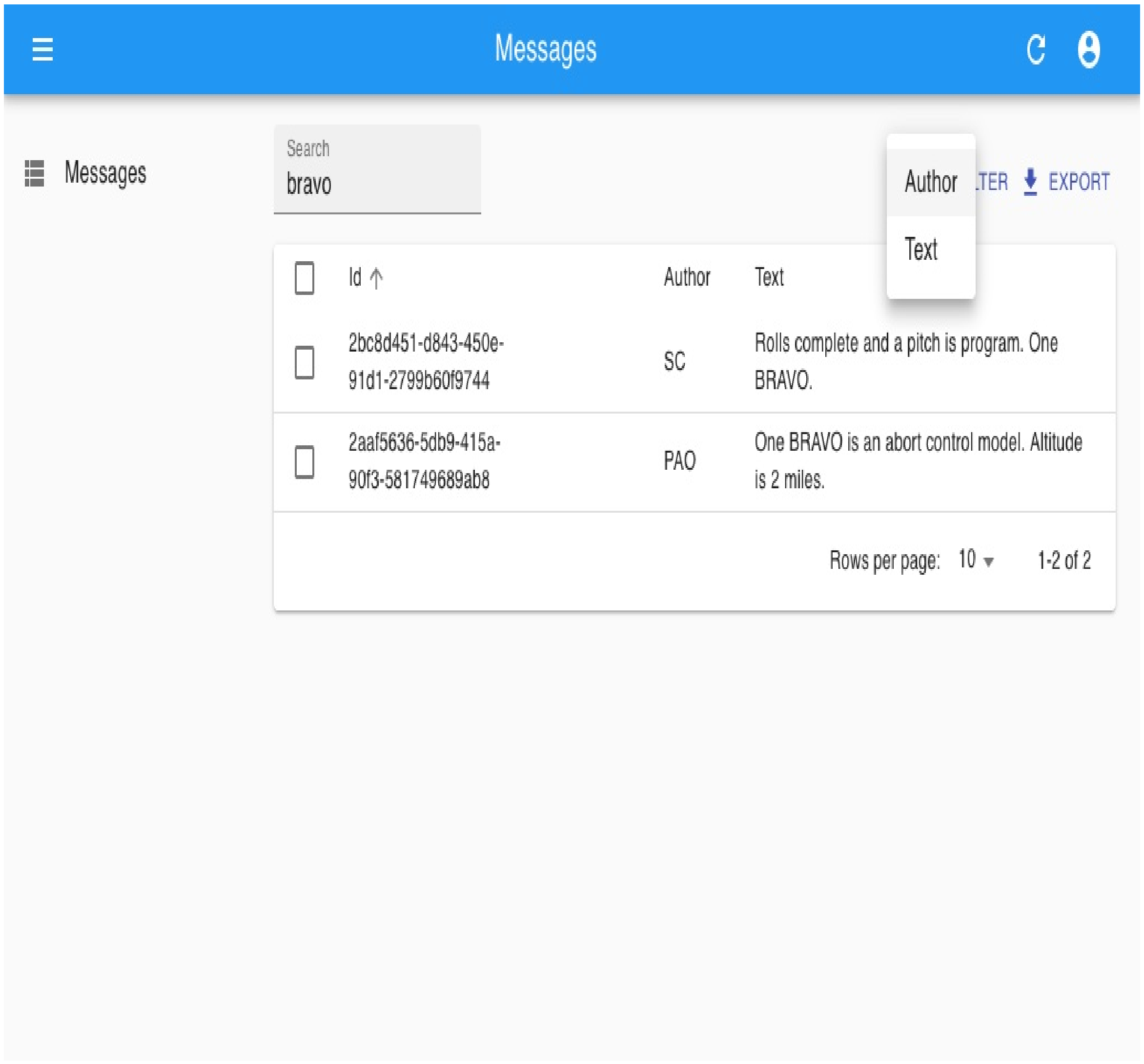
```
input MessageFilter {
    q: String
    author: String
    text: String
}
```

If we want to enable filtering and searching in the front-end, we will first need to create some filtering fields in a React component we'll call `MessageFilter`. This is quite distinct from the `MessageFilter` in the schema, although you will notice it contains matching fields:

```
const MessageFilter = (props) => (
    <Filter {...props}>
        <TextInput label="Author" source="author" />
        <TextInput label="Text" source="text" />
        <TextInput label="Search" source="q" alwaysOn />
    </Filter>
);
```

We can now add the `MessageFilter` to the `ListMessages` component, and we will suddenly find that we can page, search and filter the information in the admin console (see *figure 6-8*):

```
const ListMessages = (props) => {
    return <List {...props} filters={<MessageFilter/>}>
        <Datagrid>
            <TextField source="id"/>
            <TextField source="author"/>
            <TextField source="text"/>
        </Datagrid>
    </List>;
};
```



Id	Author	Text
2bc8d451-d843-450e-91d1-2799b60f9744	SC	Rolls complete and a pitch is program. One BRAVO.
2aaf5636-5db9-415a-90f3-581749689ab8	PAO	One BRAVO is an abort control model. Altitude is 2 miles.

Rows per page: 10 1-2 of 2

Figure 6-8. Filtering the messages table.

We can also add the ability to create a new messages, by adding a `CreateMessage` component:

```
const CreateMessage = (props) => {
    return <Create title="Create a Message" {...props}>
        <SimpleForm>
            <TextInput source="author" />
            <TextInput multiline source="text" />
        </SimpleForm>
    </Create>
};
```

And then add the `CreateMessage` component to the `Resource` (see *figure 6-9*):

```
<Resource name="Message" list={ListMessages} create={CreateMessage}/>
```

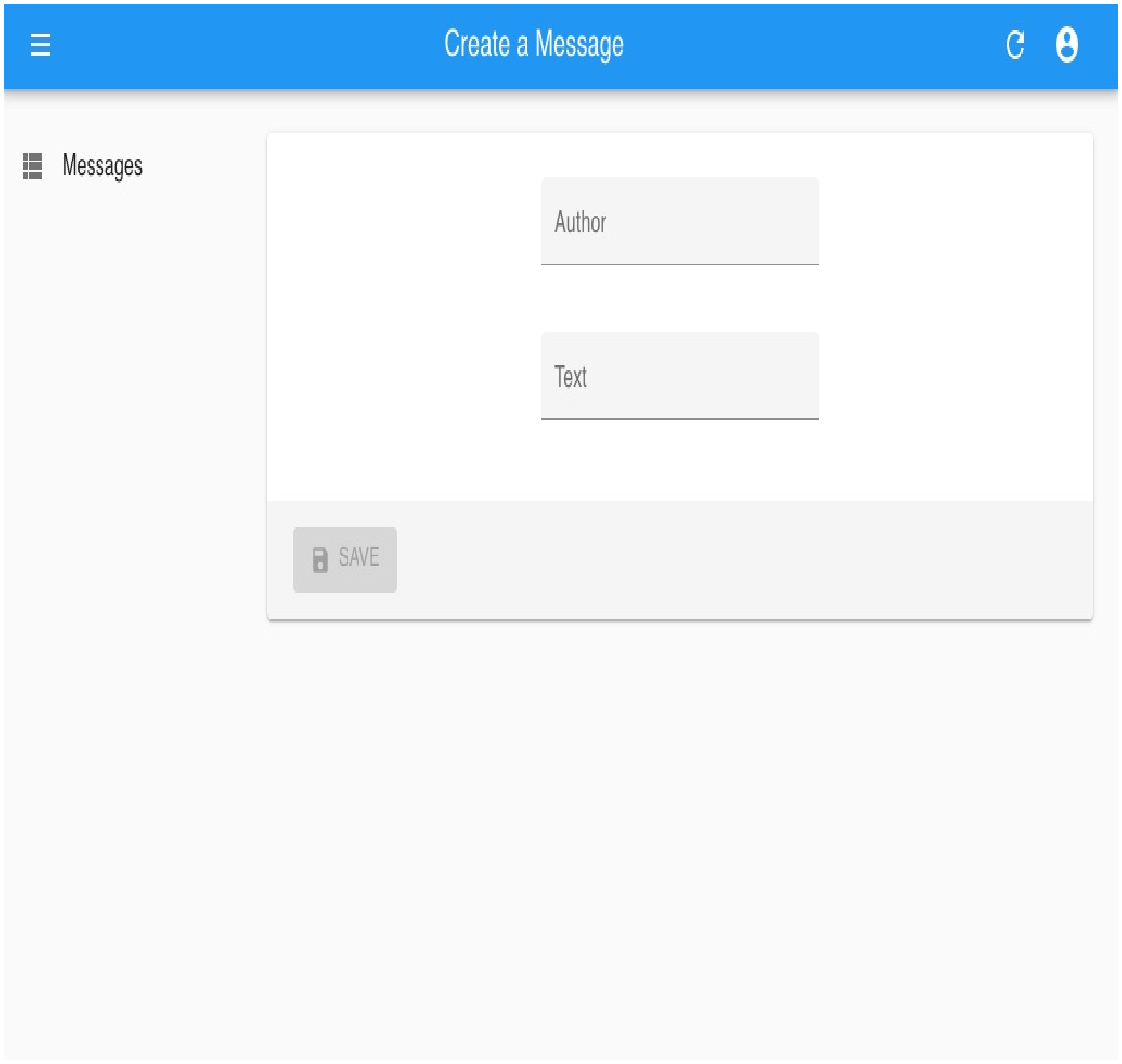
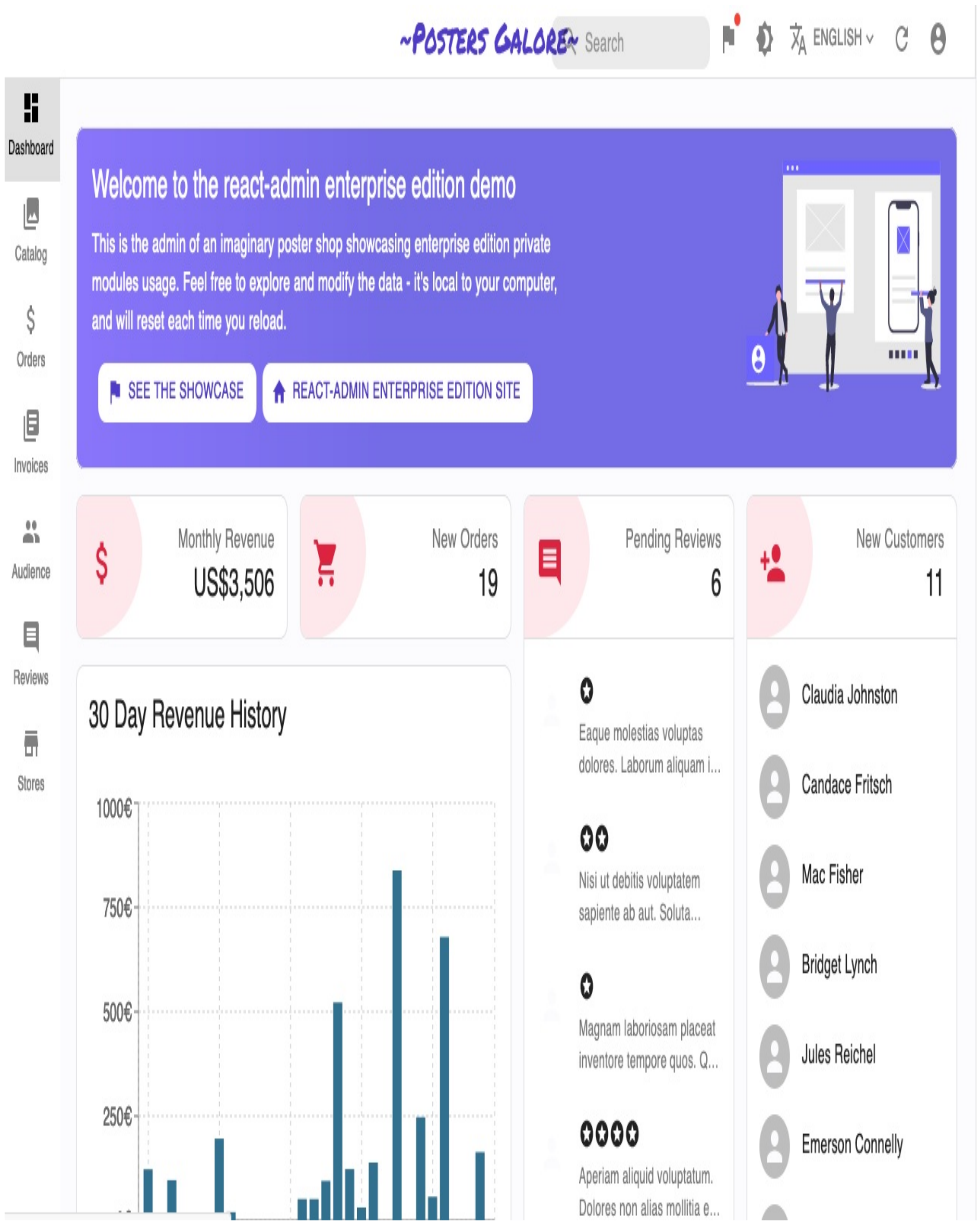


Figure 6-9. Creating messages on the console.

The GraphQL data provider will create messages by passing the contents of the `CreateMessage` form to a mutation called `CreateMessage`:

```
type Mutation {
    createMessage(
        author: String!
        text: String!
    ): Message
}
```

In a similar way, you can add the ability to update or delete messages. If you have a complex schema with sub-resources, `react-admin` has the ability to display sub-items within a table. It can also handle different display types. It can show images and links. There are components available[7] which can display resources on calendars, or in charts (see *figure 6-10* for examples from the online demo application). Admin consoles can also be made to work with your existing security model.



Welcome to the react-admin enterprise edition demo

This is the admin of an imaginary poster shop showcasing enterprise edition private modules usage. Feel free to explore and modify the data - it's local to your computer, and will reset each time you reload.

Eaque molestias voluptas dolores. Laborum aliquam i...

Nisi ut debitis voluptatem sapiente ab aut. Soluta...

Magnam laboriosam placeat inventore tempore quos. Q...

Aperiam aliquid voluptatum. Dolores non alias mollitia e...

Figure 6-10. Different view types in the online demo.

Discussion

Whilst you will have to make some additional changes to your back-end services to make `react-admin` work for you, there is a very good cahnce that this additional services will prove useful elsewhere in your main application. Even if they aren't, the building blocks that `react-admin` provides will likely slash the development time needed to create a back office system.

You can download the source for this recipe from the Github site.

6.6 No Designer? Use Semantic UI

Problem

Well-designed styling can add a lot of visual appeal to an application. But poor styling can make even a good application appear cheap and amateurish. Many developers[8] have very limited design sense. In cases where you have little or no access to professional UX help, a simple, clear UI component library can allow you to focus on the functionality of the application, without spending endless hours tweaking the location of buttons and borders.

Tried-and-tested frameworks like Bootstrap[9] can provide a good, no-glass, foundation for most applications. But even they often require a lot of focus on visual appearance. If you really want to focus on the functionaliuty of an application, and want to get a clear functional visual appearance then the *Semantic UI* library is a good choice.

But the *Semantic UI* library is old. It was written in the days when jQuery ruled the roost. At the time of writing[10] it has not been updated in over two years. What do you do if you want to use a reliable and well established library like Semantic UI with React.

Solution

The *Semantic-UI-React* library is a wrapper that makes the Semantic UI library available for React users.

As the name suggests, Semantic UI focuses on the *meaning* of the interface. There are very few components that describe the visual appearance; that is left to tweaking with CSS. Instead, Smenatic UI components focus on the functionality.

When you are creating a form, for example, you say which fields to include, rather than saying anything about their layout. That leads a to very clean, consistent appearance, which needs little or no visual adjustment.

To get started, let's install the Semantic library and its styling support:

```
$ npm install semantic-ui-react semantic-ui-css
```

In addition, we also need to include a reference to the stylesheet in the `index.js` of the application:

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

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

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

We're going to re-create our message posting application. We'll need a form with a text field for the author's name, and a text-area for posting a message. Semantic components are designed to be as similar to simple HTML elements as possible. So if we're building a form, we'll import a `Form`, `Input`, `TextArea`, and a `Button` to post the message:

```
import {Button, Form, Input, TextArea} from 'semantic-ui-react'
import './App.css';
import {useState} from "react";

function App() {
    const [author, setAuthor] = useState('');
    const [text, setText] = useState('');

    return (
        <div className="App">
            <Form>
                <Form.Field>
                    <label htmlFor='author'>Author</label>
                    <Input value={author} id='author' onChange={evt =>
setAuthor(evt.target.value)}/>
                </Form.Field>
                <Form.Field>
                    <label htmlFor='text'>Message</label>
                    <TextArea value={text} id='text' onChange={evt =>
setText(evt.target.value)}/>
                </Form.Field>
                <Button basic
                        onClick={() => {
                            setMessages(m => [{
                                icon: 'pencil',
                                date: new Date().toString(),
                                summary: author,
                                extraText: text
                            }, ...m]);
                            setAuthor('');
                            setText('');
                        }}
                >
                    Post
                </Button>
            </Form>
        </div>
    );
}

export default App;
```

This should feel very familiar. The `Form` component does have a `Field` helper which makes it a little easier to group labels and fields together, but beyond that the code looks very similar to a very simple HTML form.

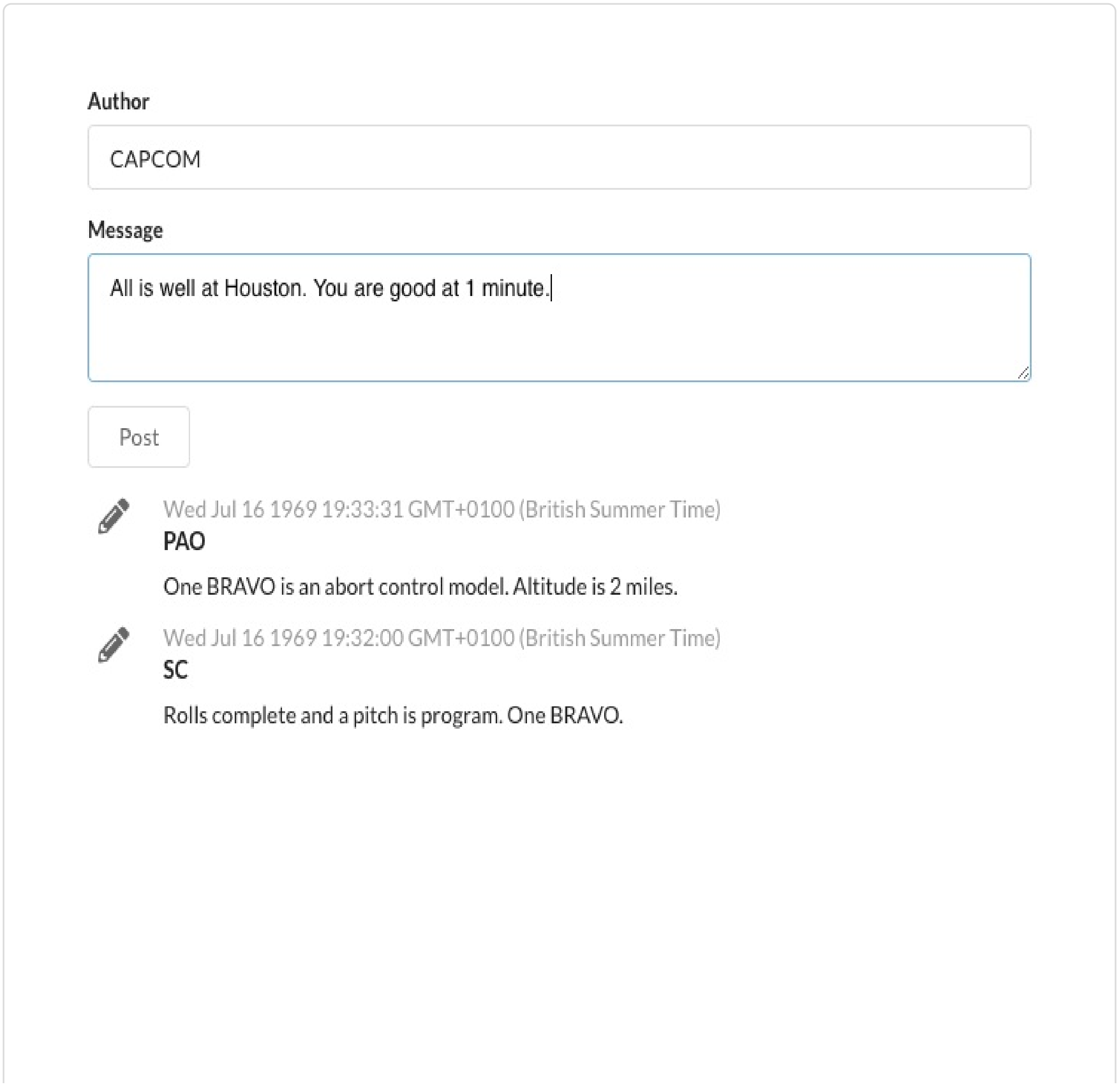
In the example application, we're "posting" messages by adding them to array called `messages`. You may have noticed that we're adding messages to the array in a very particular object structure:

```
setMessages(m => [{
    icon: 'pencil',
    date: new Date().toString(),
        summary: author,
        extraText: text
    }, ...m]);
```

This is not by accident. Although most of the components in Semantic are very simple, there are some more complex examples, which are there to support some common use-cases. One such example of the `Feed` component. The `Feed` component is there to render any kind of social message stream, such as you might see on Twitter or Instagram. It will render a clean series of messages, with date-stamps, headlines, icons and so on. This kind of visual component if very easy to build badly: i.e. in a way that works, but just looks ugly. Here's what our final code looks like with the `Feed` included:

```
import {Button, Form, Input, TextArea, Feed} from 'semantic-ui-react'
import './App.css';
import {useState} from "react";

function App() {
    const [author, setAuthor] = useState('');
    const [text, setText] = useState('');
    const [messages, setMessages] = useState([]);

    return (
        <div className="App">
            <Form>
                <Form.Field>
                    <label htmlFor='author'>Author</label>
                    <Input value={author} id='author' onChange={evt =>
setAuthor(evt.target.value)}/>
                </Form.Field>
                <Form.Field>
                    <label htmlFor='text'>Message</label>
                    <TextArea value={text} id='text' onChange={evt =>
setText(evt.target.value)}/>
                </Form.Field>
                <Button basic
                        onClick={() => {
                            setMessages(m => [{
                                icon: 'pencil',
                                date: new Date().toString(),
                                summary: author,
                                extraText: text
                            }, ...m]);
                            setAuthor('');
                            setText('');
                        }}
                >
                    Post
                </Button>
            </Form>
            <Feed events={messages}/>
        </div>
    );
}

export default App;
```

When you run the application, the interface is clean and un-fussy (see *figure 6-11*.)



Author

CAPCOM

Message

All is well at Houston. You are good at 1 minute.

Wed Jul 16 1969 19:33:31 GMT+0100 (British Summer Time)
PAO
One BRAVO is an abort control model. Altitude is 2 miles.

Wed Jul 16 1969 19:32:00 GMT+0100 (British Summer Time)
SC
Rolls complete and a pitch is program. One BRAVO.

Figure 6-11. The Semantic UI interface in action

Discussion

Semantic UI is an old library. But that's not a bad thing. It's battle-tested interface is clean and functional and is one of the best ways of getting your application up and running without the support of a visual designer. It's particularly useful if you're creating a Lean Startup[11] and want to throw something together quickly to test if there is a market for your product.

You can download the source for this recipe from the Github site.

---

1 For full details of the entire component set, see *https://material-ui.com*

2 For more information on these components, see the Material-UI site.

3 See the first recipe in this chapter for more information.

4 Including, dealing with all the nasty edge cases that can occur when a window is resized.

5 The library can be used for variable and sized lists and grids. See the documentation for more details.

6 You will find the server in the source code for this chapter. You can run the server by typing "node ./server.js"

7 Some of them available only if you subscribe to the Enterprise edition.

8 Including at least one of the authors…

9 See earlier in this chapter for guidance on how to use Bootstrap with your application.

10 November, 2020.

11 For more details, see *The Lean Startup* published by *O'Reilly Media*.

Chapter 7. Security

In this chapter we look at various ways of securing your application.

7.1 Secure requests not routes

Problem

In chapter 2, recipe 6 showed how you can use React Router to create shared routes. That means that if the user tries to get to certain paths within your application, you can force them to submit a login form before they can see the contents of that page.

This is a good, fairly general approach to take when you are first building an application. However, some applications do not fall so easily into this static model of security. Some pages will be *secure*, and some will be *insecure*. But in many applications, security is not defined by the particular pages peope are on, but by the contents of the data they are looking at. Sometimes page contents are only available to users when they've been granted access to particular roles. Sometimes what was once secure data becomes insecure data. You can also have different views of data for people who are logged in, compared to people who are not.

All of these complexities are usually straightforward to define at the API level. But it's the kind of complexity that you don't want to have to reproduce in the logic of your front end client. For these reasons, the simple approach of marking some routes secure and others as insecure is not good enough.

Solution

If defining routes as secure or insecure is not sufficient for the security of your client, you might want to consider driving access to app by the security responses you receive from the back-end server.

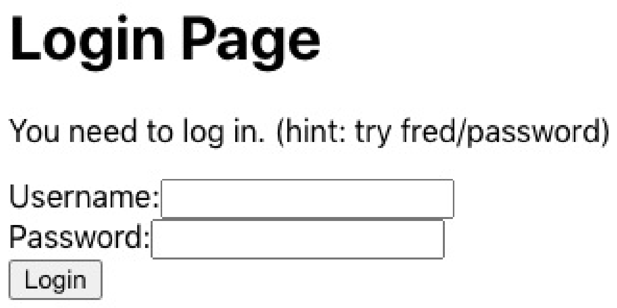
With this approach, you begin by assuming in the client can go anywhere in your app. You don’t worry about secure routes and insecure routes. You just have routes. If a user visits a path which contains private data, the server will return some kind of errors response, typically a HTTP status 401, and in the event of that error, the user is redirected to a login form.

With this approach, the policy of which data is private and which data is public is driven by the server. If the security policies change, you only need to modify the code on the server, without needing to change the client code.

Let’s take a look at the code for the original secured-routes recipe again. In our application, we inject a `SecurityProvider`, which controls the security of all of its child components. In the example application we do this is the `App.js` file:

```
import './App.css';
import {BrowserRouter, Route, Switch} from "react-router-dom";
import Public from "./Public";
import Private1 from "./Private1";
import Private2 from "./Private2";
import Home from "./Home";
import SecurityProvider from "./SecurityProvider";
import SecureRoute from "./SecureRoute";

function App() {
    return (
        <div className="App">
            <BrowserRouter>
                <SecurityProvider>
                    <Switch>
                        <Route exact path='/'>
                            <Home/>
                        </Route>
                        <SecureRoute path='/private1'>
                            <Private1/>
                        </SecureRoute>
                        <SecureRoute path='/private2'>
                            <Private2/>
                        </SecureRoute>
                        <Route exact path='/public'>
                            <Public/>
                        </Route>
                    </Switch>
                </SecurityProvider>
            </BrowserRouter>
        </div>
    );
}

export default App;
```

You can see that the application has simple `Routes` and `SecuredRoutes`. If an un-logged in user tries to access a secured route, they are redirected to a login, as you can see in *figure 7-1*



Login Page

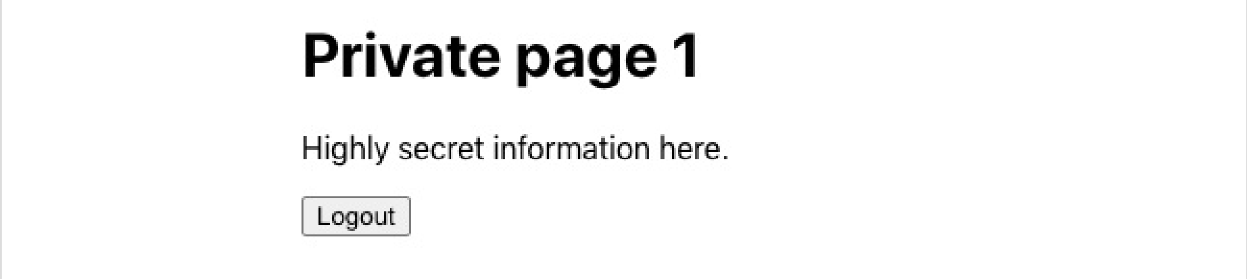
You need to log in. (hint: try fred/password)

Username:

Password:

Figure 7-1. When you first access a secured route, you see a login form

Once they are logged in (see *figure 7-2*) they then have access to the secured content.


Private page 1

Highly secret information here.

Figure 7-2. Once you are logged in, secured routes are visible

If we want to base our security upon the security of the back-end API, we'll begin by replacing all of the `SecuredRoutes` with simple `Routes`. The application simply doesn't know, until it's informed by the API, which data is private and which data is public. For the example app in this recipe, we'll have two pages on the application that contain a mix of public and private data. The *Transactions* will read secure data from the server. The *Offers* page will read insecure data from the server. This is what the new version of our `App.js` file will look like:

```
import './App.css';
import {BrowserRouter, Route, Switch} from "react-router-dom";
import Transactions from "./Transactions";
import Offers from "./Offers";
import Home from "./Home";
import SecurityProvider from "./SecurityProvider";

function App() {
    return (
        <div className="App">
            <BrowserRouter>
                <SecurityProvider>
                    <Switch>
                        <Route exact path='/'>
                            <Home/>
                        </Route>
                        <Route exact path='/transactions'>
                            <Transactions/>
                        </Route>
                        <Route exact path='/offers'>
                            <Offers/>
                        </Route>
                    </Switch>
                </SecurityProvider>
            </BrowserRouter>
        </div>
    );
}

export default App;
```

We'll also need to make a change to our `SecurityProvider`. In an API-security model, the client begins by assuming that all data is public. This is the opposite of the secured-routes approach, which assumes you don't have access until you prove that you do by logging in.

This means our new `SecurityProvider` has to default its initial logged-in state to `true`:

```
import {useState} from "react";
import SecurityContext from "./SecurityContext";
import Login from "./Login";
import axios from 'axios';

export default (props) => {
    const [loggedIn, setLoggedIn] = useState(true);

    return <SecurityContext.Provider
        value={{
            login: async (username, password) => {
                await axios.post('/api/login', {username, password});
                setLoggedIn(true);
            },
            logout: async () => {
                await axios.post('/api/logout');
                return setLoggedIn(false);
            },
            onFailure() {
                return setLoggedIn(false);
            },
            loggedIn
        }}>
        {loggedIn ? props.children : <Login/>}
    </SecurityContext.Provider>
};
```

We’ve also made several other changes. First, the code which decides whether the user should see the `Login` form is now in the `SecurityProvider`. This used to live inside the `SecuredRoute` component, but now we display it centrally. Second, we’ve replace the dummy username/password checks with calls to the back-end services called _*api/login* and //api/logout_. You should replace these with whatever security code is applicable to your system. Finally, the `SecurityProvider` now provides a new function called `onFailure`, which simply marks the person as logged-out. If this function is called, the user will be forced to log in.

If we no longer have secured-routes, at what point do we perform the security checks? We don’t them in the API calls themselves.

WARNING

For a real implementation, you would want to add code the deals with an invalid login attempt. To keep the code short, we’ve omitted any special handling here. In the example application, a failed login will simply leave you in the login form, without any error messages.

Let’s take a look at our new *Transactions* page, as defined in the `src/Transactions.js`. This component reads the transactions data and displays it on the screen:

```
import useTransactions from "./useTransactions";

export default () => {
    const {data: transactions} = useTransactions();

    return <div>
        <h1>Transactions</h1>
        <main>
            <table>
                <thead>
                <tr>
                    <th>Date</th>
                    <th>Amount</th>
                    <th>Description</th>
                </tr>
                </thead>
                <tbody>
                {
                    transactions && transactions.map(trx => <tr>
                        <td>{trx.date}</td>
                        <td>{trx.amount}</td>
                        <td>{trx.description}</td>
                    </tr>)
                }
                </tbody>
            </table>
        </main>
    </div>;
};
```

The network code that reads the data from the server is hidden inside the `useTransactions` hook. It's inside this hook that we need to add our check for a failed-access response from the server:

```
import {useEffect, useState} from 'react';
import axios from 'axios';
import useSecurity from "./useSecurity";

const useTransactions = () => {
    const security = useSecurity();
    const [transactions, setTransactions] = useState([]);

    useEffect(() => {
        (async () => {
            try {
                const result = await axios.get('/api/transactions');
                setTransactions(result.data);
            } catch (err) {
                const status = err.response && err.response.status;
                if (status === 401) {
                    security.onFailure();
                }
                // Handle other exceptions here (consider a shared
                // error handler -- see elsewhere in the book)
            }
        })();
    }, []);

    return {data: transactions};
};

export default useTransactions;
```

In the example application, we're using the `axios` library to contact the server.

`axios` handles HTTP errors such as `401` (the HTTP status for *Unauthorized*) as exceptions. That makes it a little clearer which code is dealing with an unexpected response. If you were using a different API standard, like GraphQL, you would be able to deal with security errors in an analogous way, by examining the contents of the error object that GraphQL returns.

In the event that there's an *Unauthorized* response from the server, the `useTransactions` hook makes a call to the `onFailure` function in the `SecurityProvider`.

We'll build the *Offers* page in exactly the same way. The `src/Offers.js` component will format the `offers` data from the server:

```
import useOffers from "./useOffers";

export default () => {
    const {data: offers} = useOffers();

    return <div>
        <h1>Offers</h1>
        <main>
            <ul>
                {
                    offers && offers.map(offer => <li
className='offer'>{offer}</li>)
                }
            </ul>
        </main>
    </div>;
};
```

And the code that reads the data is inside the `src/useOffers.js` hook:

```
import {useEffect, useState} from 'react';
import axios from 'axios';
import useSecurity from "./useSecurity";

const useOffers = () => {
    const security = useSecurity();
    const [offers, setOffers] = useState([]);

    useEffect(() => {
        (async () => {
            try {
                const result = await axios.get('/api/offers');
                setOffers(result.data);
            } catch (err) {
                const status = err.response && err.response.status;
                if (status === 401) {
                    security.onFailure();
                }
                // Handle other exceptions here (consider a shared
                // error handler -- see elsewhere in the book)
            }
        })();
    }, []);

    return {data: offers};
};

export default useOffers;
```

NOTE

Even though the data from the //api/offers/ end-point is not secured, we still have code that checks for security errors. One consequence of the API security approach, is that you have to treat all end-points as if they are secure, just in case the become secure in the future.

