React Development

The Big Nerd Ranch Guide

Table of Contents

Acknowledgements

Loren Klingman guides the content for the book.

Authors By Section

- Ottergram was originally written by Chris Aquino (`@radishmouse`) and Todd Gandee (`@tgandee`).
- React Coffee Shop was written by Loren Klingman.

Thanks

- Chris Aquino and Todd Gandee the authors of *Front-End Web Development: The Big Nerd Ranch Guide* for the base of content for this book.
- Jake Sower for contributing to previous React sections of the book.
- Eric Wilson for guiding the process and handling the logistics for the required time and reviewers.
- Our proofreaders, technical reviewers, and guinea pigs: Josh Justice, Liv Vitale, Taylor Martin, Jeremy Sherman, Sean Farahdel, Adam Friedman. Thank you for volunteering as tribute.

Lastly, thank you to the countless students who have taken the training. Without your curiosity and your questions, none of this matters. This work is a reflection of the insight and inspiration you have given us over the span of those many weeks. We hope the coffee made the training a little lighter.

Introduction

Learning Web Development

Front-end development requires a shift in perspective if you've never built anything for the browser. Here are a few things to keep in mind as you get started.

The browser is the ubiquitous platform.

Perhaps you have done native development for iOS or Android; written server-side code in Go, Ruby or PHP; or built desktop applications for macOS or Windows. As a front-end developer, your code will target the browser – the only platform available on nearly every phone, tablet, and personal computer in the world.

Front-end development requires visual and programatic thinking.

At one end of the spectrum is the look and feel of a web page: rounded corners, shadows, colors, fonts, whitespace, and so on. At the other end of the spectrum is the logic that governs the intricate behaviors of that web page: swapping images in an interactive photo gallery, validating data entered into a form, sending messages across a chat network, etc. You will need to gain proficiency in several core technologies, and understand how those technologies work together to build great web applications.

Web technologies are open.

No one company controls the standards for web browsers. This means that front-end developers do not get a yearly SDK release that contains all the changes they will need to deal with for the next twelve months. Native platforms are a frozen pond on which you can comfortably skate. The web is a river; it curves, moves quickly, and is rocky in some places – but that is part of its appeal. The web is the most rapidly evolving platform available. Adapting to change is a way of life for a front-end developer.

Learning React

The purpose of this book is to teach you how to use React to write programs for the browser, and as experience is the best teacher, you will build a React application as your work through this course. Other front-end technologies such as HTML and CSS will be touched on, but our focus will be on React.

Prerequisites

This book is not an introduction to programming. It assumes you have experience with the fundamentals of writing code. You are expected to be familiar with basic types, functions, and objects.

That said, it also does not assume you already know JavaScript. It introduces you to JavaScript concepts in context, as you need them.

This book contains a brief introduction to web development in Ottergram, which will be enough for the course, but if you have not worked on front-end development before, you would benefit from books that take a deeper dive into HTML and CSS.

How This Book Is Organized

This book walks you through writing two different web applications. Each application has its own section of the book. Each chapter in a section adds new features to the application you are building.

As you go through each section of this book you will build a new web application, with each chapter focusing on a couple of features.

- *Ottergram* - This is an optional introduction to HTML and CSS.
- *Coffee Shop* - Part coffee order form, part checklist, Coffee Shop takes you through a number of React techniques for state, routing, and communicating with a server.

As you work through these applications, you will be introduced to a number of tools, including:

- the Visual Studio Code text editor for working with code
- documentation resources like the Mozilla Developer Network
- the command line, using the macOS `Terminal` app or the Windows command prompt
- Google Chrome's Developer Tools
- React, Redux, React Router
- Node.js and npm (the Node package manager)
- WebSockets and the wscat module

How to Use This Book

This is not a reference book. Its goal is to get you started with React development, so you can get the most out of the reference and recipe books available. It is based on our class at Big Nerd Ranch, and, as such, it is meant to be worked through in succession.

In our classes, students work through these materials, but they also benefit from the right environment – a dedicated classroom, good food, comfortable lodging, a group of motivated peers, and an instructor to answer questions.

As a reader, you want your environment to be similar. That means getting a good night's rest and finding a quiet place to work. These things can help, too:

- Start a reading group with your friends or coworkers.
- Arrange to have blocks of focused time to work on chapters.
- Participate in the forum for this book at forums.bignerdranch.com, where you can discuss the book and find errata and solutions.
- Find someone who knows React to help you out.

🎖 Challenges

Most chapters in this book end with at least one challenge. Challenges are opportunities to review what you have learned and take your work in the chapter one step further. We recommend that you tackle as many of them as you can to cement your knowledge and gain a deeper understanding of the concepts discussed.

Challenges come in three levels of difficulty:

- 🥉 Bronze challenges typically ask you to do something very similar to what you did in the chapter. These challenges reinforce what you learned in the chapter and force you to type in similar code without having it laid out in front of you. Practice makes perfect.
- 🥈 Silver challenges require you to do more digging and more thinking. Sometimes you will need to use functions, events, markup, and styles that you have not seen before, but the tasks are still similar to what you did in the chapter.
- 🥇 Gold challenges are difficult and can take hours to complete. They require you to understand the concepts from the chapter and then do some quality thinking and problem solving on your own. Tackling these challenges will prepare you for the real-world work of React development.

You should make a copy of your code before you work on the challenges for a given chapter. This will ensure your code remains compatible with subsequent exercises in the course.

🔎 For the More Curious

Many chapters also have "For the More Curious" sections. These sections offer deeper explanations or additional information about topics presented in the chapter. The information in these sections is not absolutely essential to understanding or completing this course, but we hope you will find it interesting and useful.

Setup

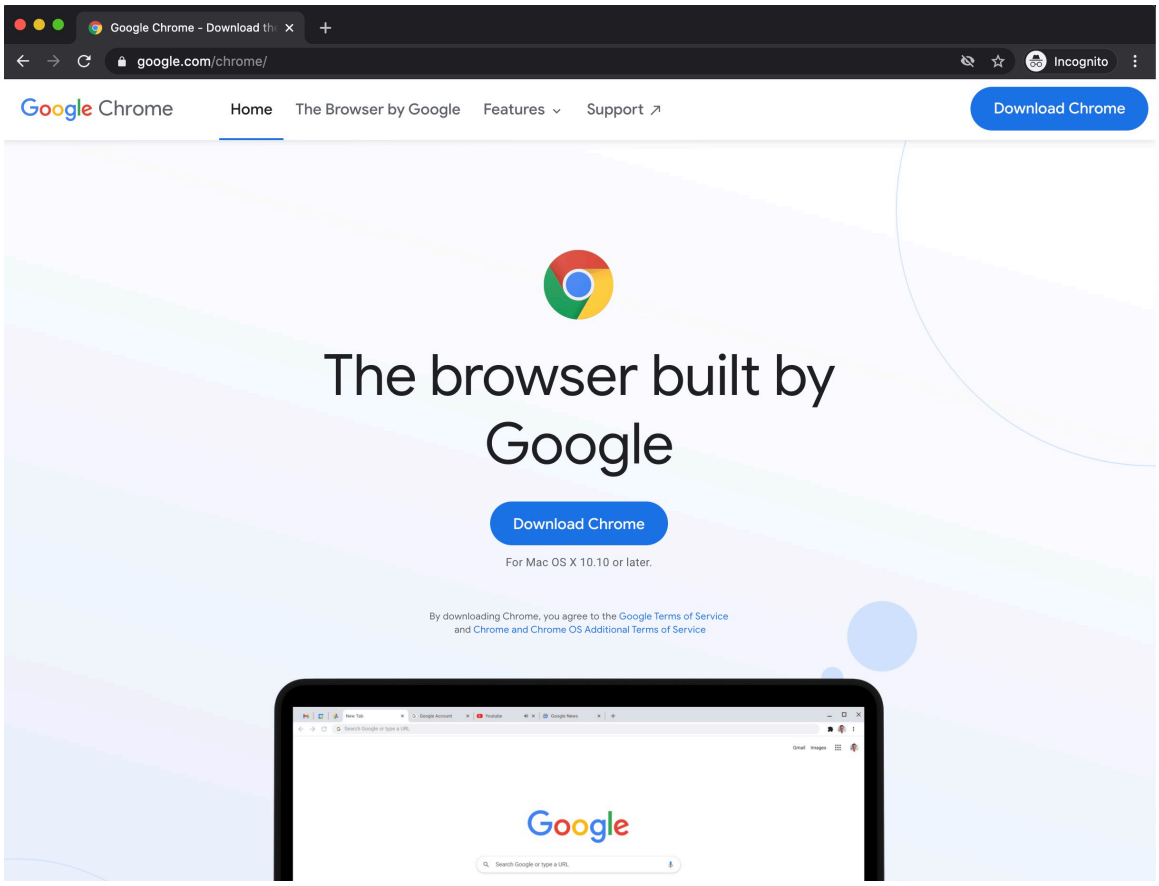
There are countless tools and resources for front-end development, with more being built all the time. Choosing the best ones is challenging for developers of all skill levels. We will guide through the use of some of our favorites in this course.

To get started, you will need three basic tools: a browser, a text editor, and good reference documentation for the many technologies used in front-end development. There are also several extras that – while not essential – will make your development experience smoother and more enjoyable.

For the purposes of this book we recommend that you use the same software we use to get the most benefit from our directions and screenshots. This chapter walks you through installing and configuring the `Google Chrome` browser, the `Visual Studio Code` text editor, `Node.js`, and a number of plug-ins and extras. You will also find out about good documentation options and get a crash course in using the *command line* on Mac and Windows. In the next chapter, you will put all these resources to use as you begin your first project.

Installing Google Chrome

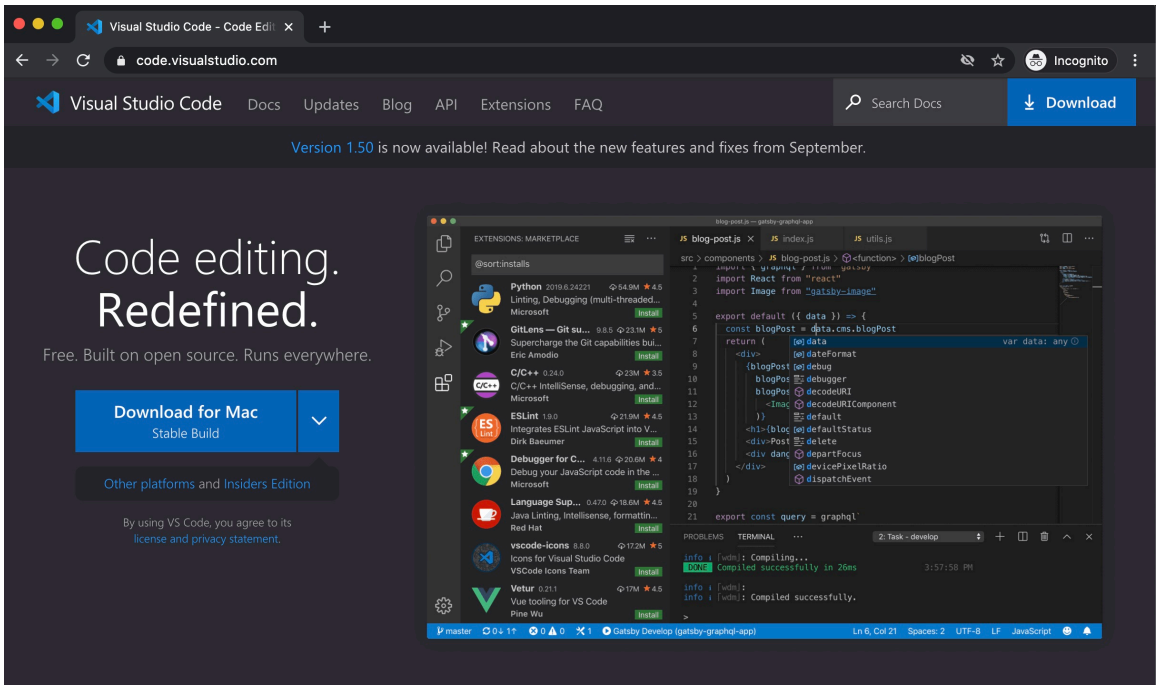
Your computer should already have a browser installed by default, but the best one to use for front-end development is `Google Chrome`. If you do not already have the latest version of `Chrome`, you can get it from www.google.com/chrome/ .



Installing Visual Studio Code

Of the many text editor programs out there, one of the best for front-end development is the `Visual Studio Code` editor by Microsoft. It is a good choice because it is highly configurable, has many extensions to help with writing code, and is free to download and use.

You can download `Visual Studio Code` for Mac or Windows from code.visualstudio.com .



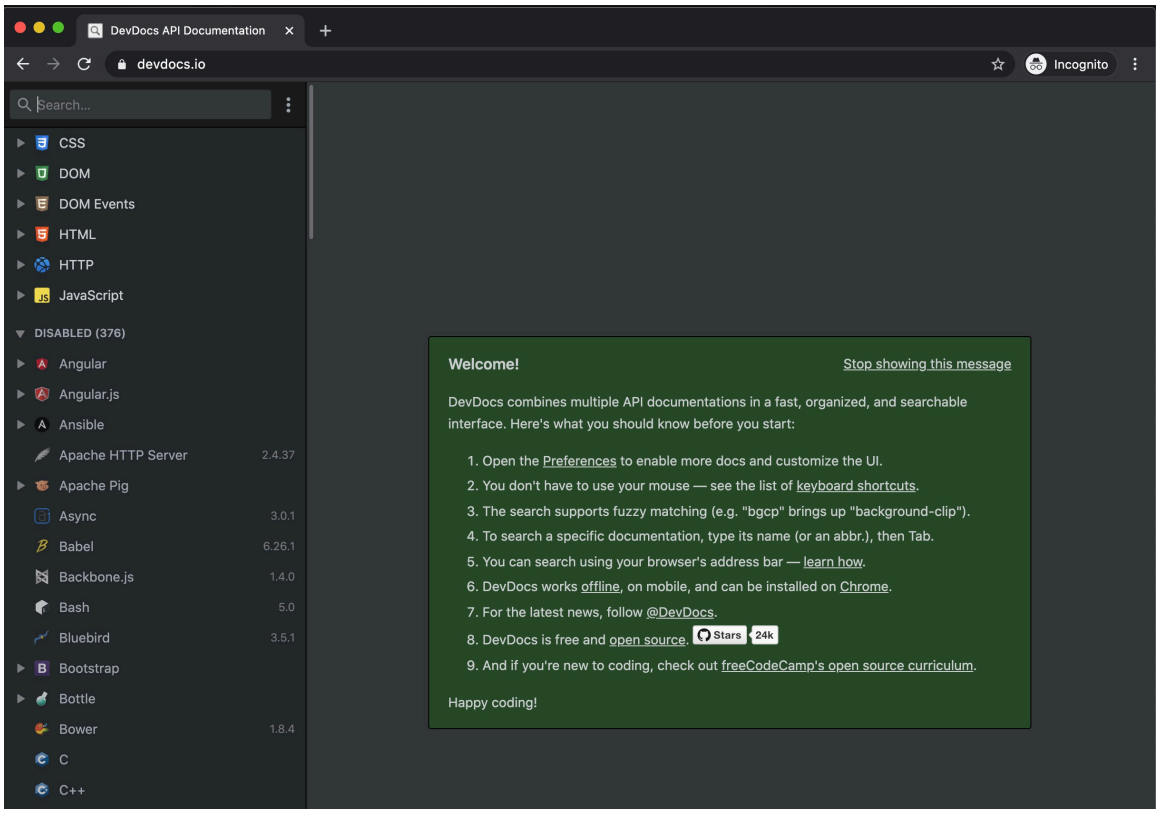
Follow the installation instructions for your platform.

`Chrome` and `Visual Studio Code` are now ready for front-end development. There are just a few more steps to completing your coding environment: accessing documentation, learning command-line basics, and downloading two final tools.

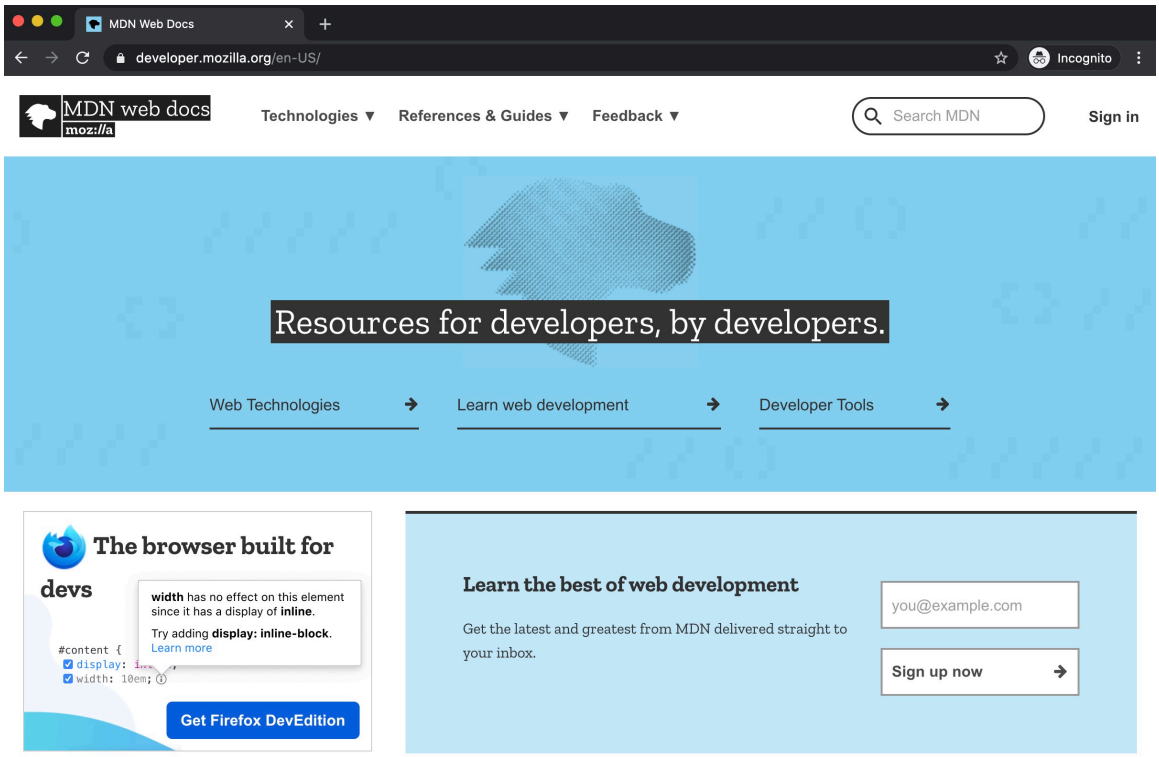
Finding Documentation and Help

Front-end development is different from programming for platforms like iOS and Android. Aside from the obvious differences, front-end technologies have no official developer documentation other than the technical specifications. This means that you will need to look elsewhere for guidance. We recommend that you familiarize yourself with the resources below and consult them regularly as you work through the book and continue on with front-end development.

The Mozilla Developer Network (MDN) is the best reference for anything to do with HTML, CSS, and JavaScript. You can access it through the following link, devdocs.io . Devdocs pulls documentation from MDN for core front-end technologies – and it can work offline, so you can check it even when you do not have an internet connection.

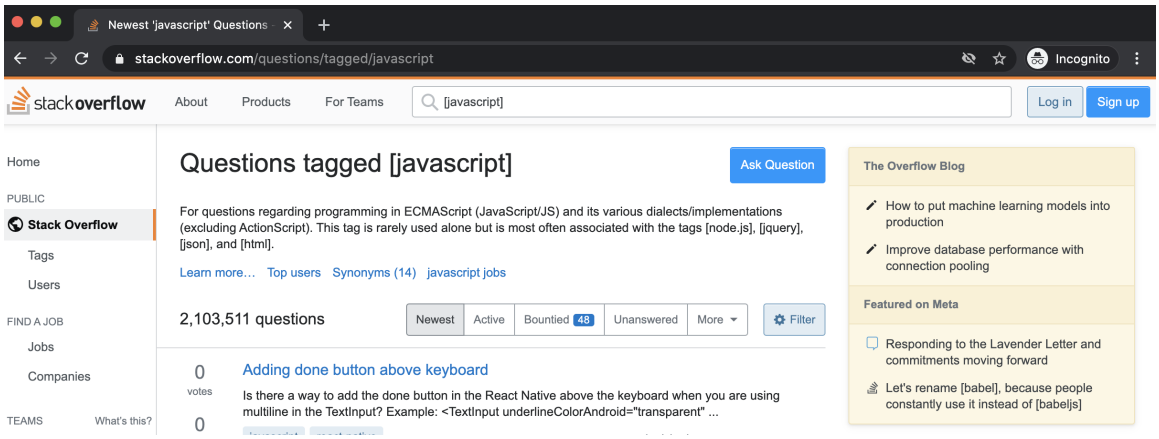


You can also use the Mozilla Developer Network (MDN) website, developer.mozilla.org .



Another site to know about is stackoverflow.com .

This is not an official source of documentation, but as a place where developers can ask questions and discusses various techniques and approaches it is invaluable. Because the answers on Stack Overflow are not official documentation, there are several things you should pay attention to when reading through the various posts. Pay attention to the date on the answers you are reading as certain answers may be correct at the time but can easily become outdated if too much time has passed. Because answers vary in quality, you should also note the number of up votes an answer receives and take time to review the comments on the various answers that are posted for a given question.

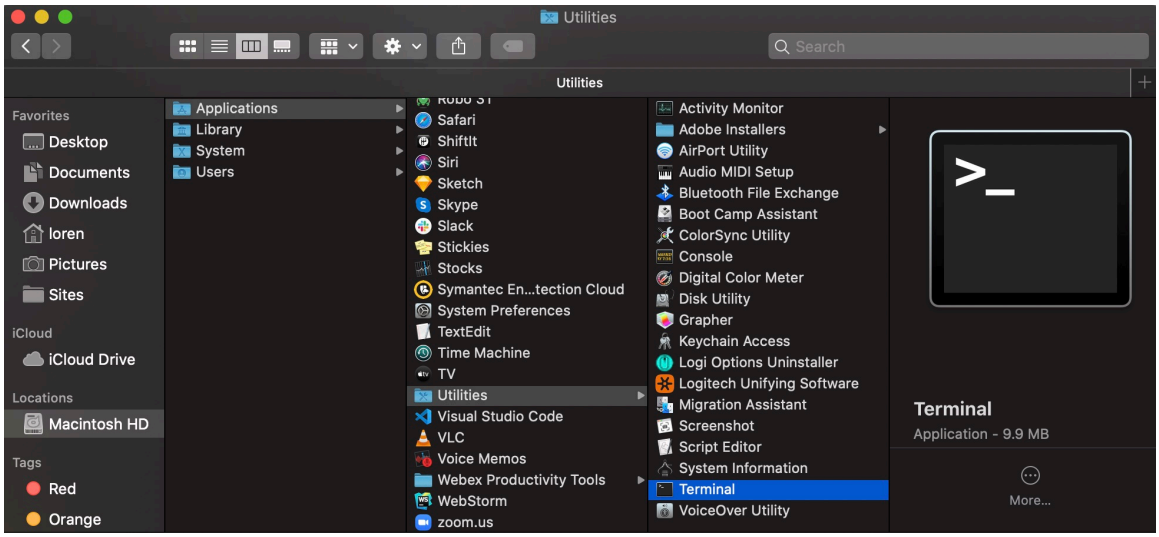


Web technologies are ever-changing. Support for features and APIs will vary from browser to browser and change over time. Two websites that can help you determine which browsers (and versions of those browsers) support what features are html5please.com and caniuse.com . When you need information about feature support, we suggest starting with html5please.com to know whether a feature is recommended for use. For more detailed information about which browser versions support a specific feature, go to caniuse.com .

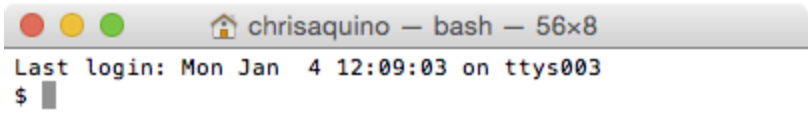
Crash Course in the Command Line

Throughout this book, you will be instructed to use the *command line* or *terminal*. Many of the tools you will be using run exclusively as command-line programs.

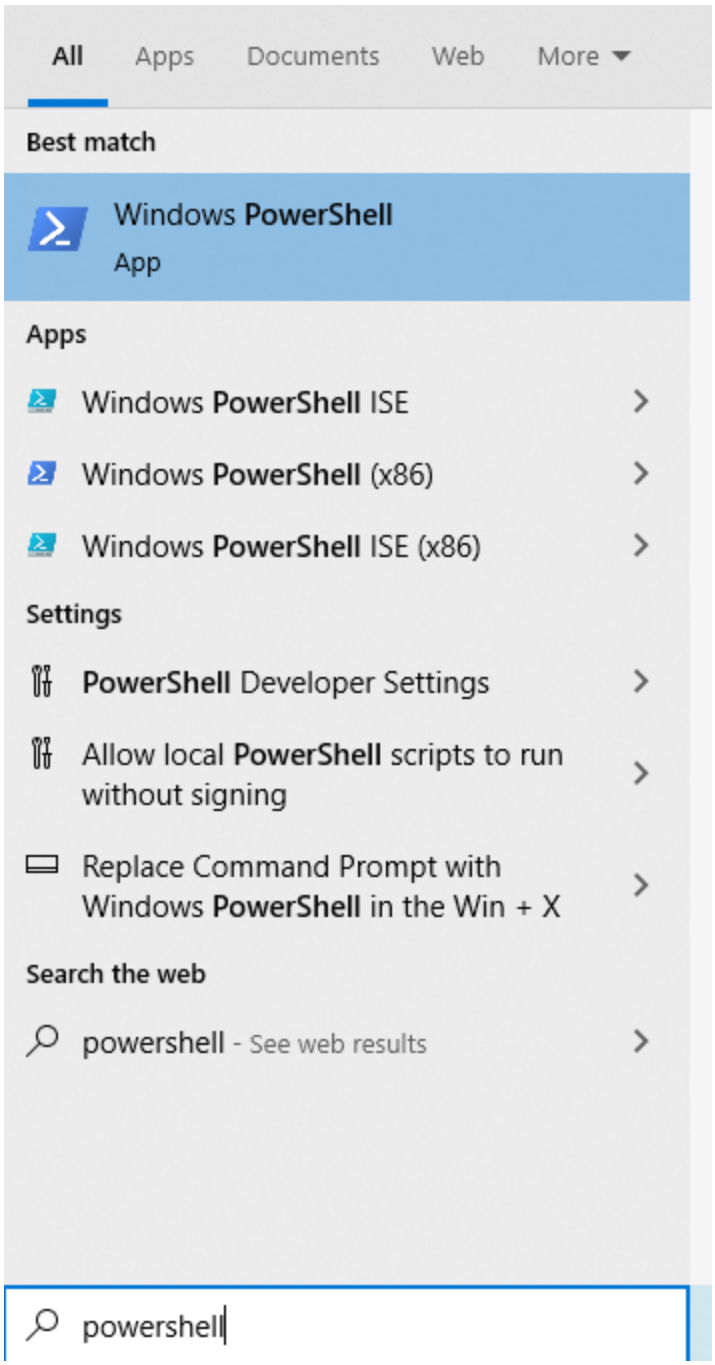
To access the command line on a Mac, open `Finder` and go to the **Applications** folder. Then, go to the **Utilities** folder. Find and open the program named `Terminal`.



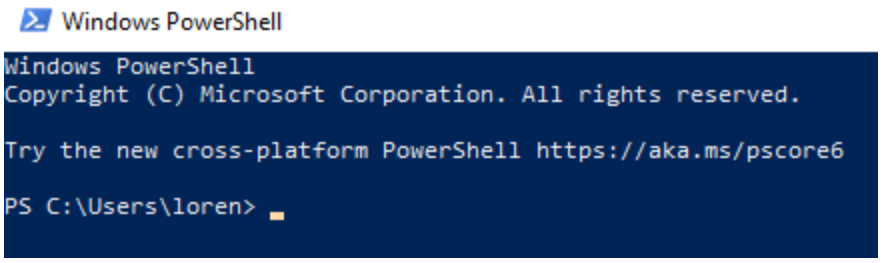
You should see a window that looks like this.



To access the command line on Windows, go to the Start menu and search for "PowerShell". Find and open the program named `Windows PowerShell`.



Click it to run the standard Windows command-line interface, which looks like this.



From now on, we will refer to "the terminal" or "the command line" to mean both the Mac `Terminal` and the Windows `Command Prompt`. If you are unfamiliar with using the command line, here is a short walk-through of some common tasks. All commands are entered by typing at the prompt and pressing the Return key.

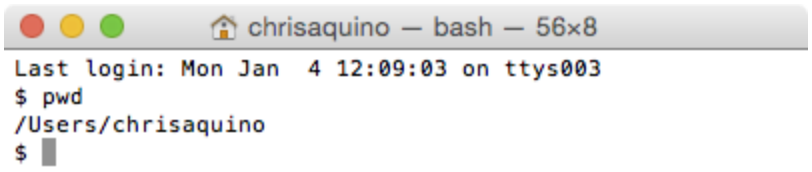
Windows Subsystem for Linux

If you are on Windows and it's possible, we recommend using the Windows System for Linux as it provides the best compatibility with Node modules.

You can learn more about installing it here: https://docs.microsoft.com/en-us/windows/wsl/ .

Finding the Current Directory

The command line is location based. That means that at any given time it is "in" a particular directory within the file structure, and any commands you enter will be applied within that directory. The command-line prompt shows an abbreviated version of the directory it is in. To see the whole path, enter the command `pwd` (which stands for "print working directory").



Changing Directories

To move around the file structure, you use the command `cd`, or "change directory," followed by the path of the directory you want to move into.

You do not always need to use the complete directory path in your `cd` command. For example, to move down into any subdirectory of the directory you are in, you simply use the name of the subdirectory. So when you are in your home directory (`/Users/YourName` or `C:\Users\YourName\`), the path to the Documents is just `Documents`.

Move into the directory where you would like to create your files:

```sh
cd Documents
```

To move up to the parent directory, use the command `cd ..` (that is, `cd` followed by a space and two periods). The pair of periods represents the path of the parent directory.

```sh
cd ..
```

Navigate to the directory where you would like to create your book. This might be `Documents` or `Desktop` or `Projects` or `Sites` depending on how you like to organize things on your machine.

```sh
cd Documents
```

Creating a Directory

The directory structure of front-end projects is important. Your projects can grow quickly, and it is best to keep them organized from the beginning. You will create new directories regularly during your development. This is done using the `mkdir` or "make directory" command followed by the name of the new directory.