OK, let's try our example application out. We'll begin by opening the front page (see *figure 7-3*.)

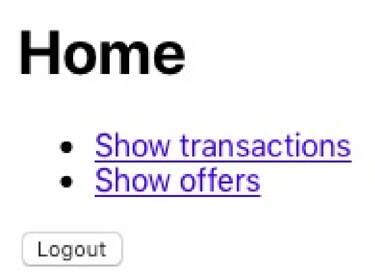
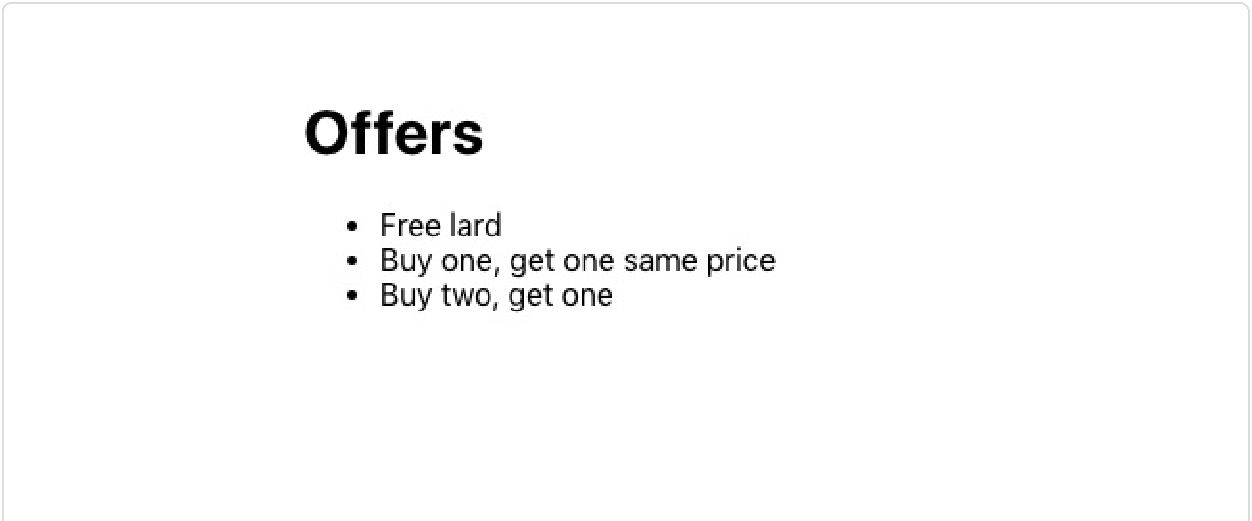


Figure 7-3. This is the front page of the application

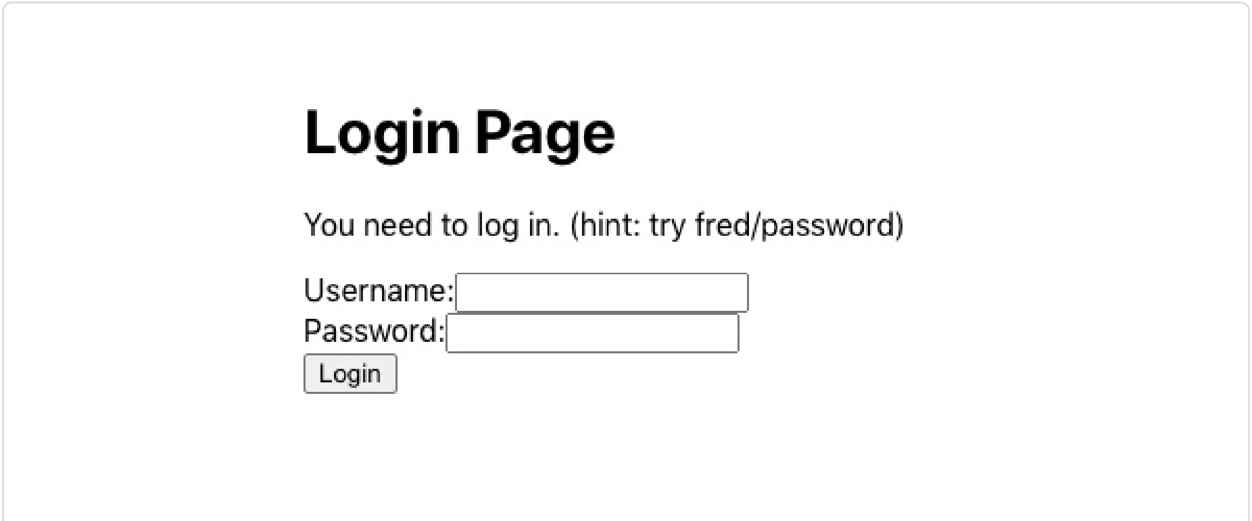
If we click on the *Offers* link, we see the offers read from the server (see *figure 7-4*). This data is unsecured, and we’ve not been asked to log in to see it.


Offers

- Free lard
- Buy one, get one same price
- Buy two, get one

Figure 7-4. If we click on the offers link, we can see the contents

If we now go back to the home page, and click on the *Transactions* link, we are immediately asked to log in (see *figure 7-5*). The transactions page has attempted to download transaction data from the server, which resulted in a 401 (*Unauthorized*) response. The code catches this as an exception, and calls the `onFailure` function in the `SecurityProvider`, which then displays the login form.

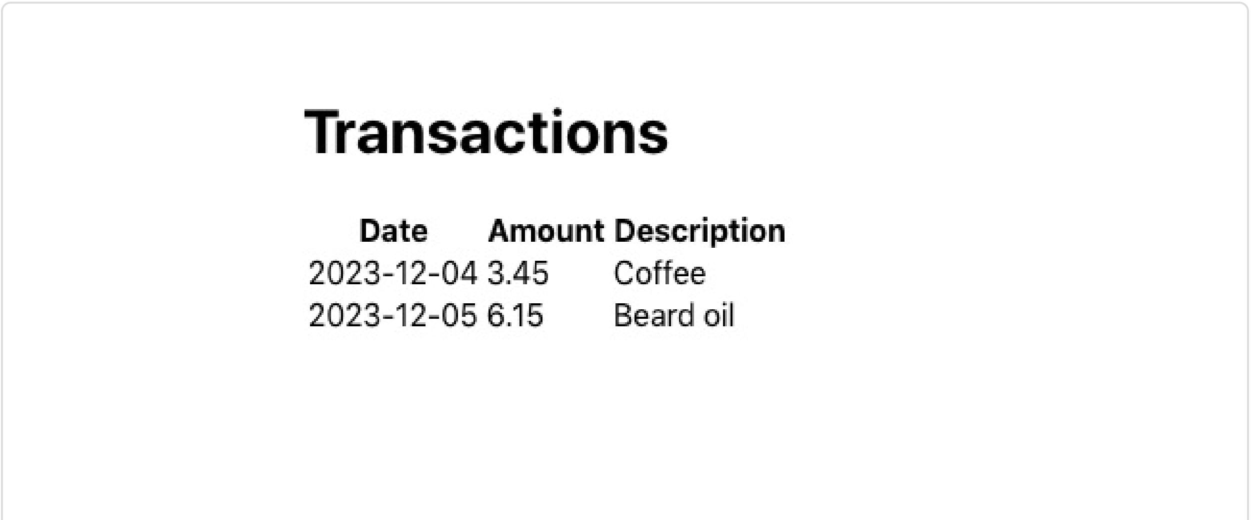

Login Page

You need to log in. (hint: try fred/password)

Username:
Password:

Figure 7-5. image

If we log in, our username and password are sent to the server. Assuming that doesn't result in an error, the `SecurityProvider` hides the login form, and the *Transactions* page is re-rendered and the data is now able to be read as we've logged in (see *figure 7-6*).


Transactions

Date	Amount	Description
2023-12-04	3.45	Coffee
2023-12-05	6.15	Beard oil

Figure 7-6. Once we are logged in, we can see the page

Discussion

Our example app now contains nothing to indicate which APIs are secured, and which are unsecured. All of that work is now handled by the server. The API endpoints are entirely in charge of the security of the application.

Using this approach, you should apply the same security handling to all API calls. One of the benefits of extracting API calls into custom hooks, is that this code can be shared. Hooks can call other hooks, and a common approach is to create hooks which act as general purpose `GET` and `POST` calls[1]. A general-purpose `GET` hook could not only handle access failures, it can also include request cancellations, and debouncing (recipes 3 and 6 in chapter 5) as well as shared error handling (recipe 1 in chapter 4).

Another advantage to the secured-API approach is that it's possible to entirely disable security in some circumstances. For example, during development you could do away with the need for developers to have an identity provider configured. You can also choose to have different security configurations in different deployments.

Finally, for automated testing systems, like Cypress, which can simulated network responses, you can split the testing of application functionality from non-functional security testing.

You can download the source for this recipe from the Github site.

7.2 Enable two factor authentication with physical tokens

Problem

Usernames and passwords are not always enough; they might be stolen or guessed. They might be shared online. Some applications may require additional levels of security to protect their data. Some users might only choose applications that provide additional security.

An increasing number of systems now provide *two-factor authentication*. A two-factor system requires the user to login with a form, and then provide some additional information. This might be a code sent to them by an SMS text message. Or an application on their phone which generates a pseudo-random key. Or, perhaps most securely, it might involve the use of a physical hardware device, like a Yubikey[2], which is attached to the computer when required and pressed.

These physical tokens work using public key cryptography. They can generate a public key for use with a given application. They can also encrypt strings using a private key. An application can send a random “challenge” string to the device, which will then generate a signature using the private key. The application can then use the public key to checked that the string signed correctly.

But how do you integrate them with you React application?

Solution

Web Authentication (also known as *WebAuthn*) is a widely[3] supported W3C standard that allows a browser to communicate with a physical device, like a Yubikey.

There are two *flows* in web authentication. The first is called *attestation*. During attestation, a user registers a security device with an application. During *assertion*, the user is able to verify their identity to log in to a system.

First, let's look at *attestation*. During this flow, the user registers a physical device against their account. That means that user should always be logged in during attestation.

The code for this recipe includes a dummy node server, which you can run from the //server/ directory within the application:

```
$ cd server
$ npm install
$ npm run start
```

There are three steps to attestation:

- The server generates an attestation request, saying what can kind of device is acceptable
- The user connects the device and activates it. Probably by pressing a button on it.
- A response is generated from the device which includes the public key, and is then returned to the server where it can be stored against the user's account

We can tell if the browser supports WebAuthn, by checking for the existence of `window.PublicKeyCredential`. If it exists, you're good to go.

On the server, there is an end-point at //startRegister/ which will create the attestation request for us. So we'll begin by calling that:

```
import axios from "axios";
...
// Ask to start registering a physical token for the current user
const response = await axios.post("/startRegister");
```

What does an attestation request look like? This is an example.

```
{
    "rpName": "Physical Token Server",
    "rpID": "localhost",
    "userID": "1234",
    "userName": "freda",
    "excludeCredentials": [
        {"id": "existingKey1", "type": "public-key"}
    ],
    "authenticatorSelection": {
        "userVerification": "discouraged"
    },
    "extensions": {
        "credProps": true,
    },
}
```

Some of the attributes begin with the letters *rp…*, which stands for *Relying party* and refers to the application that has generated the request.

The `rpName` is a free form text string that describes the application. You should set the `rpId` to the current domain name. Here it’s `localhost`, because we’re running on a development server. The `userID` is a string which uniquely identifies the user. The `userName` is the name of user.

`excludeCredentials` is an interesting attribute. Users might record multiple devices against their account. This value list the devices which are already recorded, to avoid the user registering the same device twice. If you attempt to register the same device more than once, the browser will immediately throw an exception saying that the device has been registered elsewhere.

The `authenticatorSelection` allows you to set various options about what the user needs to do when they activate their device. Here we’re setting `userVerification` to `false`, to prevent the user having to perform any additional steps (such as entering a pin number) when activating their device. This means that when asked to plug in their device, they will just insert it into the USB socket and press the button, with nothing else needed.

The `credProps` extension asks the device to return additional credential properties, which might be useful to the server.

Once the attestation request has been generated by the server, we need to ask the user to connect their security device. This is done with a browser function called

```
    navigator.credentials.create()
```

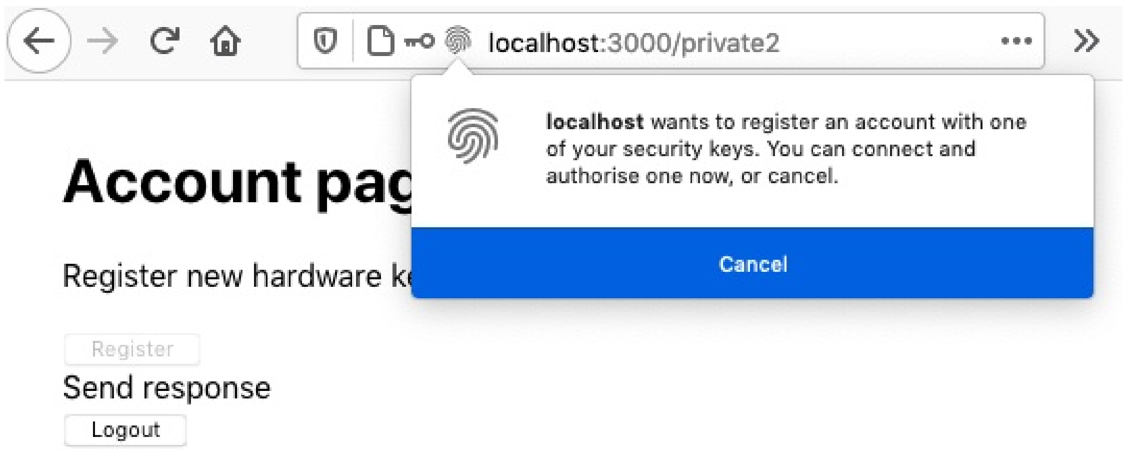
The `create` function accepts an attestation request object. Unfortunately, the data within the object needs to be in a variety of low-level binary forms, such as byte arrays. We can make out life significantly easier by installing a library from Github called `webauthn-json`, which will allows to use JSON to specify the request.

```
npm install "@github/webauthn-json"
```

We can then pass the contents of the WebAuthn request to the Github version of the `create` function

```
import {create} from "@github/webauthn-json";
import axios from "axios";
...
// Ask to start registering a physical token for the current user
const response = await axios.post("/startRegister");
// Pass the WebAuthn config to webauthn-json 'create' function
const attestation = await create({publicKey: response.data});
```

This is the point where the user will be asked to insert and activate their security device (see *figure 7-7*)



localhost wants to register an account with one of your security keys. You can connect and authorise one now, or cancel.

Figure 7-7. The browser asks for the token when create() is called

The `create` function resolves to an *attestation object*, which you can think of as the registration information for the device. The attestation object can be used by the server to verify the user’s identity when they log in. We need to record the attestation object against the user’s account. We’ll do that by posting it back to an endpoint on the example server at //register/

```
import {create} from "@github/webauthn-json";
import axios from "axios";
...
// Ask to start registering a physical token for the current user
const response = await axios.post("/startRegister");
// Pass the WebAuthn config to webauthn-json 'create' function
const attestation = await create({publicKey: response.data});
// Send the details of the physical YubiKey to be stored against the
user
const attestationResponse = await axios.post("/register",
{attestation});
```

That's the overview of how we register a new device for a user. But where do we put that in the code?

The example application has an *Account* page (see *figure 7-8*) and we'll add a button in there to register a new key

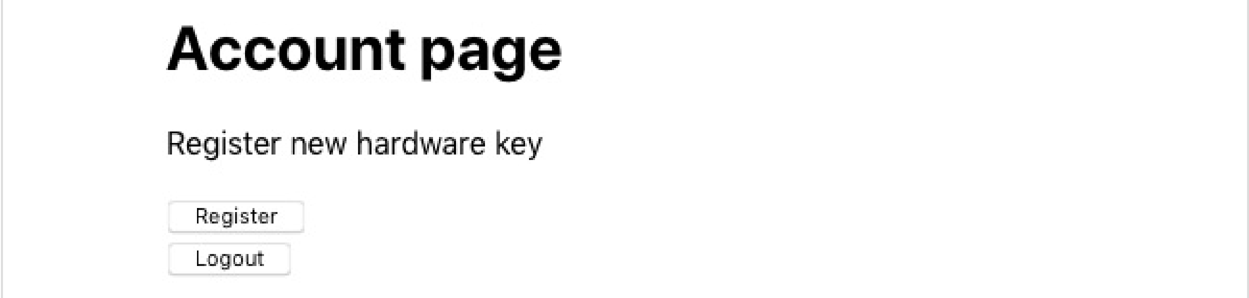
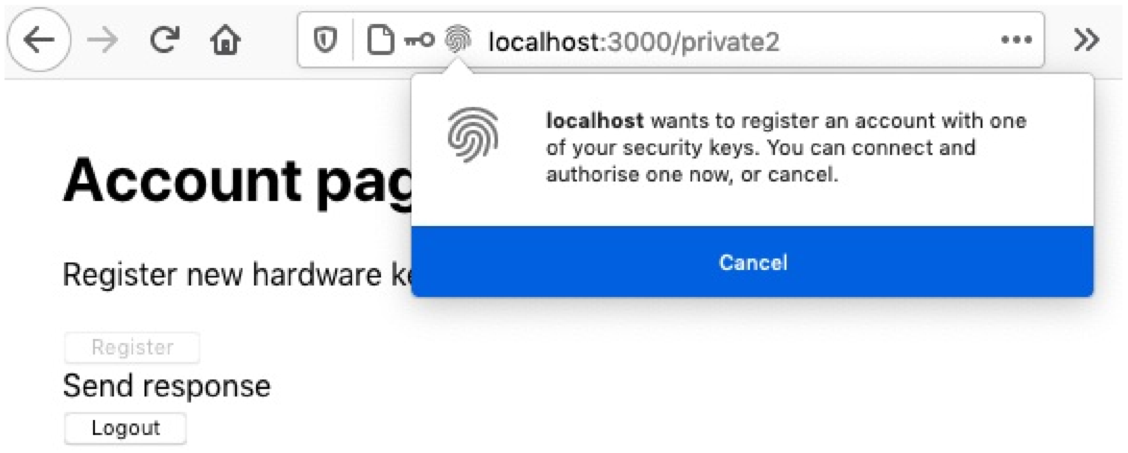


Figure 7-8. We'll add a button to the account page to register a new device

This is the registration code in place: .src/Private2.js

```
import {useState} from 'react';
import Logout from "./Logout";
import axios from "axios";
import {create} from "@github/webauthn-json";

const Private2 = () => {
    const [busy, setBusy] = useState(false);
    const [message, setMessage] = useState();

    return <div className='Private2'>
        <h1>Account page</h1>

        {
            window.PublicKeyCredential && <>
                <p>
                    Register new hardware key
                </p>
                <button onClick={async () => {
                    setBusy(true);
                    try {
                        const response = await
axios.post("/startRegister");
                        setMessage('Send response')
                        const attestation = await create({publicKey:
response.data});
                        setMessage('Create attestation')
                        const attestationResponse = await
axios.post("/register", {attestation});
                        setMessage('registered!')
                        if (attestationResponse.data &&
attestationResponse.data.verified) {
                            alert("New key registered");
                        }
                    } catch (err) {
                        setMessage('' + err);
                    } finally {
                        setBusy(false);
                    }
                }}
                        disabled={busy}
                >Register
                </button>
            </>
        }
        <div className='Account-message'>
            {message}
        </div>

        <Logout/>
    </div>;
};

export default Private2;
```

If we press the registration button on the account page, we’re asked to connect the security device (see *figure 7-9*), once we do that the device's credentials are sent to the server, and we're told that a new device has been recorded against our account (see *figure 7-10*.)


localhost:3000/private2

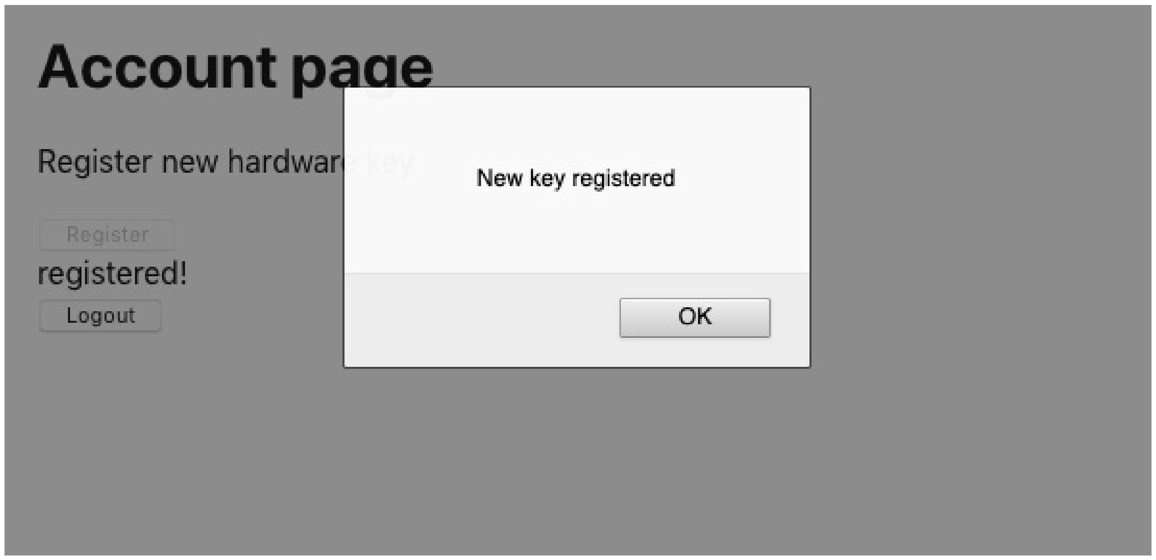
localhost wants to register an account with one of your security keys. You can connect and authorise one now, or cancel.

Account pag

Register new hardware k

Send response

Figure 7-9. When you choose to register a new device, you are asked to activate it


Account page

Register new hardware key

registered!

New key registered

Figure 7-10. We are told when a new device is registered

The next flow we need to think about is *assertion*. Assertion happens when a user verifies their identity when logging in.

The steps are quite similar to attestation. First, the application asks the server to create an *assertion request*. Second, the user converts that request into an *assertion* object, by activating their security device. Third, the server checks the assertion against its stored credentials, to prove the person is who they say they are.

Let’s begin with the first stage, when we create an assertion request. This is what an assertion request looks like:

```
{
    "allowCredentials": [
        {id: "existingTokenID", "type": "public-key"}
    ],
    "attestation": "direct",
    "extensions": {
        "credProps": true,
    },
    "rpID": "localhost",
    "timeout": 60000,
    "challenge": "someRandomString"
}
```

The `allowCredentials` attribute is an array of registered devices which will be acceptable. The browser will be able to use this array to check that the user has connected the correct device.

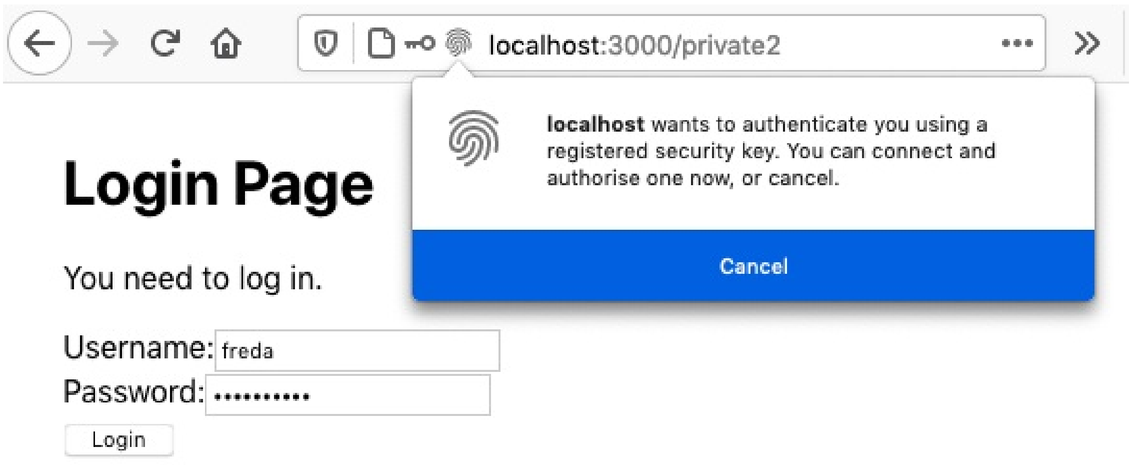
The assertion request also includes a `challenge` string. This is a randomly generated string which the device will need to create a signature for with its private key. The server will be able to check this signature with the public key, to ensure that the correct device was used. The `timeout` specifies how long the user will have to prove their identity.

The example server generates an assertion request when you call the //startVerify/ endpoint with a specified user-id

```
import axios from "axios";
...
// Ask for a challenge to verify user userID
const response = await axios.post("/startVerify", {userID});
```

We can then pass the assertion request to the `get()` WebAuthn function, which will ask the user to verify their identity by connecting an acceptable device (see /figure 7-11).

```
import {get} from "@github/webauthn-json"
import axios from "axios";
...
const response = await axios.post("/startVerify", {userID});
const assertion = await get({publicKey: response.data});
```



localhost wants to authenticate you using a registered security key. You can connect and authorise one now, or cancel.

Figure 7-11. image

The `get()` function returns an assertion object, which contains a signature for the challenge string. This is sent back to the server’s //verify/ endpoint, to check the signature. The response to that call will tell us if the user has correctly verified their identity.

```
import {get} from "@github/webauthn-json"
import axios from "axios";
...
    const response = await axios.post("/startVerify", {userID});
const assertion = await get({publicKey: response.data});
const resp2 = await axios.post("/verify", {userID, assertion});
if (resp2.data && resp2.data.verified) {
    // User is verified
}
```

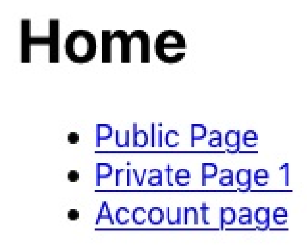
Where do we put this code in the application?

The example application is based on the secured-routes recipe[4]. It contains a `SecurityProvider`, which manages the security for all of its child components. The `SecurityProvider` provides a function called `login`, which is called with the username and password when the user submits a login form. We'll put the verification code in here.

```
import {useState} from "react";
import SecurityContext from "./SecurityContext";
import {get} from "@github/webauthn-json"
import axios from "axios";

const SecurityProvider = (props) => {
    const [loggedIn, setLoggedIn] = useState(false);

    return <SecurityContext.Provider
        value={{
            login: async (username, password) => {
                const response = await axios.post('/login', {username,
password});
                const {data} = response;
                if (data.twoFactorNeeded) {
                    const userID = data.userID;
                    const response = await axios.post("/startVerify",
{userID});
                    const assertion = await get({publicKey:
response.data});
                    const resp2 = await axios.post("/verify", {userID,
assertion});
                    if (resp2.data && resp2.data.verified) {
                        setLoggedIn(true);
                    }
                } else {
                    setLoggedIn(true);
                }
            },
            logout: async () => {
                await axios.post('/logout');
                setLoggedIn(false);
            },
            loggedIn
        }}>
        {props.children}
    </SecurityContext.Provider>
};
export default SecurityProvider;
```

We first send the username and password to the //login/ endpoint. If the user has registered a security device, the response to the _*login* will have a `twoFactorNeeded` attribute set to true. We can call the //startVerify_ endpoint with the user's id, use the resulting assertion request to ask the use to activate their device. We can then the assertion back to the server. And if all is well, we set `loggedIn` to true, and the user will then see the page.

Let's look at it in action. We'll assume we've already registered the device against our account. We open the application, and click on the *Account* page (see *figure 7-12*.)

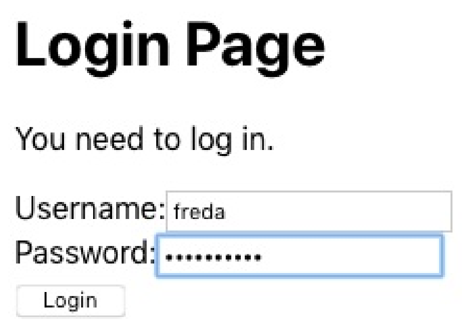


Home

- Public Page
- Private Page 1
- Account page

Figure 7-12. When the application opens, click the Account link

The *Account* page is secured, so we're asked for a username and password (see *figure 7-13*.) In the example application you can enter "freda" as the username, and "mypassword" as the password.



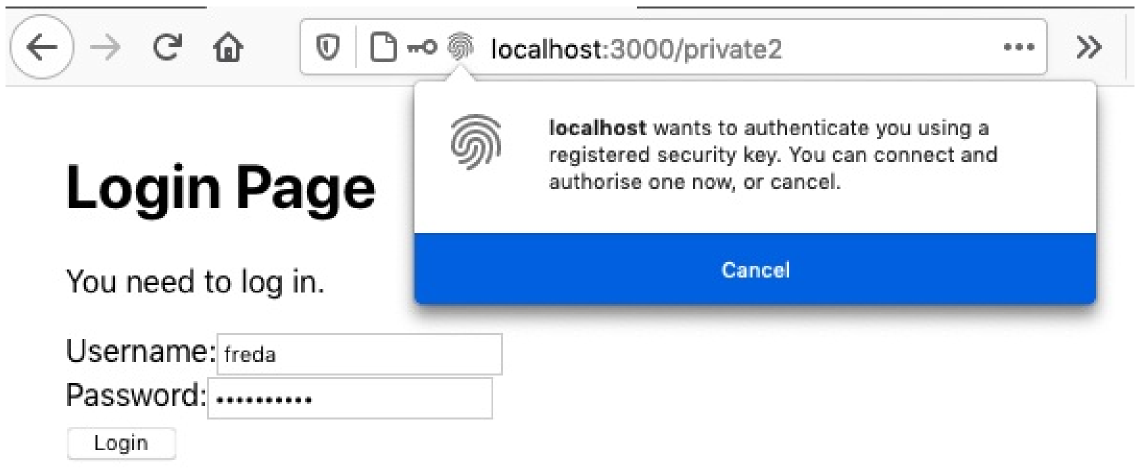
Login Page

You need to log in.

Username: freda
Password: ••••••••••
Login

Figure 7-13. The login form appears

Once we’ve entered the username and password, the browser asks us to connect the security device (see *figure 7-14*.)



localhost:3000/private2

localhost wants to authenticate you using a registered security key. You can connect and authorise one now, or cancel.

Cancel

Login Page

You need to log in.

Username: freda
Password: ••••••••••
Login

Figure 7-14. The browser asks the user to activate their security device

If they connect their device, and activate it, the user is then able to see the secured page (see *figure 7-15*.)

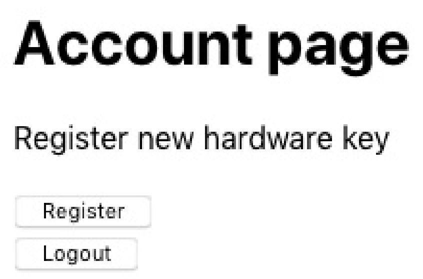


Figure 7-15. The Account page is visible once the user has verified their identity

Discussion

As you can probably tell, *WebAuthn* is quite a complex API. It uses quite obscure language (*attestation* for *registration*, and *assertion* for *verification*) and uses some low-level data-types, which fortunately the Github `webauthn-json` allows us to avoid.

The *real* complexity lives on the server. The example server in the downloadable source code uses a library called *SimpleWebAuthn* to handle most of the cryptological *stuff* for us. If you are planning on using *SimpleWebAuthn* for the server-side of your application, be aware that there is also a client *SimpleWebAuthn* library that works with it. We've avoided using it in the example client source, to avoid making our code too *SimpleWebAuthn*-specific.

If you implement two-factor authentication, you will need to think about what you will do if a user loses their security. Technically, all you will have to do re-enable their account is to remove the device that's registered against their name. But you need to be extremely careful. A common attack against two-factor authentication is to call the service desk and pretend to be a user who has lost their token.

Instead, you will need to create a sufficiently rigorous process that will check the identity of any person asking for an account reset.

You can download the source for this recipe from the Github site.

7.3 Enable https in your development system

Problem

HTTPS is often used during production environments, but there are circumstances where it can useful to use HTTPS during development. Some networked services will only work from within pages secured with HTTPS, WebAuthn will only work remotely with HTTPS[5], and numerous bugs and other issues can creep into your code if your application uses a proxy server with HTTPS.

Enabling HTTPS on production servers is now quite straightforward[6], but how do you enable HTTPS on a development server?

Solution

If you've created your application with `create-react-app`, you can enable HTTPS by:

- Generating a self-signed SSL certificate, and
- Registering the certificate with your development server

To generate a self-signed certificate, we need to understand a little about how HTTPS works.

HTTPS is really just HTTP that is tunneled through an encrypted Secure Sockets Layer (SSL) connection. When a browser connects to a HTTPS address, it opens a connection to a secure socket on the server[7]. In order to make this connection, the server has to provide a certificate that has been issued by some organisation that the browser trusts. If the browser accepts the certificate, it will then send encrypted data to the secure socket on the server, which will then be decrypted on the server, and forwarded to a HTTP server.

The main difficulty in setting up a HTTPS server, is getting a certificate that a web browser will trust. Browsers maintain a set of **root certificates**. These are certificates that are issued by large, trustworthy organizations. When a HTTPS server presents a certificate to a browser, that certificate must be signed by one of the browser's root certificates.

If we want to generate our own SSL certificate, we will first need to create a root certificate and tell the browser to trust it. Then we must generate a certificate for our development server, that has been signed by the root certificate.

If this sounds complicated, it’s because it is.

Let’s begin by creating a root certificate. To do this, you will need a tool called *OpenSSL* installed on your machine.

We’ll use the `openssl` command to create a key file. It will ask your for a pass phrase, which you will have to enter twice:

```
$ openssl genrsa -des3 -out mykey.key 2048
Generating RSA private key, 2048 bit long modulus
......................................................+++
.................................+++
e is 65537 (0x10001)
Enter pass phrase for mykey.key:
Verifying - Enter pass phrase for mykey.key:
$
```

The `mykey.key` file now contains a private key. This key can be used for encrypting data. We can use the key file to create a certificate file. A certificate file contains information about an organization, and an end-date after which it is no longer valid.