To see this command in action, set up a directory for the projects you will build as you work through this book. Enter this command:

```sh
mkdir react-book
```

Next, create a new directory for your first project, Ottergram, which you will begin in the next chapter. You want this new directory to be a subdirectory of the `react-book` directory you just created. You

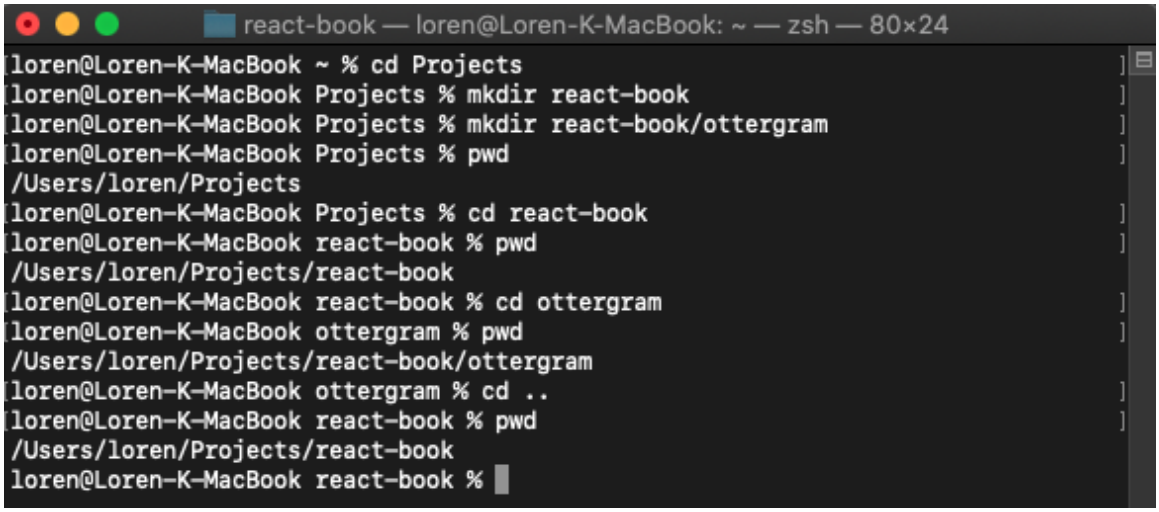
can do this from your home directory by prefacing the new directory name with the name of the projects directory and, on a Mac, a slash:

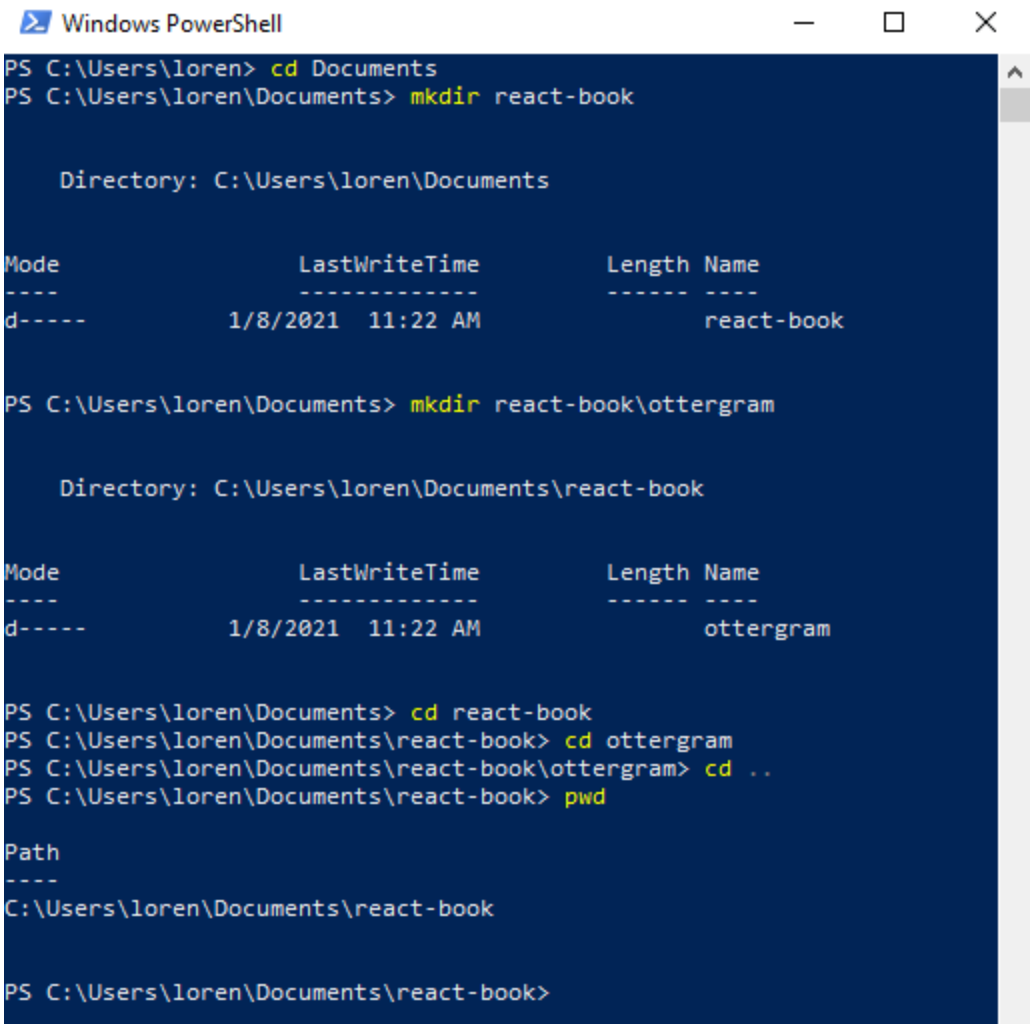
```sh
mkdir react-book/ottergram
```

On Windows, you use the backslash instead:

```sh
mkdir react-book\ottergram
```

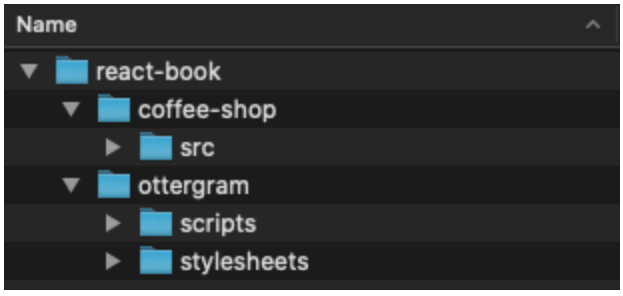
Remember that you can check your current directory by using the `pwd` command. Here is an example of the author creating directories, moving between them, and checking the current directory.





Changing Multiple Directories

You are not limited to moving up or down one directory at a time. Let's say that you had a more complex directory structure, like this one.



Suppose you are in the `ottergram` directory and you want to go directly to the `src` directory inside of `coffee-shop`. You would do this with `cd` followed by a path that means "the `src` directory

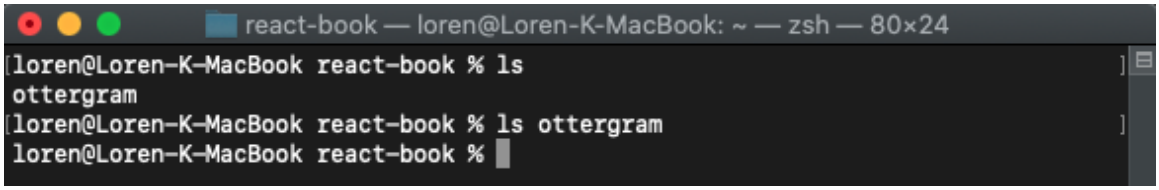
inside the `coffee-shop` directory inside the parent directory of where I am now":

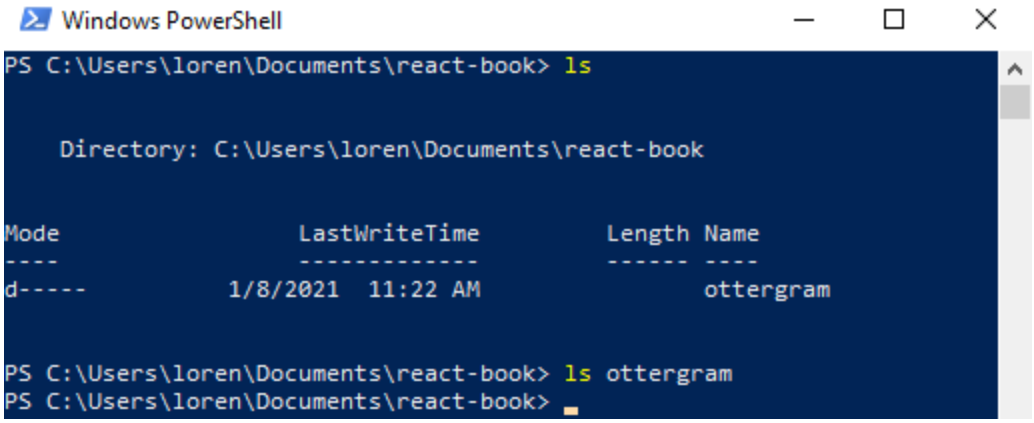
```sh
cd ../coffee-shop/src
```

Listing Files

You may need to see a list of files in your current directory. On a Mac, you use the `ls` command for that. If you want to list the files in another directory, you can supply a path:

```sh
ls
ls ottergram
```



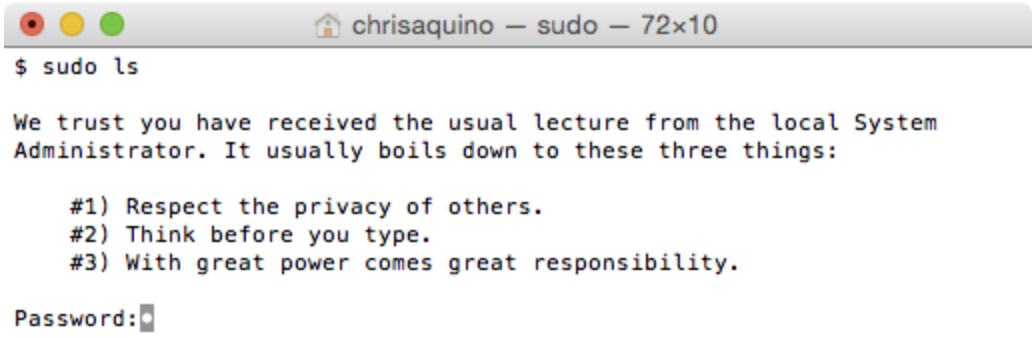


By default, `ls` will not print anything if a directory is empty.

Getting Administrator Privileges

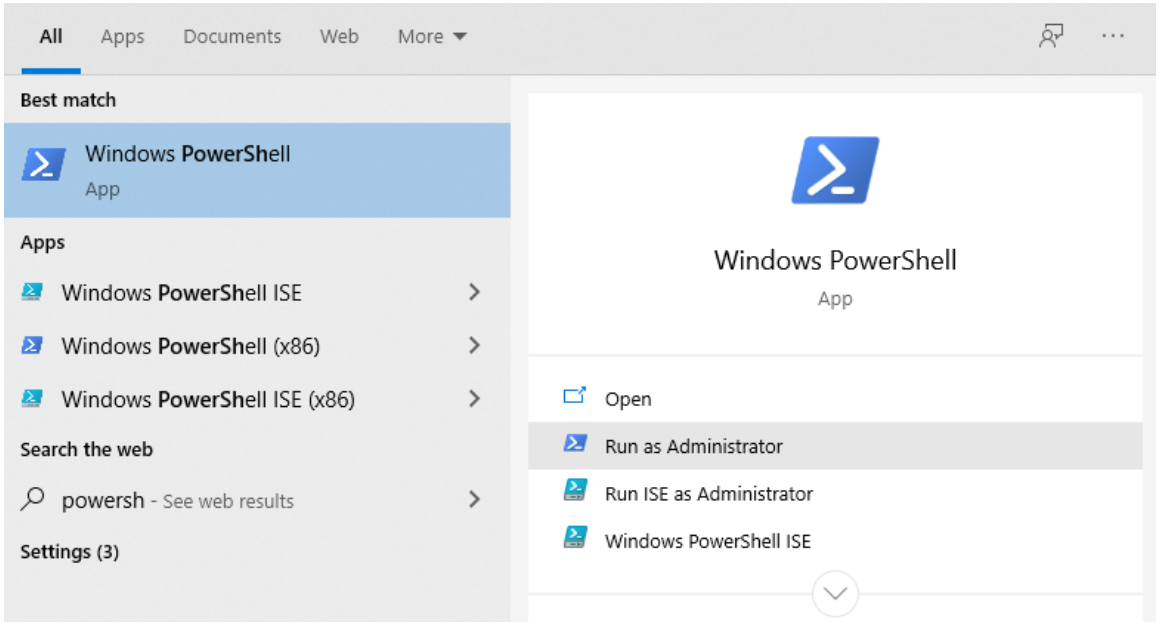
On some versions of macOS and Windows, you will need superuser or administrator privileges in order to run some commands, such as commands that install software or make changes to protected files.

On a Mac, you can give yourself privileges by prefixing a command with `sudo`. The first time you use `sudo` on a Mac, it will give you a stern warning.



`sudo` will prompt you for your password before it runs the command as the superuser. As you type, your keystrokes will not be echoed back, so type carefully.

On Windows, if you need to give yourself privileges you do so in the process of opening the PowerShell. Find the PowerShell in the Windows. Then, from the pane on the right, select "Run as Administrator". Any commands you run in this command prompt will be run as the superuser, so be careful.



> **Tip**
>
> For Windows users, you may need to change your script execution policy.
>
> If you get a warning about not being able to run one of the scripts for the class, you can change the policy as follows.
>
> 1. Start Windows PowerShell as an administrator.
> 2. Run `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned`.
> 3. Confirm with `Y` and press enter.
> 4. Close PowerShell and reopen normally, without administrator privileges.

Quitting a Program

As you proceed through the book, you will run many apps from the command line. Some of them will do their job and quit automatically, but others will run until you stop them. To quit a command-line program, press `Ctrl + C`. (Note: This command uses `Ctrl` on both Mac and Windows.)

Installing Node.js

There is one final set-up step before you begin your first project.

`Node.js` (or simply `Node`) lets you use programs written in JavaScript from the command line. Most front-end development tools are written for use with `Node.js`. You will learn lots more about `Node.js` later in the book.

Install `Node` by downloading the installer from nodejs.org. We recommend downloading the LTS or "Long Term Support" version. The version of `Node.js` used in this book is 16.13.0, and you will likely see a different version available for download.

Double-click the installer and follow the prompts.

When you install `Node`, it provides two command-line programs: `node` and `npm`. The `node` program does the work of running programs written in JavaScript. You will not need this until later. The other program is the `Node` package manager, `npm`, which is needed for installing open-source development tools from the internet.

🔎 Alternatives to Visual Studio Code

There are many, many text editors to choose from. If you are not that keen on `Visual Studio Code`, when you are done working through the projects in this book you may want to try out one of the following two options.

Atom is an open source text editor, made by GitHub specifically for developing web applications. It can be downloaded from atom.io. It is free and available for macOS and Windows and has a large number of plug-ins to customize your development experience. It is also built using HTML, CSS, and JavaScript, but runs as a desktop application.

Another popular editor is WebStorm by JetBrains. Unlike the others, it is a paid application available for Windows, macOS, and Linux. It also has a number of plug-ins for customization. JetBrains also makes Android Studio, PyCharm, IntelliJ, and many other IDEs. All of their IDEs have a similar look and feel and share the same keyboard shortcuts. It can be downloaded from https://www.jetbrains.com/webstorm/.

Ottergram

Foundation

When you visit a website, your browser has a conversation with a server, another computer on the internet.

Browser: "Hey there! Can I please have the contents of the file named `cat-videos.html`?"

Server: "Certainly. Let me take a look around ... here it is!"

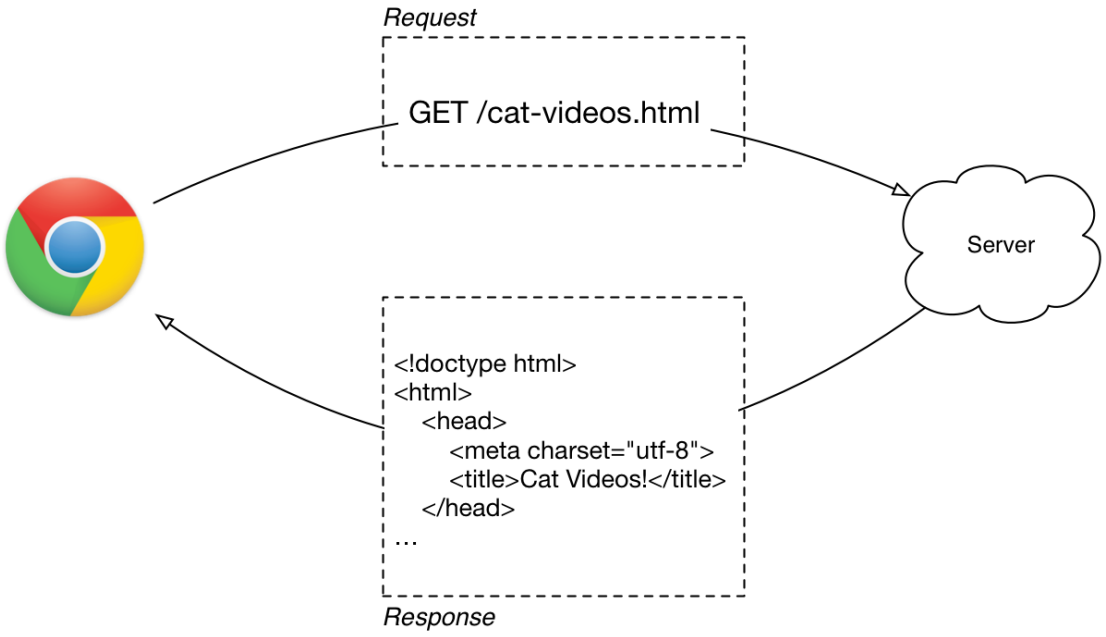
Browser: "Ah, it's telling me that I need another file named `styles.css`."

Server: "Sure thing. Let me take a look around ... here it is!"

Browser: "OK, that file says that I need another file named `animated-background.gif`."

Server: "No problem. Let me take a look around ... here it is!"

That conversation goes on for some time, sometimes lasting thousands of milliseconds.



It is the browser's job to send requests to the server; interpret the HTML, CSS, and JavaScript it receives in the response from the server; and present the result to the user. Each of these three technologies plays a part in the user's experience of a website. If your app were a living creature, the

HTML would be its skeleton and organs (the mechanics), the CSS would be its skin (the visible layer), and the JavaScript would be its personality (how it behaves).

In this chapter, you are going to set up the basic HTML for your first project, Ottergram. In the next chapter, you will set up your CSS, which you will refine later with flexbox. In later, you will begin adding JavaScript.

Installing Browser Sync

`browser-sync` makes your example code easier to run in the browser and automatically reloads the browser when you save changes to your code.