You can create a certificate, using the following command:

```
$ openssl req -x509 -new -nodes -key mykey.key -sha256 -days 2048 -out
mypem.pem
Enter pass phrase for mykey.key:
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or
a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:US
State or Province Name (full name) []:Massachusetts
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) []:O'Reilly Media
Organizational Unit Name (eg, section) []:Harmless scribes
Common Name (eg, fully qualified host name) []:Local
Email Address []:me@example.com
```

```
$
```

Here we are creating a certificate that will be valid for the next 2048 days. The pass phrase you are asked for is the pass phrase you set when you created the `mykey.key` file. It doesn't really matter what you enter for the organization details, as you will only be using it on your local machine.

The certificate is now stored in a file called `mypem.pem`[8], and we need to install this file as a root certificate on our machine. There are a number of ways of installing root certificates on your machine[9]. Root certificates can be used to sign web site certificates, which is what we'll do next.

We'll create a local key file, and a Certificate Signing Request (CSR) file, with the following command:

```
$ openssl req -new -sha256 -nodes -out myprivate.csr -newkey rsa:2048
-keyout myprivate.key
-subj "/C=US/ST=Massachusetts/L=Cambridge/O=O'Reilly Media/OU=Harmless
scribes/CN=Local/ema
ilAddress=me@example.com"
Generating a 2048 bit RSA private key
....................+++
..+++
writing new private key to 'myprivate.key'
-----
$
```

Next, create a file called `extfile.txt`, containing the following:

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage=digitalSignature,nonRepudiation,keyEncipherment,dataEncipherm
ent
subjectAltName=DNS:localhost
```

We can now run a command that will generate a SSL certificate for our application:

```
$ openssl x509 -req -in myprivate.csr -CA mypem.pem -CAkey mykey.key -
CAcreateserial -out
myprivate.crt -days 500 -sha256 -extfile  ./extfile.txt
Signature ok
subject=/C=US/ST=Massachusetts/L=Cambridge/O=O'Reilly
```

```
Media/OU=Harmless scribes/CN=Local/
emailAddress=me@example.com
Getting CA Private Key
Enter pass phrase for mykey.key:
$
```

Remember, the pass phrase is the one you created when you first created the `mykey.key` file.

OK, the result of going through all of those steps if that we have two files that we can use to secure our development server:

- The `myprivate.crt` file is a certificate signed by the root certificate. This is the file that reassures the browser that our app can be trusted, and
- The `myprivate.key` file will be used to encrypt connections between the development server and the browser.

If you created your application with `create-react-app`, you can enable HTTPS by putting this in a `.env` file in your main application directory:

..env

```
HTTPS=true
SSL_CRT_FILE=myprivate.crt
SSL_KEY_FILE=myprivate.key
```

If you restart your server, you should be able to access your application at *https://localhost:3000*

Discussion

Self-signed certificates are quite complex things to create, but there are circumstances when they are required. However, even if you don’t need to run HTTPS in your development environment, it can still be worth understanding what HTTPS is, how it works, and why you should trust it.

You can download the source for this recipe from the Github site.

7.4 Enable two factor authentication with fingerprints

Problem

Elsewhere in this chapter[10], we look at how physical tokens, such as Yubikeys, can be used for two-factor authentication. But physical tokens are still fairly uncommon, and can be quite expensive. Most people already have mobile devices, such as cell phones and tablets. Many of those have built in fingerprint sensors. But how can we get a React application to use a fingerprint sensor for two-factor authentication?

Solution

We can also use fingerprint sensors as WebAuthn authentication tokens. The connect to the API in the same way, although there are a number of configuration changes that are required.

This recipe is a based on recipe 2 in this chapter, on how to use removable tokens for two-factor authentication. We saw in recipe 2 that there are two main flows in WebAuthn authentication:

- Registering a token. This is called *attestation*, and
- Verifying a token, this is called *assertion*

Both attestation and assertion have three stages. First the server generates a request. Second, the user uses the token, which generates a response. Third, the response to the server.

If we want to switch from using a removable physical token, to using the built-in fingerprint sensor in a device, we will only need to change the attestation request stage. The attestation request says what kind of token can be registered for a user. For removable physical tokens, like Yubikeys, we generated an attestation request that looked like this:

```
{
    "rpName": "Physical Token Server",
    "rpID": "localhost",
    "userID": "1234",
    "userName": "freda",
    "excludeCredentials": [
        {"id": "existingKey1", "type": "public-key"}
    ],
    "authenticatorSelection": {
        "userVerification": "discouraged"
    },
    "extensions": {
        "credProps": true,
    },
}
```

We need to change this slightly to allow the user to use a fingerprint sensor:

```
    {
        "rpName": "Physical Token Server",
        "rpID": "localhost",
        "userID": "1234",
        "userName": "freda",
        "excludeCredentials": [
            {"id": "existingKey1", "type": "public-key"}
        ],
        "authenticatorSelection": {
            "authenticatorAttachment": "platform",
            "userVerification": "required"
        },
        "attestation": "direct",
        "extensions": {
            "credProps": true,
        },
    }
```

The two requests are almost the same. The first change is in the authenticator selection. We now want to use a *platform* authenticator, because fingerprint sensors are built-in to the device and not removable. This means we are effectively limiting the user to their current physical. Whereas, a Yubikey can be disconnected from one machine, and then connected to another. We’re also saying that we want to use direct attestation. This means that we won’t want to use any additional verification that can be configured against other tokens. For example, we won’t be asking the user to press the fingerprint sensor **and** enter a pin.

Beyond changing this initial attestation request object, all of the other code remains the same. Once a user responds to the attestation request by pressing the fingerprint sensor, it will generate a public key which we can store against the user. When the user logs back in and confirms their identity by pressing the fingerprint sensor, it will sign the challenge string in the same way that a Yubikey would.

Therefore, if you’re going to support one type of authenticator, it’s worth allowing the user to support both fingerprint sensors and removable tokens.

NOTE

Unless a user has a removable token that also works on mobile devices–for example by using Near-Field Communication (NFC)–it’s unlikely that any one user will register both removable tokens and fingerprints. As soon as they have registered a fingerprint, they won’t be able to log in and register a removable token. And vice-versa.

This is how we’ll update the component that allows a user to register a token:

```
import {useState} from 'react';
import Logout from "./Logout";
import axios from "axios";
import {create} from "@github/webauthn-json";

const Private2 = () => {
    const [busy, setBusy] = useState(false);
    const [message, setMessage] = useState();

    const registerToken = async (startRegistrationEndpoint) => {
        setBusy(true);
        try {
            const response = await
axios.post(startRegistrationEndpoint);
            setMessage('Send response')
            const attestation = await create({publicKey:
response.data});
            setMessage('Create attestation')
            const attestationResponse = await axios.post("/register",
{attestation});
            setMessage('registered!')
            if (attestationResponse.data &&
attestationResponse.data.verified) {
                alert("New key registered");
            }
        } catch (err) {
            setMessage('' + err);
        } finally {
            setBusy(false);
        }
    };
    return <div className='Private2'>
        <h1>Account page</h1>
        {
            window.PublicKeyCredential && <>
                <p>
                    Register new hardware key
                </p>
                <button onClick={() =>
registerToken("/startRegister")}
                        disabled={busy}
                >Register Removable Token
                </button>
                <button onClick={() =>
registerToken("/startFingerprint")}
                        disabled={busy}
                >Register Fingerprint
                </button>
            </>
        }
        <div className='Account-message'>
            {message}
        </div>

        <Logout/>
    </div>;
};

export default Private2;
```

We’re calling a slightly different end-point when we want to register a fingerprint, otherwise the rest of the code remains the same.

To try it out, you’ll need to use a device with a fingerprint sensor. WebAuthn can only be used when the browser is connected to either a *localhost* server, or to a remote server using *HTTPS*. To test this code from a mobile device, you will either need to configure *HTTPS* on on your development server[11], or you will need to configure your device to proxy *localhost* connections to your development machines[12].

To run the example application, you will need to change into the application directory and start the development server with:

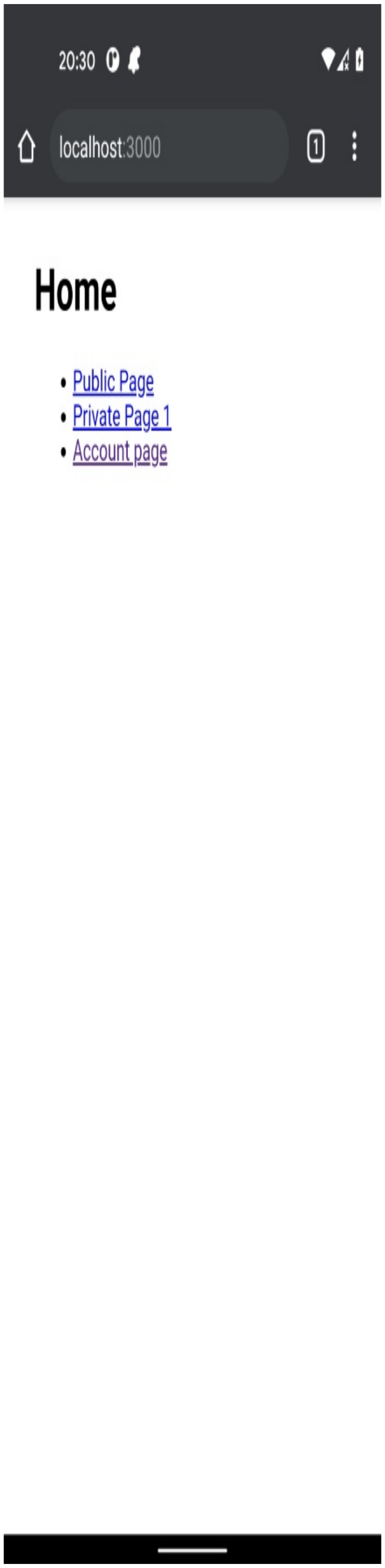
```
npm run start
```

You will also need to run the API server. Open a separate terminal for this and the run it from the *server* sub-directory:

```
cd server
npm run start
```

The development server will run on port 3000, and the API on port 5000. The development server is configured to proxy API requests to the API server.

When you open the application, you should click the *“Account page”* link (see *figure 7-16*).

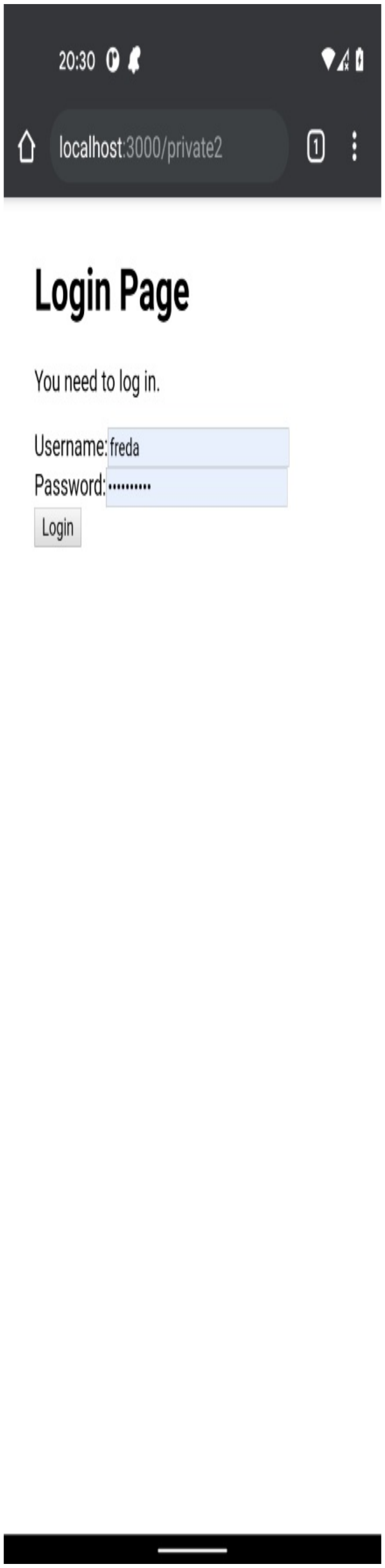


Home

- Public Page
- Private Page 1
- Account page

Figure 7-16. Click on the Account page link on the home page

The application will ask you to sign-in. Enter the username *freda* and the password *mypassword* (see *figure 7-17*.) These values have been hard-coded in the example server.


20:30

localhost:3000/private2

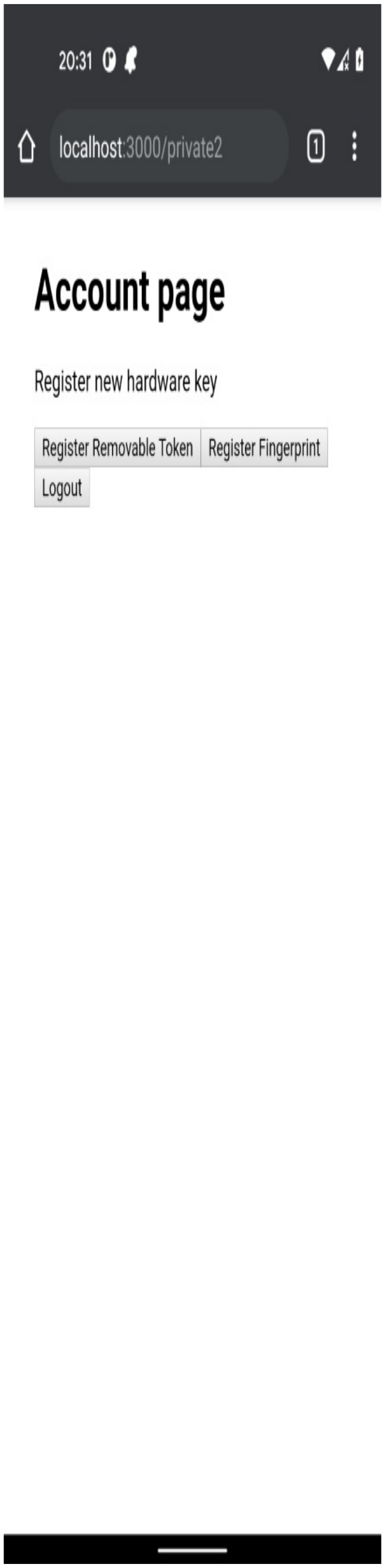
Login Page

You need to log in.

Username: freda
Password: ••••••••••
Login

Figure 7-17. Enter freda/mypassword into the login form

You will now see two buttons for registering tokens against your account. One for fingerprints, the other for removable tokens (see *figure 7-18*.)

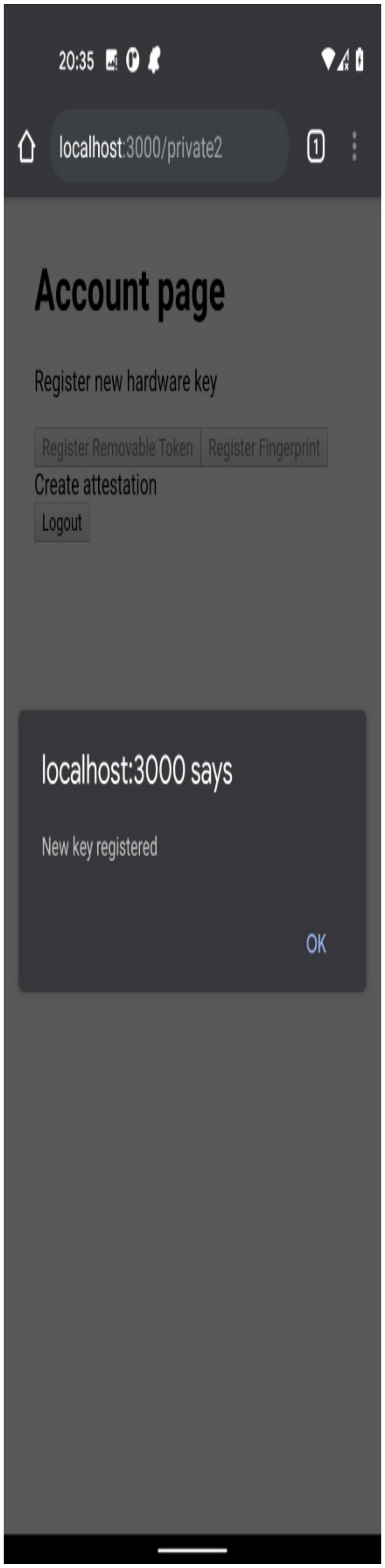


Account page

Register new hardware key

Figure 7-18. There are buttons to register fingerprints and removable tokens

Press the button to register a fingerprint. Your mobile device will ask you to press the fingerprint sensor. Your fingerprint sensor will generate a public key that can be stored against the *freda* account. A message box will appear to tell you when this is done, as you can see in *figure 7-19*.



Account page

Register new hardware key

Create attestation

localhost:3000 says

New key registered

Figure 7-19. The application will confirm when the token is registered

Now log out. When you log back in again, enter *freda* and *mypassword* in the form. You will now be asked to confirm your identity by pressing the fingerprint sensor, and you will then be logged back in to the system.

Discussion

Built-in fingerprint sensors are much more common that removable tokens like Yubikeys. There is a difference in the usage pattern of the two devices. Yubikeys can be moved from device to device, whereas fingerprints are typically limited to a single device[13]. Removable tokens therefore have additional flexibility for users who might want to connect from several devices. The downside to removable devices is that they are far easier to lose than a cell phone. In most cases, it is worthwhile supporting both types of device, and leave it to the users to decide which option is best for them.

You can download the source for this recipe from the Github site.

7.5 Use confirmation logins

Problem

Sometimes a user might want to perform operations that are more dangerous or are not easily reversible. They might want to delete data, remove a user account, or do something that will send an email. How do you prevent a third party from carrying out these operations if they find a logged-in, but unattended machine?

Solution

Many systems force users to confirm their login credentials before being able to perform sensitive operations. You will mostly likely want to do this for several operations, and so it would be useful if there was a way of doing the confirmation centrally.

We’ll base this recipe on the code for the *Secured Routes* recipe from chapter 4. In the *Secured Routes* recipe we built a *SecurityProvider* component which provided *login* and *logout* functions to its child components:

```
import {useState} from "react";
import SecurityContext from "./SecurityContext";

export default (props) => {
    const [loggedIn, setLoggedIn] = useState(false);

    return <SecurityContext.Provider
        value={{
            login: (username, password) => {
                // Note to engineering team:
                // Maybe make this more secure...
                if (username === 'fred' && password === 'password') {
                    setLoggedIn(true);
                }
            },
            logout: () => setLoggedIn(false),
            loggedIn
        }}>
        {props.children}
    </SecurityContext.Provider>
};
```

Components that needed to use the *login* or *logout* functions could access them from the from *useSecurity* hook:

```
const security = useSecurity();
...
// Anywhere that we need to logout...
security.logout();
```

For this recipe, we’ll add an extra function to *SecurityProvider* that will allow a child component to confirm that the user is logged in. Once they’ve provided the username and password, we allow them to perform the dangerous operation.

We could do this by creating a function that accepts a callback function containing the dangerous operation, which is then called after the user confirms their login details. This will be easier to implement in the *SecurityProvider*, but will have some issues when we call it from a component. That’s because we will either have to return some sort of success/failure flag:

```
// We WON'T do it like this
confirmLogin((success) => {
    if (success) {
        // Do dangerous thing here
    } else {
        // Handle the user canceling the login
    }
})
```

This has the disadvantage that if you forget to check the value of the `success` flag, the code will perform the dangerous operation, even if the user cancels the login form.

Or else, we will have to pass two separate callbacks: one for success, and one for cancellation.

```
    // We WON'T do it like this either
    confirmLogin(
        () => {
            // Do dangerous thing here
        },
        () => {
            // Handle the user canceling the login
        });
```

This has the disadvantage that the code is, well, a little ugly.

Instead, we’ll implement the code with a promise. This will make the implementation a little more complex, but it will simplify any code that calls it.

This is a version of *SecurityProvider*, complete with the new `confirmLogin` function:

```
    import {useRef, useState} from "react";
    import SecurityContext from "./SecurityContext";
    import LoginForm from "./LoginForm";

    export default (props) => {
        const [showLogin, setShowLogin] = useState(false);
        const [loggedIn, setLoggedIn] = useState(false);
        const resolver = useRef();
        const rejecter = useRef();

        const onLogin = async (username, password) => {
            // Note to engineering team:
            // Maybe make this more secure...
            if (username === 'fred' && password === 'password') {
                setLoggedIn(true);
            }
        };
        const onConfirmLogin = async (username, password) => {
            // Note to engineering team:
            // Same here...
            return (username === 'fred' && password === 'password');
        };

        return <SecurityContext.Provider
            value={{
                login: onLogin,
                confirmLogin: async (callback) => {
                    setShowLogin(true);
                    return new Promise((res, rej) => {
                        resolver.current = res;
                        rejecter.current = rej;
                    });
                },
                logout: () => setLoggedIn(false),
                loggedIn
            }}>
            {
                showLogin ?
                    <LoginForm
                        onLogin={async (username, password) => {
                            const valid = await onConfirmLogin(username,
    password);
                            if (valid) {
                                setShowLogin(false);
                                resolver.current();
                            }
                        }}
                        onCancel={() => {
                            setShowLogin(false);
                            rejecter.current();
                        }}
                    />
                    : null
            }
            {props.children}
        </SecurityContext.Provider>
    };
```

If the user calls the `confirmLogin` function, the *SecurityProvider* will display a login form to allow the user to confirm their username and password. The `confirmLogin` function returns a promise which will only resolve if the user types in the username and password correctly. If the user cancels the login form, the promise will be rejected.

We’re not showing the details of the `LoginForm` component here, but you can find it in the downloadable source for this recipe.

Our example code here is just checking the username and password against static strings to see if they’re correct. In your version of the code, you will replace this with a call to some security service.

NOTE

When we call the `confirmLogin`, we’re creating a storing the promise in a *ref*. Refs are commonly used to acquire references to elements in the DOM are render time, but they can also be used whenever to want to store a piece of state immediately. In general, it’s not good practice to use a lot of refs in your code, and we’re only using them here so we can record the promise immediately, without wait for a `useState` operation to resolve.

How would you use the `confirmLogin` function in practice? Let’s say we have a component that contains a button which performs some dangerous operation:

```
import {useState} from 'react';
import Logout from "./Logout";

export default () => {
    const [message, setMessage] = useState();

    const doDangerousThing = () => {
        setMessage('DANGEROUS ACTION!')
    }

    return <div className='Private1'>
        <h1>Private page 1</h1>

        <button onClick={() => {
            doDangerousThing();
        }}>
            Do dangerous thing
        </button>

        <p className='message'>{message}</p>

        <Logout/>
        </div>;
};
```

If we want the user to confirm their login details before performing this operation, we can first get hold of the context provided by the *SecurityProvider*:

```
const security = useSecurity();
```

In the code that performs the dangerous operation, when can then *await* the promise returned by `confirmLogin`:

```
const security = useSecurity();
...
await security.confirmLogin();
setMessage('DANGEROUS ACTION!')
```

The code following the call to `confirmLogin` will only be performed once the promise has been resolved. This only happens if the user provides the correct username and password.

If the user cancels the login dialog, the promise will be rejected, and we can handle the cancellation in a *catch* block.

This is a modified version of the component performing dangerous code, that now confirms the user's login before proceeding:

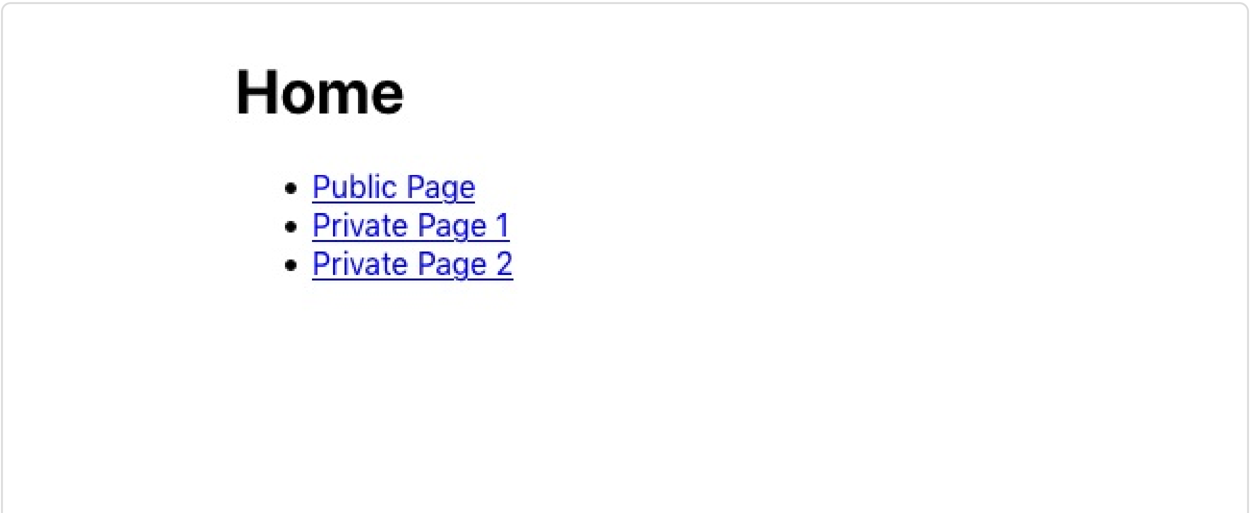
```
import {useState} from 'react';
import Logout from "./Logout";
import useSecurity from "./useSecurity";

export default () => {
    const security = useSecurity();
    const [message, setMessage] = useState();

    const doDangerousThing = async () => {
        try {
            await security.confirmLogin();
            setMessage('DANGEROUS ACTION!')
        } catch (err) {
            setMessage('DANGEROUS ACTION CANCELLED!')
        }
    }

    return <div className='Private1'>
```

```
          <h1>Private page 1</h1>

          <button onClick={() => {
              doDangerousThing();
          }}>
              Do dangerous thing
          </button>

          <p className='message'>{message}</p>

          <Logout/>
      </div>;
  };
```

If we try out the code, we will first need to run the application from the app directory:

```
npm run start
```

When the application opens (see *figure 7-20*), you will need to click on *Private page 1*.



Home

- Public Page
- Private Page 1
- Private Page 2

Figure 7-20. Begin by clicking the Private Page 1 link

You will now be asked to login (see *figure 7-21*). You should log in with `fred/password`.

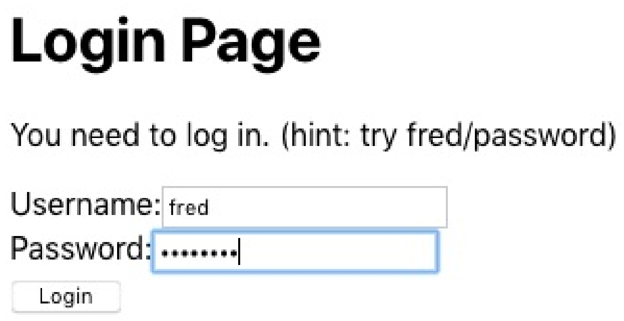


Figure 7-21. The page is secured, so you will need to log in

If you now click the button to perform the dangerous operation, you will be asked to confirm your credentials before continuing (as show in *figure 7-22*):

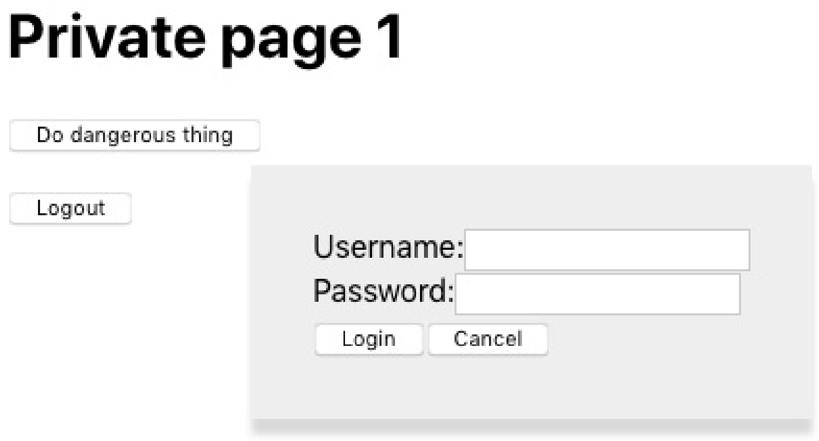


Figure 7-22. You must confirm your login details before continuing

Discussion

This recipe allows you to centralize your confirmation code, because all of the user-interface for the confirmation process is hidden away in the

SecurityProvider.

This has an additional advantage that might be immediately clear. Not only does this lighten the code in our components, but it has the added advantage that user confirmation can take place inside custom hooks. If you abstract a set of operations into some hook-based service[14], you can also include the confirmation logic into that service. This will leave your components completely unaware of which operations are classed as dangerous, and which are not.

You can download the source for this recipe from the Github site.

7.6 Use single factor authentication

Problem

We've already seen that removable tokens and fingerprints can be in a two-factor authentication system to provide additional security to a user's account.

However, they can also be used as a simple login convenience. Many mobile applications allow a user to log in by simply pressing the fingerprint sensor, without needing to enter a username or password. Whilst this doesn't increase the security of the application, neither does it decrease it. The user still needs access to a particular security device, which will be far harder to copy than a username and password.

But how do you enable single-factor authentication for a React application?

Solution

Security tokens, such as fingerprint sensors and USB-devices like Yubikeys, need to be recorded against a user-account on the server. The problem with single-factor authentication is that we don't know who the user is supposed to be when they tap the fingerprint sensor. In a two-factor system, they have just typed their username into a form. But in a single-factor system we need to know who the user is supposed to be when we create the assertion request[15]

We can avoid this problem by setting a cookie in the browser containing the user-id whenever a person with a token-enabled account logs in[16]

When the time comes in the application to display the login form, the app can check for the existence or the cookie, and then use it to create a assertion request which can be used to then ask for the security token. If the user does not wish to use the token, or if they are a different user, then can cancel the request and simply use the login form.[17]

WARNING

The user-id is not, in itself, particularly private information. User-ids are often machine-generated internal keys which provide very little in the way of secured information. However, if your user-ids are more easily identifiable, such as an email address, you might not want to use this approach.

We're basing the code for this recipe, on the *Secured Routes* code from chapter 2. We manage all of our security through a wrapper component called *SecurityProvider*. This provide child components with *login* and *logout* functions. We'll add another functions called *loginWithToken*:

```
import {useState} from "react";
import SecurityContext from "./SecurityContext";
import {get} from "@github/webauthn-json"
import axios from "axios";

const SecurityProvider = (props) => {
    const [loggedIn, setLoggedIn] = useState(false);

    return <SecurityContext.Provider
        value={{
            login: async (username, password) => {
                const response = await axios.post('/login', {username,
password});
                setLoggedIn(true);
            },
            loginWithToken: async (userID) => {
                const response = await axios.post("/startVerify",
{userID});
                const assertion = await get({publicKey:
response.data});
                await axios.post("/verify", {userID, assertion});
                setLoggedIn(true);
            },
            logout: async () => {
                await axios.post('/logout');
                setLoggedIn(false);
            },
            loggedIn
        }}>
        {props.children}
    </SecurityContext.Provider>
};
export default SecurityProvider;
```

The *loginWithToken* accepts a user-id and then asks the user to verify their identity with a token by:

- Calling a *startVerify* function on the server, to create an assertion request
- Passing the request to WebAuthn so that the user will be asked to activate their token, for example, by pressing the fingerprint sensor, then
- Passing the generate assertion back to an endpoint called *verify* to check that a valid token has been provided

In your implementation, you will need to provide your own equivalent code to the *startVerify* and *verify* endpoints.

In order to call the *loginWithToken* function in *SecurityProvider*, we will need to find the current user's ID from the cookies. We'll do this by installing the *js-cookie* library:

```
npm install js-cookie
```

This will allow us to read a *userID* cookie like this:

```
import Cookies from 'js-cookie';
...
const userIDCookie = Cookies.get('userID');
```

We can now use this code in a *Login* component, which will check for a *userID* cookie. If one exists, it will ask to log in by token. Otherwise, it will allow the user to log in using a username and password.

```
import {useEffect, useState} from "react";
import useSecurity from "./useSecurity";
import Cookies from 'js-cookie';

const Login = () => {
    const {login, loginWithToken} = useSecurity();
    const [username, setUsername] = useState();
    const [password, setPassword] = useState();
    const userIDCookie = Cookies.get('userID');

    useEffect(() => {
        (async () => {
            if (userIDCookie) {
                loginWithToken(userIDCookie);
            }
        })();
    }, [userIDCookie]);

    return <div>
        <h1>Login Page</h1>

        <p>You need to log in.</p>

        <label htmlFor='username'>Username:</label>
        <input
            id='username'
            name='username'
            type='text'
            value={username}
            onChange={(evt) => setUsername(evt.target.value)}/>

        <br/>
        <label htmlFor='password'>Password:</label>
        <input
            id='password'
            name='password'
            type='password'
            value={password}
            onChange={(evt) => setPassword(evt.target.value)}/>

        <br/>
        <button onClick={() => login(username,
password)}>Login</button>
    </div>;
};

export default Login;
```

Let’s try out the example application. We must first, start the development server from the application directory:

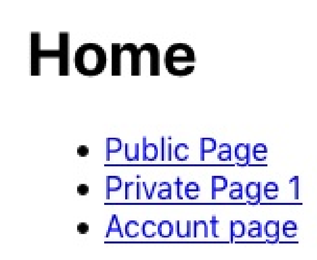
```
npm run start
```

Then in a separate terminal, we can start the example API server.

```
cd server
npm run start
```

The development server runs on port 3000; the API server on port 5000.

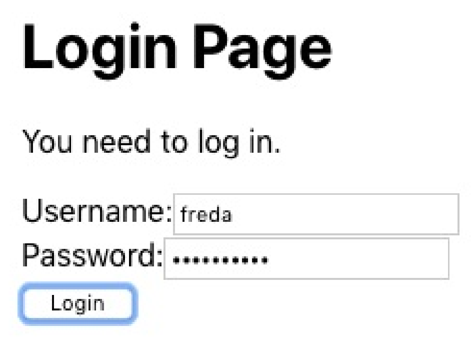
When the application starts, click on the link to the *Account page* (as show in *figure 7-23*).


Home

- Public Page
- Private Page 1
- Account page

Figure 7-23. When the app opens, click on the link to the account page

We're now asked to log in (see *figure 7-24*). Use the username *freda* and the password *mypassword*.

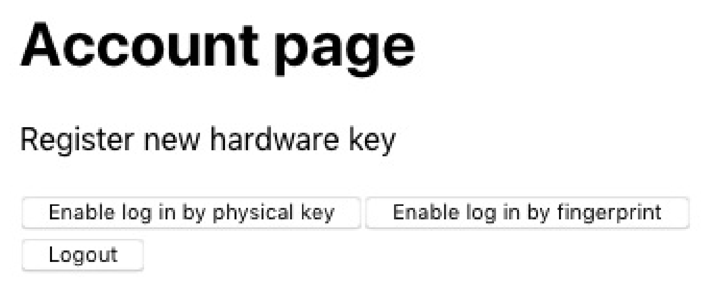

Login Page

You need to log in.

Username: freda
Password: ••••••••••

Figure 7-24. Log in with freda/mypassword

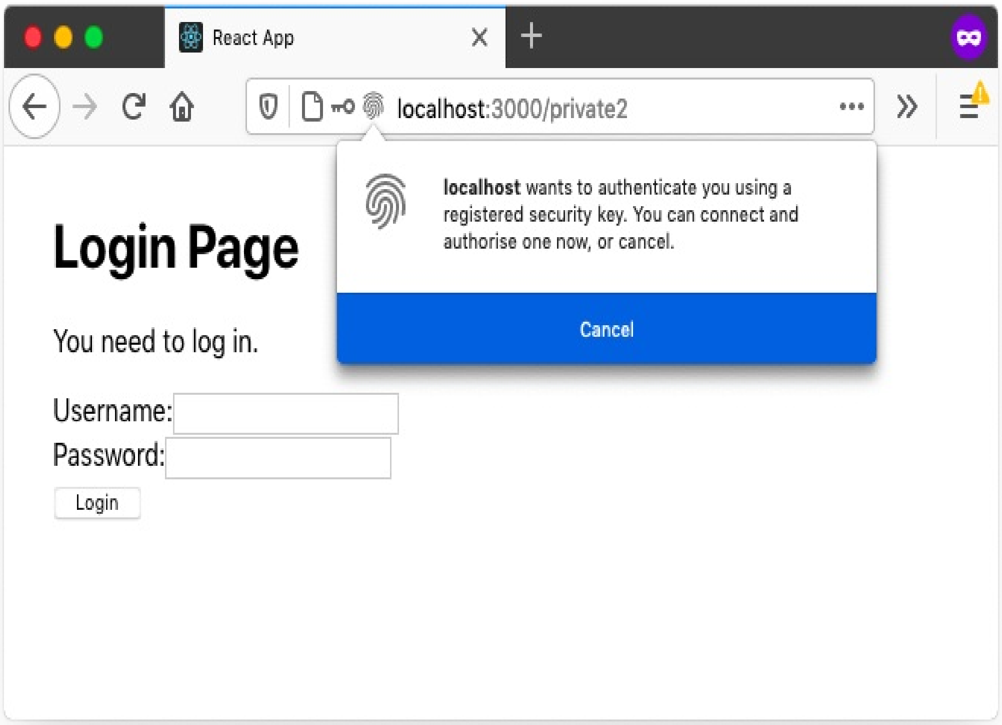
The account page asks if we want to enable log-in with a fingerprint sensor or physical token (see *figure 7-25*). You can register a token and then log out.


Account page

Register new hardware key

Figure 7-25. Choose to enable log in with a physical token or fingerprint

The next time we log in, we will immediately see the request to activate a token (see *figure 7-26*).



localhost wants to authenticate you using a registered security key. You can connect and authorise one now, or cancel.

Login Page

You need to log in.

Username:
Password:

Figure 7-26. Once enabled, you can log in with just the token

If we activate the token, we are then logged in, without needing to use the login form.

Discussion

It’s important to note that single-factor authentication is about increasing convenience rather than security. Fingerprint sensors are particularly useful for this, as logging in literally involves moving one finger.

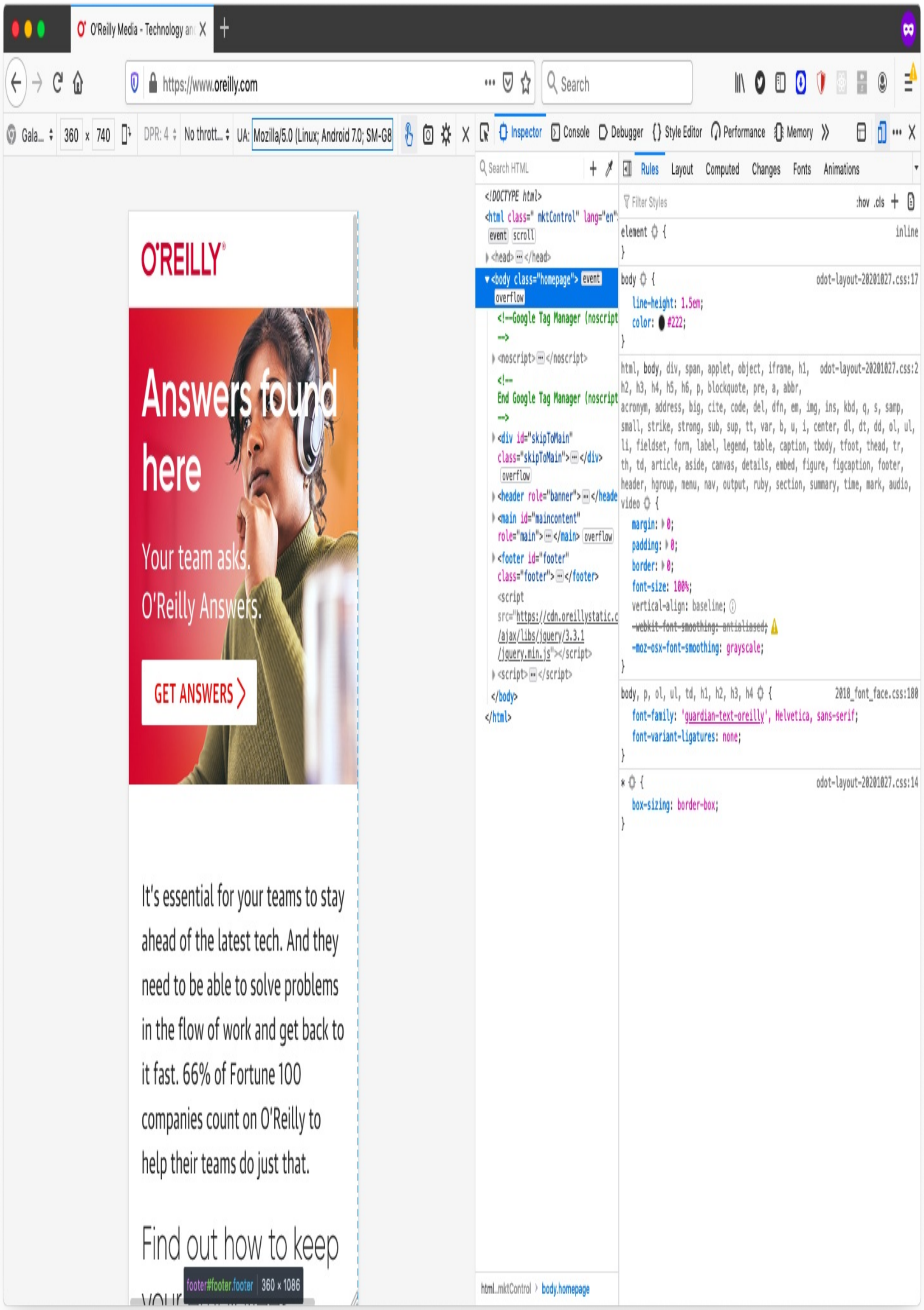
You should always provide the ability to fall back to using the login form. Doing so will not reduce the security of your application, as a wily hacker could always delete the cookie and fall back to using the form anyway.

You can download the source for this recipe from the Github site.

7.7 Test local React apps on an Android device

Problem

Most mobile browser testing can be carried out using a desktop browser simulating the appearance of a mobile device (see /figure 7-27).


O'Reilly Media - Technology an
https://www.oreilly.com
Search
Gala... 360 × 740 DPR: 4 No thrott... UA: Mozilla/5.0 (Linux; Android 7.0; SM-G8
Inspector Console Debugger Style Editor Performance Memory
Search HTML
Rules Layout Computed Changes Fonts Animations
Filter Styles :hov .cls
O'REILLY
Answers found here
Your team asks. O'Reilly Answers.
GET ANSWERS
Find out how to keep your
footer#footer.footer 360 × 1086
html.mktControl body.homepage

It's essential for your teams to stay ahead of the latest tech. And they need to be able to solve problems in the flow of work and get back to it fast. 66% of Fortune 100 companies count on O'Reilly to help their teams do just that.

```
<!DOCTYPE html>
<html class=" mktControl" lang="en"  event  scroll
  <head>…</head>
  <body class="homepage">  event  overflow
    <!--Google Tag Manager (noscript
    -->
    <noscript>…</noscript>
    <!--
    End Google Tag Manager (noscript
    -->
    <div id="skipToMain"
    class="skipToMain">…</div>
     overflow
    <header role="banner">…</heade
    <main id="maincontent"
    role="main">…</main>  overflow
    <footer id="footer"
    class="footer">…</footer>
    <script
    src="https://cdn.oreillystatic.c
    /ajax/libs/jquery/3.3.1
    /jquery.min.js"></script>
    <script>…</script>
  </body>
</html>
```

```
element {                                                   inline
}
body {                                     odot-layout-20201027.css:17
  line-height: 1.5em;
  color: ● #222;
}
html, body, div, span, applet, object, iframe, h1,  odot-layout-20201027.css:2
h2, h3, h4, h5, h6, p, blockquote, pre, a, abbr,
acronym, address, big, cite, code, del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var, b, u, i, center, dl, dt, dd, ol, ul,
li, fieldset, form, label, legend, table, caption, tbody, tfoot, thead, tr,
th, td, article, aside, canvas, details, embed, figure, figcaption, footer,
header, hgroup, menu, nav, output, ruby, section, summary, time, mark, audio,
video {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  vertical-align: baseline;
  ~~-webkit-font-smoothing: antialiased;~~
  -moz-osx-font-smoothing: grayscale;
}
body, p, ol, ul, td, h1, h2, h3, h4 {             2018_font_face.css:180
  font-family: 'guardian-text-oreilly', Helvetica, sans-serif;
  font-variant-ligatures: none;
}
* {                                        odot-layout-20201027.css:14
  box-sizing: border-box;
}
```

Figure 7-27. A desktop browser can be used for most mobile testing

But there are times when it is best to test a React application on a real physical. This is usually not a problem; the React application can be accessed remotely using the IP address of the development machine.

There are, however, circumstances where that is not true:

- Your mobile device might not be able to connect to the same network as your development machine, or
- You might be using a technology, like WebAuthn, that requires HTTPS for domains other than *localhost*

Is it possible to configure a mobile device to access a React app running on *localhost*, even though it is running on a separate machine?

Solution

In this recipe we will look at how we can proxy the network on an Android-based device so that connections to *localhost* will automatically be routed to the server on your development machine.

The first thing you'll need is an Android device that has USB debugging enabled[18]. You will also need a copy of the Android SDK installed. This will allow you to use a tool called the *Android Debug Bridge* (`adb`). The Android Debug Bridge opens a communication channel between your development machine and an Android device.

You will then need to connect your Android device to your development machine with a USB cable, and ensure that the `adb` command is available on your command path[19] You can then list the Android devices connected to your machine:

```
$ adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
25PRIFFEJZWWDFWO        device
$
```

Here you can see there is a single device connected, with a device-id `25PRIFFEJZWWDFWO`.

You can now use the `adb` command to configure a proxy on the Android which will redirect all HTTP traffic to its internal port 3000:

```
adb shell settings put global http_proxy localhost:3000
```

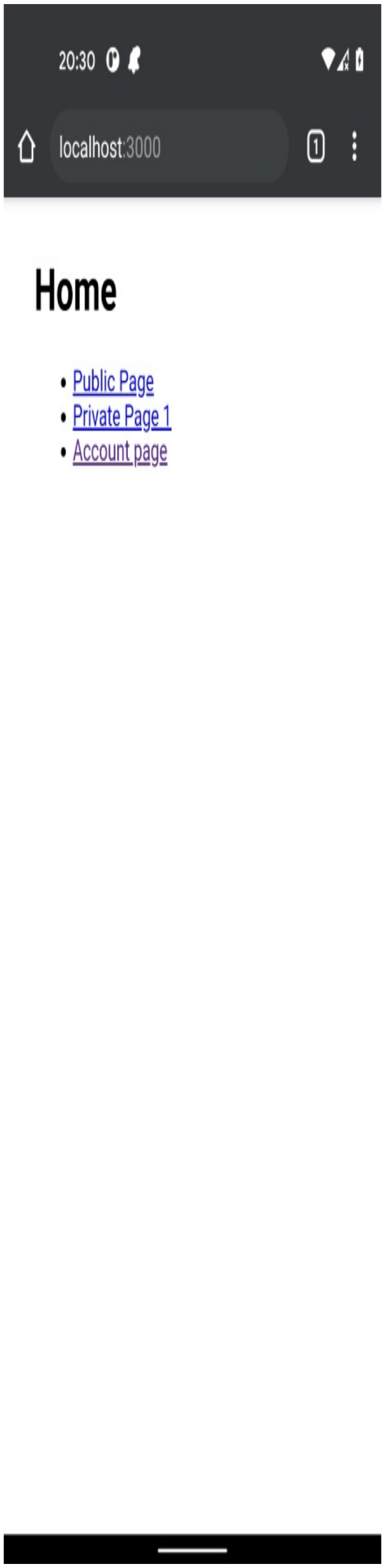
WARNING

If you have more than one Android device connected to your machine, you will also need pass the option `-s <device-id>` to the `adb`, so that it will know which device to talk to.

You will next need to tell `adb` to run a proxy service on the Android device, which will forward any traffic from port 3000 on the device, to port 3000 on the development machine:

```
adb reverse tcp:3000 tcp:3000
```

If you now open a browser on the Android device, and tell it to go to *http://localhost:3000*, it will display the app running on your development machine, as if it's running inside the device (see *figure 7-27*).



Home

- Public Page
- Private Page 1
- Account page

Figure 7-28. If you open a mobile browser to localhost, it will connect to the development machine

Once you have finished using the app, you will need to disable the proxy setting on the Android device.

WARNING

If you fail to disable the proxy on the Android device, it will no longer be able to access the network.

You can do this by resetting the proxy back to `:0`

```
adb shell settings put global http_proxy :0
```

Discussion

This recipe requires a lot of work the first time you use it, because it involves installing an entire Android SDK on your development machine. But then it will be very easy to connect and disconnect real Android devices to your machine.

7.8 Use eslint to check for security flaws

Problem

Security threats in JavaScript are frequently caused by just a few common coding issues. You can decide to create a set of coding standards that will avoid those errors, but you will then need to frequently review the standards to keep them up to date with the latest changes in technology, and you will also need to introduce slow and expensive code review processes.

Is there a way to check for poor security practices in code, that will not slow down your development processes?

Solution

The best way to introduce security reviews is to try to automate them. One tool that will allow you to do this is `eslint`. If you’ve created your application with a tool like `create-react-app`, you have probably already got `eslint` installed. In fact, `create-react-app` runs `eslint` each time it restarts its development server. If you ever see coding issues being flagged in the terminal, that output has come from `eslint`:

```
Compiled with warnings.

src/App.js
  Line 5:9:  'x' is assigned a value but never used  no-unused-vars

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.
```

If you don’t have `eslint` installed, you can add install it through `npm`:

```
npm install --save-dev eslint
```

Once installed, you can initialize it like this:

```
$ node_mobule/.bin/eslint --init
✔ How would you like to use ESLint? · problems
✔ What type of modules does your project use? · esm
✔ Which framework does your project use? · react
✔ Does your project use TypeScript? · No / Yes
✔ Where does your code run? · browser
✔ What format do you want your config file to be in? · JavaScript
Local ESLint installation not found.
The config that you've selected requires the following dependencies:

eslint-plugin-react@latest eslint@latest
✔ Would you like to install them now with npm? · No / Yes
$
```

Remember: you don’t need to initialize `eslint` if you’re using `create-react-app`; it’s already done for you.

At this point, you could choose to write your own set of `eslint` rules to check for breaches of any security practices. However, it’s far easier to install an `eslint` plugin with a set of security rules already written for you.

As an example, let’s install the eslint-plugin-react-security package, which is created and managed by *https://slyk.io*

```
npm install --save-dev eslint-plugin-react-security
```

Once installed, we can enable this plugin by editing the `eslintConfig` section of `package.json` (if you're using `create-react-app`) or the `eslintrc*` file in your app directory.

You should change it from this:

```
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ]
},
```

to this:

```
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ],
  "plugins": [
    "react-security"
  ],
  "rules": {
    "react-security/no-javascript-urls": "warn",
    "react-security/no-dangerously-set-innerhtml": "warn",
    "react-security/no-find-dom-node": "warn",
    "react-security/no-refs": "warn"
  }
},
```

This will enable four rules from the `react-security` plugin.

To check that they work, let's add some code to an application that will contravene the *no-dangerously-set-innerhtml* rule:

```
import logo from './logo.svg';
import './App.css';

function App() {
    return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
          <div dangerouslySetInnerHTML={{__html: '<p>This is a bad
idea</p>'}} />
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

If you’ve installed eslint manually, you can now scan this file with:

```
node_modules/.bin/eslint src/App.js
```

If you’re using create-react-app you just need to restart the server, to ensure that it reloads the `eslint` config:

```
Compiled with warnings.

src/App.js
  Line 12:16:  dangrouslySetInnerHTML prop usage detected  react-
security/no-dangerously-set-innerhtml

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.
```

Discussion

Another advantage of automating your security checks, if that you can add them to your build-and-deploy pipeline. If you run the checks at the start of the pipeline, you can reject a commit immediately and notify the developer.

If you have a team of developers, you might also want to run the `eslint` checks using a git *pre-commit* hook. This will prevent developers ever checking-in code that fails the audit. This will give faster feedback to the developer, and prevent them failing the build for everyone else.

If you want to configure pre-commit hooks through your `package.json` file, consider installing Husky code hooks

You can download the source for this recipe from the Github site.

7.9 Make login forms browser-friendly

Problem

Many security solutions rely on username/password forms, but there are a number of usability traps that are easy to fall into when creating them. On some devices, in an attempt to be helpful, automated capitalization and autocorrect can corrupt usernames and passwords. Some browsers will attempt to autocomplete username fields, but it is often not clear what rules they use, and so autocomplete works on some sites but not others.

What practices should you follow when building login forms, so that they will work with the browser, rather than against it?

Solution

There are several HTML attributes that can greatly improve the usability of your login forms.

First, it can be useful to disable autocorrect for username fields. Autocorrect is frequently applied on mobile devices, to compensate for the small keyboards, and the spelling mistakes that inevitably occur. But autocorrect is of little use when typing usernames. You can disable autocorrect using the `autoCorrect` attribute:

```
<input autoCorrect="off"/>
```

Next, if your username is an email address, consider setting the `type` to `email`.

On mobile devices this might launch an email-specific keyboard, which will make typing email addresses far easier. Some browsers may even show recent email addresses in an autocomplete window, or in the header of an email-specific keyboard:

```
<input type="email"/>
```

You might also consider using `j_username` as the `id` and `name` of the username field. Why? It's because `j_username` is commonly used by Java-based and is likely to have been used at some point in the past by the user. This increases the likelihood that the browser might offer the email address in an autocomplete window:

```
<input id="j_username" name="j_username"/>
```

For many browsers, you can explicitly say that a field represents a username field, making it **very** likely that you will trigger an autocomplete response from the browser.

```
<input autoComplete="username"/>
```

Now, what to do about passwords?

First, always set the type to `password`.

```
<input type="password"/>
```

Never be tempted to reproduce the visual appearance of a password field in some other way, for example, by custom CSS styling. Doing so will prevent the browser applying standard security features to the password field, such as disabling the *copy* function inside it. Also, if you don't set the type to `password`, the browser will not offer to store the value in its password manager.

Password fields are typically used for two different things: for current passwords (when logging in) and for new passwords (when signing up, or when changing a password).

Why is this relevant? It’s because the HTML `autoComplete` attribute can indicate to the browser how you intend to use the password field.

If it’s a login form, you will want to say that the password is a *current password*:

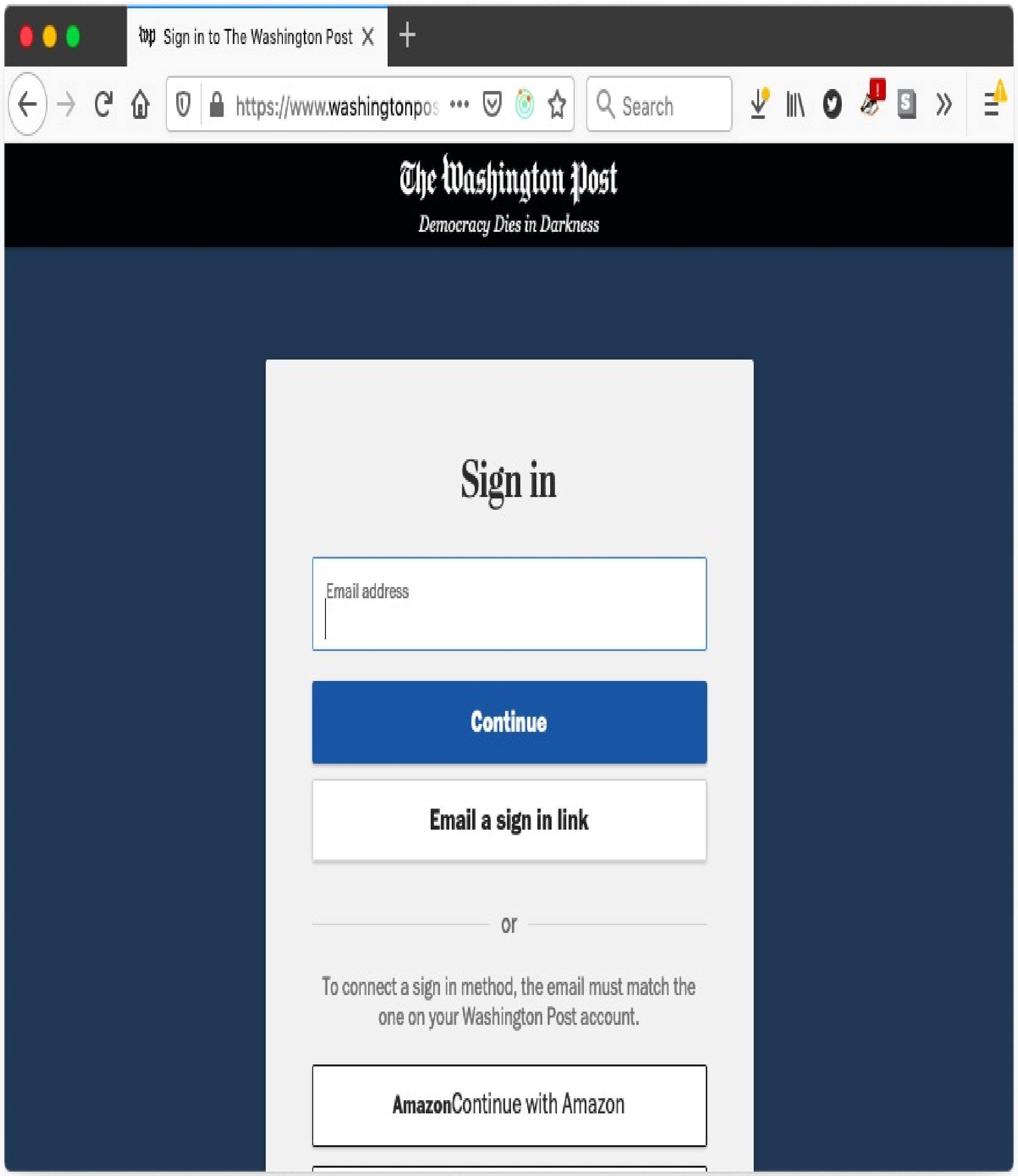
```
<input type="password" autoComplete="current-password"/>
```

If it’s a registration, or change password form, you should set it to `new-password`:

```
<input type="password" autoComplete="new-password"/>
```

This will encourage the browser to autocomplete stored passwords in a login form. It will also trigger any built-in, or third-party, password generation tools.

Finally, avoid using wizard-style login screens (see *figure 7-30* for an example from the *Washington Post*).


Sign in to The Washington Post
https://www.washingtonpos
Search
The Washington Post
Democracy Dies in Darkness
Sign in
Email address
Continue
Email a sign in link
or
AmazonContinue with Amazon

To connect a sign in method, the email must match the one on your Washington Post account.

Figure 7-29. Multi-step login forms can prevent a browser using autocomplete

Browsers are less likely to recognize a single username field as a login form, and so are less likely to offer to complete the details for you.

Discussion

The `autocomplete` attribute has many other seldom-used values that are useful for many types of form-fields, from address details and phone numbers to credit card numbers. For further information see the Mozilla development site.

---

[1] Or, in the case of GraphQL, accessors and mutators.

[2] *https://www.yubico.com*

[3] With the notable exception of Internet Explorer

[4] Recipe 6, in chapter 2.

[5] It is possible to get around this problem from Android devices, by proxying your phone through your development machine. There is a recipe for doing this, elsewhere in this chapter.

[6] See *https://letsencrypt.org*

[7] By default, this will be on port 443.

[8] The .pem extension stands for Privacy-Enhanced Mail. The PEM format was originally designed for use with email, but it now used as a general certificate storage format.

[9] For a detailed guide, see *https://www.bounca.org/tutorials/install_root_certificate.html*

[10] See recipe 2 in the chapter for how to use physical tokens for two-factor authentication.

[11] See recipe 3 in this chapter.

[12] See recipe 7 in this chapter.

[13] This is obviously not the case if the user has connected and external fingerprint sensor.

[14] For an example of such a service, see the `useForum` hook in the second recipe of chapter 5.

[15] The assertion request is needed when the browser asks the user to scan their fingerprint, or activate their token. It includes a list of the devices that can be used, and therefore will be unique to a given user.

[16] A consequence of this approach is that the user will only be able to perform single-factor authentication on the browser where they registered the token. If they use a different browser, or if they have recently cleared their cookies, they will have to fall back to using the login form.

[17] This assumes that you are using a cookie that is readable by JavaScript. It’s also possible to use a HTTP-only cookie, which will only be read by server code. If you use a HTTP-only cookie, you will need code on the server to check if the user should provide a token, or not.

[18] For details, see the Android developer site

[19] You will need to locate the Android SDK installation on your machine. The `adb` command can be found in a sub-directory within this installation.

Chapter 8. Testing

In this chapter we'll look at various techniques for testing your React applications. In general, we've found that it is a bad idea to be too prescriptive about the precise mix of tests you should have. A good guiding principle is to follow these two rules:

- Never write code unless you have a failing test, and
- If a test passes the first time you run it, then delete it.

These two rules will help you build code that works, whilst avoiding creating redundant tests that provide very little value.

We have found that early in a project it is easier to write more browser-based tests. These tests tend to be higher-level and help capture the principle business requirements for an application. Later on, when the architecture of the application starts to emerge and stabilize, it becomes easier to right more unit tests of individual components. They are faster to write, quicker to run and once you have a stable structure to your code, you will not need to continuously update them.

Sometimes it's worth loosening the definition of what a test *is*. When you are working on layout code, whose main value is visual, you might consider a Storybook story to be a "test". The assertion is done by your eye, looking at the component as you create. Of course, this kind of test will not automatically pick up regression failures, but we present a technique in recipe that will allow you to turn these visual checks into actual automated tests.

If you write tests *before* you write code, you will find that tests are tools for design. They will become executable examples of how you would like your application to work. They become the principle tools of development.

Instead, if you write tests *after* you write the code, they will be simply artifacts. Pieces of code that you must slavishly create because they feel like the sorts of things a professional developer should write.

There are four key tools that we focus on in this chapter: the React testing library, Storybook, the Selenium library and Cypress.

The React testing library is a great way of creating very detailed unit tests.

Storybook is a gallery tool that we have looked at previously. We include it in this chapter because a gallery is a set of code examples, and that's what tests are. You will find ways of using Storybook as part of your testing/development process.

Selenium is one of the most established libraries testing your application in a real browser.

Finally, what is quickly becoming our favorite tool for testing: Cypress. Cypress is similar to Selenium, in that it runs inside a browser. But it includes a whole host of additional features, such as test replays, generated videos of test runnings, and a significantly simpler programming model. If you use only one tool from this chapter, let it be Cypress.

8.1 Using the react testing library

Problem

There are many ways that you can test a React application. Early on in a project, when you are still defining the basic purpose and function of an application, you might choose to create tests in some very high-level form, such as Cucumber tests. If you are looking at some isolated piece of the system (such as the creation and maintenance of a data item) you might want o create functional tests using a tool like Cypress.

But if you are deep into the detail of creating a single component, then you will probably want to create unit tests. Unit tests are so called because they attempt to test a single piece of code as an isolated unit. While it’s debatable whether *unit test* is the correct term for testing components (which often contain sub-components and so are not isolated), it’s the name usually applied to tests of components that are tested outside of a browser.

But how do you unit test React components? There have histrically been several approaches. Early unit tests relied on rendering the component into an HTML string. This required very little testing infrastructure, but there were mutliple downsides:

- Managing re-renders when the component state changed,
- Making assertion on HTML elements that must first be parsed from a string
- Creating complex code to simulate UI interactions

It was not long before various libraries were created to take care of the details of each of these problems.

However, tests created in this way still lack the reality of tests created in browsers. The subtleties of the interaction between the virtual *Document Object Model* (DOM) and the browser DOM are lost. Often sub-components were not rendered to reduce the complexity of the tests.

The result was that React applications often had very few unit tests. Developers would refactor their code so that complex logic was moved into easily testable JavaScript functions. Anything more complex would have to be tested with a real browser, and that would lead to tests that were slow to run. Because they were slow, developers would be discouraged from testing too many scenarios.

So how can you unit test React components in a realistic way, but without the overheading of launching the entire app, and running the tests in a real browser?

Solution

The *testing-library* by Kent C. Dodds attempts to avoid the issues with previous unit testing libraries by providing a standalone implementation of the Document Object Model. This means that you React component can render to the virtual DOM, and that result can then be synchronized with the testing-library’s DOM and create a tree of HTML elements that behave in the same way that they would in a real browser.

You can inspect the elements in the same way that would within a browser. They have the same attributes and properties. You can even pass keystrokes to `input` fields and have them behave the same way as fields in the browser.

If you created your application with *create-react-app*, you should already have the *testing-library* installed. If not, you can install it from the command line:

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

These three libraries will allow us to unit test components.

The `@testing-library/react` library allows us to render components. To do this, it will use the DOM implementation in `@testing-library/jest-dom`. The `@testing-library/user-event` library greatly simplifies the process of interacting with the generate DOM elements. This is the library that will allow us to click the buttons and type in the fields of our components.

In order to show how to unit test components, we will need an application to test. We'll be using the same application through much of this chapter. When the application is launched, the user is asked to performance a simple calculation. They will be told if the answer they provide is right or wrong (see *figure 8-1*).

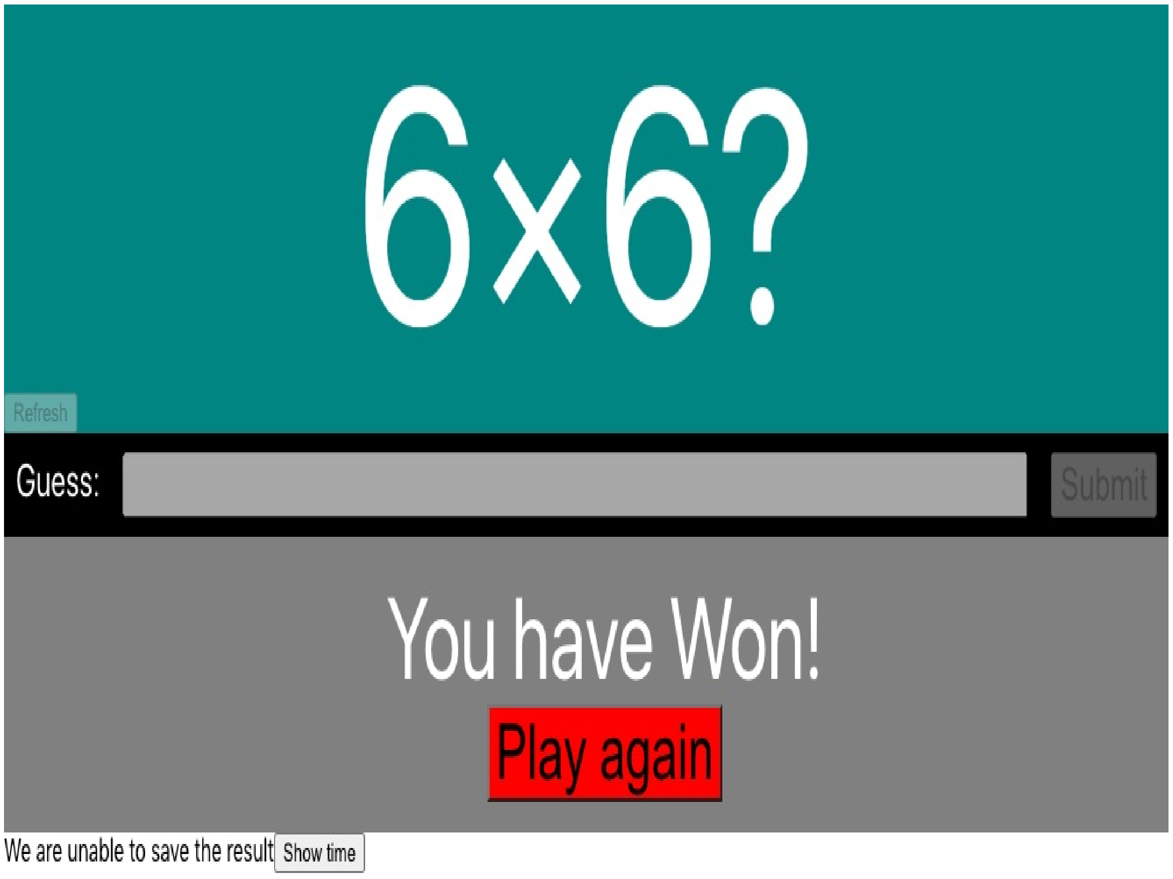


Figure 8-1. The application under test

The main component of the application is called `App`. We can create a unit test for this component by writing a new file called *App.test.js*

```
describe('App', () => {
    it('should tell you when you win', () => {
        // Given we've rendered the app
        // When we enter the correct answer
        // Then we are told that we've won
    })
});
```

This is a *Jest* test, with a single scenario that tests that the *App* component will tell us we’ve won, if we enter the correct answer. We’ve put placeholder comments for the structure of the test.