Install `browser-sync` using this command at the command line:

```sh
npm install -g browser-sync
```

(The `-g` in the command stands for "global." Installing the package globally means that you will be able to run `browser-sync` from any directory.)

It does not matter what directory you are in when you run this command, but you will may need superuser privileges. If that is the case, run the command using `sudo` on a Mac:

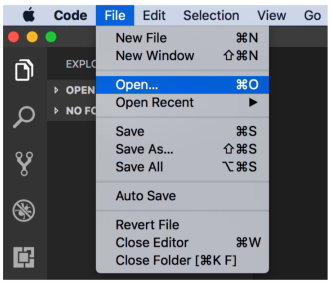
```sh
sudo npm install -g browser-sync
```

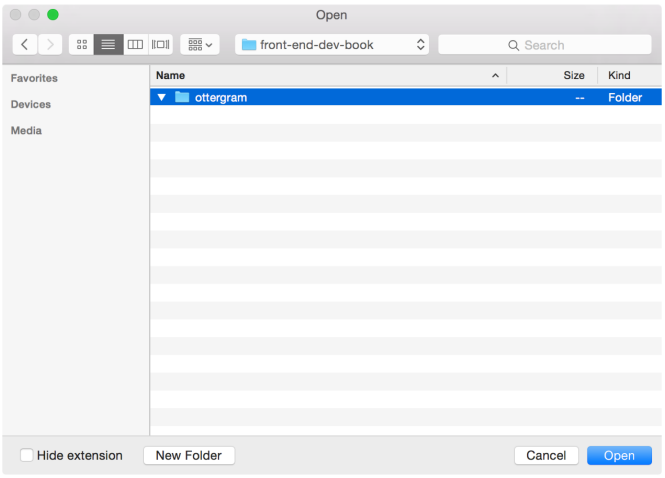
If you are on Windows, first open a command prompt as the administrator, as shown above.

When you start `browser-sync`, as you will later in this chapter, it will run until you press Control-C. It is a good idea to quit `browser-sync` when you are done working on a project for a while. That means that you will need to start `browser-sync` each time you begin work on Ottergram.

Setting Up Ottergram

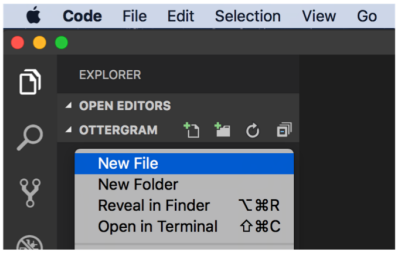
In the last chapter, you created a folder for the projects in this book as well as a folder for Ottergram. Start your `Visual Studio Code` text editor and open the `ottergram` folder by clicking `File` -> `Open...`. In the dialog box, navigate to the `react-book` folder and choose the `ottergram` folder. Click `Open` to tell `Visual Studio Code` to use this folder.

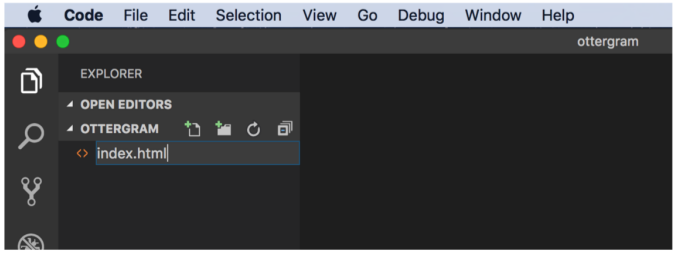




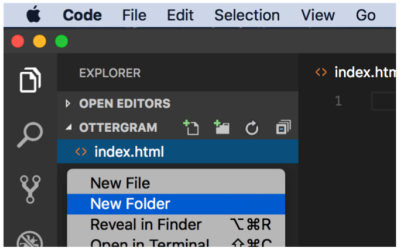
You will see the `ottergram` folder in the left-hand panel of `Visual Studio Code`. This panel is for navigating among the files and folders in your project.

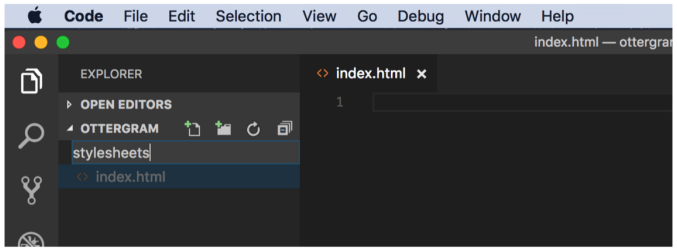
You are going to create some files and folders within the `ottergram` project folder using `Visual Studio Code`. Control-click (right-click) below `ottergram` in the left-hand panel and click `New File` in the pop-up menu. You will be prompted for a name for the new file. Enter `index.html` and press the Return key.



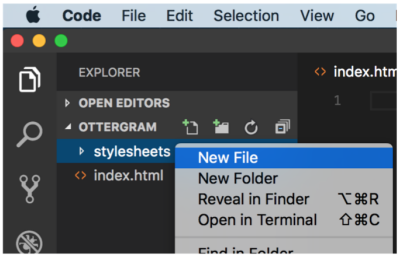


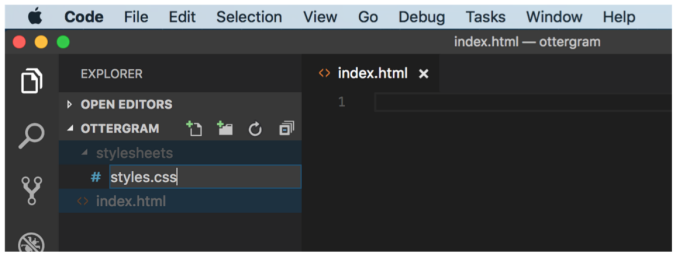
You can use the same process to create folders using `Visual Studio Code`. Control-click (right-click) below `ottergram` in the left-hand panel again, but this time click `New Folder` in the pop-up. Enter the name `stylesheets` in the prompt that appears.



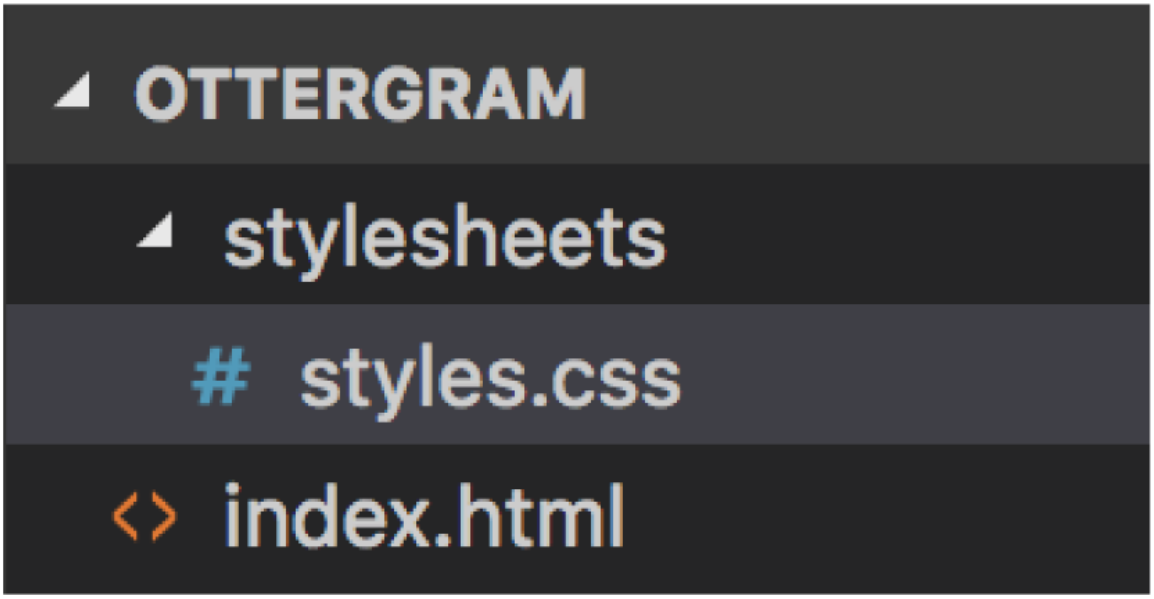


Finally, create a file named `styles.css` in the `stylesheets` folder: Control-click (right-click) `stylesheets` in the left-hand panel and choose `New File`. Enter `styles.css` and press the Return key.





When you are finished, your project folder should look like this.



There are no rules about how to structure your files and folders or what to name them. However, Ottergram (like the other projects in this book) follows conventions used by many front-end developers. Your `index.html` file will hold your HTML code. Naming the main HTML file `index.html` dates back to the early days of the web, and the convention continues today.

The `stylesheets` folder, as the name suggests, will hold one or more files with styling information for Ottergram. These will be CSS, or "cascading style sheets," files. Sometimes developers give their CSS files names that describe what part of the page or site they pertain to, such as `header.css` or `blog.css`. Ottergram is a simple project and only needs one CSS file, so you have named it `styles.css` to reflect its global role.

Initial HTML

Time to get coding. Let's create the foundation for your first project.

Open `index.html` in `Visual Studio Code` and add the basic HTML skeleton of your document.

```
+ <!DOCTYPE html>
+ <html>
+   <head>
+     <meta charset="utf-8">
+     <title></title>
+   </head>
+   <body>
+
+   </body>
+ </html>
```

Move your cursor between `<title>` and `</title>` – the opening and closing `title` tags. Type "ottergram" to give the project a name. Now, click to put your cursor in the blank line between the opening and closing `body` tags. There, type "header" and press the Tab key. `Visual Studio Code` will convert the text "header" into opening and closing `header` tags.

Next, add a blank line between the header tags. Then, type "h1" and press Tab. Again, your text is converted into tags. Enter the text "ottergram" again. This will be the heading that will appear on your web page.

Your file should look like this:

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
  </head>
  <body>
    <header>
      <h1>ottergram</h1>
    </header>
  </body>
</html>
```

`Visual Studio Code` has saved you some typing and helped you build well-formed initial HTML.

Let's examine your code. The first line, `<!DOCTYPE html>`, defines the *doctype* – it tells the browser which version of HTML the document is written in. The browser may render, or draw, the page a little

differently based the doctype. Here, the doctype specifies HTML5.

Earlier versions of HTML often had long, convoluted, and hard to remember doctypes, such as:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Often, folks had to look up the doctype each time they created a new document.

With HTML5, the doctype is short and sweet.

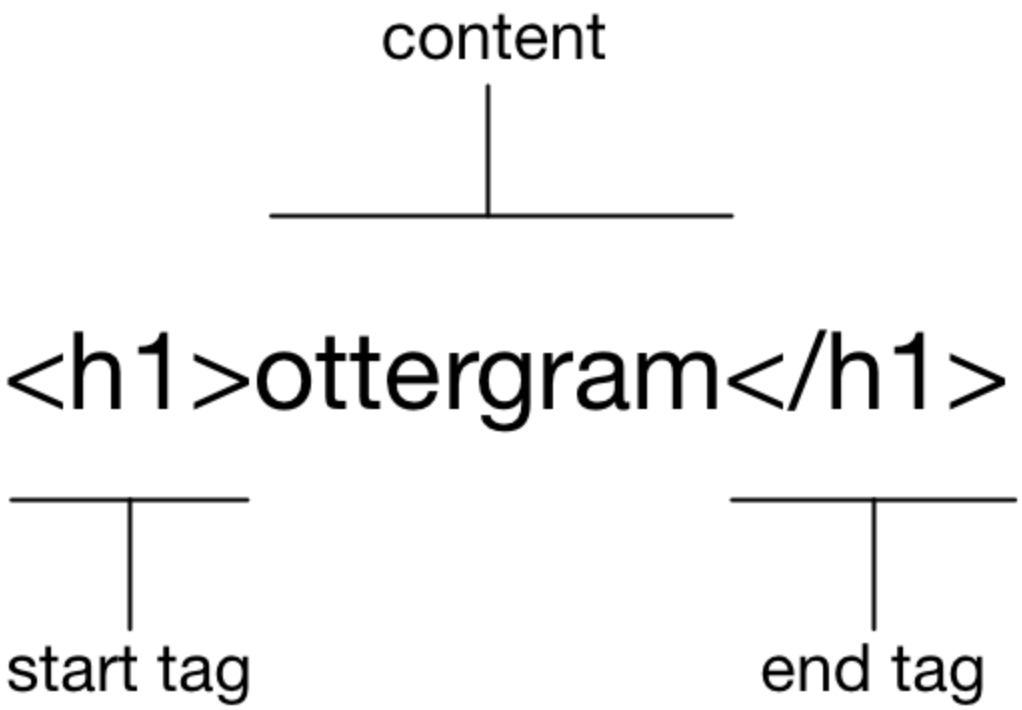
After the doctype is some basic HTML markup consisting of a `head` and a `body`.

The `head` will hold information about the document and how the browser should handle the document. For example, the title of the document, what CSS or JavaScript files the page uses, and when the document was last modified are all included in the `head`.

Here, the `head` contains a `<meta>` tag. `<meta>` tags provide the browser with information about the document itself, such as the name of the document's author or keywords for search engines. The `<meta>` tag in Ottergram, `<meta charset="utf-8">`, specifies that the document is encoded using the UTF-8 character set, which encompasses all Unicode characters. Use this tag in your documents so that the widest range of browsers can interpret them correctly, especially if you expect international traffic.

The `body` will hold all of the HTML code that represents the content of your page: all the images, links, text, buttons, and videos that will appear on the page.

Most tags enclose some other content. Take a look at the `h1` heading you included; its anatomy is shown below.



HTML stands for "hypertext markup language." Tags are used to "mark up" your content, designating their purpose – such as headings, list items, and links.

The content enclosed by a set of tags can also include other HTML. Notice, for example, that the `<header>` tags enclose the `<h1>` tag shown above (and the `<body>` tags enclose the `<header>` !).

There are a lot of tags to choose from – more than 140. To see a list of them, visit the MDN HTML element reference, located at developer.mozilla.org/en-US/docs/Web/HTML/Element . This reference includes a brief description of each element and groups elements by usage (e.g., text content, content sectioning, or multimedia).

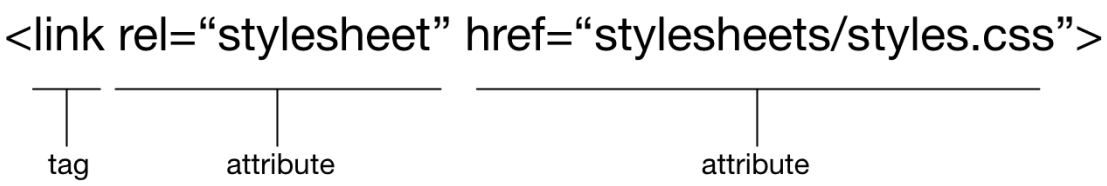
Linking a stylesheet

Later, you will write styling rules in your stylesheet, `styles.css`. But remember the conversation between the browser and the server at the beginning of this chapter? The browser only knows to ask for a file from the server if it has been told that the file exists. You have to *link* to your stylesheet so that the browser knows to ask for it. Update the `head` of `index.html` with a link to your `styles.css` file.

```
<!DOCTYPE html>
<html>
  <head>
```

```
      <meta charset="utf-8">
      <title>ottergram</title>
+     <link rel="stylesheet" href="stylesheets/styles.css">
    </head>
    <body>
  ...
```

The `<link>` tag is how you attach an external stylesheet to an HTML document. It has two *attributes*, which give the browser more information about the tag's purpose. (The order of HTML attributes does not matter.)



You set the `rel` (or "relationship") attribute to `"stylesheet"`, which lets the browser know that the linked document provides styling information. The `href` attribute tells the browser to send a request to the server for the `styles.css` file located in the `stylesheets` folder. Note that this file path is *relative* to the current document.

Save `index.html` before you move on.

Adding content

A web page without content is like a day without coffee. Add a list after your `header` to give your project a reason for living.

You are going to add an *unordered list* (that is, a bulleted list) using the `<ul>` tag. In the list, you will include five list items using `<li>` tags, and in each list item you will include some text surrounded by `<span>` tags.

The updated `index.html` is shown below. Note that throughout this book we show new code that you are adding has a `+` before the line. Code that you are to delete has a `-` before the line. Existing code is shown with nothing before the line to help you position your changes within the file.

We encourage you to make use of Visual Studio Code's autocompletion and autoformatting features. With your cursor in position, type "ul" and press Tab. Next, press Return to create a new line. Type "li" and press Tab. Press Return to create another new line. Then, type "span" and press Tab. Enter the name of an otter. Then, create four more list items and spans in the same way.

```
<!DOCTYPE html>
<html>
  <head>
```

```
      <meta charset="utf-8">
      <title>ottergram</title>
      <link rel="stylesheet" href="stylesheets/styles.css">
    </head>
    <body>
      <header>
        <h1>ottergram</h1>
      </header>
+     <ul>
+       <li>
+         <span>Barry</span>
+       </li>
+       <li>
+         <span>Robin</span>
+       </li>
+       <li>
+         <span>Maurice</span>
+       </li>
+       <li>
+         <span>Lesley</span>
+       </li>
+       <li>
+         <span>Barbara</span>
+       </li>
+     </ul>
    </body>
  </html>
```

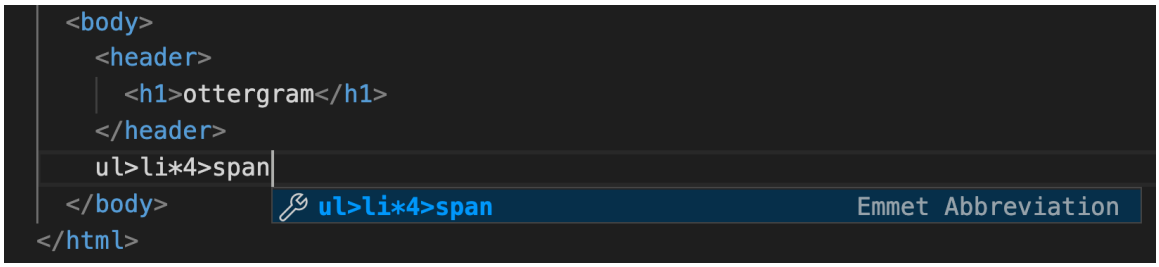
The `<span>` tags nested inside each `<li>` tag do not have any special meaning. They are generic containers for other content. You will be using them in Ottergram for styling purposes, and you will see other examples of container elements as you continue through this book.

Next, you will add images of otters to go with the names you have entered.

🔎 For the more curious: Emmet

Emmet is a syntax for quickly creating code. You'll notice when you've used tab to expand that VS Code has had a note "Emmet Abbreviation" in the popup window. Emmet uses syntax similar to CSS selectors to generate code. (We'll discuss CSS selectors in detail in a later chapter in this section.)

Using Emmet, you can quickly generate the same structure as above in one command by typing `ul>li*4>span` and followed by `tab`.



Once, you've pressed tab to expand the abbreviation, you can type the first otter's name. Then, press tab and VS Code will jump you into the space between the next two `span` tags so you can type the name of the second otter. Press tab again, and type the third otter's name. Press tab one last time, and enter the name of the fourth otter.

You can find out more about Emmet at https://docs.emmet.io/ .

Adding images

The resource files for all the projects in this book are at https://reactbook.bignerdranch.com/resources.zip . They include five Creative Commons-licensed otter images taken by Michael L. Baird, Joe Robertson, and Agunther that were found on commons.wikimedia.org .

Download and unzip the resources. Inside the `ottergram-resources` folder, locate the `img` folder. Copy the `img` folder to your `ottergram/` project directory. (The .zip contains other resources, but for now you will only need the `img` folder.)

You want your list to include clickable thumbnail images in addition to the titles. You will achieve this by adding anchor and image tags to each item in your `ul`. We will explain these changes in more detail after you enter them. (If you use autocompletion, note that you will need to move the `</a>` tags so that they follow the `span` s.)

```
  ...
      <ul>
        <li>
+         <a href="#">
+           <img src="img/otter1.jpg" alt="Barry the Otter" />
            <span>Barry</span>
+         </a>
        </li>
        <li>
+         <a href="#">
+           <img src="img/otter2.jpg" alt="Robin the Otter" />
            <span>Robin</span>
+         </a>
        </li>
        <li>
+         <a href="#">
+           <img src="img/otter3.jpg" alt="Maurice the Otter" />
```

```
            <span>Maurice</span>
+         </a>
        </li>
        <li>
+         <a href="#">
+           <img src="img/otter4.jpg" alt="Lesley the Otter" />
            <span>Lesley</span>
+         </a>
        </li>
        <li>
+         <a href="#">
+           <img src="img/otter5.jpg" alt="Barbara the Otter" />
            <span>Barbara</span>
+         </a>
        </li>
      </ul>
  ...
```

If your lines are not nicely indented, you can take advantage of Visual Studio Code's built-in HTML formatting support. Select the code you want to format. Then, press `Cmd+K Cmd+F` and your code will be aligned and indented for you. On Windows, press `Ctrl+K Ctrl+F`.

Let's look at what you have added.

The `<a>` tag is the *anchor* tag. Anchor tags make elements on the page clickable, so that they take the user to another page. They are commonly referred to as "links," but beware: They are not like the `<link>` tag you used earlier.

Anchor tags have an `href` attribute, which indicates the resource the anchor points to. Usually the value is a web address. Sometimes, though, you do not want a link to go anywhere. That is the case for now, so you assigned the "dummy" value `#` to the `href` attributes. This will make the browser scroll to the top of the page when the image is clicked. Later you will use the anchor tags to open a larger copy of an image when the thumbnail is clicked.

Inside the anchor tags you added `<img>`, or *image*, tags with `src` attributes indicating filenames in the `img` directory you added earlier. You also added a descriptive `alt` attribute to your image tags. `alt` attributes contain text that replaces the image if it is unable to load. `alt` text is also what screen readers use to describe an image to a user with a visual impairment.

Image tags are different from most other elements in that they do not wrap other elements, but instead refer to a resource. When the browser encounters an `<img>` tag, it draws the image to the page. This is known as a *replaced element*. Other replaced elements include embedded documents and applets.

Because they do not wrap content or other elements, `<img>` tags do not have a corresponding closing tag. This makes them *self-closing* tags (also known as *void* tags). You will sometimes see self-closing tags written with a slash before the right angle-bracket, like `<img src="otter.jpg"/>`. Whether to include the slash is a matter of preference and does not make a difference to the browser.

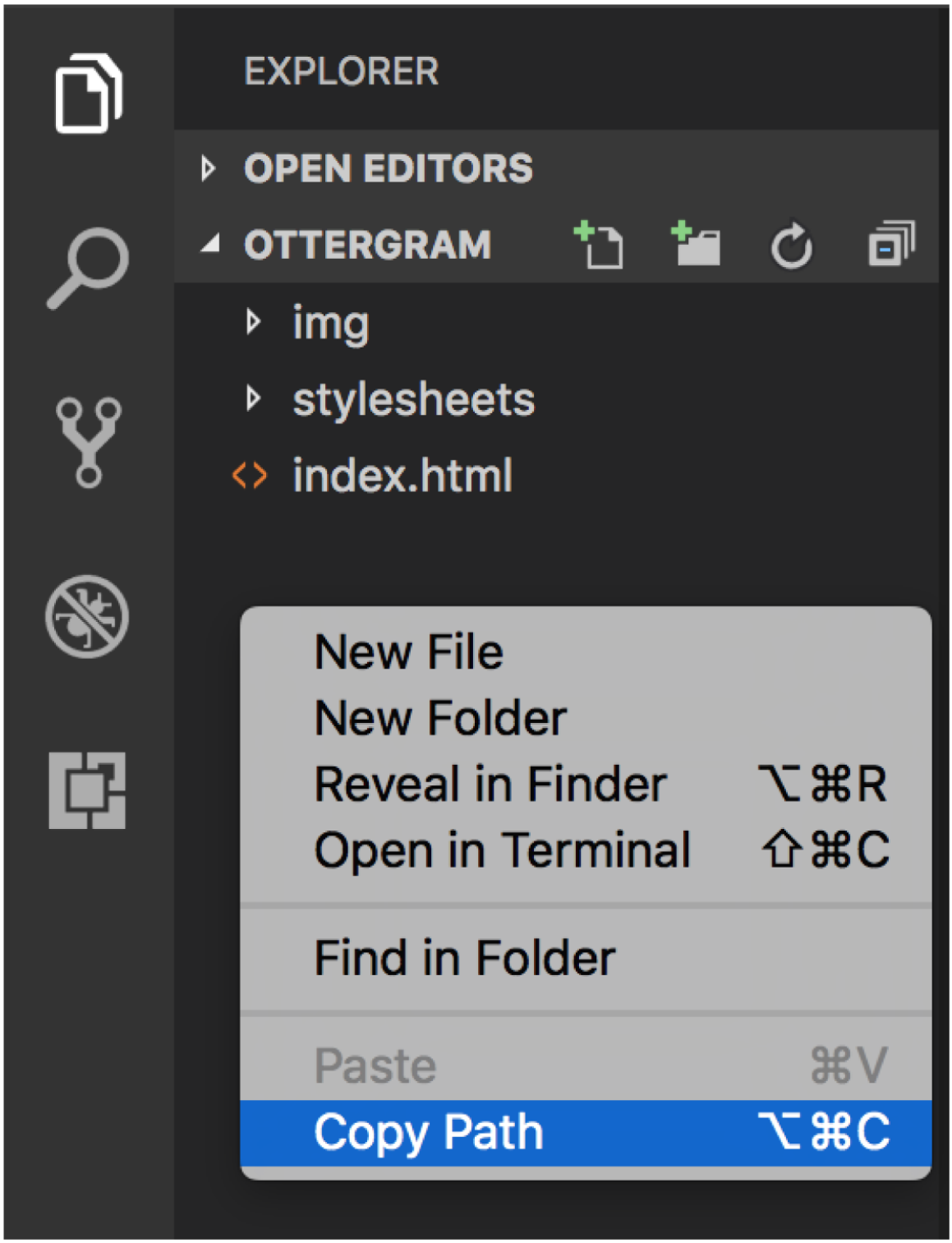
When using React, including the self-closing slash will be required by the JSX processor. Therefore, we will include the slash in this section of the book as well.

Save `index.html`. In a moment, you will see the results of your coding.

Viewing the Web Page in the Browser

To view your web page, you need to be running the `browser-sync` tool that you installed during setup.

Open the terminal and change directory to your `ottergram` folder. Recall from the setup chapter that you change the directory using the `cd` command followed by the path of the folder you are moving into. One easy way to get the `ottergram` path is to Control-click (right-click) below the `ottergram` files in `Visual Studio Code`'s left-hand panel and choose `Copy Path`. Then, at the command line, type `cd `, paste the path, and press Return.



The path you enter might look something like this:

```sh
cd /Users/chrisaquino/Documents/react-book/ottergram
```

Once you are in the `ottergram` directory, run the following command to open Ottergram in the Chrome Browser. (We have broken the command across two lines so that it fits on the page. You should enter it on a single line.)

```sh
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

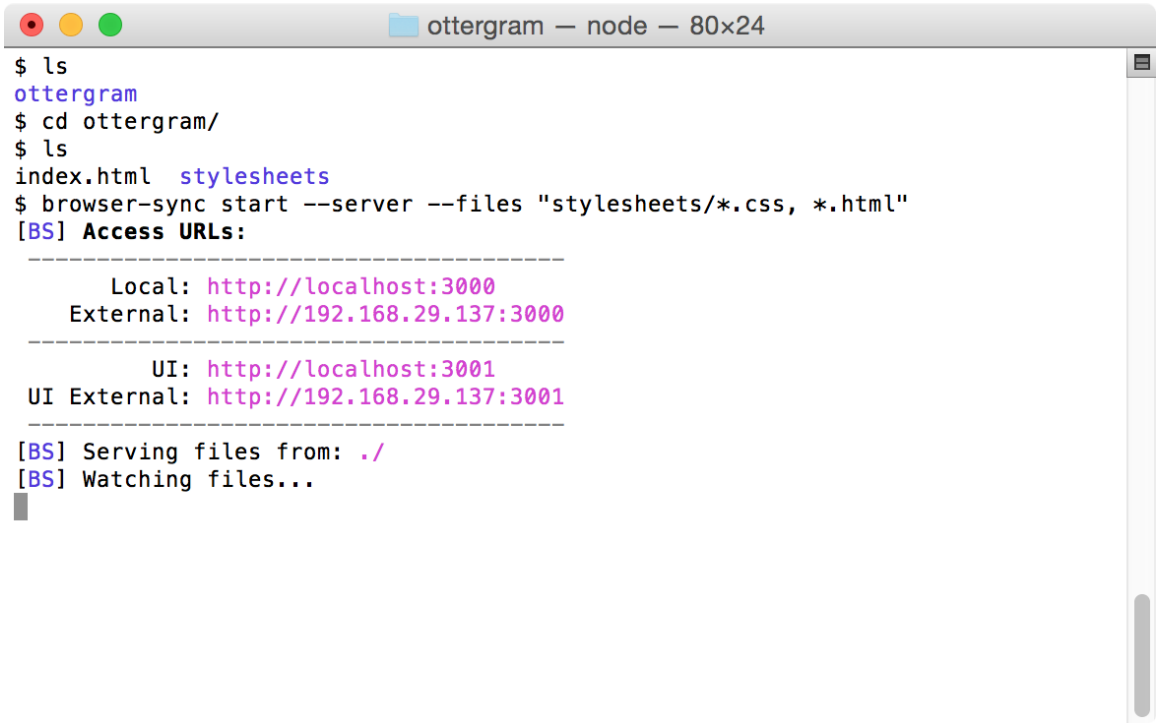
If Chrome is your default browser, you can leave out the `--browser "Google Chrome"` portion of the command:

```sh
browser-sync start --server --files "stylesheets/*.css, *.html"
```

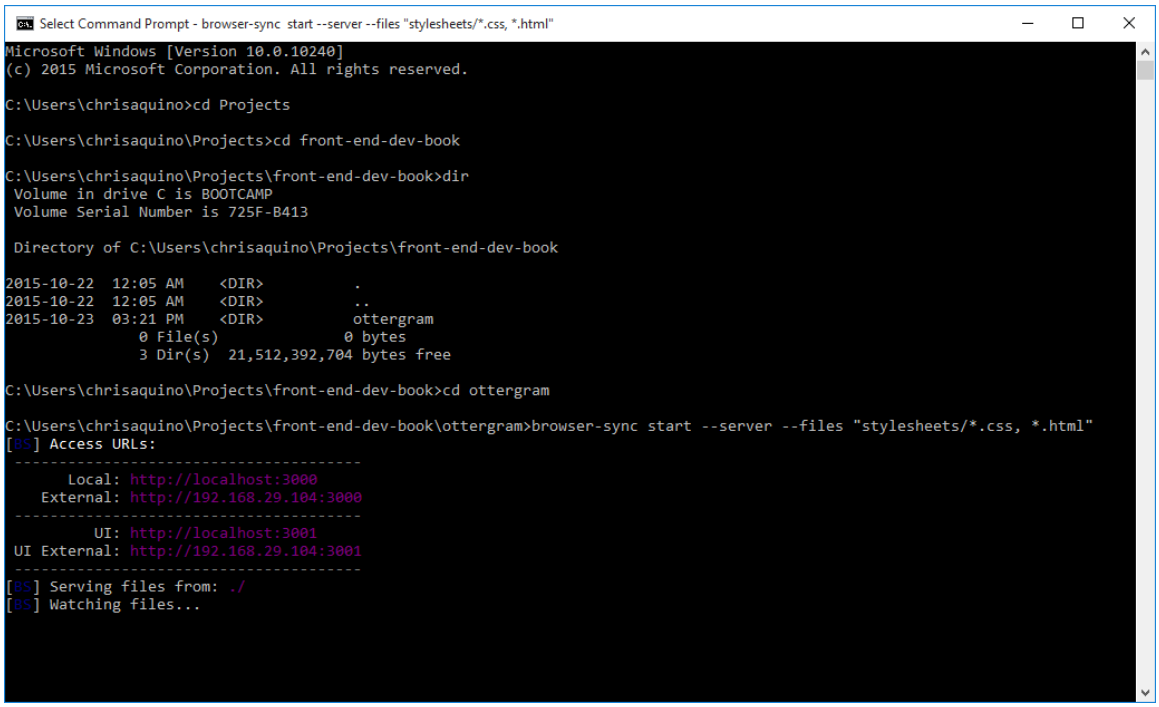
This command starts `browser-sync` in server mode, allowing it to send responses when a browser sends a request to get a file, such as the `index.html` file you created.

The command you entered also tells `browser-sync` to automatically reload the browser if any HTML or CSS files are changed. This makes the development experience much nicer. Before tools like `browser-sync`, you had to manually reload the page after every change.