We will begin by rendering the *App* component. We can do this by importing the component and passing it to the testing library’s *render* function:

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

describe('App', () => {
    it('should tell you when you win', () => {
        // Given we've rendered the app
       render(<App/>);

       // When we enter the correct answer
       // Then we are told that we've won
    })
});
```

Notice that we pass actual *JSX* to the *render* function. This means that we could, if we wish, test the components behavior when passed different sets of properties.

For the next part of the test, we’ll need to enter the correct answer. In order to do that, we must first know what the correct answer is. The puzzle that user is shown is always a randomly generated multiplication, so we can capture the numbers from the page and then type in the product into the *Guess* field[1].

We will need to look at the elements generated by the *App* component. The *render* function returns and object that not only contains the elements, but also a set of functions for filtering them. Instead of using this returned value, we’ll instead use the testing-library’s *screen* object.

You can think of the *screen* object as the contents of the browser window. It allows use to find elements within the page so that we can interact with them. For example, if we want to find the input field labeled *Guess*, we can do it like this:

```
const input = screen.getByLabelText(/guess:/i);
```

The filter methods in the *screen* object typically begin with:

- *getBy…* if you know that the DOM contains a single instance of the matching element
- *queryBy…* if you know there are 0 or 1 elements that match
- *getAllBy…* if you know there are one or more matching elements (returns an array)
- *queryAllBy…* to find 0 or more elements (returns an array)

These methods will throw an exception if they find more or fewer elements that they were expecting. There are also *findBy…* and *findAllBy…* methods which are asynchronous versions of *getBy…* and *getAllBy…* which returns promises.

For each of these filter method types, you can search:

Function name ends	Description
…ByLabelText	Find matching field
…ByPlaceHolderText	Find with text
…ByText	With matching text content
…ByDisplayValue	Find by value
…ByAltText	Matching the alt attribute
…ByTitle	Matching the title attribute
…ByRoie	Find by aria role
…ByTestId	Find by data-testid attribute

This means there are nearly 50 ways to find elements within the page. However, you might have noticed that **none** of them use a CSS selector to track an element down. This is deliberate. The testing-library restricts the number of ways that you can find elements within the DOM. It doesn't allow you to, for example, find elements by class-name. This is to reduce the fragility of the test. Class names are frequently used for cosmetic styling and are subject to frequent change.

It is still possible to track down elements with selectors, by using the *container* returned by the render method:

```
const {container} = render(<App/>);
const theInput = container.querySelector('#guess');
```

But this approach is frowned upon. If you are using the testing-library, it’s probably best to follow the standard approach and find elements based upon their content.

There is one small concession to this approach made by the filter functions: the …*ByTestId* functions. If you really have no practical way of finding an element by its content, then you can always add a `data-testid` attribute to the relevant tag. That is useful for the test we are currently writing because we need to find two numbers displayed on the page. And these numbers are randomly generated, so we don’t know what their content is (*figure 8-2*).

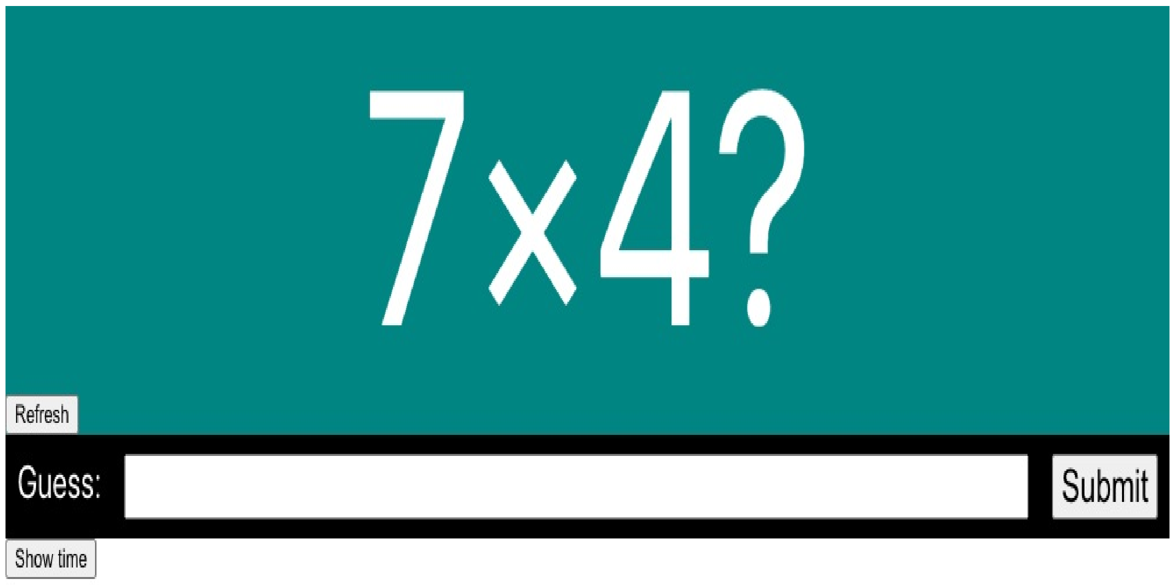


Figure 8-2. We cannot find the numbers by content, because we won’t know what they are

So we make a small amendment to the code, and add test-ids:

```
<div className='Question-detail'>
    <div data-testid='number1' className='number1'>{pair && pair[0]}
</div>
    &times;
    <div data-testid='number2' className='number2'>{pair && pair[1]}
</div>
```

```
      ?
  </div>
```

This means we can start to implement the next part of our test:

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

describe('App', () => {
    it('should tell you when you win', () => {
        // Given we've rendered the app
        render(<App/>);

        // When we enter the correct answer
        const number1 = screen.getByTestId('number1').textContent;
        const number2 = screen.getByTestId('number2').textContent;
        const input = screen.getByLabelText(/guess:/i);
        const submitButton = screen.getByText('Submit');
        // Err...

        // Then we are told that we've won
    })
});
```

We have the text for each of the numbers, and we have the `input` element. We now need to type the correct number into the field and then submit the answer. We'll do this with the `@testing-library/user-event` library. The user-event library simplifies the process of generating JavaScript events for HTML elements. You will often see the user-event library imported with the alias `user`. This is because you can think of the calls to the user-event library as the actions a user is making:

```
import {render, screen} from '@testing-library/react';
import user from '@testing-library/user-event';
import App from './App';

describe('App', () => {
    it('should tell you when you win', () => {
        // Given we've rendered the app
        render(<App/>);

        // When we enter the correct answer
        const number1 = screen.getByTestId('number1').textContent;
        const number2 = screen.getByTestId('number2').textContent;
        const input = screen.getByLabelText(/guess:/i);
        const submitButton = screen.getByText('Submit');
        user.type(input, '' + (parseFloat(number1) *
parseFloat(number2)));
        user.click(submitButton);

        // Then we are told that we've won
    })
});
```

Finally, we need to make an assertion that we have won. We can write this very simply by looking for some element containing the word “won”[2].

```
// Then we are told that we've won
screen.getByText(/won/i);
```

This assertion will work because *getByText* throws an exception if it does not find exactly one matching element.

TIP

If you are unsure about the current HTML state at some point in a test, try adding `screen.getByTestId('NONEXISTANT')` into the code. The exception that’s thrown will show you the current HTML.

However, it’s liable to break if your application is running slowly. This because the *get…* and *query…* functions look at the existing state of the DOM. If the result takes a couple seconds to appear, the assertion will fail. For this reason, it’s a good idea to make some assertions asynchronous. It makes the code a little more complex, but the test will be more stable when running against slow moving code.

The *find…* methods are asynchronous versions of the *get…* methods, and the testing-library’s *waitFor* will allow you to re-run code for a period of time. Combining the two functions together, we can create the final part of our test:

```
import {render, screen, waitFor} from '@testing-library/react';
import user from '@testing-library/user-event';
import App from './App';

describe('App', () => {
    it('should tell you when you win', async () => {
        // Given we've rendered the app
        render(<App/>);

        // When we enter the correct answer
        const number1 = screen.getByTestId('number1').textContent;
        const number2 = screen.getByTestId('number2').textContent;
        const input = screen.getByLabelText(/guess:/i);
        const submitButton = screen.getByText('Submit');
        user.type(input, '' + (parseFloat(number1) *
parseFloat(number2)));
        user.click(submitButton);

        // Then we are told that we've won
        await waitFor(() => screen.findByText(/won/i), {timeout:
4000});
    })
});
```

WARNING

Unit tests should run very quickly, but if some reason your test takes longer than 5 seconds, you will need to pass a second *timeout* value in milliseconds to the `it` function.

Discussion

Working with different teams, we found that early on in a project the developers would write unit tests for each of the components. But over time, they would write fewer and fewer unit tests. If we revisited them several months later we might even discover them removing some of the unit tests they had originally created.

This is partly because unit tests are more abstract than browser tests. They are doing the same kinds of things as browser tests, but they do them invisibly. When they are interacting with components, you don’t **see** them.

A second reason is that teams often see tests as deliverable artifacts within a project. The team might even have builds that fail if a certain percentage of the code isn’t covered by unit tests.

These issues generally disappear if developers write tests *before* they write code. If you write the tests first, a line at a time, you will have a much better grasp of the current state of HTML. If you stop seeing tests as development artifacts, and start to look at them as tools for designing your code, they stop becoming a time-consuming burden and start to become tools which make your work easier.

The important thing when writing code, is that you begin with a failing test. In the early days of a project, that might be a failing browser test. As the project matures, and the architecture stabilizes, you should find that you are creating more and more unit tests.

8.2 Use storybook for render tests

Problem

Tests are really nothing more or less than examples that you can execute. In that way, tests have a lot in common with component gallery systems like *Storybook*. Both tests and galleries are examples of components running in particular circumstances. Whereas a test will make assertions with a code, a developer will make an *assertion* of a library example by looking at it and checking that it appears as expected. In both galleries and tests, exceptions will be easily visible.

There are obvious differences. Tests can automatically interact with components; gallery components require a person to press buttons and type text. Tests can all be exercised with a single command; galleries have to be manually viewed, one example at a time. Gallery components are visual and easy to understand; tests are quite abstract and less fun to create.

Is there some way that galleries like Storybook can be combined with automated tests, to get the best of both worlds?

Solution

We’re going to look at how you can re-use your Storybook stories inside tests. You can install Storybook into your application with this command:

```
npx -p @storybook/cli sb init
```

The example application we are using in this chapter is a simple mathematical game, in which the user needs to calculate the answer to a multiplication (see *figure 8-3*).

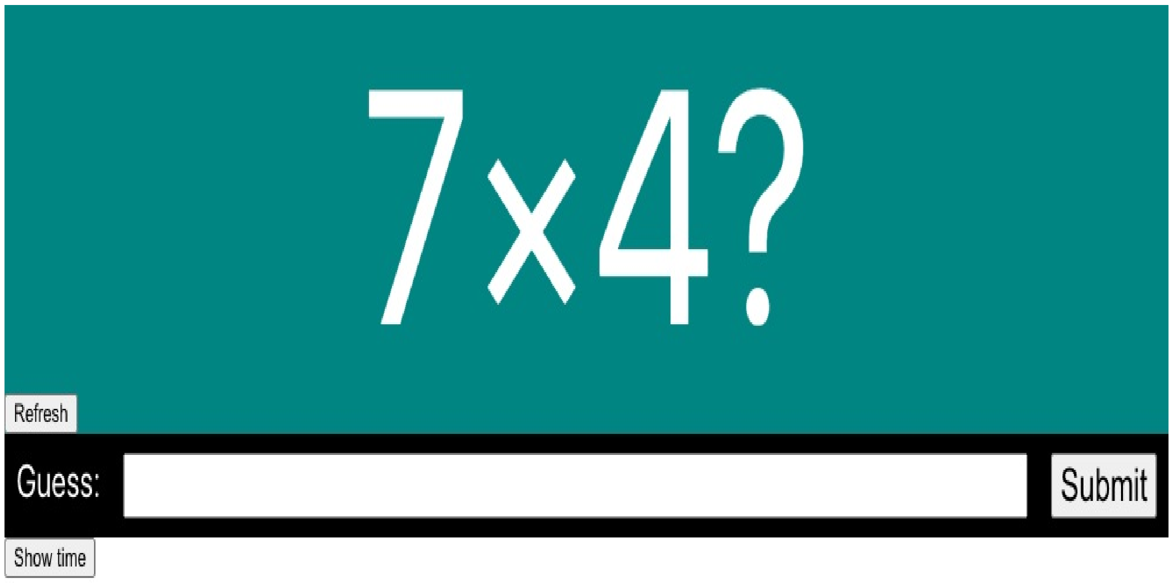


Figure 8-3. For more information on the game, see the downloadable source.

One of the components in the game is called *Question*, and it displays a randomly generated multiplication question (*figure 8-4*).

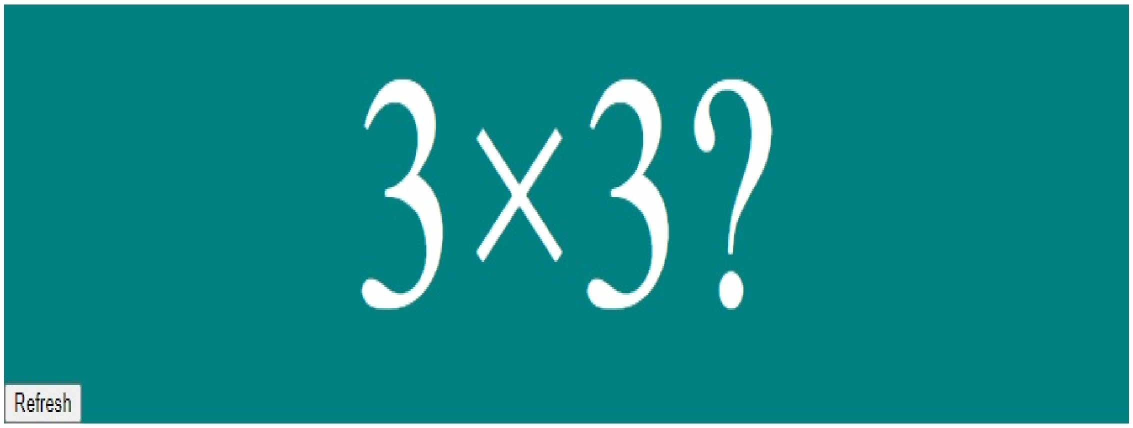


Figure 8-4. The Question component

Let’s say we don’t worry too much about tests for this component. Let’s just build it by creating some Storybook stories. We’ll write a new *Question.stories.js* file:

```
import Question from "./Question";

const Info = {
    title: 'Question'
};

export default Info;

export const Basic = () => <Question/>;
```

And then we’ll create an initial version of the component, that we can look at in Storybook and be happy with:

```
import {useEffect, useState} from 'react';
import './Question.css';

const RANGE = 10;

function rand() {
    return Math.floor((Math.random() * RANGE) + 1);
}

const Question = ({refreshTime}) => {
    const [pair, setPair] = useState();

    const refresh = () => {
        setPair(pair => {
            return [rand(), rand()];
        });
    };

    useEffect(refresh, [refreshTime]);

    return <div className='Question'>
        <div className='Question-detail'>
            <div data-testid='number1' className='number1'>{pair &&
pair[0]}</div>
            &times;
            <div data-testid='number2' className='number2'>{pair &&
pair[1]}</div>
            ?
        </div>
        <button
            onClick={refresh}
        >Refresh
        </button>
    </div>;
};

export default Question;
```

This component displays a randomly generated question if the user presses the *Refresh* button, or if a parent component passes in a new *refreshTime* value.

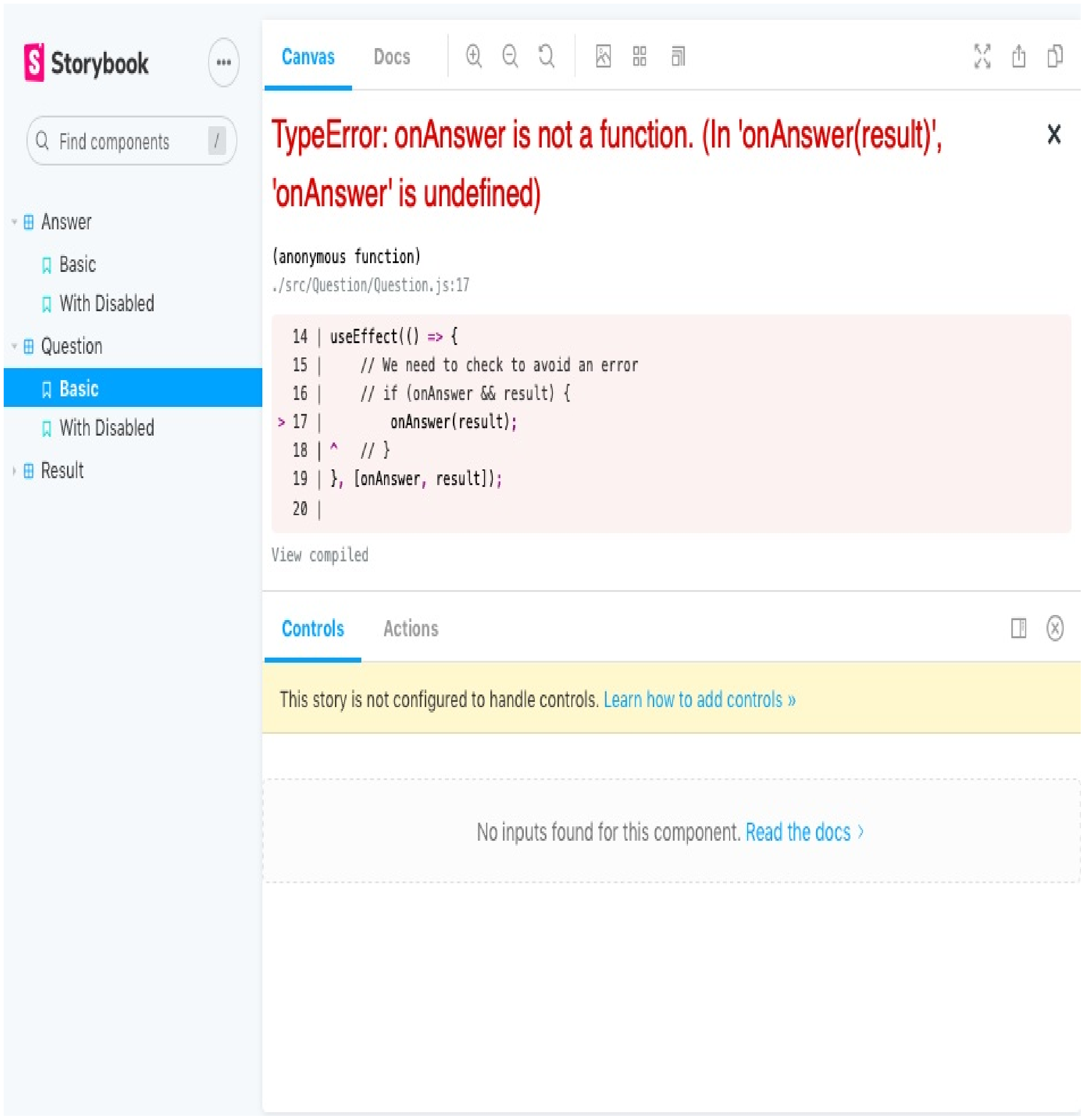
We display the component in Storybook, and it looks like it works fine. We can click the refresh button and it refreshes. So at that point we start to use the component in the main application. After a while, we start to add in a few extra features, but none of them are really visual changes, so we don't look at the Storybook stories for it again. After all, it will still look the same, right?

This is a modified version of the component, after we've wired it into the rest of the application:

```
import {useEffect, useState} from 'react';
import './Question.css';

const RANGE = 10;

function rand(notThis) {
    return Math.floor((Math.random() * RANGE) + 1);
}

const Question = ({onAnswer, refreshTime}) => {
    const [pair, setPair] = useState();
    const result = pair && (pair[0] * pair[1]);

    useEffect(() => {
       onAnswer(result);
    }, [onAnswer, result]);

    const refresh = () => {
        setPair(pair => {
            return [rand(), rand()];
        });
    };

    useEffect(refresh, [refreshTime]);

    return <div className='Question'>
        <div className='Question-detail'>
            <div data-testid='number1' className='number1'>{pair &&
pair[0]}</div>
            &times;
            <div data-testid='number2' className='number2'>{pair &&
pair[1]}</div>
            ?
        </div>
        <button
            onClick={refresh}
        >Refresh
        </button>
    </div>;
};

export default Question;
```

This version is only *slightly* longer than before. We’ve added in an `onAnswer` callback function that will return the correct answer to the parent component each time a new question is generated.

The new component appears to work well in the application, but then an odd thing occurs. The next time someone looks at Storybook, they notice an error as shown in *figure 8-5*.



TypeError: onAnswer is not a function. (In 'onAnswer(result)', 'onAnswer' is undefined)

```
(anonymous function)
./src/Question/Question.js:17
  14 | useEffect(() => {
  15 |     // We need to check to avoid an error
  16 |     // if (onAnswer && result) {
> 17 |         onAnswer(result);
  18 | ^   // }
  19 | }, [onAnswer, result]);
  20 |
View compiled
```

This story is not configured to handle controls. Learn how to add controls »

No inputs found for this component. Read the docs ›

Figure 8-5. An error occurs when we look at the new version of the component.

What happened? We’ve added an implicit assumption into the code that the parent component will always pass an *onAnswer* callback into the component. Because the Storybook stories rendered *Basic* story without an *onAnswer*, we got the error:

```
<Question/>
```

Does this really matter? Not for a simple component like this. After all, the application itself still worked. But failure to cope with missing properties, such as the missing callback here or more frequently missing data, is one of the commonest causes of errors in React.

React properties are frequently generated using data from the network, and that means that the initial properties you pass to components will often be null or undefined. It's generally a good idea to either use a type-safe language, like Typescript, to avoid this issues, or else to write tests that check that your components can cope with missing properties.

We created this component without any tests, but we did create it with a Storybook story–and that story *did* catch the issue. So is there some way that we can write a test that will automatically check that all the Storybook stories can be rendered?

We're going to create a test for this component in a file called *Question.test.js*

TIP

Consider creating a folder for each component. Instead of simply having a file called *Question.js* in the *src/* directory, create a folder called *src/Question/*, and inside there you can place *Question.js*, *Question.stories.js* and *Question.test.js*. If you then add an *src/Question/index.js* file, which does a default export of the *Question* component, the rest of your code will be unaffected, and you will reduce the number of files other developers have to deal with[3].

In the test file we can then create a Jest test that loads each of the stories, and then passes them to the testing-library *render* function[4].

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

const stories = require('./Question.stories');

describe('Question', () => {
    it('should render all storybook stories without error', () => {
        for (let story in stories) {
            if (story !== 'default') {
                let C = stories[story];
                render(<C/>);
            }
        }
    });
});
```

WARNING

If your stories are using *decorators* to provide such things as routers or styling, this technique will not pick them up automatically. You should add them into the *render* method within the test.

When you run this test, you will get a failure:

```
onAnswer is not a function
TypeError: onAnswer is not a function
```

We can fix the error by checking if there is a callback before calling it:

```
useEffect(() => {
    // We need to check to avoid an error
    if (onAnswer && result) {
        onAnswer(result);
    }
}, [onAnswer, result]);
```

This technique you to create some extremely basic tests for a component with very little effort. It’s worth creating a story for the component which includes no properties whatsoever. Then, before you add a new property, create a story that uses it and think about how you will expect the component to behave.

Even though the test will only perform a simple render of each story, there is no reason why you can’t import a single story and create a test using that story:

```
import {render, screen} from '@testing-library/react';
import user from '@testing-library/user-event';
import Question from "./Question";
import {Basic, WithDisabled} from './Question.stories'
...
it('should disable the button when asked', () => {
  render(<WithDisabled/>);
  const refreshButton = screen.getByRole('button');
  expect(refreshButton.disabled).toEqual(true);
});
```

Discussion

This is a technique for introducing very rudimentary unit testing into your application and in practice we've found that it can find a surprising number of regression bugs. It also helps you think of tests as examples, which are there to help you design your code[5] rather than coding artifacts that must be produced to keep the team-lead happy[6]. Creating render tests for stories is also useful if you have a team who are new to unit testing. By creating visual examples it avoids the problems that can arise from non-visual tests feeling very abstract. It can also get developers into the habit of having a test file for each component in the system. At some future point where a small change needs to be made on the component, it will then be much easier to add a small unit test function before adding the change.

8.3 Using Cypress for network testing

Problem

One of the principle features of high quality code is in the way it responds to errors. The first of Peter Deutsch's Eight Fallacies of Distributed Computing is: *The network is reliable*. Not only is the network *not* reliable, neither are the servers or databases that are connected to it. At some point your application is going to have to deal with some kind of network failure. It might be that the phone loses its connection, or the server goes down, or the database crashes, or the data you are updating has just been deleted by somebody else. Whatever the causes, you will need to decide what your application will do when terrible things happen.

Network issues can be extremely difficult to simulate in testing environments. If you write code that puts the server into some error state, that is likely to causes problems for other tests or users who are connected to the server.

How can you create automated tests for network failure cases?

Solution

For this recipe we are going to use *Cypress*. We mentioned the Cypress testing system back in chapter 1. It's a truly remarkable testing system which is rapidly becoming our go-to tool in many development projects.

To install Cypress into your project, type the following:

```
npm install --save-dev cypress
```

Cypress works by automating a web browser. In that sense, it is similar to other systems like Selenium, but the difference is that Cypress does not require you to install a separate driver, and it has the ability to both run the browser remotely, and also it can inject itself into the JavaScript engine of the browser.

The reason this is significant, is that Cypress can actively replace core parts of the JavaScript infrastructure with faked versions which it can control. For example, Cypress has the ability to replace the JavaScript `fetch` function, which is used[7] to make network calls to the server. Cypress tests can therefore spoof the behavior of a network server and allow a client-side developer to artificially craft responses from the server.

For this recipe we will use the example game application that we use for other recipes in this chapter. We will add a network call to store the result each time a user answers a question. We can do this without creating the actual server code, by faking the responses in Cypress.

In order to show how this works, we will first create a test which simulates the server responding correctly. Then we will create a test to simulate the server failing.

Once Cypress is installed, create a file in *cypress/integration/* called *0001-basic-game-functions.js*[8]:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        // When I enter an incorrect answer
        // Then the server will be told that I have lost
    });
});
```

We’ve put placeholder comments for each of the steps we will need to write.

Each command and assertion in Cypress begins with *cy.*. If we want to open the browser at location *http://localhost:3000* we can do it with:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        cy.visit('http://localhost:3000');

        // When I enter an incorrect answer
        // Then the server will be told that I have lost
    });
});
```

To run the test we can either type:

```
npx cypress run
```

This will run all tests without showing the browser[9] or we can type:

```
npx cypress open
```

This will open the Cypress application window (as you can see in *figure 8-6*). If we double-click the test file, the test will open in a browser (as you can see in *figure 8-7*)

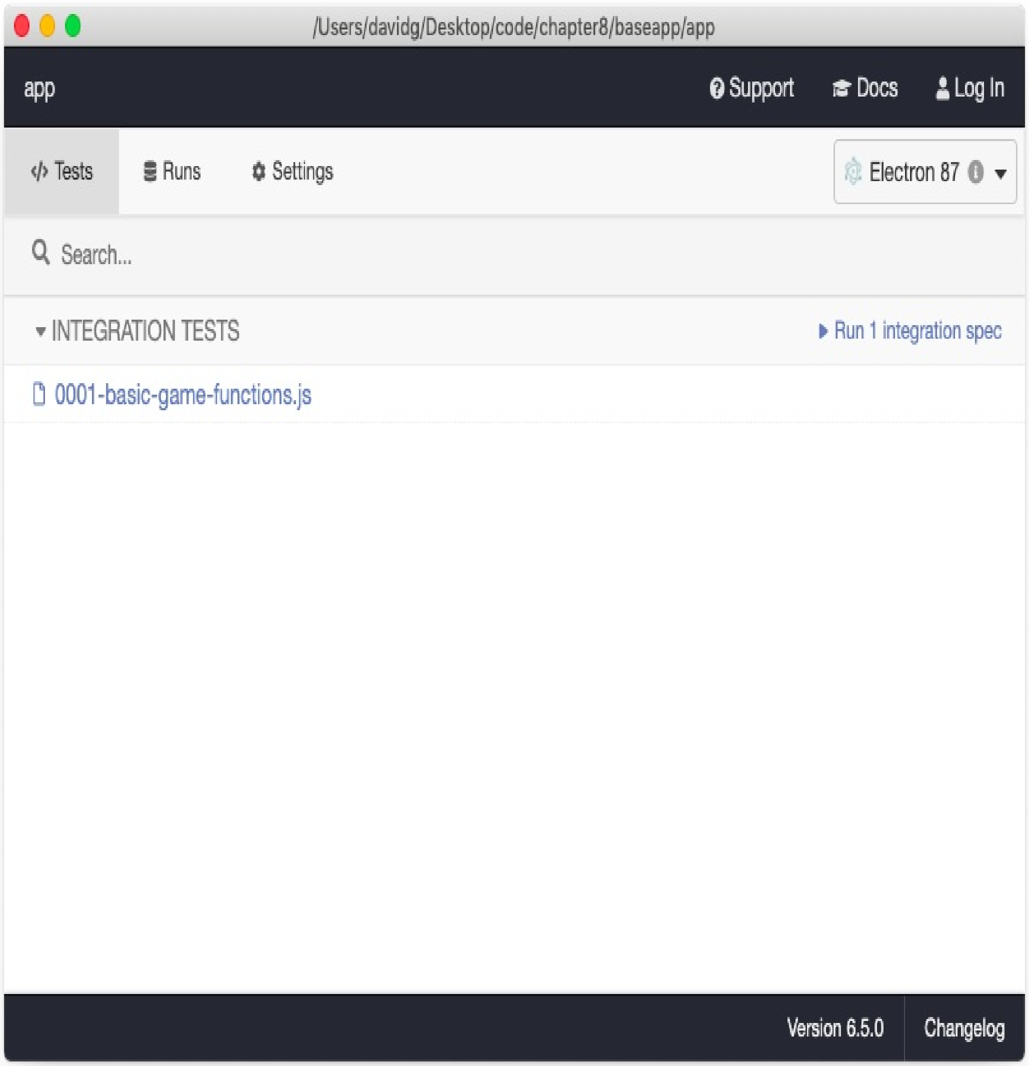


Figure 8-6. The test will appear in the Cypress window when you type npx cypress open

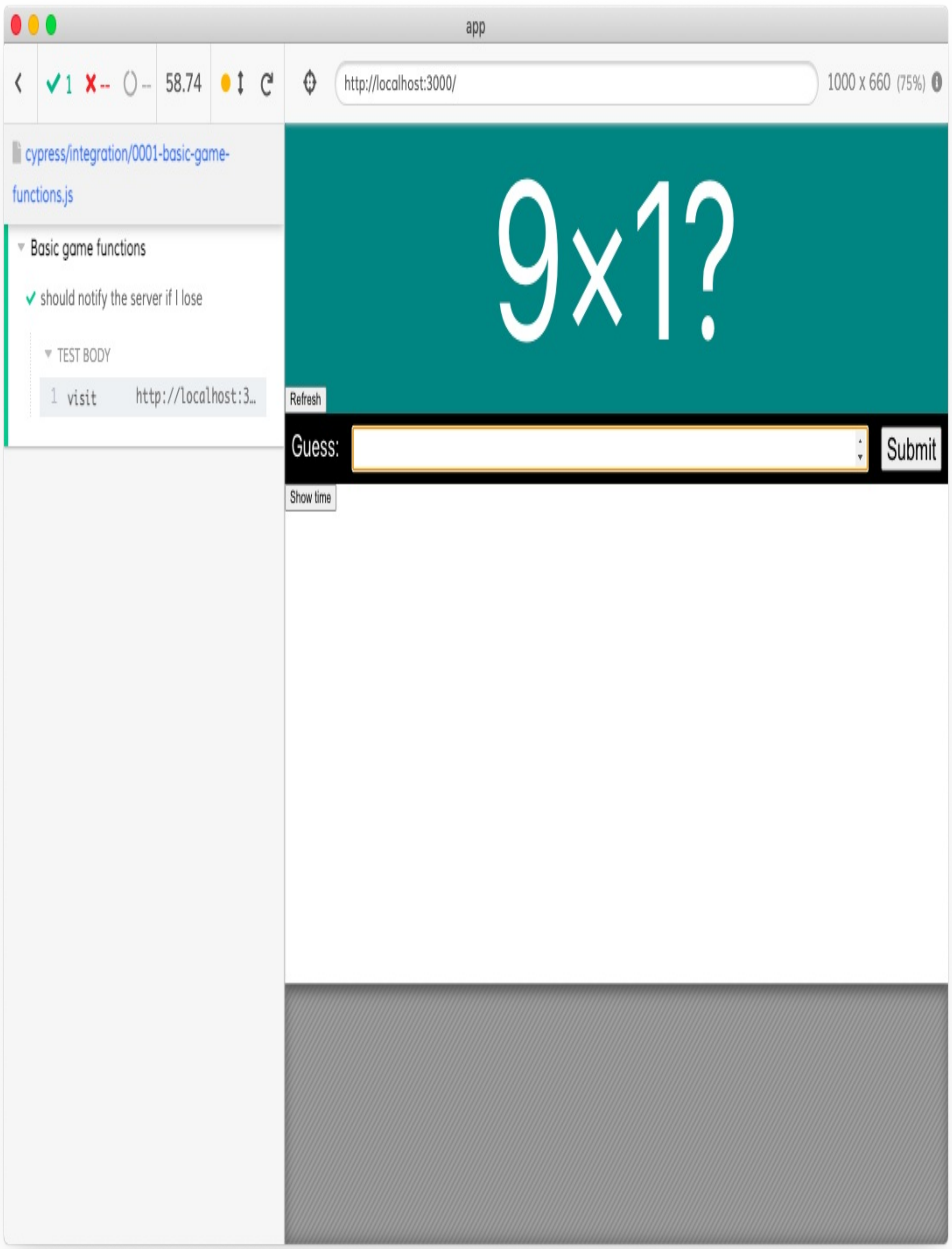


Figure 8-7. Cypress will run a test running in a browser

The example application asks the user to perform a multiplication of two random numbers (see *figure 8-8*). The numbers will be in the range 1..10, so if we enter the value *101* we can be sure that the answer will be incorrect.

NOTE

Cypress does not allow to capture textual content from the screen directly. So we cannot simply read the values of the two numbers and store them in variables. This is because the commands in Cypress do not immediately perform the actions in the browser. Instead, when you run a command, Cypress adds it to a *chain* of instructions which are performed at the end of the test. This might seem a little odd, but these *chainable* instructions[10] allow Cypress to handle most of the problems caused by asynchronous interfaces. The downside is that no command can return the contents of the page as the page will not exist at the time the command is run. We will see elsewhere in this chapter how we can remove randomness in test scenarios and make this test deterministic.

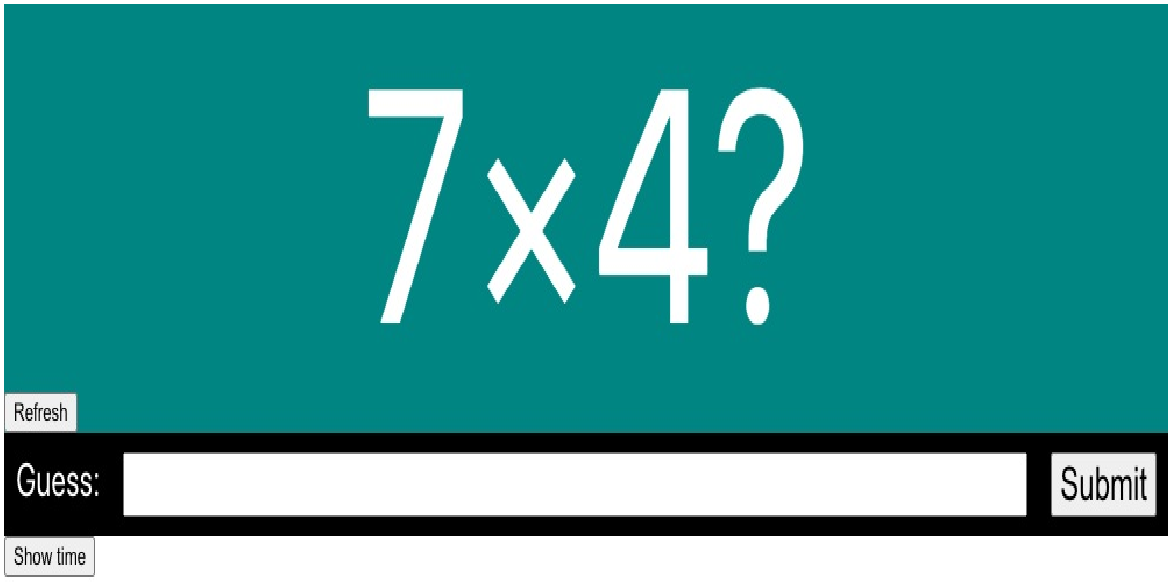


Figure 8-8. The application asks the user to calculate the product of two random numbers

We can use the *cy.get()* command to find the input-field by a CSS selector. We can also use the *cy.contains()* command to find the *Submit* button:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        cy.visit('http://localhost:3000');
        // When I enter an incorrect answer
        cy.get('input').type('101');
        cy.contains('Submit').click();

        // Then the server will be told that I have lost
    });
});
```

Now we just need to test that the application contacts the server with the result of the game.