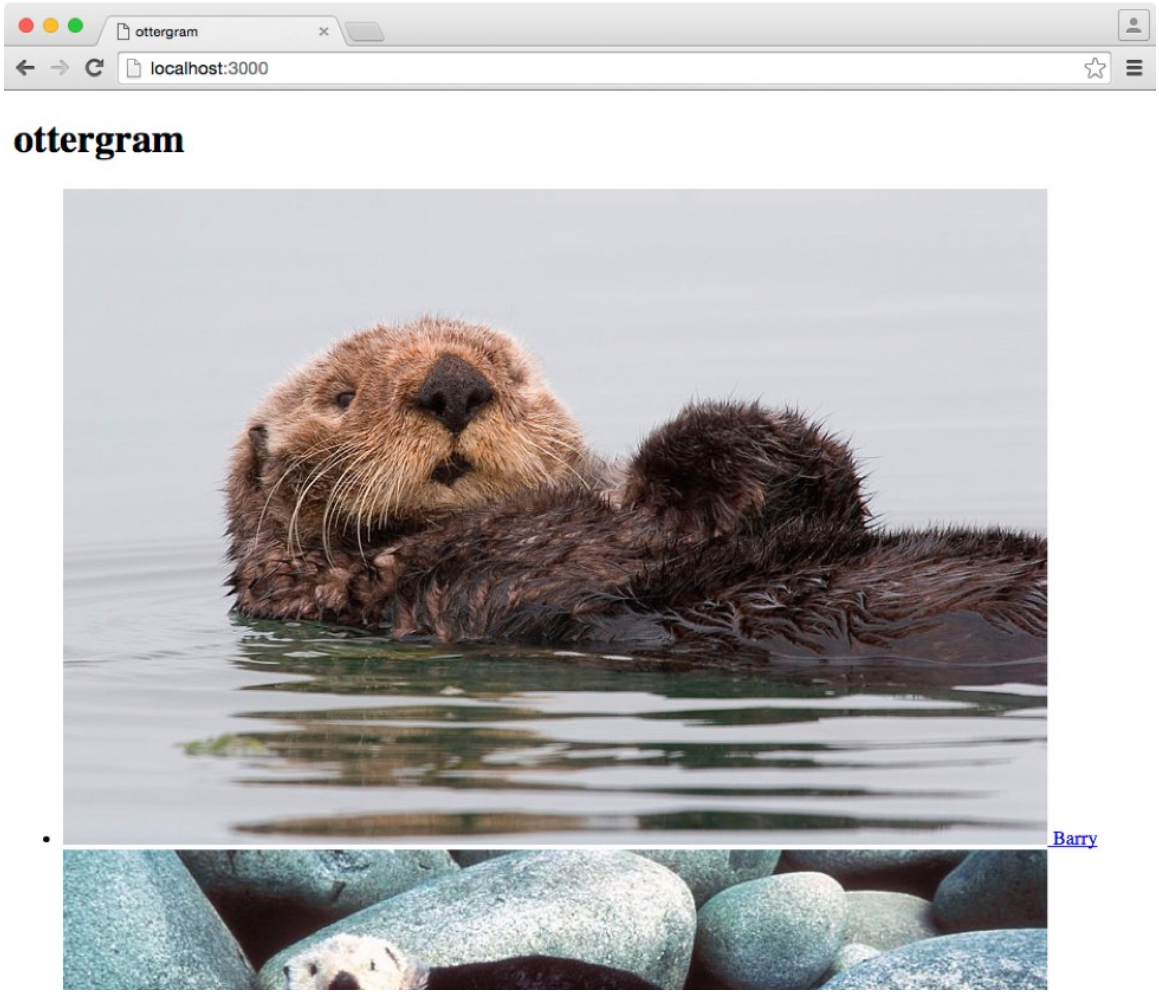
The image below shows the result of entering this command on a Mac.



You should see the same output on Windows:



Once the Ottergram page has loaded in `Chrome`, you should see your page with the "ottergram" heading, "ottergram" as the tab label, and a series of otter photos and names:

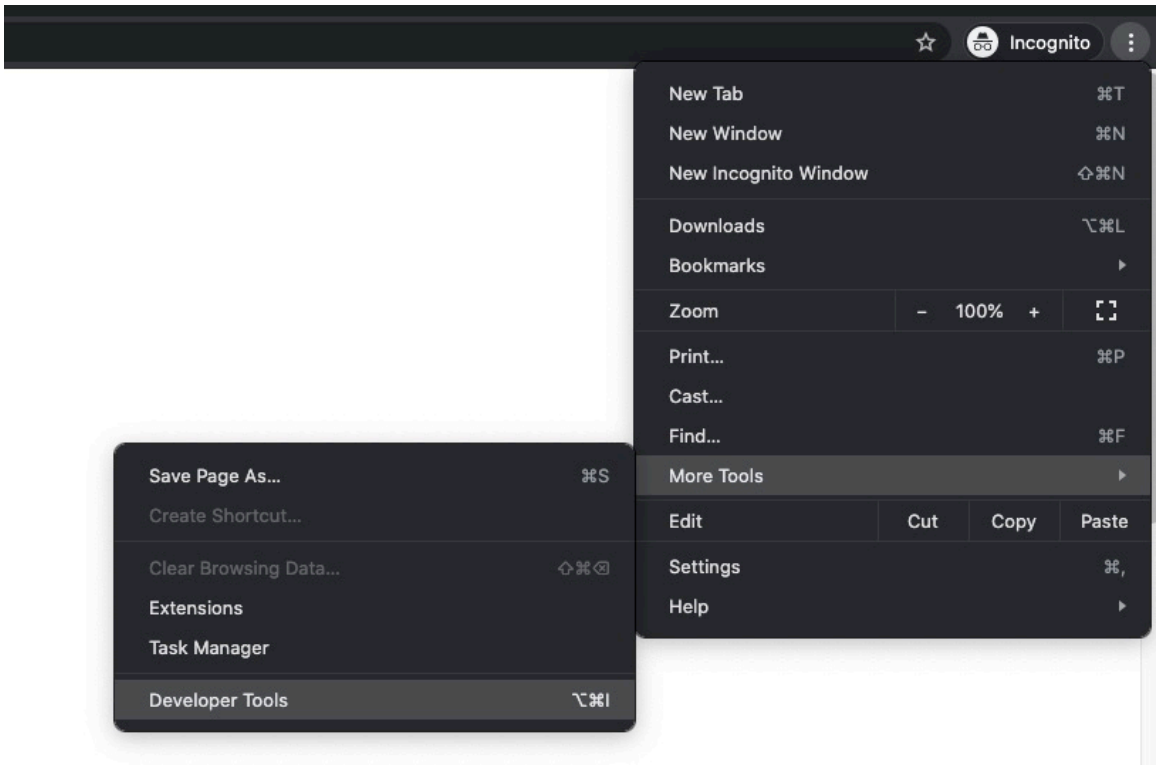


The Chrome Developer Tools

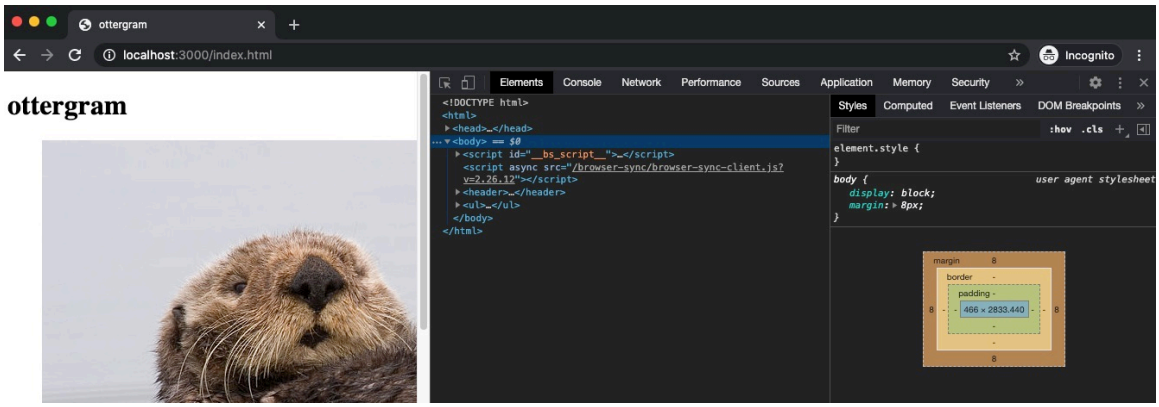
Chrome has built-in Developer Tools (commonly known as "DevTools") that are among the best available for testing styles, layouts, and more on the fly. Using the DevTools is much more efficient than trying things out in code. The DevTools are very powerful and will be your constant companion as you do front-end development.

You will start using the DevTools in the next chapter. For now, open the window and familiarize yourself with its major areas.

To open the DevTools, click the triple dot icon to the right of the address bar in Chrome. Next, click `More Tools` -> `Developer Tools`.



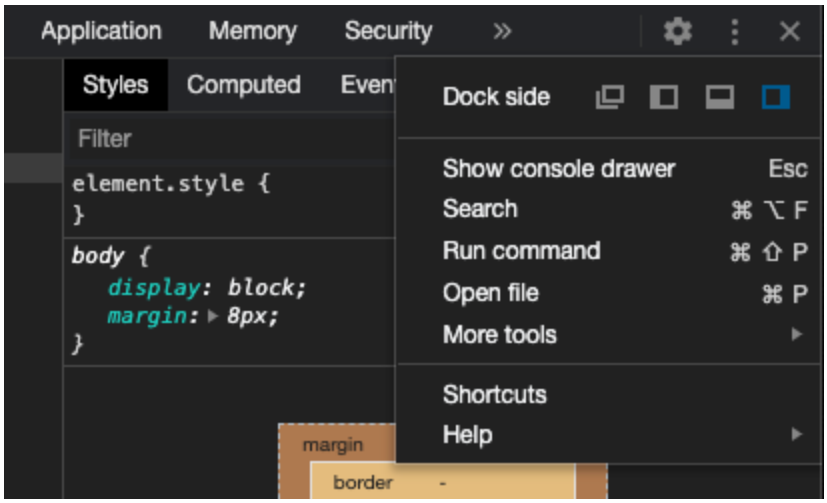
Chrome displays the DevTools to the right by default. Your screen will look something like:



The DevTools show the relationship between the code and the resulting page elements. They let you inspect individual elements' attributes and styles and see immediately how the browser is interpreting your code. Seeing this relationship is critical for both development and debugging.

In the image above, you can see the DevTools next to the web page, displaying the elements panel. The elements panel is divided into two sections. On the left is the *DOM tree view*. This is a representation of the HTML, interpreted as DOM elements. (You will learn much more about the DOM, which stands for "document object model," in upcoming chapters.) On the right-hand side of the elements panel is the styles pane. This shows any visual styles applied to individual elements.

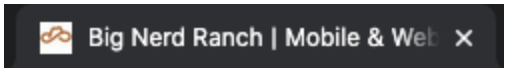
Having the DevTools docked on the right side of the screen while you are working is usually convenient. If you want to change the location of the DevTools, you can click the triple dot button near the upper-right corner. This will show you a menu of options, including buttons for the **Dock side**, which will change the anchor location of the DevTools.



With your otters and markup in place and the DevTools open, you are ready to begin styling your project in the next chapter.

🔎 For the More Curious: The favicon

Have you ever noticed the icons that appear on the left side of each tab in Chrome?



That is the favicon image file.

Some browsers request one by default, so you may see a 404 (Not Found) error about the request. Do not worry about this error if it appears. It will not affect your project. However, you can easily add a favicon image – and that is your first challenge.

Favicons used to be in a special `ico` format, but most browsers now support several formats including `PNG` and `SVG`.

🥈 Silver Challenge: Adding a favicon

You have decided that your otters deserve a favicon. You are going to create a favicon file using one of the otter images.

Using software on your computer or an online photo editor, edit an otter image to create a square PNG file.

Save the resulting `favicon.png` file in the same folder as your `index.html` file. Then, add `<link rel="shortcut icon" type="image/png" href="/favicon.png"/>` after the `link` tag for your `styles.css` file. Finally, reload your browser. Your browser tab will look something like:

Note: You can keep your solution to this challenge in your code, and it will not interfere with future code changes for Ottergram.

Styles

In this chapter, you will design a static version of Ottergram. In the chapters that follow, you will make Ottergram interactive.

When you reach the end of this chapter, your website will look like this:

This chapter introduces a number of concepts and examples. Do not worry if you do not feel that you have mastered all of them when you get to the end. You will be encountering them again and again as you progress through this book, and your work in this chapter will provide a solid foundation on which you will build deeper understanding.

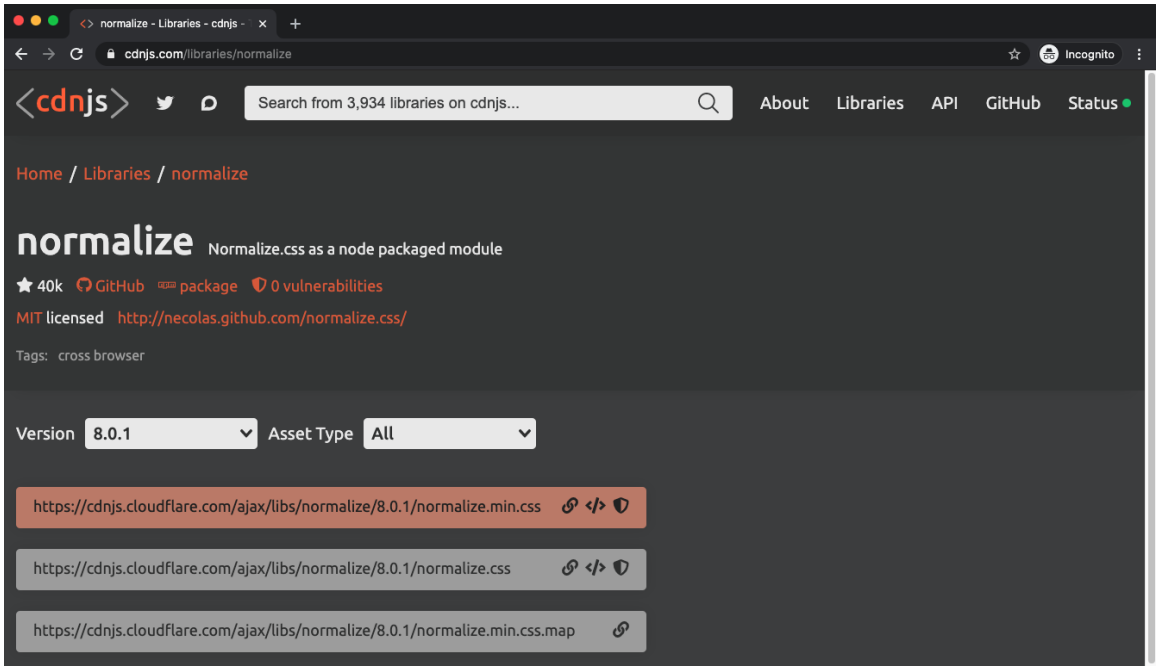
Of course, we can only introduce you to a tiny fraction of all the styles that are available in CSS. You will want to consult MDN for information about the full set of properties and their values.

Creating a Styling Baseline

You are going to begin by adding the `normalize.css` file to your project. `normalize.css` helps the CSS you write display consistently across browsers. All browsers come with a set of default styles, but the defaults are different from browser to browser. `normalize.css` gives you a good starting point for developing your own custom CSS for a website or web app.

`normalize.css` is freely available online. You do not need to download it. To add it to Ottergram, you only need to link to it in `index.html`.

To ensure that you are using the most current version of `normalize.css`, you are going to get its address from a content sharing site. Go to cdnjs.com/libraries/normalize and find the version of the file ending with `.min.css`. (This version is a smaller download than the others, with the extra whitespace stripped out.) Click the `Copy` button to copy its address.



The current version at the time of this writing is 8.0.1, but the version you use may be more recent.

Open your Ottergram folder in `Visual Studio Code`. Then, open `index.html`. Add a new `<link>` tag and paste in the address. (In the code below, the `<link>` has been broken into two lines to fit on the page. You can leave yours on a single line.)