We will use the *cy.intercept()* command to do this. The *cy.intercept()* command will change the behavior of network requests in the application so that we can fake responses for a given request. If the result is going to be POSTed to the endpoint //api/result/ we generate a faked response like this:

```
cy.intercept('POST', '/api/result', {
    statusCode: 200,
    body: ''
});
```

Once this command takes effect, network requests to //api/result/ will receive the fakes response. That means we need to run the command *before* the network request is made. We will do it at the start of the test:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        cy.intercept('POST', '/api/result', {
            statusCode: 200,
            body: ''
        });
        cy.visit('http://localhost:3000');

        // When I enter an incorrect answer
        cy.get('input').type('101');
        cy.contains('Submit').click();

        // Then the server will be told that I have lost
    });
});
```

We’ve now specified the network response. But how do we assert that the network call has been made, and how do we know that it has sent the correct data the //api/result/ endpoint?

We will need to given the network request an *alias*. This will allow us to refer to the request later in the test[11]:

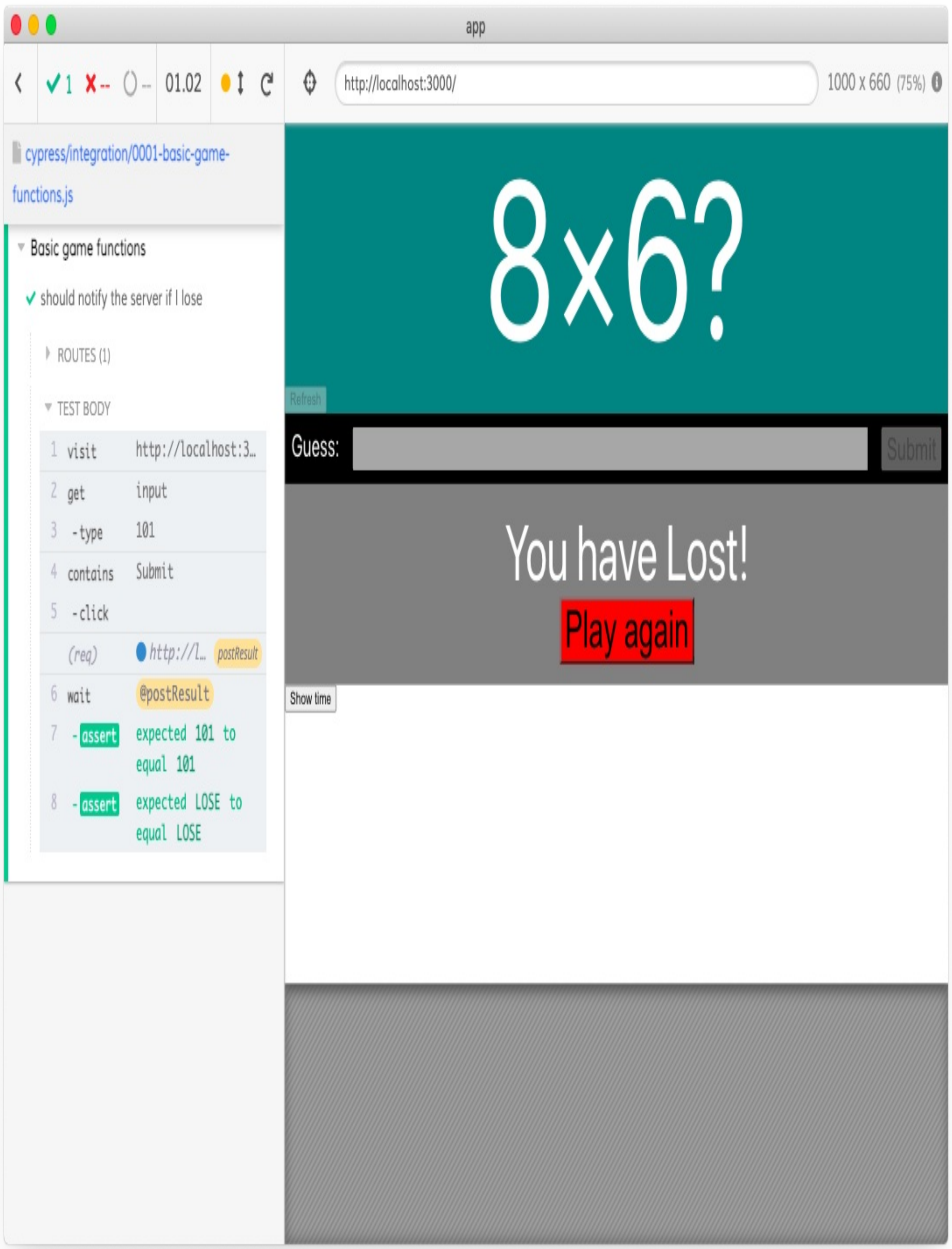
```
cy.intercept('POST', '/api/result', {
    statusCode: 200,
    body: ''
}).as('postResult');
```

We can then make an assertion at the end of the test, which will wait for the network call to be made, and will check the contents of the data sent in the request body:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        cy.intercept('POST', '/api/result', {
            statusCode: 200,
            body: ''
        }).as('postResult');
        cy.visit('http://localhost:3000');

        // When I enter an incorrect answer
        cy.get('input').type('101');
        cy.contains('Submit').click();

        // Then the server will be told that I have lost
        cy.wait('@postResult').then(xhr => {
            expect(xhr.request.body.guess).equal(101)
            expect(xhr.request.body.result).equal('LOSE')
        });
    });
});
```

This assertion is checking two of the attributes of the request body for the expected values.

If we run the test now, it will pass (as you can see in *figure 8-9*)



cypress/integration/0001-basic-game-functions.js

Basic game functions

✓ should notify the server if I lose

ROUTES (1)

TEST BODY

```
1 visit      http://localhost:3...
2 get        input
3 -type      101
4 contains   Submit
5 -click
  (req)      http://l... postResult
6 wait       @postResult
7 -assert    expected 101 to
             equal 101
8 -assert    expected LOSE to
             equal LOSE
```

Figure 8-9. The completed test passes.

Now that we’ve created a test for the successful case, we can now write a test for the failure case. The application should display a message on screen if the network call fails. We don’t actually care what details are sent to the server in this test, but we still need to wait for the network request to complete before checking for the existence of the error message:

```
it('should display a message if I cannot post the result', () => {
    // Given I started the application
    cy.intercept('POST', '/api/result', {
        statusCode: 500,
        body: {message: 'Bad thing happened!'}
    }).as('postResult');
    cy.visit('http://localhost:3000');

    // When I enter an answer
    cy.get('input').type('16');
    cy.contains('We are unable to save the
result').should('not.exist');
    cy.contains('Submit').click();

    // Then I will see an error message
    cy.wait('@postResult');
    cy.contains('We are unable to save the result');
});
```

Notice that we check for the error message *not* existing before we make the network call, just to ensure that the network call *causes* the error.

In addition to generating stubbed responses and status codes, *cy.intercept()* can perform other tricks, such as slowing response times, throttling network speed or even generating responses from test functions. For further details, see the cy.intercept documentation

Discussion

Cypress testing can transform the way a development team works, specifically in its ability to mock network calls. APIs are often reused across an application and this can mean they are developed at a different cadence to front-end code. Some teams may even has developers who specialize in front-end or server code. Cypress allows front-end developers to write code against network that do not currently exist, as well as handling all of the pathological failure cases.

One area of network code that can introduce very subtle bugs if network performance. Development environments are generally configured to use local servers with little or no data in them. This means that API performance is far better at development time that it is in a more realistic environment. It is very easy to write code that assumes that data is immediately available. This code is likely to break in a realistic environment where the data may take a second or so to arrive.

It is therefore worth having at least one test for each API call where the response is slowed by a second or so:

```
cy.intercept('GET', '/api/widgets', {
    statusCode: 200,
    body: [{id: 1, name: 'Flange'}],
    delay: 1000
}).as('getWidgets');
```

This will often flush out a whole plethora of asynchronous bugs that might otherwise creep into your code.

Almost as importantly, creating artificially slow network responses will give you a sense of the impact on overall performance of each of the API calls.

8.4 Use Cypress for testing offline

Problem

This recipe uses a custom Cyporess command invented by Etienne Bruines.

Applications need to cope with being disconnected from the network. We've seen elsewhere[12] that we can create a hook to detect if we are currently offline. But how are we test for offline behavior?

Solution

We can simulate offline working using Cypress. Cypress tests have the ability to inject code that modifies the internal behavior of the browser under test. We should therefore be able to modify the network code in order to simulate offline conditions.

For this recipe, you will need to install Cypress in your application. If you don't already have Cypress, you can install it by running this command in your application directory.

```
npm install --save-dev cypress
```

You can then add a *0002-offline-working.js* file to the //cypress/integration/ directory:

```
describe('Offline working', () => {
    it('should tell us when we are offline', {browser: '!firefox'}, ()
=> {
        // Given we have started the application
        // When the application is offline
        // Then we will see a warning
        // When the application is back online
        // Then we will not see a warning
    });
});
```

WARNING

This test will be ignored if the test is run with Firefox. The offline-simulation code relies upon the remote debugging protocol which is not currently available in the Firefox browser.

We have marked out the structure of the test as a series of comments. Cypress commands all begin with *cy.*., so we can open the application like this:

```
describe('Offline working', () => {
    it('should tell us when we are offline', {browser: '!firefox'}, ()
=> {
        // Given we have started the application
        cy.visit('http://localhost:3000');

        // When the application is offline
        // Then we will see a warning
        // When the application is back online
        // Then we will not see a warning
    });
});
```

The question is, how do we force the browser to simulate offline working?

We can do it because Cypress is designed to be extensible. We can add a custom Cypress command that will allow to go offline and back online:

```
cy.network({ offline: true });
cy.network({ offline: false });
```

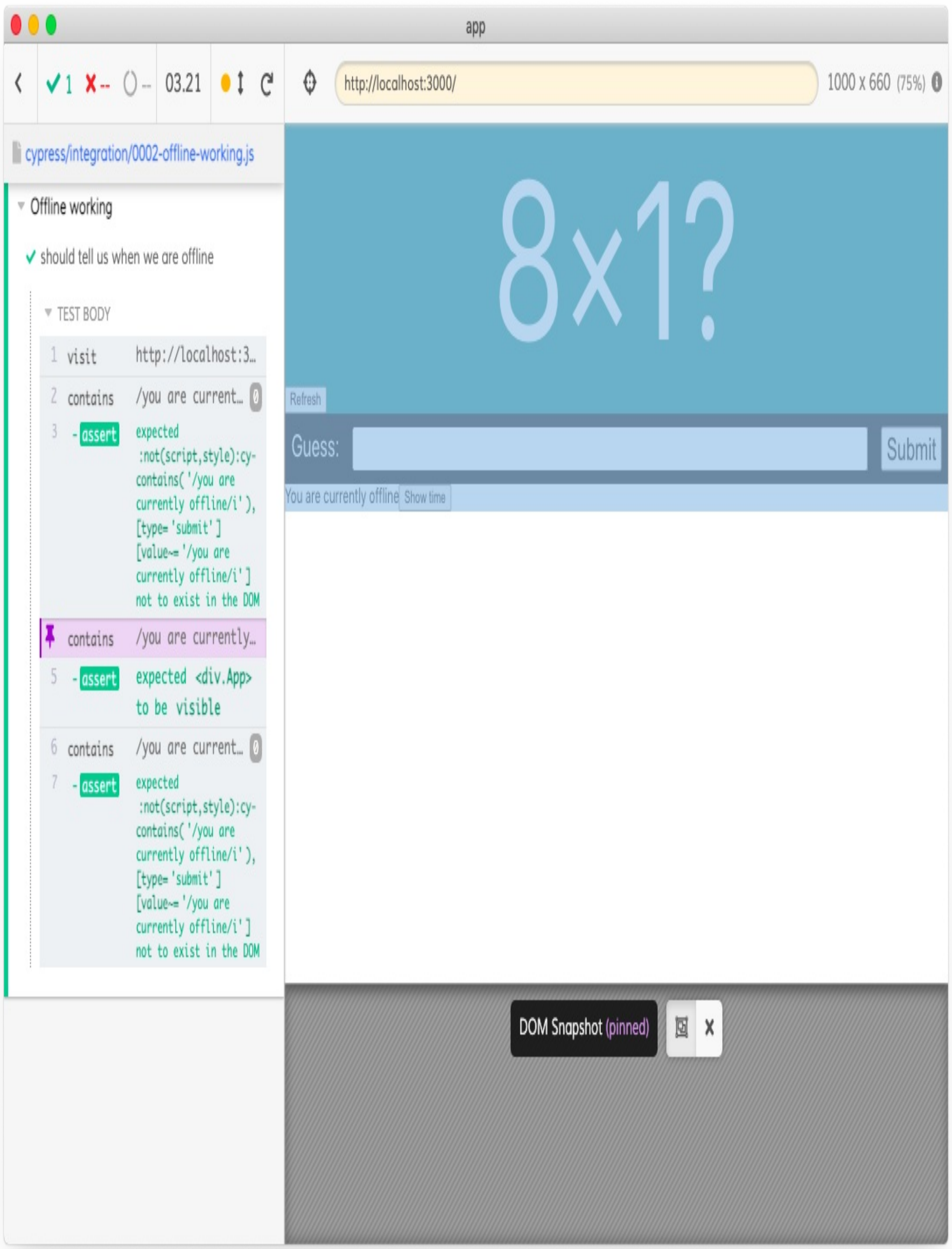
To add a custom command, open the //cypress/support/commands.js/ file, and add the following code:

```
Cypress.Commands.add('network', (options = {}) => {
        Cypress.automation('remote:debugger:protocol', {
            command: 'Network.enable',
        })

        Cypress.automation('remote:debugger:protocol', {
            command: 'Network.emulateNetworkConditions',
            params: {
                offline: options.offline,
                'latency': 0,
                'downloadThroughput': 0,
                'uploadThroughput': 0,
                'connectionType': 'none',
            },
        })
    }
);
```

This command uses the remote debugging protocol in DevTools to emulate offline network conditions. Once you have saved this file, you can then implement the rest of the test:

```
describe('Offline working', () => {
    it('should tell us when we are offline', {browser: '!firefox'}, ()
=> {
        // Given we have started the application
        cy.visit('http://localhost:3000');
        cy.contains(/you are currently offline/i).should('not.exist');

        // When the application is offline
        cy.network({ offline: true });

        // Then we will see a warning
        cy.contains(/you are currently
  offline/i).should('be.visible');

        // When the application is back online
        cy.network({ offline: false });

        // Then we will not see a warning
        cy.contains(/you are currently offline/i).should('not.exist');
    });
  });
```

If you run the test now, in a browser like Electron, it will pass (see *figure 8-10*.)



```
cypress/integration/0002-offline-working.js

▼ Offline working
  ✓ should tell us when we are offline

    ▼ TEST BODY
    1  visit       http://localhost:3…
    2  contains    /you are current… 0
    3  - assert    expected
                    :not(script,style):cy-
                   contains('/you are
                   currently offline/i'),
                   [type='submit']
                   [value~='/you are
                   currently offline/i']
                   not to exist in the DOM
       contains    /you are currently…
    5  - assert    expected <div.App>
                   to be visible
    6  contains    /you are current… 0
    7  - assert    expected
                    :not(script,style):cy-
                   contains('/you are
                   currently offline/i'),
                   [type='submit']
                   [value~='/you are
                   currently offline/i']
                   not to exist in the DOM
```

Figure 8-10. The online/offline test. You can view each stage by clicking in the left panel.

Discussion

It should be possible to create similar commands that simulate various network conditions. This would make it possible to simulate a user opening your application whilst connected to a cell network, then losing the connection as they wonder into a subway. and then reestablishing their connection when they come in range of wifi.

For more information on how the network command works, see the Cypress.io blog article.

8.5 Use Selenium for browser-based testing

Problem

Nothing beats running your code inside a real browser, and the most common way of writing automated browser-based tests is by using a *web driver*. Most browsers can be controlled by sending a command to a network port. Different browsers have different commands, and a web driver is a command line tool which simplifies the process of controlling the browser.

But how can we write a test for a React application that uses a web driver?

Solution

In this recipe we are going to use the *Selenium* library. Selenium is a framework that provides a consistent API for a whole set of different web drivers. This means that you can write a test for Firefox, and the same code should work in the same way for Chrome, Safari and Edge[13].

In this recipe we are going to use the same example application that we are using for all recipes in this chapter. It's a game that asks the user for the answer to a simple multiplication.

The Selenium library is available for a whole set of different lanuages, such as Python Java and C#. We will be using the JavaScript version: *SeleniumJS*.

We'll begin by installing Selenium:

```
npm install --save-dev selenium-webdriver
```

We will also need to install at least one web driver. Web drivers at the operating system level, but it more manageable to make the installation part of your application. We could install a driver like *geckodriver* for Firefox, but for now we will install *chromedriver* for Chrome:

```
npm install --save-dev chromedriver
```

We can now start to create a test. It’s useful to include Selenium tests inside the *src_ folder of the application, because it will make it easier to use an IDE to run the tests manually. So we’ll create a folder called //src/selenium_ and then add a file inside it called* 0001-basic-game-functions.spec.js__[14]:

```
describe('Basic game functions', () => {
    it('should tell me if I won', () => {
        // Given I have started the application
        // When I enter the correct answer
        // Then I will be told that I have won
    });
});
```

We have sketched out the outline a test in the comments.

TIP

Whilst it’s convenient to include Selenium tests in the main *src* tree, it will mean that a tool like *Jest* will run it as if it were a unit test. This is a problem if you leave a process running tests in the background. For example, if you created your application with *create-react-app* and leave a *npm run test* command running, you will find that a browser will suddenly appear on your screen each time you save the Selenium test. To avoid this, adopt some sort of naming convention to distinguish between Selenium and unit tests. If you name all your Selenium tests **.spec.js*, you can modify your test script to avoid them by setting it to *react-scripts test ‘.*.test.js’*

Selenium uses a web driver to automate the web browser. We can create an instance of the driver at the start of each test:

```
import {Builder} from "selenium-webdriver";
let driver;

describe('Basic game functions', () => {
    beforeEach(() => {
        driver = new Builder().forBrowser('chrome').build();
    });

    afterEach(() => {
        driver.quit();
    });

    it('should tell me if I won', () => {
        // Given I have started the application
        // When I enter the correct answer
        // Then I will be told that I have won
    });
});
```

In this example, we are creating a *Chrome* driver.

NOTE

By creating a driver for each test, we will also create a fresh instance of the browser for each test. This ensures that no browser state is carried between tests. This is important because it will allow us to run the tests in any order. We have no such guarantee on shared server state. If your tests are reliant upon, for example, database data, you should ensure that each test initializes the server correctly when it starts.

In order for Selenium to create an instance of the driver, we should also explicitly *require* the driver. This will ensure that Selenium can find it in $node_{modules}$:

```
import {Builder} from "selenium-webdriver";
require('chromedriver');

let driver;

describe('Basic game functions', () => {
    beforeEach(() => {
        driver = new Builder().forBrowser('chrome').build();
    });

    afterEach(() => {
        driver.quit();
    });

    it('should tell me if I won', () => {
        // Given I have started the application
        // When I enter the correct answer
        // Then I will be told that I have won
    });
});
```

We can now start to fill out the test. The JavaScript version of Selenium is highly asynchronous. Virtually all commands return promises, which means that it is very efficient, but it is also extremely easy to introduce testing bugs.

Let’s begin our test by opening the application:

```
import {Builder} from "selenium-webdriver";
require('chromedriver');

let driver;

describe('Basic game functions', async () => {
    beforeEach(() => {
        driver = new Builder().forBrowser('chrome').build();
    });

    afterEach(() => {
        driver.quit();
    });

    it('should tell me if I won', () => {
        // Given I have started the application
        await driver.get('http://localhost:3000');
        // When I enter the correct answer
        // Then I will be told that I have won
    }, 60000);
});
```

The *driver.get* command tells the browser to open the given URL. In order for this to work, we’ve also had to make two other changes. First, we’ve had to mark the test function with *async*. This will allow us to *await* the promise returned by *driver.get*. Second, we’ve added a time-out value of 60,000 milliseconds to the test. This is almost far longer than you will need, but it’s to avoid the implicit 5 second limit of Jest tests. Without it, you will find your test failing before it starts.

In order to enter the correct value into game, we will need to read the two numbers that appear in the question (as shown in *figure 8-11*)

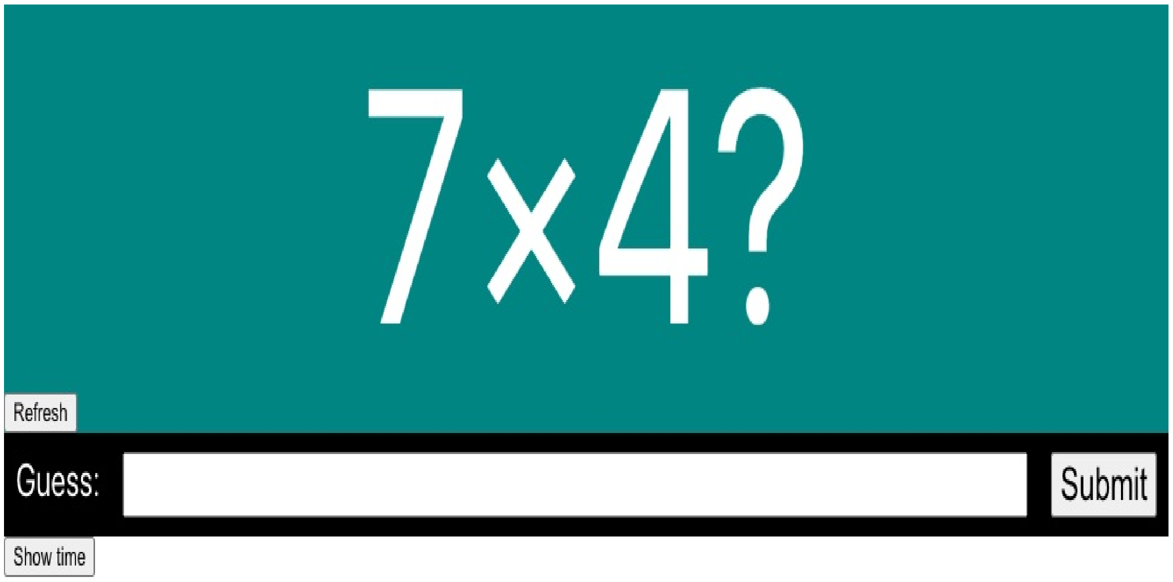


Figure 8-11. The game asks the user to calculate a random product.

We can find the two numbers on the page, and the *input* and *submit* buttons using a command called *findElement*:

```
const number1 = await
driver.findElement(By.css('.number1')).getText();
const number2 = await
driver.findElement(By.css('.number2')).getText();
const input = await driver.findElement(By.css('input'));
const submit = await
driver.findElement(By.xpath('//button[text()='Submit']'));
```

If you are ever reading a set of elements from the page, and don't care about resolving them in a strict order, you can use the *Promise.all* function to combine them into a single promise which you can then await:

```
const [number1, number2, input, submit] = await Promise.all([
    driver.findElement(By.css('.number1')).getText(),
    driver.findElement(By.css('.number2')).getText(),
    driver.findElement(By.css('input')),
    driver.findElement(By.xpath('//button[text()='Submit']'))
]);
```

In the example application, this optimization will save virtually no time, but if you have a page that renders different components in uncertain orders, combining the promises can improve test performance.

This means we can now complete the next part of our test:

```
import {Builder, By} from "selenium-webdriver";
require('chromedriver');

let driver;

describe('Basic game functions', async () => {
    beforeEach(() => {
        driver = new Builder().forBrowser('chrome').build();
    });

    afterEach(() => {
        driver.quit();
    });

    it('should tell me if I won', () => {
        // Given I have started the application
        await driver.get('http://localhost:3000');
        // When I enter the correct answer
        const [number1, number2, input, submit] = await Promise.all([
            driver.findElement(By.css('.number1')).getText(),
            driver.findElement(By.css('.number2')).getText(),
            driver.findElement(By.css('input')),
            driver.findElement(By.xpath('//button[text()='Submit']'))
        ]);
        await input.sendKeys('' + (number1 * number2));
        await submit.click();
```

```
            // Then I will be told that I have won
        }, 60000);
    });
```

Notice that we are not combining the promises returned by *sendKeys* and *click*, because we care that the answer is entered into the input field *before* we submit it.

Finally, we want to make the assertion that a *You have won!* message appears on the screen (see *figure 8-12*.)

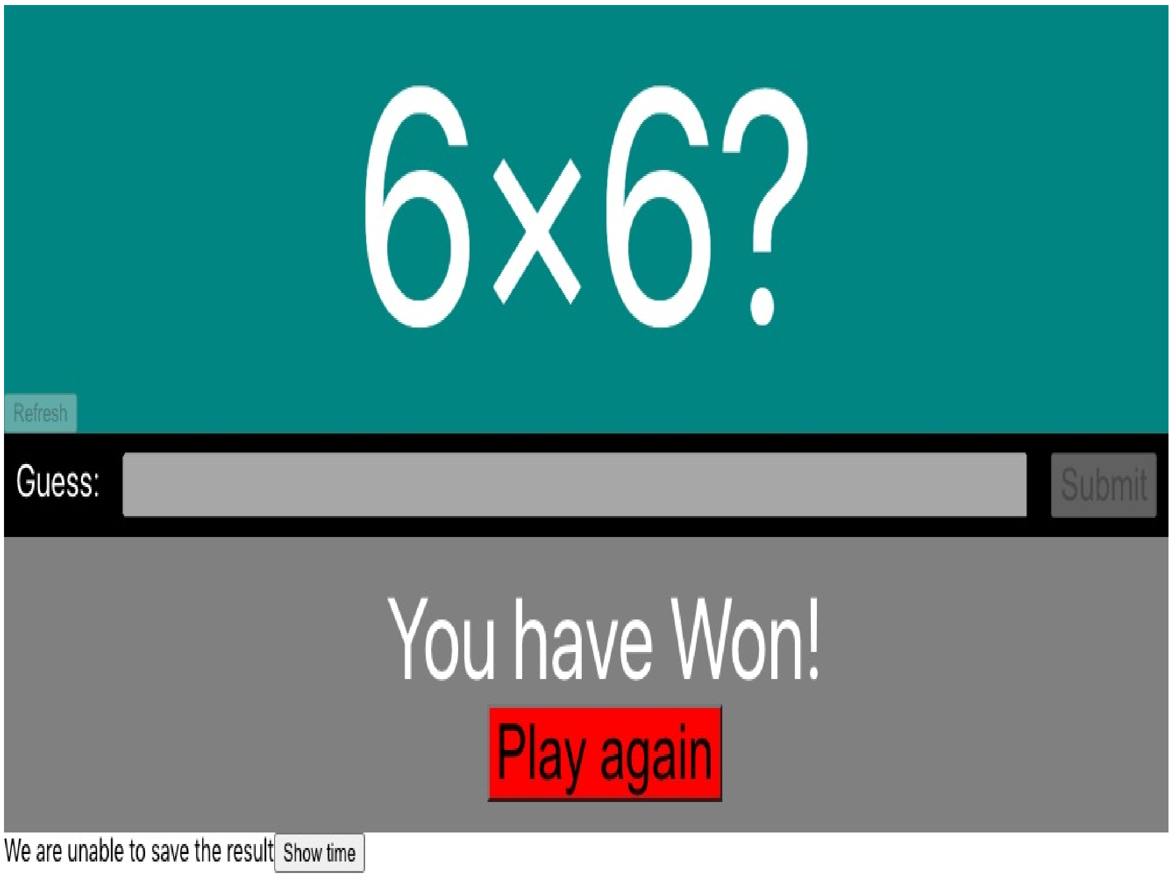


Figure 8-12. The user will be told if they got the correct answer.

Now we could write our assertion like this:

```
const resultText = await
driver.findElement(By.css('.Result')).getText();
expect(resultText).toMatch(/won/i);
```

This code will almost certainly work, because the result is displayed very quickly after the user submits an answer. Quite often, React applications will display dynamic results slowly, particularly if they rely upon data from the network. If we modify the application code to simulate a 2 second delay before the result appears[15], our test will produce the following error:

```
no such element: Unable to locate element: {"method":"css
selector","selector":".Result"}
  (Session info: chrome=88.0.4324.192)
NoSuchElementError: no such element: Unable to locate element:
{"method":"css selector","selector":".Result"}
  (Session info: chrome=88.0.4324.192)
```

We can avoid this problem by waiting until the element appears on the screen, and then waiting until the text matches the expected result. We can dow both of those things with the *driver.wait* function:

```
await driver.wait(until.elementLocated(By.css('.Result')));
const resultElement = driver.findElement(By.css('.Result'));
await driver.wait(until.elementTextMatches(resultElement, /won/i));
```

This gives us the final version of our test:

```
import {Builder, By} from "selenium-webdriver";
require('chromedriver');

let driver;

describe('Basic game functions', async () => {
    beforeEach(() => {
        driver = new Builder().forBrowser('chrome').build();
    });

    afterEach(() => {
        driver.quit();
    });

    it('should tell me if I won', () => {
        // Given I have started the application
        await driver.get('http://localhost:3000');
        // When I enter the correct answer
        const [number1, number2, input, submit] = await Promise.all([
            driver.findElement(By.css('.number1')).getText(),
            driver.findElement(By.css('.number2')).getText(),
            driver.findElement(By.css('input')),
            driver.findElement(By.xpath('//button[text()='Submit']'))
        ]);
        await input.sendKeys('' + (number1 * number2));
        await submit.click();
        // Then I will be told that I have won
        await driver.wait(until.elementLocated(By.css('.Result')));
        const resultElement = driver.findElement(By.css('.Result'));
        await driver.wait(until.elementTextMatches(resultElement,
/won/i));
    }, 60000);
});
```

Discussion

In our experience, web driver tests are the most popular form of automated test for web applications. *Popular* that is, in the sense of *frequently used*. They are inevitably dependent upon matching versions of browsers and web drivers, and they do have a reputation for failing intermittently. These failures are generally caused by timing issues and are particularly common in Single Page Applications which can update their contents asynchronously.

Although it is possible to avoid these problems by carefully adding timing delays and retires into the code, this can them make your tests sensitive to environmental changes, such as running your application on a different testing server.

This is probably why testing systems like Cypress are becoming more popular.

8.6 Automatically find visual differences

Problem

Applications can look very different when viewed on different browsers. Applications can even look different if viewed on the same browser but on a *different* operating system. One example of this would be Chorme, which tends to hide scrollbars when viewed on a Mac, but display them when displayed on Windows. Thankfully, very old browsers like *Internet Explorer* are finally disappearing, but even modern browsers can apply CSS in subtly different ways which can radically change the appearance of a page.

It can be a time consuming job to constantly check an application manually across a range of browsers and platforms.

What can be done to automate this compatibility process?

Solution

In this recipe, we're going to combine three tools to check for visual consistency across different browsers and platforms:

- Storybook. This will give us a basic gallery of all of the components, in all relevant configurations, that we need to check.
- Selenium. This will allow us to capture the visual appearance of all of the components in Storybook. The *Selenium Grid* will also allow us to remotely connect to browsers on different operating systems in order to make comparisons between operating systems.
- ImageMagick. Specifically we will use a command line tool called *compare*, to generate visual differences between screenshots, and a numerical metric of how far appart two images are.

We'll begin by installing Storybook. You can do this in your application with this command:

```
npx -p @storybook/cli sb init
```

You will then need to create *stories* for each of the components and configurations that you interested in tracking. You can find out how to do this from other recipes in this book or from the Storybook tutorials.