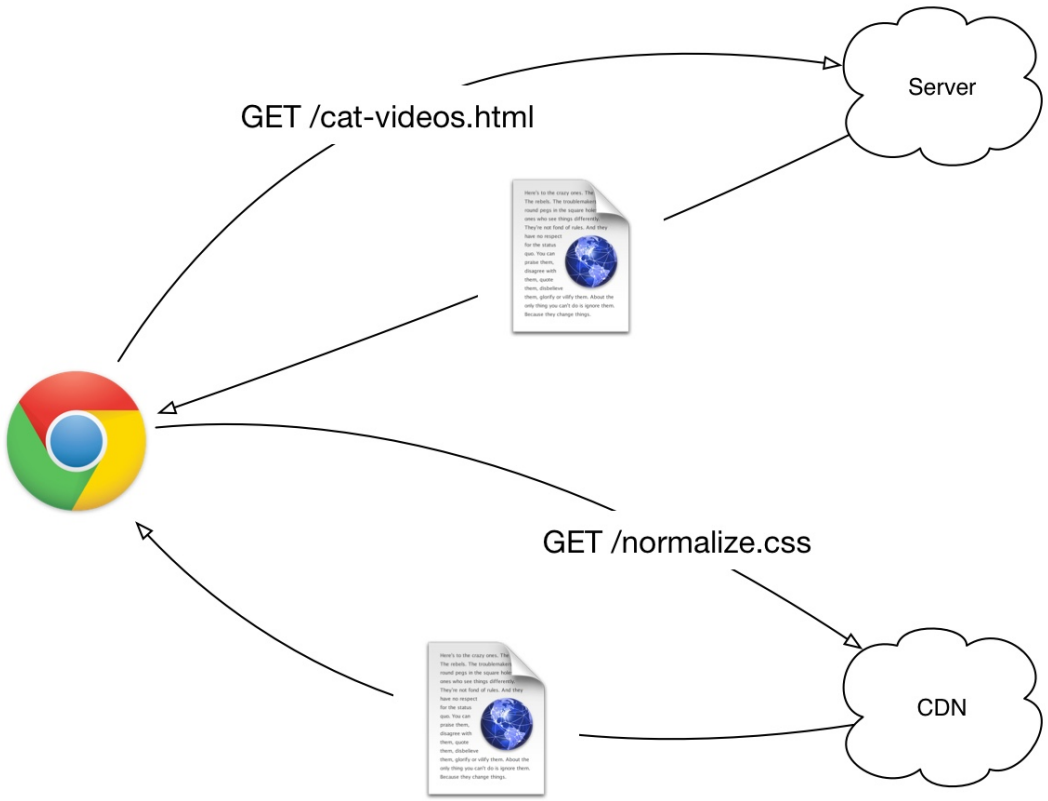
```
<!doctype html>
<html>
  <head>
```

```
      <meta charset="utf-8">
      <title>ottergram</title>
+     <link rel="stylesheet"
+       href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css">
      <link rel="stylesheet" href="stylesheets/styles.css">
    </head>
  ...
```

Make sure that you add the `<link>` tag for `normalize.css` *before* the `<link>` tag for `styles.css`. The browser needs to read the styles found in `normalize.css` before it reads yours.

And, just like that, your project can take advantage of this useful tool. No other setup required.

You may be wondering why you are linking to an address on a completely different server. In fact, it is not unusual for an HTML file to specify resources located on different servers.



In this case, `normalize.css` is hosted on cdnjs.com, a public server that is part of a *content delivery network*, or CDN. CDNs have servers all around the world, each with copies of the same files. When users request a file, they receive it from a server nearby, cutting down on the load time for that file. cdnjs.com hosts many versions of popular front-end libraries and frameworks.

Preparing the HTML for Styling

In the last chapter, you created a stylesheet called `styles.css`, and in this chapter you will add a number of CSS *styling rules* to it. But before you get started adding styles, you need to set up your HTML with targets for your styling rules to refer to.

You are going to add `class` attributes identifying the `span` elements with the otters' names as `thumbnail-title`. `class` attributes are a way to identify a group of HTML elements, usually for styling. Your "thumbnail title" `class` will allow you to easily style all the names at once.

In `index.html`, add the class name `thumbnail-title` as an attribute of the `span`s inside the `li` elements, as shown:

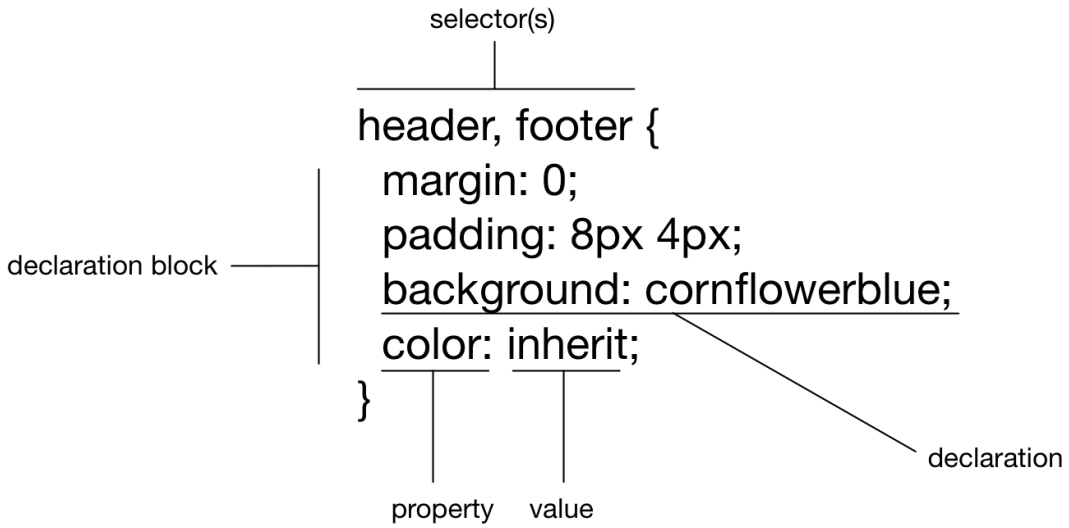
```
  ...
      <ul>
        <li>
          <a href="#">
            <img src="img/otter1.jpg" alt="Barry the Otter" />
-           <span>Barry</span>
+           <span class="thumbnail-title">Barry</span>
          </a>
        </li>
        <li>
          <a href="#">
            <img src="img/otter2.jpg" alt="Robin the Otter" />
-           <span>Robin</span>
+           <span class="thumbnail-title">Robin</span>
          </a>
        </li>
        <li>
          <a href="#">
            <img src="img/otter1.jpg" alt="Maurice the Otter" />
-           <span>Maurice</span>
+           <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
        <li>
          <a href="#">
            <img src="img/otter4.jpg" alt="Lesley the Otter" />
-           <span>Lesley</span>
+           <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
        <li>
          <a href="#">
            <img src="img/otter5.jpg" alt="Barbara the Otter" />
-           <span>Barbara</span>
```

```
+           <span class="thumbnail-title">Barbara</span>
          </a>
      </ul>
  ...
```

In a moment, you will use this class name to style all the image titles.

Anatomy of a Style

When you create individual styles, you do so by writing styling rules, which consist of two main parts: *selectors* and *declarations*.



The first part of a styling rule is one or more selectors. Selectors describe the elements that the style should be applied to, like `h1`, `span`, or `img`. But selectors are not limited to tag names. You can write selectors that apply to a more targeted set of elements by increasing the selector's *specificity*.

For example, you can write selectors based on attributes – such as the `thumbnail-title` class attribute you just added to the `<span>` tags. Selectors based on attributes are more specific than selectors based on element names.

In addition to making sure styles are only applied to a limited set of elements (e.g., elements with the class name `thumbnail-title` versus all `<span>` elements), specificity also determines the selector's relative priority. If a stylesheet contains multiple styles that could apply to the same element, the styles with a selector of higher specificity will be used instead styles whose selector has a lower specificity. You can read more about specificity in a For the More Curious section at the end of this chapter.

Throughout this chapter, you will be introduced to a number of different kinds of selectors that vary in their specificity. Though there are often many ways to target the same element for styling,

understanding specificity is key to choosing the best selector to use so that your styles are maintainable.

The second part of a styling rule is the declaration block, wrapped in curly braces, which defines the styles to be applied. The individual declarations within the block each include a property name and a value for that property.

In your first styling rule, you will use the `class` attribute you just added as a selector to apply styles around the otters' names.

Your First Styling Rule

To use a `class` as a selector in a styling rule, you prefix the class name with a dot (period), as in `.thumbnail-title`. The first styles you are going to add will set the background and foreground colors for the `.thumbnail-title` class.

Open `styles.css` and add your styling rule:

```
+ .thumbnail-title {
+   background: rgb(96, 125, 139);
+   color: rgb(202, 238, 255);
+ }
```

You will learn more about color later in this chapter. For now, just take a look at your changes. Save `styles.css` and make sure `browser-sync` is running. If you need to restart it, the command is:

```sh
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

This will open your web page in `Chrome`.

You can see that you have set the background for the thumbnail titles to a deep gray-blue and the font color to a lighter blue. Nice.

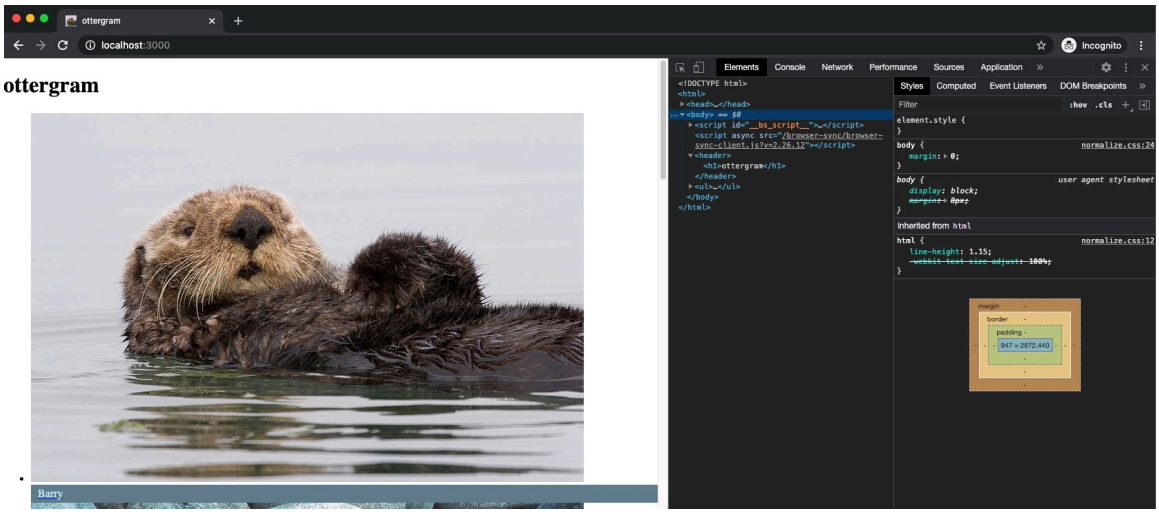
Continue styling the thumbnail titles: Return to `styles.css` and add to your existing styling rule for the `.thumbnail-title` class, as shown:

```
  .thumbnail-title {
+   display: block;
+   margin: 0;
+   padding: 4px 10px;
    background: rgb(96, 125, 139);
    color: rgb(202, 238, 255);
  }
```

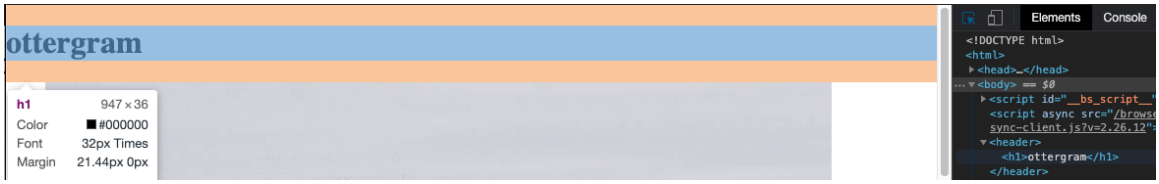
The three declarations you have added all affect an element's *box*. For every HTML tag that has a visual representation, the browser draws a rectangle to the page. The browser uses a scheme called the *standard box model* (or just "box model") to determine the dimensions of that rectangle.

The box model

To understand the box model, you are going to look at its representation in the DevTools. Save `styles.css`, switch to `Chrome`, and make sure the DevTools are open.

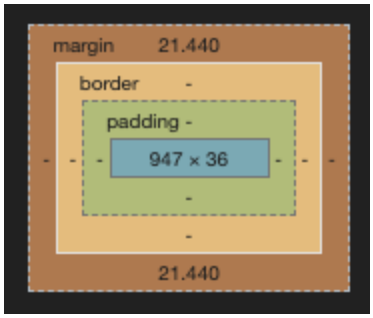


Click the  button in the upper-left of the elements panel. This is the *Inspect Element* button. Now move your cursor over the word "ottergram" on the web page. As you hover over the word, the DevTools surrounds the heading with a blue- and peach-colored rectangle.



Click the word "ottergram" on the web page. Although you no longer see the multicolored overlay, the element is now selected and the DOM tree view in the elements panel will expand to show and highlight the corresponding `<h1>` tag.

The rectangular diagram in the lower-right of the elements panel represents the box model for the `h1` element. You can see that the regions of the diagram have some of the same colors as the rectangle you saw overlaying the heading when you inspected it.

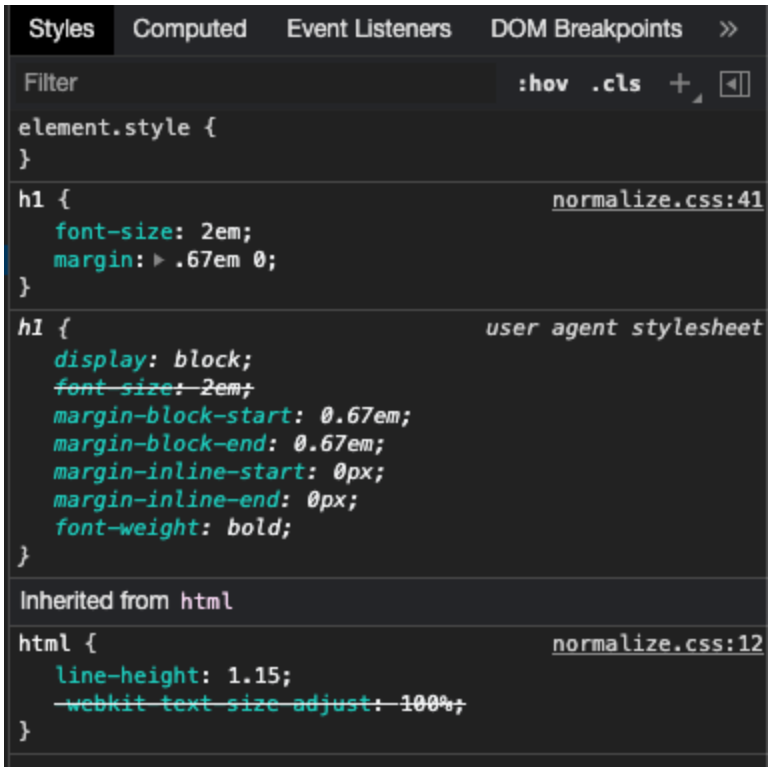


The box model incorporates four aspects of the rectangle drawn for an element (which the DevTools renders in four different colors in the diagram).

Type	Description
content (shown in blue)	the visual content – here, the text
padding (shown in green)	transparent space around the content
border (shown in yellow)	a border, which can be made visible, around the content and padding
margin (shown in peach)	transparent space around the border

The numbers are *pixel* values; a pixel is a unit corresponding to the smallest rectangular area of a computer screen that can display a single color. In the case of the `h1` element, the content area has been allocated an area of 947 pixels by 36 pixels (your values may be different, depending on the size of your browser window). There is no padding or border, and there is a margin of 21.44 pixels above and below the element.

Where did that margin value come from? Look just above the box model.



Chrome tells you that no style was applied directly to the element. `normalize.css` set the `font-size` and `margin`. After that, the *user agent stylesheet*, applied some styles. The browser provides these styles in case the user does not specify a style. Styles you specify override the defaults.

Because you have not specified some values for the `h1` element's box, the default styles have been applied. You'll notice `normalize.css` specified the font size and overrode the browser's default. Chrome shows overridden styles with a strikethrough, so you know it was there and was overridden by a style with higher priority.

Now you are ready to understand the styling declarations you added:

```css
.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;
  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

The `display: block` declaration changes the box for all elements of the class `.thumbnail-title` so that they occupy the entire width allowed by their containing element. (Notice that the background color for the titles now covers a wider area.) Other `display` values, such as the `display: inline` property you will see later, make an element's width fit to its content.

You also set the margin for the thumbnail titles to 0 and the padding to two different values: 4px and 10px (*px* is the abbreviation for "pixels"). This sets the padding to specific pixel values, overriding the default size set by the user agent stylesheet.

Padding, margin, and certain other styles can be written as *shorthand properties*, in which one value is applied to multiple properties. You are taking advantage of this here: When two values are provided for the padding, the first is applied to both vertical values (top and bottom), and the second is applied to both horizontal values (left and right). It is also possible to provide a single value to be applied to all four sides or to specify a separate value for each side.

To sum up, your new declarations say that the box for all elements of the `.thumbnail-title` class will fill the width of its container with no margin and with padding that is 4 pixels at the top and bottom and 10 pixels at the left and right sides.

🔎 Margin Collapsing

In certain cases, the browser will automatically combine the margins of two blocks. For example, if a webpage has an `h1` with `20px` of bottom margin followed by an `h2` with `15px` of top margin. The margin will be collapsed, and they will be displayed with `20px` between them.