Next, we will need *Selenium*, in order to automate the capture of screenshots. You can install Selenium with this command:

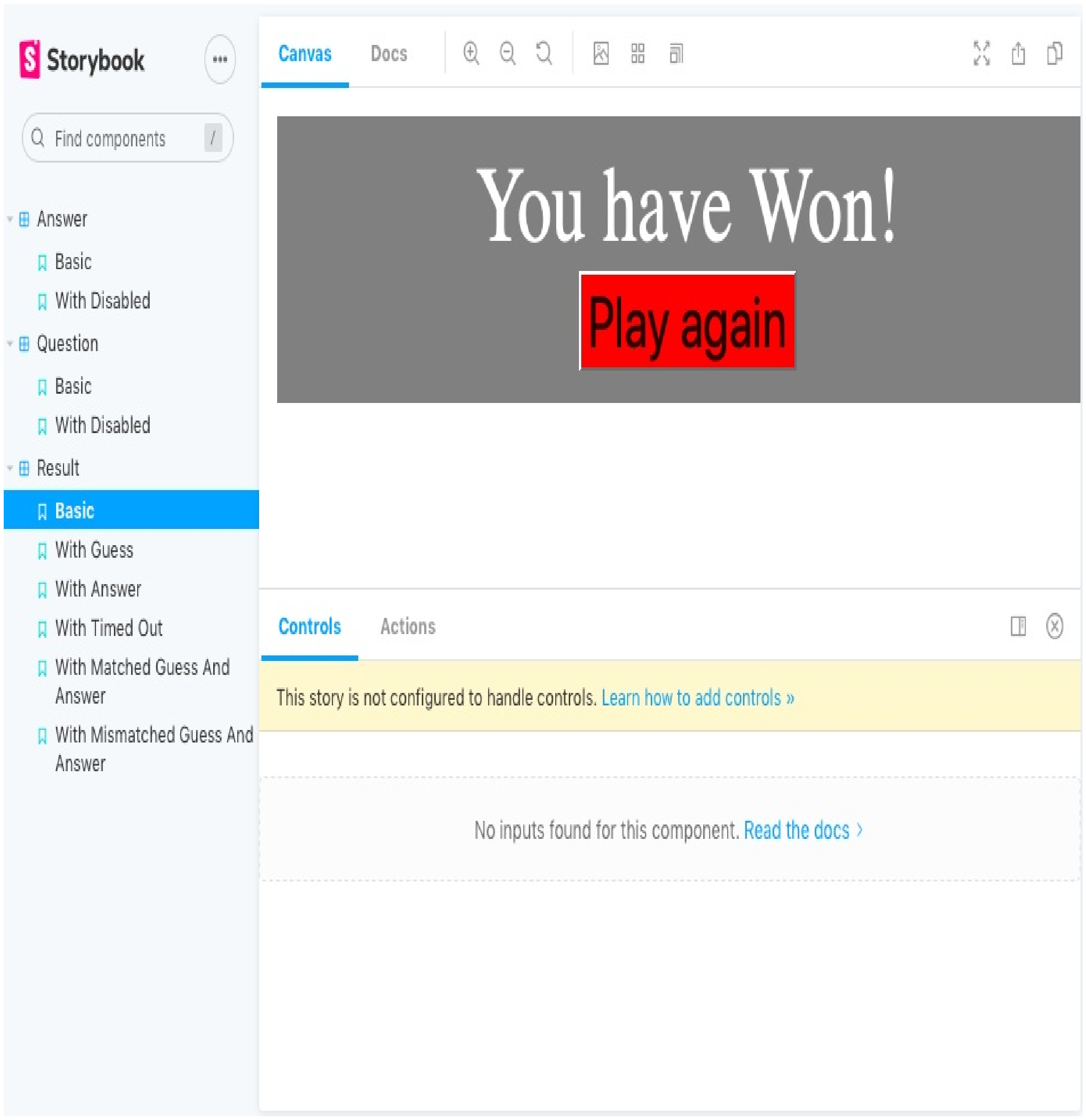
```
npm install --save-dev selenium-webdriver
```

You will also need to install the relevant web drivers. For example, to automate Firefox and Chrome you will need:

```
npm install --save-dev geckodriver
npm install --save-dev chromedriver
```

Finally, you will need to install *ImageMagick*. This is a set of command-line image manipulation tools. For details on how to install ImageMagick, see the ImageMagick download page.

We are going to use the same example game application that we've used previously in this chapter. You can see the components from the application displayed inside Storybook in *figure 8-13*.



This story is not configured to handle controls. Learn how to add controls »

No inputs found for this component. Read the docs ›

Figure 8-13. Components from the application displayed in Storybook.

You can run the Storybook server on your application by typing:

```
npm run storybook
```

Next, we will create a test, that will actually just be a script for capturing screenshots of each of the components inside Storybook. In a folder called //src/selenium/ [16] create a script called *shots.spec.js*:

```
import {Builder, By, until} from "selenium-webdriver";

require('chromedriver');
let fs = require('fs');

describe('shots', () => {
    it('should take screenshots of storybook components',
        async () => {
            const browserEnv = process.env.SELENIUM_BROWSER ||
'chrome';
            const url = process.env.START_URL ||
'http://localhost:6006';
            const driver = new Builder().forBrowser('chrome').build();
            driver.manage().window().setRect({
                width: 1200,
                height: 900,
                x: 0,
                y: 0
            })

            const outputDir = './screenshots/' + browserEnv;
            fs.mkdirSync(outputDir, {recursive: true});

            await driver.get(url);

            await driver.wait(
                until.elementLocated(By.className("sidebar-item")),
                60000
            );
            let elements = await driver.findElements(
                By.css("button.sidebar-item")
            );
            for (let e of elements) {
                const expanded = await e.getAttribute('aria-
expanded');
                if (expanded !== 'true') {
                    await e.click();
                }
            }
            let links = await driver.findElements(
                By.css("a.sidebar-item"));
            for (let link of links) {
                await link.click();
                const s = await link.getAttribute('id');
                let encodedString = await driver.findElement(
                    By.css('#storybook-preview-wrapper')
                ).takeScreenshot();
                await fs.writeFileSync(`${outputDir}/${s}.png`,
                    encodedString,
                    'base64'
                );
            }

            driver.quit();
        }, 60000);
});
```

This is quite a long script. What it does is open a browser to the Storybook server, then open each of the components and taking a screenshot of each story, which it stores in a sub-directory within *screenshots*. You can see example screenshots in *figure 8-14*.

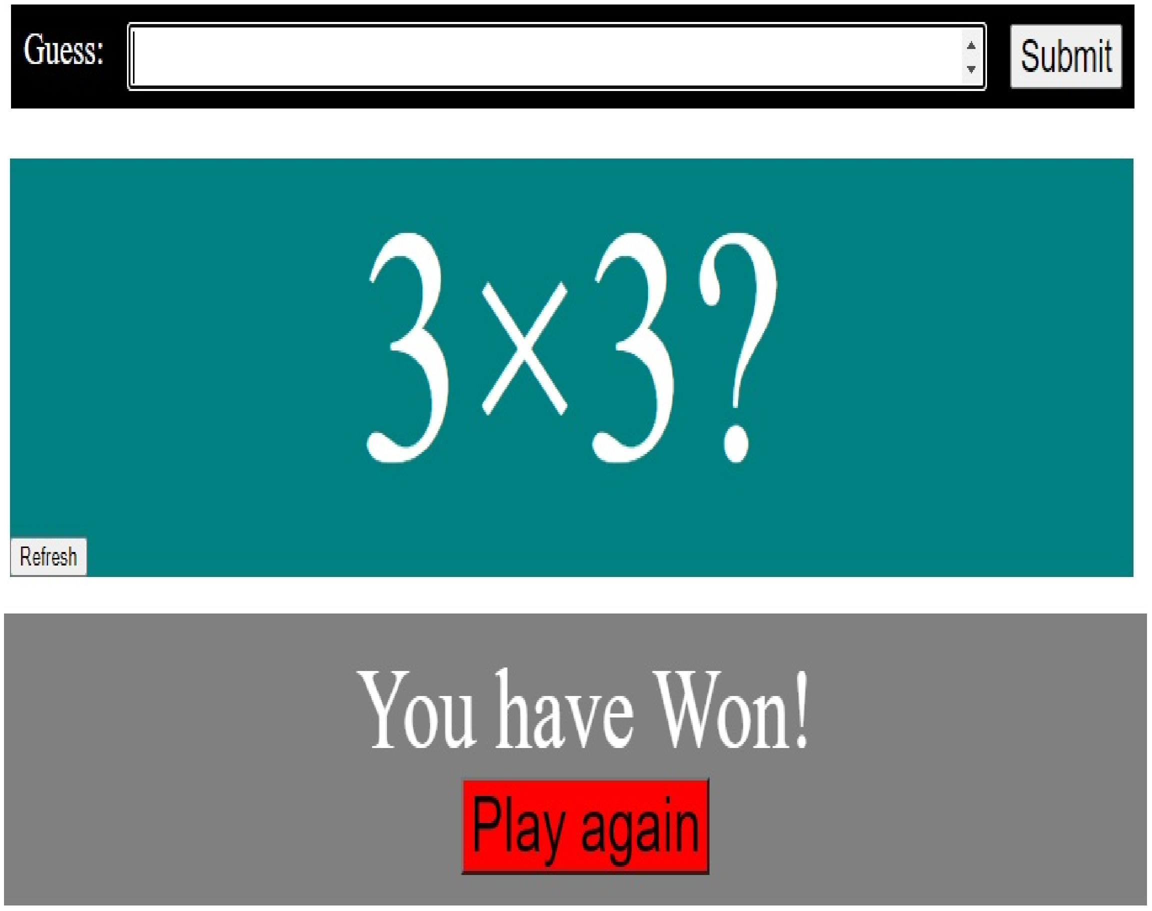


Figure 8-14. A set of example screenshots of components produced by the test.

We could use a different testing system to take screenshots of each component, such a Cypress. The advantage of using Selenium is that we can remotely open a browser session on a remote machine.

By default the *shots.spec.js* test will take screenshots of Storybook at address *http://localhost:6006* using the Chrome browser. Let's say we are running the *shots* test on a Mac. If we have a Windows machine on the same network, we can install a *Selenium Grid* server. This is a proxy server which allows remote machines to start a web driver session.

If the Windows machine has address *192.168.1.16*, we can set this environment variable on the command line before run the *shots.spec.js* test:

```
export SELENIUM_REMOTE_URL=http://192.168.1.16:4444/wd/hub
```

Because the Windows machine will be accessing the Storybook server back on the Mac, with an IP address of *192.168.1.14*, we will also need to set an environment variable that on the command line:

```
export START_URL=http://192.168.1.14:6006
```

We can also choose which browser we want the Windows machine to use[17]:

```
export SELENIUM_BROWSER=firefox
```

If we create a script to run *shots.spec.js* in *package.json*:

```
"scripts": {
  ...
  "testShots": "CI=true react-scripts test --detectOpenHandles
'selenium/shots.spec.js'"
}
```

We can run the test and capture the screenshots of each component:

```
npm run testShots
```

The test will use the environment variables we created to contact the Selenium Grid server on the remote machine. It will ask Selenium Grid to open a Firefox browser to our local Storybook server, and it will then take a screenshot of each of the components, which will then be sent back over the network where they will be stored in a folder called //screenshots/firefox/.

Once we've run it for Firefox, we can then run it for Chrome:

```
export SELENIUM_BROWSER=chrome
npm run testShots
```

The screen shots for that session will be stored in //screenshots/chrome/.

NOTE

A fuller implementation of this technique would also record the operating system and type of client (e.g. screen size) used.

We now need some way to check for visual differences between the screenshots from Chrome and the screenshots from Firefox. This is where ImageMagick is useful. The *compare* command in ImageMagick can generate an image that highlights the visual differences between two other images. For example, consider the two screenshots from Firefox and Chrome in *figure 8-15*.

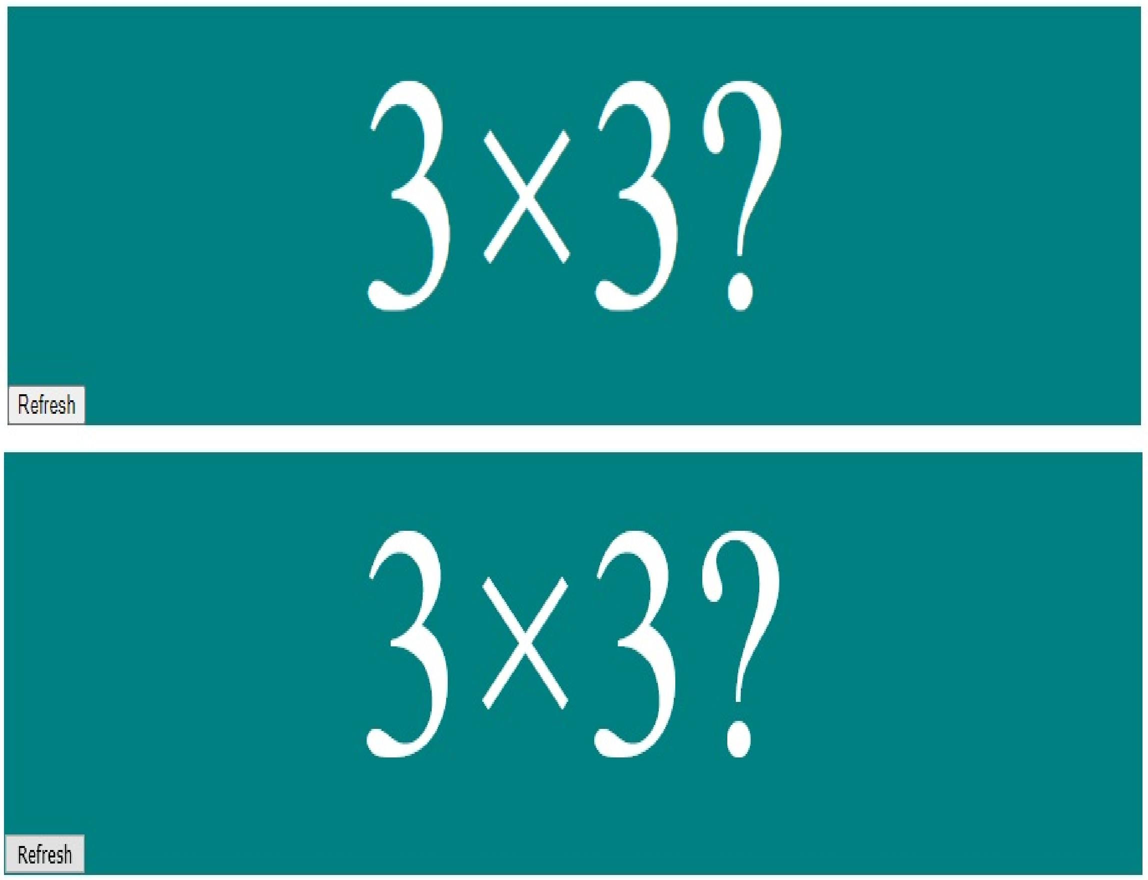


Figure 8-15. The same component captured in Chrome and Firefox.

These two images appear to be identical. If we type in this command from the main application directory:

```
$ compare -fuzz 15% screenshots/firefox/question--basic.png
screenshots/chrome/question--basic.png difference.png
```

We will generate a new image that shows the differences between the two screenshots, which you can see in *figure 8-16*.

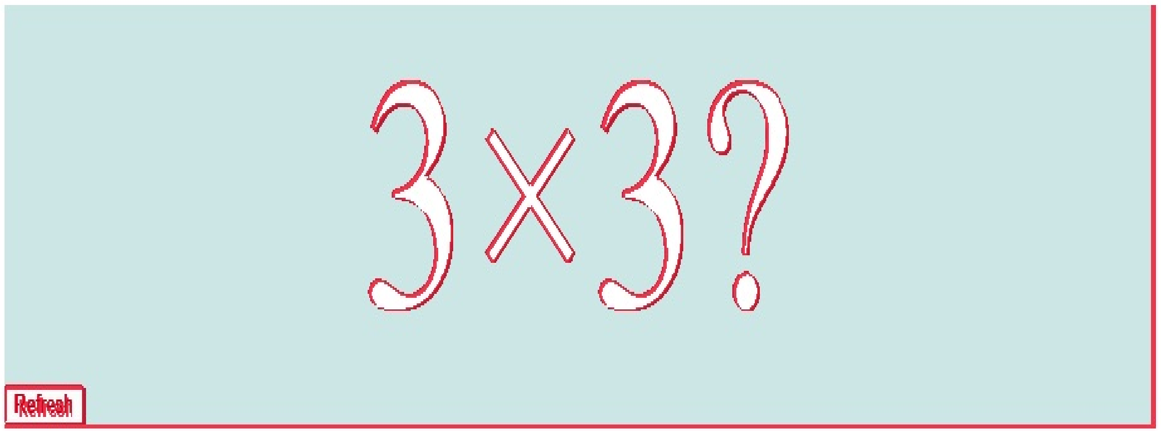


Figure 8-16. The generated image showing the differences between two screen captures.

The generated image shows pixels that are more than 15% visually different between the two images. And you can see that the screenshots are virtually identical.

That’s good, but it still requires a human being to look at the images and assess whether the differences are significant. What else can we do?

The *compare* command also has the ability to display a numerical measure of the difference between two images:

```
$ compare -metric AE -fuzz 15% screenshots/firefox/question--basic.png
screenshots/chrome/question--basic.png difference.png
6774
$
```

The value *6774* is a numerical measure of the visual difference between the two images. For another example, consider these two screenshots in *figure 8-17* which show the *Answer* component when it has been passed the *disabled* property.

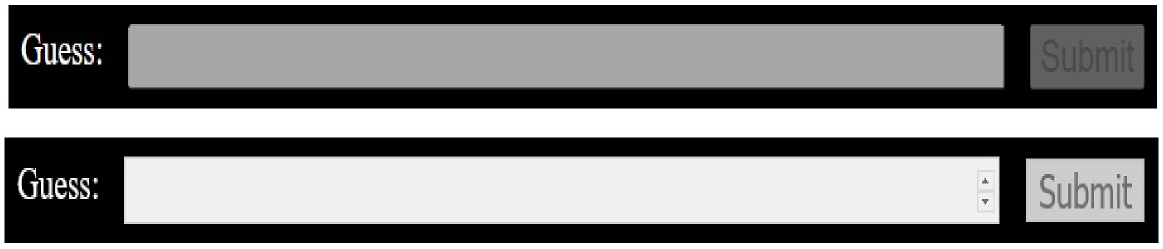


Figure 8-17. Disabled form rendered by Chrome and Firefox.

Comparing these two images returns a much larger number:

```
$ compare -metric AE -fuzz 15% screenshots/firefox/answer--with-
disabled.png     screenshots/chrome/answer--with-disabled.png
difference3.png
28713
$
```

And indeed the generated image (see *figure 8-18*) shows exactly where the difference lies: in the appearance of the disabled input field.

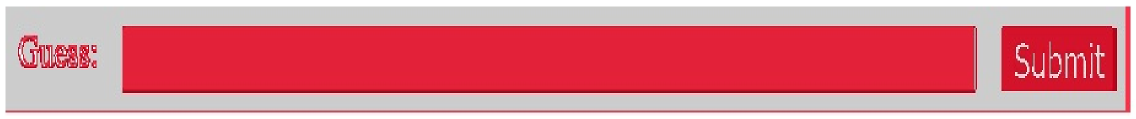


Figure 8-18. The visual difference between the Chrome and Firefox forms.

Figure 8-19 shows a similarly large difference[18] for a component that displays different font styling between the browsers, which is actually the result of some

Mozilla-specific CSS attributes.

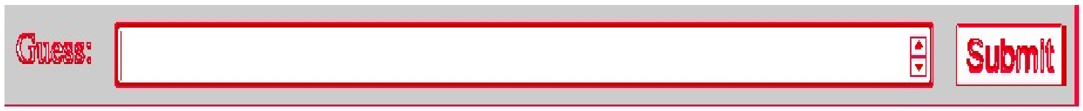
bottom. image::images/ch10-text-comparison.png["A component with different text styling in Chrome and Firefox. The difference is show at the bottom."]

In fact, it’s possible to write a shell script to run through each of the images and generate a small web report showing the visual differences alongside their metrics:

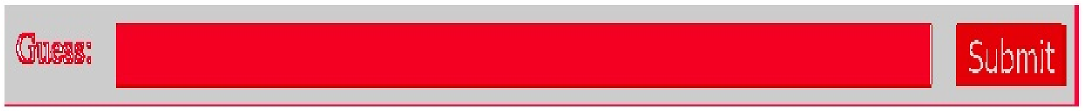
```
#!/bin/bash
mkdir -p screenshots/diff
export HTML=screenshots/compare.html
echo '<body><ul>' > $HTML
for file in screenshots/chrome/*.png
do
    FROM=$file
    TO=$(echo $file | sed 's/chrome/firefox/')
    DIFF=$(echo $file | sed 's/chrome/diff/')
    echo "FROM $FROM TO $TO"
    ls -l $FROM
    ls -l $TO
    METRIC=$(compare -metric AE -fuzz 15% $FROM $TO $DIFF 2>&1)
    echo "<li>$FROM $METRIC<br/><img src=../$DIFF/></li>" >> $HTML
done
echo "</li></body>" >> $HTML
```

Which will create the //screenshots/compare.html/ report you can see in figure *8-20*.

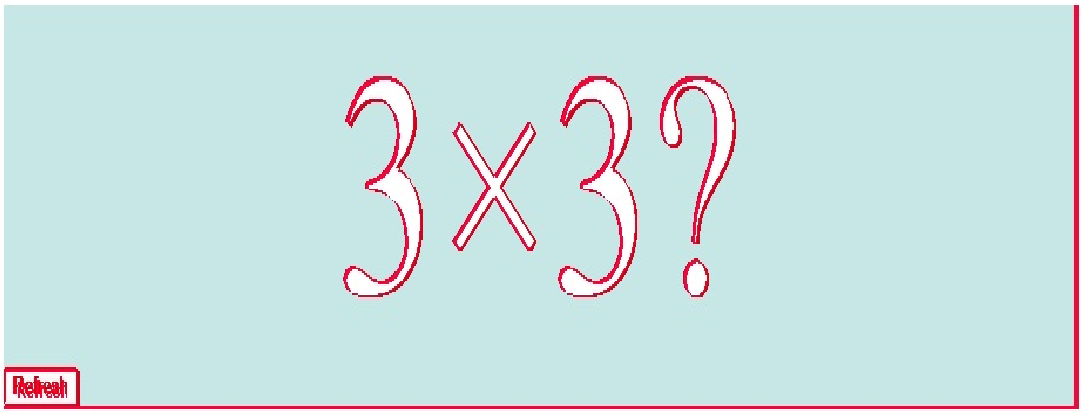
- screenshots/chrome/answer--basic.png 7174



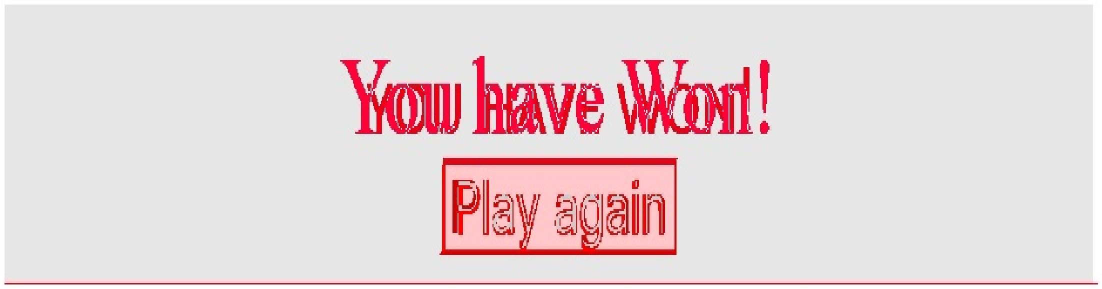
- screenshots/chrome/answer--with-disabled.png 28713



- screenshots/chrome/question--basic.png 6774



- screenshots/chrome/result--basic.png 21131



- screenshots/chrome/result--with-answer.png 20671

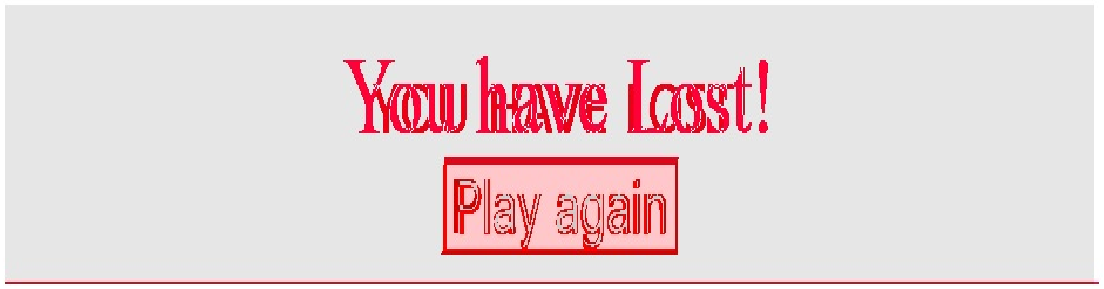


Figure 8-19. An example of the generated comparison report.

Discussion

To save space, we have shown only a very simplistic implementation of this technique. It would be possible to create a ranked report, which showed visual differences from largest to smallest. This would highlight the most significant visual differences between platforms. It would also be possible to create a continuous integration job that would set some visual threshold between images. Any components that varied by more than that threshold would fail the job, and notify someone of the inconsistency.

8.7 Add a visual console to mobile browsers

Problem

This recipe is slightly different to the others in this chapter because instead of being about automated testing, it's about manual testing. Specifically about manually testing code on mobile devices.

If you are testing an application on a mobile, you might stumble across a bug that doesn't appear in the desktop environment. Normally if a bug appears, you're able to add debug messages into the JavaScript console. But mobile browsers tend not to have a visible JavaScript console. It's true that if you are using Mobile Chrome, you can try debugging it remotely with a desktop version of Chrome. But what if the problem is discovered in another browser? Or if you simply don't want to go through the work of setting up a remote debug session?

Is there some way to get access to the JavaScript console, and other development tools, from within a mobile browser?

Solution

In this recipe we are going to a quite remarkable piece of software called Eruda.

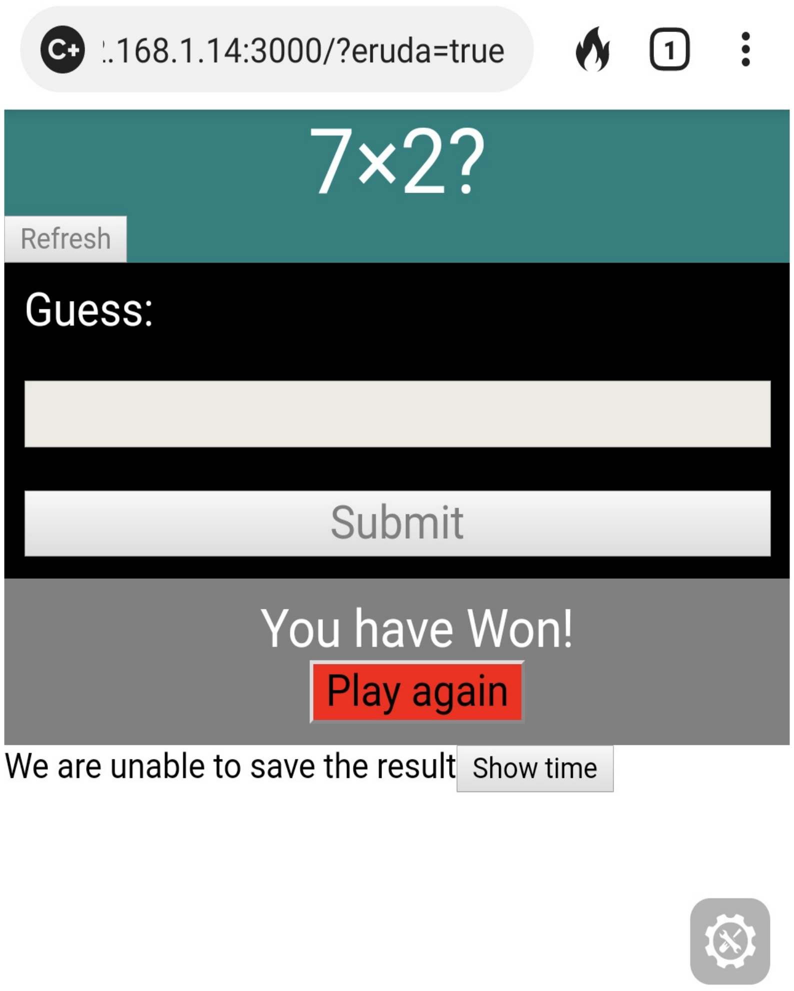
Eruda is a lightweight implementation of a development tools panel, which will allow you to view the JavaScript console, the structure of the page as well as a whole heap of other plugins and extensions.

In order to enable Eruda, you will need to install a small amount of fairly rudimentary JavaScript in the *head* section of your application. Eruda itself can be downloaded from a content distribution network, but because it can be quite large, you should only enable it if the person using the browser has indicated that they want to access it.

One way of doing this is only enabling Eruda if *eruda=true* appears in the URL. Here’s an example script that you can insert into your page container[19].

```
<script>
    ;(function () {
        var src = '//cdn.jsdelivr.net/npm/eruda';
        if (!/eruda=true/.test(window.location)
            && localStorage.getItem('active-eruda') != 'true') return;
        document.write('<scr' + 'ipt src="' + src
            + '"></scr' + 'ipt>');
        document.write('<scr' + 'ipt>');
        document.write('window.addEventListener(' +
            '"load", ' +
            'function () {' +
            '  var container=document.createElement("div"); ' +
            '  document.body.appendChild(container);' +
            '  eruda.init({' +
            '    container: container,' +
            '    tool: ["console", "elements"]' +
            '  });' +
            '})');
        document.write('</scr' + 'ipt>');
    })();
</script>
```

If you now open your application as *http://ipaddress/?eruda=true* or *http://ipaddress/#eruda=true* you will notice that an additional button has appeared in the interface, as show in *figure 8-21*.



7×2?

Refresh

Guess:

Submit

You have Won!

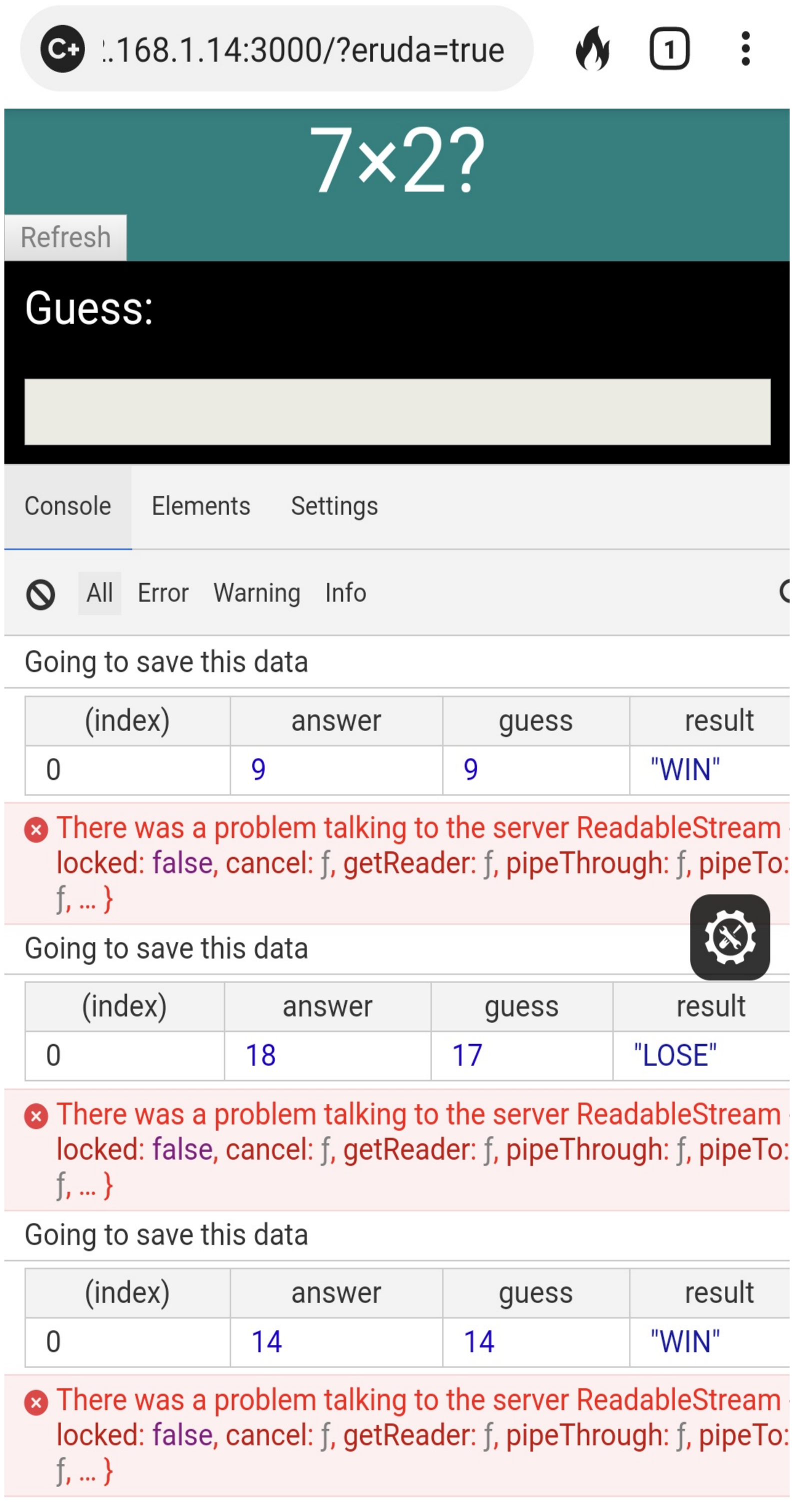
Play again

We are unable to save the result

Show time

Figure 8-20. If you add ?eruda=true to the URL, a button will appear on the right of the page.

If you are using the example application for this chapter[20], then try entering a few answers in the game. Then, click on the Eruda button. The console will appear as show in *figure 8-22*.



Going to save this data

(index)	answer	guess	result
0	9	9	"WIN"

```
There was a problem talking to the server ReadableStream
locked: false, cancel: ƒ, getReader: ƒ, pipeThrough: ƒ, pipeTo:
ƒ, … }
```

Going to save this data

(index)	answer	guess	result
0	18	17	"LOSE"

```
There was a problem talking to the server ReadableStream
locked: false, cancel: ƒ, getReader: ƒ, pipeThrough: ƒ, pipeTo:
ƒ, … }
```

Going to save this data

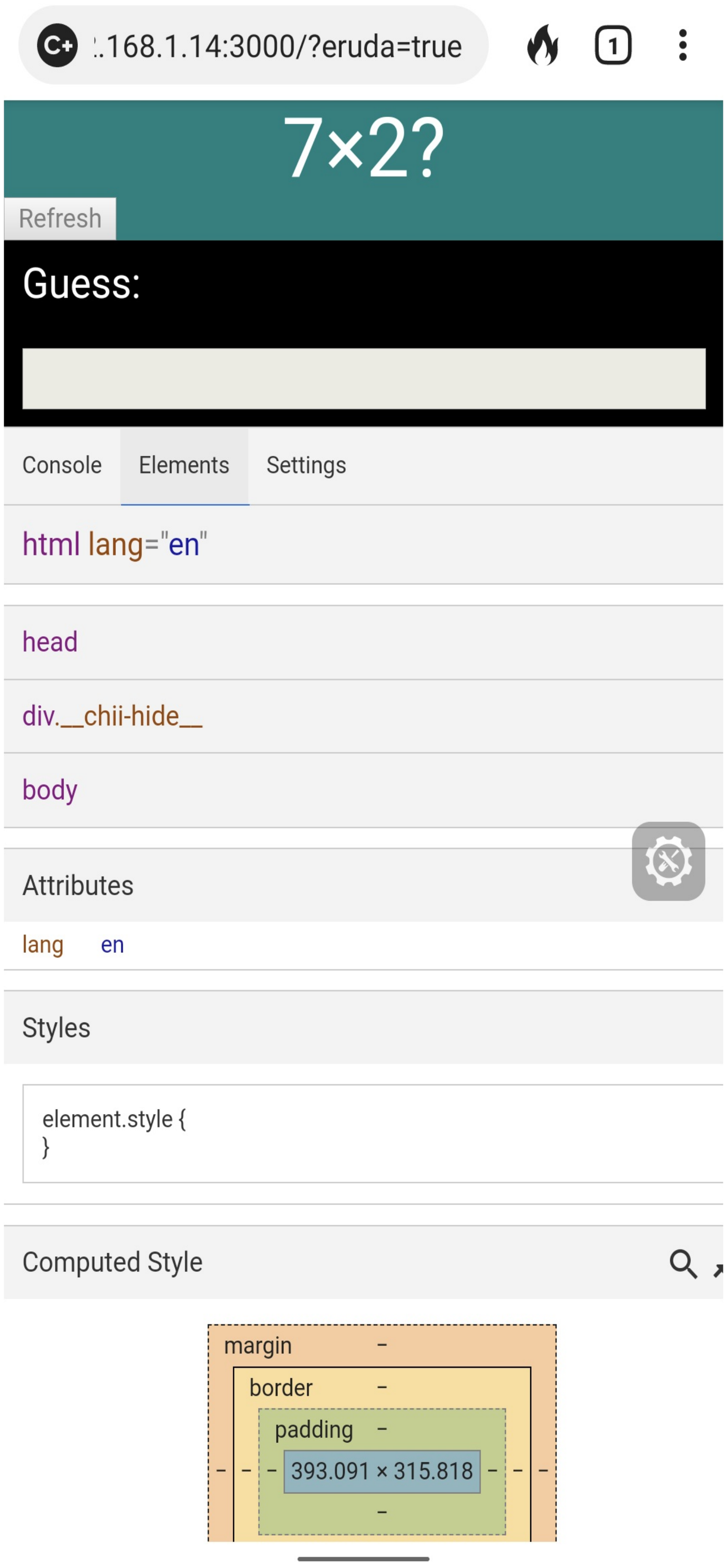
(index)	answer	guess	result
0	14	14	"WIN"

```
There was a problem talking to the server ReadableStream
locked: false, cancel: ƒ, getReader: ƒ, pipeThrough: ƒ, pipeTo:
ƒ, … }
```

Figure 8-21. Pressing the button opens the Eruda tools.

Because the server for the example application has not been implemented, you should find some errors and other logs recorded in the console. The console even supports the much underused *console.table* function, which is a useful way of displaying an array of objects in tabular format.

The *Elements* tab provides a fairly rudimentary view of the Document Object Model (see *figure 8-23*.)



:.168.1.14:3000/?eruda=true

7×2?

Refresh

Guess:

Console Elements Settings

html lang="en"

head

div.__chii-hide__

body

Attributes

lang	en

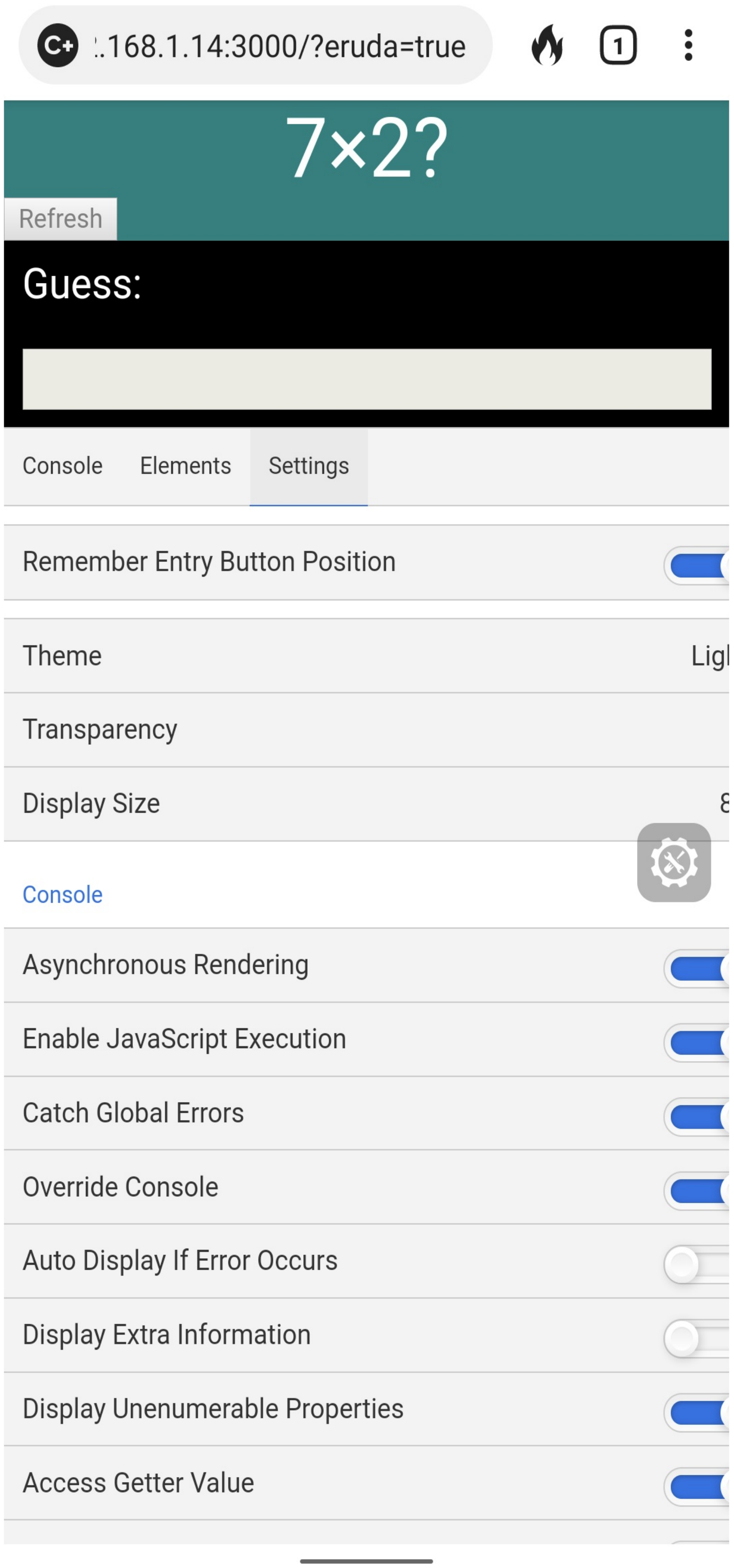
Styles

```
element.style {
}
```

Computed Style

Figure 8-22. The Eruda elements view.

Meanwhile, the *Settings* tab has a very large set of JavaScript features that you can enable and disable while interacting with the web page (see *figure 8-24*)



:.168.1.14:3000/?eruda=true

7×2?

Refresh

Guess:

Console | Elements | Settings

Remember Entry Button Position	on
Theme	Ligl
Transparency	
Display Size	8

Console

Asynchronous Rendering	on
Enable JavaScript Execution	on
Catch Global Errors	on
Override Console	on
Auto Display If Error Occurs	off
Display Extra Information	off
Display Unenumerable Properties	on
Access Getter Value	on

Figure 8-23. The Eruda settings view.

Discussion

Eruda is a delightful tool which delivers a whole bucket of functionality, with very little work required by the developer. In addition to the basic features, it also has plugins that allow you to track performance, screen refresh rate, generate fake geolocations and even write a run JavaScript, from inside the browser. Once you start to use it, you probably find that it quickly becomes a standard part of your manual testing process.

8.8 Remove randomness from tests

Problem

In a perfect world, tests would always have a completely artificial environment. Tests are examples of how you would like your application to work under explicitly defined conditions. But there are often uncertainties that tests have to cope with. For example, they might run at different times of day[21]. They might even, as in the example application that we have used throughout this chapter, have to deal with *randomness*.

Our example application is a game which presents the user with a randomly generated question that they must answer (see *figure 8-25*.)

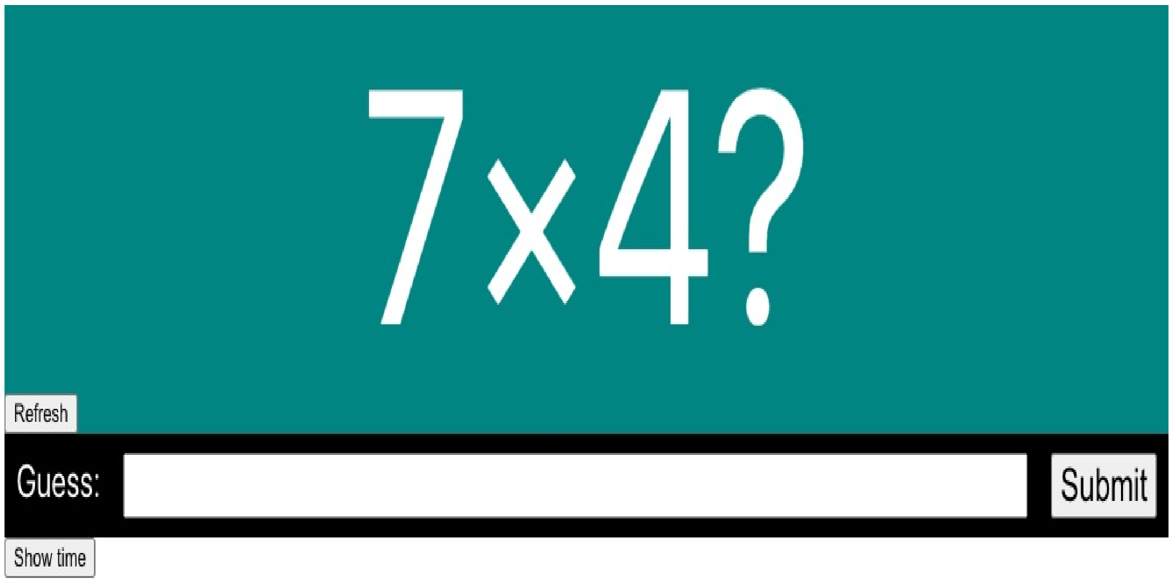


Figure 8-24. The game asks the user to calculate a random multiplication.

Randomness might also appear in the generation of identifiers within the code, or random data sets. If you are suggested a new username, your application might suggest a randomly generated string.

But randomness creates a problem for tests. This is an example test that we implemented earlier in this chapter:

```
describe('Basic game functions', () => {
    it('should notify the server if I lose', () => {
        // Given I started the application
        // When I enter an incorrect answer
        // Then the server will be told that I have lost
    });
});
```

There was actually a very good reason why that test looked at the case where the user entered an *incorrect* answer. The question asked is always to caulcate the product of 2 numbers between 1 and 10. It's therefore easy to think of an incorrect answer. 101. It will *always* be wrong. But if we want to write a test to show what happens when the user enters the *correct* answer, we have a problem. The correct answer depends upon data that is randomly generated. That either means that we have to write some code that finds the two numbers that appear on the screen, as in this example from the first Selenium recipe in this chapter:

```
const [number1, number2, input, submit] = await Promise.all([
    driver.findElement(By.css('.number1')).getText(),
    driver.findElement(By.css('.number2')).getText(),
    driver.findElement(By.css('input')),
    driver.findElement(By.xpath('//button[text()='Submit']'))
]);
await input.sendKeys('' + (number1 * number2));
await submit.click();
```

This means we have to write a bunch of extra code to find details which, in the end, we don't care about. We only care that the answer is right.

Alternatively, having tests which conditionally depend upon data that is randomly generated at runtime can have a much more severe effect. If we wanted to write a Cypress test to enter the correct answer to the multiplication, we would have very great difficulty. That's because Cypress does not allow you to capture values from the page and pass them to other steps in the test[22].

It would be much better then, if we could turn off the randomness for a while. Just during a test.

But can we?

Solution

We are going to look at how we can use the *Sinon* library temporarily replace the *Math.random* with a faked one of our own making.

Let's first consider how we can do this inside a unit test. We'll create a new test for the top-level *App.js* component which will check that entering the correct value results in a message saying that we won.

We'll first create a function that will fix the return value of *Math.random()*:

```
const sinon = require('sinon');

function makeRandomAlways(result) {
    if (Math.random.restore) {
        Math.random.restore();
    }
    sinon.stub(Math, 'random').returns(result);
}
```

This function works by replacing the *random()* method of the *Math* object with a stubbed method which always returns the same value. We can now use this in a test. The *Question* that appears on the page, always generates random numbers between 1 and 10, based upon the value of

```
Math.random() * 10 + 1
```

If we fix *Math.random()* so that it always produced the value 0.5, then the "random" number will always be 6. That means we can write a unit test like this:

```
it('should tell you that you entered the right answer', async () => {
    // Given we've rendered the app
    makeRandomAlways(0.5);
    render(<App/>);

    // When we enter the correct answer
    const input = screen.getByLabelText(/guess:/i);
    const submitButton = screen.getByText('Submit');
    user.type(input, '36');
    user.click(submitButton);

    // Then we are told that we've won
    await waitFor(() => screen.findByText(/won/i), {timeout: 4000});
})
```

And this test will always pass, because the application will always ask the question “What is 6 x 6?”

The real value of fixing *Math.random()* is when we are using a testing framework that explicitly *prevents* us from capturing a randomly generated value. Such as Cypress, as we saw above.

Cypress allows us to add custom commands[23], by adding them to the //cypress/support/commands.js/ script. If you edit that file, and add this code:

```
Cypress.Commands.add('random', (result) => {
    cy.reload().then((win) => {
        if (win.Math.random.restore) {
            win.Math.random.restore();
        }
        sinon.stub(win.Math, 'random').returns(result);
    });
});
```

You will create a new command called *cy.random()*. We can use this command to create a test that the *winning* case which we discussed in the introduction[24]:

```
describe('Basic game functions', () => {
    it('should notify the server if I win', () => {
        // Given I started the application
        cy.intercept('POST', '/api/result', {
            statusCode: 200,
            body: ''
        }).as('postResult');
        cy.visit('http://localhost:3000');
        cy.random(0.5);
        cy.contains('Refresh').click();

        // When I enter the correct answer
        cy.get('input').type('36');
        cy.contains('Submit').click();

        // Then the server will be told that I have lost
        cy.wait('@postResult').then(xhr => {
            assert.deepEqual(xhr.request.body, {
                guess: 36, answer: 36, result: 'WIN'
            });
        });
    });
});
```

NOTE

After calling the *cy.random()* command, we need to press the *Refresh* button just in case the random numbers on the page were generated after the page was loaded, but before the *Math.random()* function was replaced.

Discussion

You can never remove all randomness from a test. The basic performance of the machine can have a huge effect on when and how often your components are re-rendered for example. But removing uncertainty as much as we can is generally a good thing in a test. The more we can do to remove external dependencies from our tests, the better.

Which is a topic that will crop up again in our next recipe.

8.9 Time travel

Problem

Time can be the source of a tremendous amount of bugs. If time was simply a scientific measurement, it would relatively straightforward. But it isn’t. The representation of time is affected by national boundaries and by local laws. Some countries have their own time zones. Others have multiple time zones. One reassuring factor is that all countries have a time zone offset that can be measured in whole hours. Except for places like India, where time is offset by 05:30 from UTC.

That’s why it is useful to try to fix time within a test. But how do we do that?

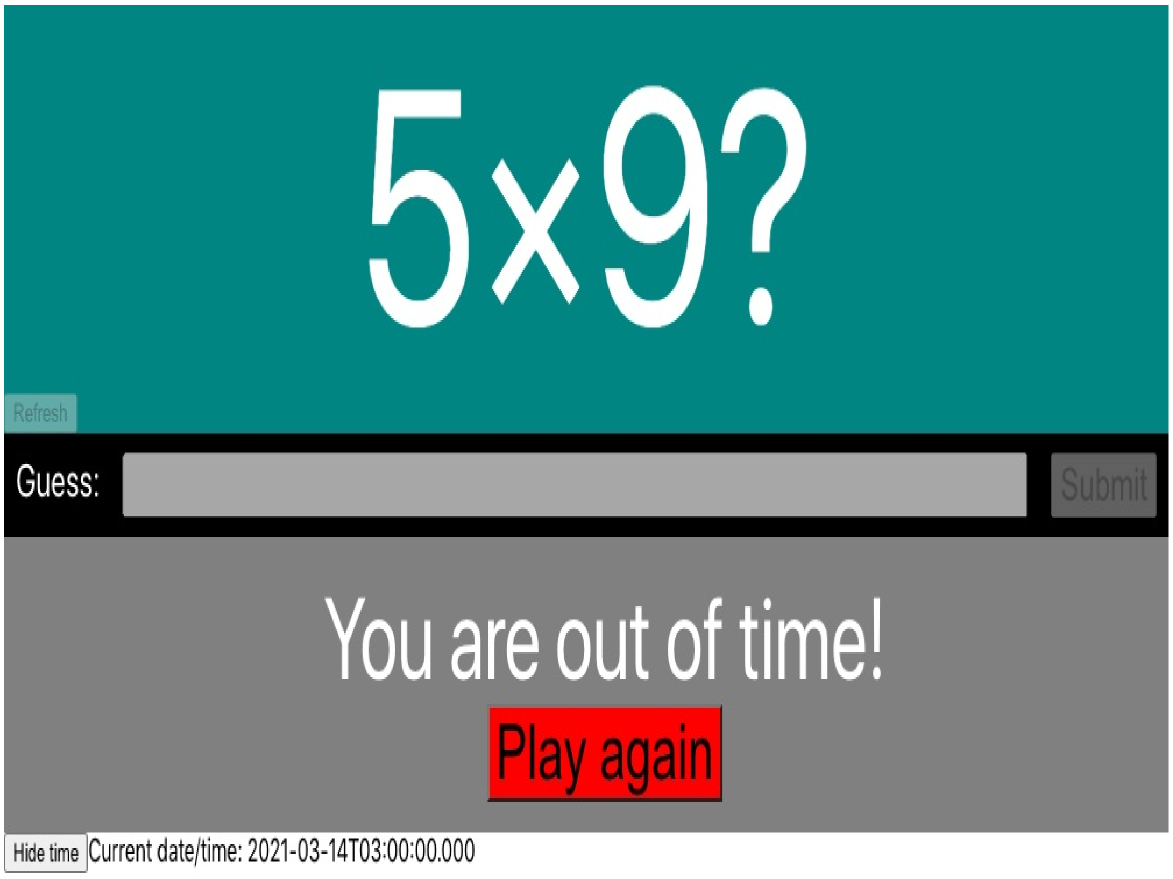
Solution

In this recipe we are going to look at how you can fix time when testing your React application. There are some issues that you need to consider when testing time-dependent. First of all, you should probably avoid changing the time on your server. In most cases, you are best to set your server to UTC and leave it that way.

That does mean that if you want to fake a date and time in your browser, you will have problems as soon as the browser makes contact with the server. That means you will either have to modify the server APIs to always accept an *effective date* in all time-sensitive code[25] or you should try to time-dependent browser code in isolation from the server.

We will adopt the latter approach for this recipe: we will use the Cypress testing system which will allow us to fake any connections with the server[26].

The example application we will use for this recipe is the same one we use for other recipes in this chapter. It’s a simple game that asks the user to calculate the product of two numbers. We’re going to test a feature of the game which gives the user 30 seconds to provide an answer. After 30 seconds they will see a message telling them they’ve run out of time (see *figure 8-26*)



You are out of time!

Current date/time: 2021-03-14T03:00:00.000

Figure 8-25. The player will lose if they do not answer within 30 seconds.

30 seconds is not very long time, so we could try writing a test that somehow pauses for 30 seconds. That has two problems. First, it will slow your test down. You don’t need many 30 second pauses before your tests will become unbearable to run. Second, adding a pause is not a very precise way of testing the feature. If you pause for 30 seconds, the time might actually pause for 30.5 seconds before looking for the message.

In order to get precision, we need to take control of time within the browser. As you saw in the previous recipe[27], Cypress has the ability to inject code into the browser which can replace key pieces of code with stubbed functions, which we can control. Cypress actually has a built in command called *cy.clock()* which will allow us to specify the current time.

Let's see how to use *cy.clock()* by creating a test for the timeout feature. This will be the structure of our test:

```
describe('Basic game functions', () => {
    it('should say if I timed out', () => {
        // Given I have started a new game
        // When 29 seconds have passed
        // Then I will not see the time-out message
        // When another second has passed
        // Then I will see the time-out message
        // And the game will be over
    });
});
```

We will can start by opening the application and starting a new game by pressing the *Refresh* button.

```
describe('Basic game functions', () => {
    it('should say if I timed out', () => {
        // Given I have started a new game
        cy.visit('http://localhost:3000');
        cy.contains('Refresh').click();

        // When 29 seconds have passed
        // Then I will not see the time-out message
        // When another second has passed
        // Then I will see the time-out message
        // And the game will be over
    });
});
```

Now we need to simulate 29 seconds of time passing. We can do this with the *cy.clock()* and *cy.tick()* commands. The *cy.clock()* allows you to either specify a new date and time, or by default it will set the time and date back to 1969. The *cy.tick()* command allows you to add a set number of milliseconds to the current date and time.

```
describe('Basic game functions', () => {
    it('should say if I timed out', () => {
        // Given I have started a new game
        cy.clock();
        cy.visit('http://localhost:3000');
        cy.contains('Refresh').click();

        // When 29 seconds have passed
        cy.tick(29000);

        // Then I will not see the time-out message
        // When another second has passed
        // Then I will see the time-out message
        // And the game will be over
    });
});
```

We can now complete the other steps in the test. For details on the other Cypress commands we're using, see the Cypress documentation.

```
describe('Basic game functions', () => {
    it('should say if I timed out', () => {
        // Given I have started a new game
        cy.clock();
        cy.visit('http://localhost:3000');
        cy.contains('Refresh').click();

        // When 29 seconds have passed
        cy.tick(29000);

        // Then I will not see the time-out message
        cy.contains(/out of time/i).should('not.exist');

        // When another second has passed
        cy.tick(1000);

        // Then I will see the time-out message
        cy.contains(/out of time/i).should('be.visible');

        // And the game will be over
        cy.get('input').should('be.disabled');
        cy.contains('Submit').should('be.disabled');
    });
});
```

If we run the test in Cypress, it passes (as you can see in figure FIGNUM).

That’s a relatively simple time-based test. But what if we wanted to test something much more complex, like Daylight Savings Time?

Daylight Savings are the bane of most development teams. They sit in your code base, silently for months and then suddenly appear in the Spring and Autumn, in the early hours of the morning.

When Daylight Savings Time occurs depends upon which time-zone you are in. And that’s a particularly awful thing to deal with inclient code, because JavaScript dates don’t really work with time-zones. They can certainly handle offsets, for example, you can create a *Date* object in a browser like Chrome[28] that is set to 5 hours before *Greenwich Mean Time*:

```
new Date("2021-03-14 01:59:30 GMT-0500")
```

But JavaScript dates are all implicitly in the time-zone of the browser. When you create a date with a time-zone name in it, the JavaScript engine will simply shift it into the browser’s own time-zone.

The browser’s time-zone is fixed at the time that the browser opens. There’s no way to say *Let’s pretend we’re in New York from now on*.

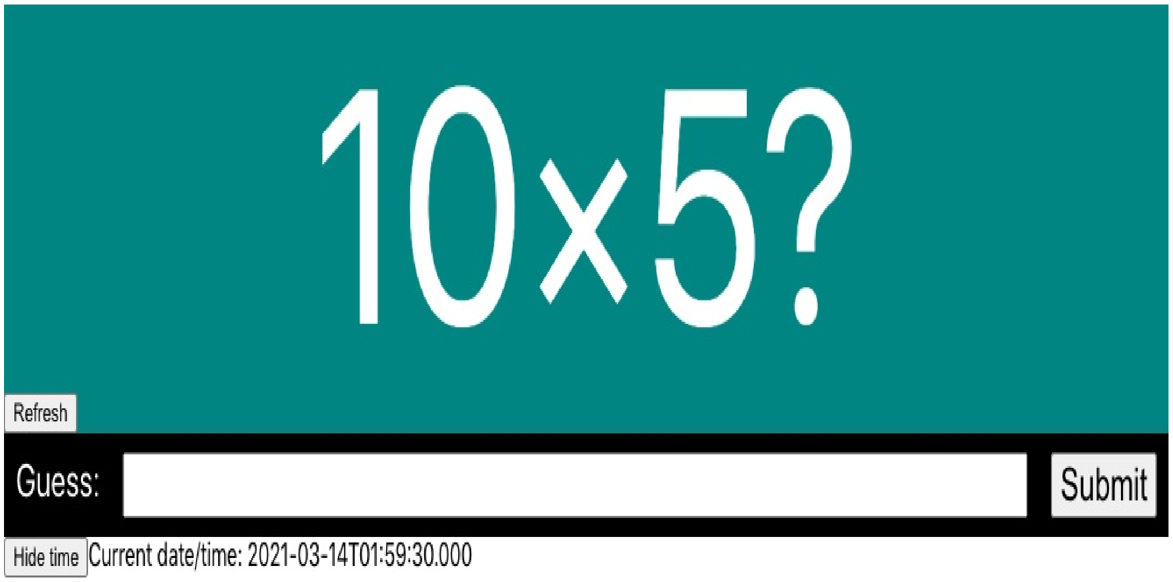
What inevitably happens if that if developers create tests for Daylight Savings Time[29] then they are likely to create tests that work in their own time-zone. And then find that these tests fail when run on an integration server that is probably set to UTC.

There is, however, a way around this problem. On Linux and Mac computers[30] you can specify the time-zone when you launch a browser by setting an environment variable called `TZ`. If we start the Cypress with the `TZ` variable set, that variable will be inherited by any browser that Cypress then launches. This means that while we can’t set the timezone for a single test, we can set it for an entire test run.

First, let’s launch Cypress with the time-zone set to New York:

```
TZ='America/New_York' npx cypress open
```

The example application has a button that allows you to see the current time (see *figure 8-27*).



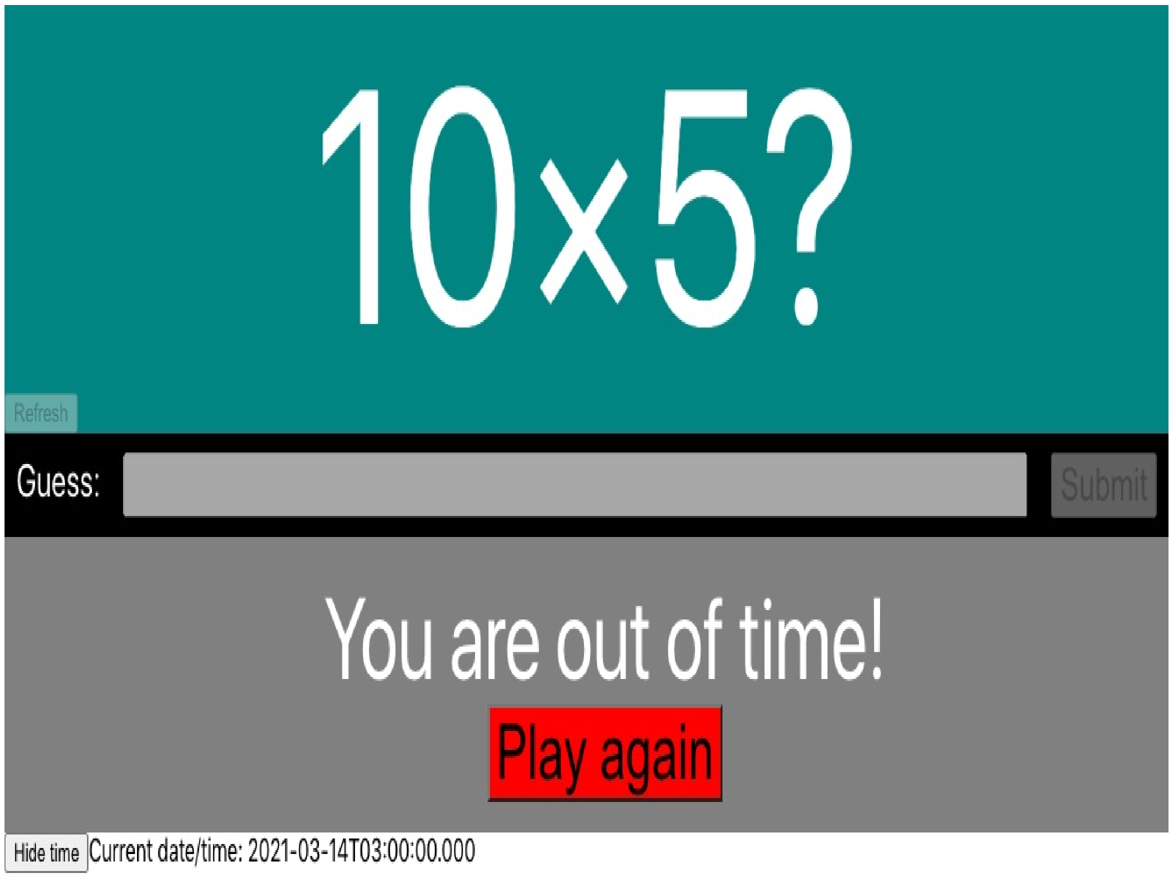
Current date/time: 2021-03-14T01:59:30.000

Figure 8-26. The current time is shown on the screen.

We can create a test that checks that the time on page correctly handles the change to Daylight Savings. This is the test we'll create:

```
describe('Timing', () => {
    it('should tell us the current time',() => {
        cy.clock(new Date("2021-03-14 01:59:30").getTime());
        cy.visit('http://localhost:3000');
        cy.contains('Show time').click();
        cy.contains('2021-03-14T01:59:30.000').should('be.visible');
        cy.tick(30000);
        cy.contains('2021-03-14T03:00:00.000').should('be.visible');
    });
});
```

In this test we are passing an explicit date to *cy.clock()*. We need to convert this to milliseconds by calling *getTime()* as *cy.clock()* only accepts numeric times. We then check that the initial time is displayed, and that 30 seconds later, the time rolls over to 3am, instead of 2am (as show in *figure 8-28*).



Current date/time: 2021-03-14T03:00:00.000

Figure 8-27. After 30 seconds, the time correctly changes from 01:59 to 03:00.

Discussion

If you do need to create tests that are sensitive to the current time zone, consider placing them into a sub-folder so that they can be run is a separately configured test run. If you wish to format dates into various time-zones, you can use the *toLocaleString()* data method:

```
(new Date()).toLocaleString('en-US', { timeZone: 'Asia/Tokyo' })
```

If you want more extensive time-zone functionality, such as inferring the name of your current time-zone, consider looking at the Moment Timezone library.

---

[1] You will see in other recipes in this chapter, that it's possible to dynamically remove the randomness from a test and fixing the correct answer, without the need the question from the page.

[2] Notice that many text comparisons are made using regular expressions. This allows, as in this example, for case-insensitive matches of substrings. This is another technique to avoid tests that frequently break.

[3] See the source code in the Github repisitory. This is how we've structured the code in the example application.

[4] If you don't have the testing-library installed, see the first recipe in this chapter.

[5] That is, they make your life easier.

[6] That is, more work.

[7] Either directly, or indirectly via libraries such as *axios*.

[8] It doesn't actually matter what you call the file, but we are following the convention of prefixing high-level tests such as this with story numbers. This general reduces the likelihood of test merge conflicts, and makes it much easier to track the intent of individual changes.

[9] This will run the tests more quickly, and record a video of the test running, in case you need to examine it later. This is useful if your test is running on an integration server.

[10] Cypress commands are similar in many ways to promises, although they are not implemented as promises. You can think of each one as a "prom-ish".

[11] The *cy.intercept()* command cannot simply return a reference to the faked network request, because of the chainable nature of Cypress commands.

[12] Recipe 5 in chapter 2.

[13] This doesn't mean that the tests will work against every browser, just that they will all run across every browser.

[14] We are following a convention where we prefix the test with it's associated story number. This is not required by Selenium.

[15] You will find the code to do this is downloadable source for this chapter from Github.

[16] You could actually put this script anywhere, but this is the location we used in the example code on the Github site.

[17] The remote machine will have to have the appropriate browser and web driver installed for this to work.

[18] The visual difference for this image is 21,131.

[19] For create-react-app applications, this should be added to //public/index.html/.

[20] The code is available in the source code repository for this book.

[21] We will see how to deal with controlling time in a later recipe in this chapter.

[22] At least, not without a good deal of extra work

[23] For further information on Cypress, see the other recipes in this chapter.

[24]

You can find out more about this test in recipe 3 of this chapter.

[25]That is, allow the browser to say to the server *Let’s pretend it’s Thursday, April 14th*.

[26]For more information on how to install and create tests for Cypress, see other recipes in this chapter.

[27]See recipe 8 in this chapter to see how to remove randomness.

[28]Firefox will not generally accept this format.

[29]This rarely happens.

[30]Sadly, not on Windows machines.

About the Authors

David Griffiths has been writing code professionally in React for five years, and has created applications for startups, retail stores, vehicle manufacturers, national sports bodies and large software vendors. He has over 10 years of JavaScript experience.

Together, David and Dawn have written several books in the *Head First* series, including *Head First Android Development* and *Head First Kotlin*, and delivered video courses for O’Reilly Media.