You can read more about margin collapsing on MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing

Style Inheritance

Next, you are going to add styles to change the size and appearance of the text.

Add a new styling rule in `styles.css` to set the font size for the `body` element. To do this, you will use a different type of selector – an *element selector* – by simply using the element's name.

```
+ body {
+   font-size: 10px;
+ }
  .thumbnail-title {
    display: block;
    margin: 0;
    padding: 4px 10px;
    background: rgb(96, 125, 139);
    color: rgb(202, 238, 255);
  }
```

This styling rule sets the `body` element's `font-size` to `10px`.

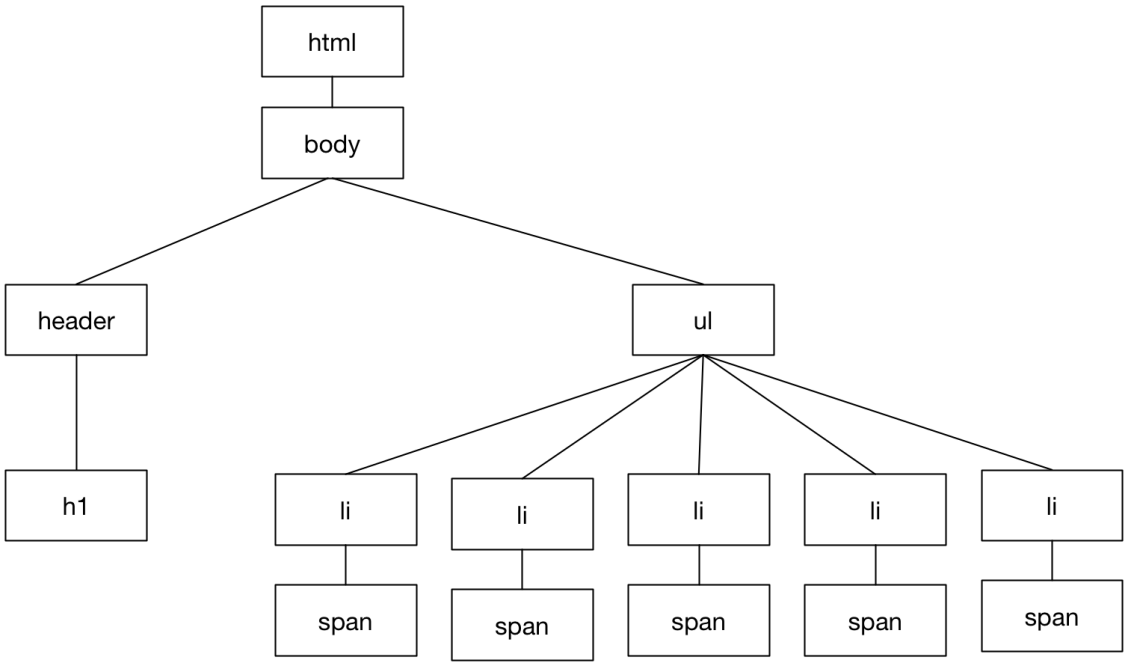
You will rarely use element selectors in your stylesheets, because you will not often want to apply the exact same styles to every occurrence of a particular tag. Also, element selectors limit your ability to reuse styles; using them means that you may end up retyping the same declarations throughout your stylesheets. This is not great for maintenance if you need to alter those styles.

But, in this case, targeting the `body` element is exactly the right amount of specificity. There can be only one `<body>` element, and you will not be reusing its styles.

Save `styles.css` and check out your web page in `Chrome`.

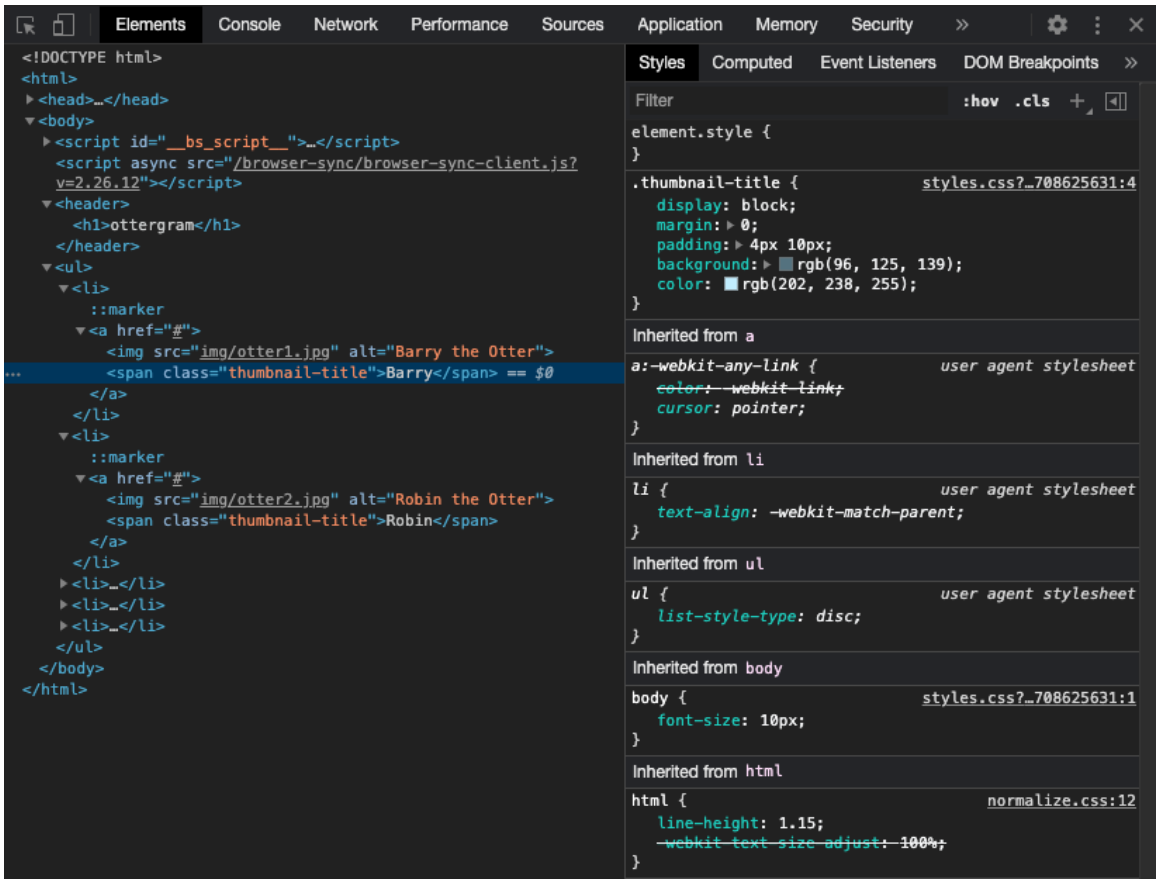
Your headline and thumbnail titles have gotten smaller. You may – or may not – have expected this. While the headline is directly within the `body` element where you declared the `font-size` property, the thumbnail titles are not. They are nested several levels deep. However, many styles, including font size, are applied to the elements specified by the styling rule as well as the *descendants* of those elements.

The structure of your document can be described using a tree diagram. Representing your elements as a tree is a good way to visualize the DOM.



An element contained within another element is said to be its descendent. In this case, your `span`s are all descendants of the `body` (as well as the `ul` and their respective `li`), so they inherit the `body`'s `font-size` style.

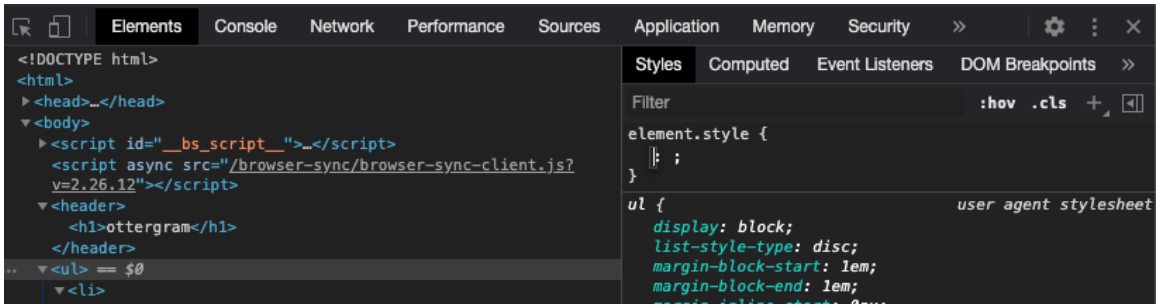
In the DevTools' DOM tree view, locate and select one of the `span` elements. In the styles pane, notice the boxes labeled **Inherited from a**, **Inherited from li**, and **Inherited from ul**. These three areas, as indicated, show styles inherited at each level from the user agent stylesheet. Under **Inherited from body**, you can see that the `font-size` property has been inherited from the style set for the `body` element in `styles.css`.



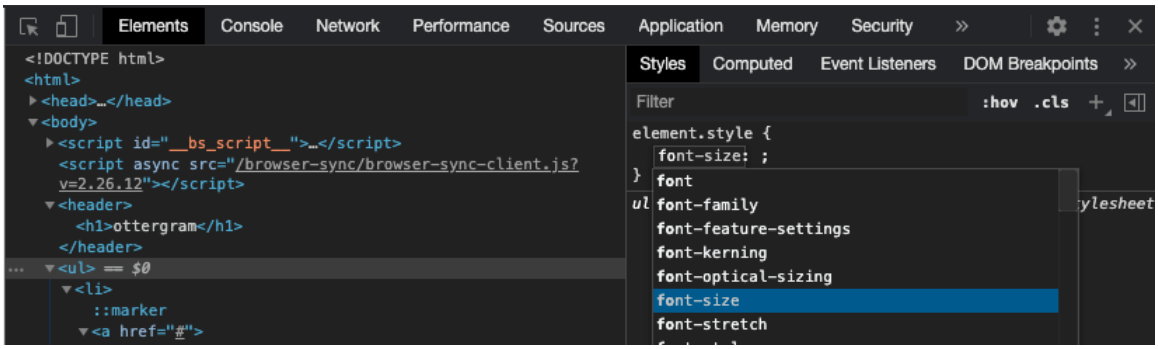
What if a different font size were set at another level, such as the `ul`? Styles from the closer ancestor take priority, so a font size set in `styles.css` for the `ul` would override one set for the `body` and a font size set for the `span` element itself would override them both.

To see this, click on the `ul` element in the DOM tree view. This will allow you to try out styles on the fly. The styles you add here will be immediately reflected in the web page view, but will not be added to your actual project files.

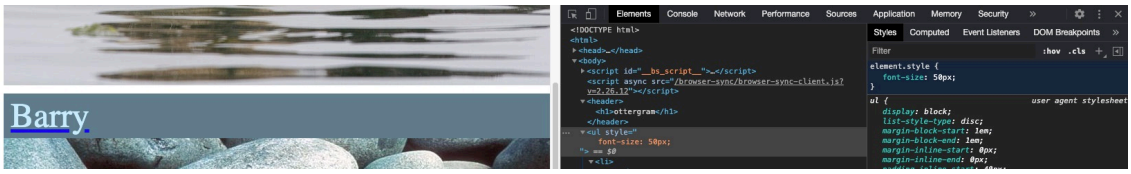
At the top of the styles pane in the elements panel, you will see a section labeled `element.style`. Click anywhere in between the curly braces of the `element.style`, and the DevTools will give you a prompt.



Start typing `font-size`, and the DevTools will suggest possible completions.



Choose `font-size`. Then, press the Tab key. Enter a large value, such as `50px`, and press Return. You may need to scroll the page, but you will see that the `ul`'s `font-size` has overridden the `body`'s.

- 

Not all style properties are inherited – `border`, for example, is not. To find out whether a property is inherited, refer to the property's MDN reference page.

Back in `styles.css`, update your declaration block for the `.thumbnail-title` class to override the `body`'s `font-size` and use a larger font.

```
  body {
    font-size: 10px;
  }
  .thumbnail-title {
    display: block;
    margin: 0;
    padding: 4px 10px;
    background: rgb(96, 125, 139);
    color: rgb(202, 238, 255);
+   font-size: 18px;
  }
```

For elements of the class `.thumbnail-title`, you changed the font size to 18 pixels.

Save `styles.css` and admire your thumbnail titles in `Chrome`.

They look good, but the user agent stylesheet is adding underlines to the `.thumbnail-title` elements. This is because you wrapped them (along with the `.thumbnail-image` elements) with an anchor tag, making them inherit the underline style.

You do not need the underlines, so you are going to remove them by changing the `text-decoration` property for the anchor tags in a new styling rule in `styles.css`. What selector should you use for this rule?

If you are confident that you want to remove the underlines from the thumbnail titles *as well as any other anchor elements in Ottergram*, you can simply use an element selector:

```css
a {
  /* style declaration */
}
```

(The text between the `/* */` indicators is a CSS *comment*. Code comments are ignored by the browser; they allow the developer to make notes in the code for future reference.)

If you think you might use anchors for another purpose (and will want to style them differently), you can pair the element selector with an *attribute selector*, like this:

```css
a[href]{
  /* style declaration */
}
```

This selector would match any anchor element with an `href` attribute. Of course, anchor elements generally do have `href` attributes, so that might not be targeted enough to match only the thumbnail images and titles. To make an attribute selector more precise, you can also specify the value of the attribute, like this:

```css
a[href="#"]{
  /* style declaration */
}
```

This selector would match only those anchor elements whose `href` attribute has a value of `#`.

By the way, you can also use attribute selectors, with or without values, on their own, such as:

```css
[href]{
  /* style declaration */
}
```

As it happens, Ottergram is a fairly simple project and you will not, in fact, be using anchor tags for anything other than the thumbnails and their titles. It is therefore safe to use an element selector, and you should do so because it is the most straightforward solution with the right amount of specificity.

Add the new style declaration to `styles.css`:

```
  body {
    font-size: 10px;
  }
+ a {
+   text-decoration: none;
+ }
  .thumbnail-title {
    ...
  }
```

Save your file and check your browser. The underlines are gone and your thumbnail titles are nicely styled.

Note that you should not remove the underlines from links that are in normal text – text that is not an obvious heading, title, or caption. The underlining of linked text is an important visual indicator that users have come to expect. You did it here because the thumbnails do not require the same visual cues. Users will reasonably expect them to be clickable, and the pointer cursor they see when they mouseover them will confirm that expectation. (The pointer cursor is applied by the `a` tag wrapping the `img`.)

In the rest of the chapter, you will use class selectors to style the thumbnail images, the unordered list of images, the list items (which include the thumbnail images and their titles), and, finally, the header. Go ahead and add class names to the `h1`, `ul`, `li`, and `img` elements in `index.html` so they are ready as you need them.

```
  ...
    </head>
    <body>
      <header>
-       <h1>ottergram</h1>
+       <h1 class="logo-text">ottergram</h1>
      </header>
-     <ul>
+     <ul class="thumbnail-list">
-       <li>
+       <li class="thumbnail-item">
          <a href="#">
-           <img src="img/otter1.jpg" alt="Barry the Otter" />
+           <img class="thumbnail-image" src="img/otter1.jpg" alt="Barry the Otter" />
            <span class="thumbnail-title">Barry</span>
          </a>
        </li>
-       <li>
+       <li class="thumbnail-item">
          <a href="#">
-           <img src="img/otter2.jpg" alt="Robin the Otter" />
```

```
+           <img class="thumbnail-image" src="img/otter2.jpg" alt="Robin the Otter" />
            <span class="thumbnail-title">Robin</span>
          </a>
        </li>
-       <li>
+       <li class="thumbnail-item">
          <a href="#">
-           <img src="img/otter3.jpg" alt="Maurice the Otter" />
+           <img class="thumbnail-image" src="img/otter3.jpg" alt="Maurice the Otter" />
            <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
-       <li>
+       <li class="thumbnail-item">
          <a href="#">
-           <img src="img/otter4.jpg" alt="Lesley the Otter" />
+           <img class="thumbnail-image" src="img/otter4.jpg" alt="Lesley the Otter" />
            <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
-       <li>
+       <li class="thumbnail-item">
          <a href="#">
-           <img src="img/otter5.jpg" alt="Barbara the Otter" />
+           <img class="thumbnail-image" src="img/otter5.jpg" alt="Barbara the Otter" />
            <span class="thumbnail-title">Barbara</span>
          </a>
        </li>
      </ul>
  ...
```

By adding class names to these elements, you have given yourself targets for the styles you will be adding.

We favor class selectors over other kinds of selectors, and you should, too. You can write very descriptive class names that make your code easy to develop and maintain. Also, you can add multiple class names to an element, making them a flexible and powerful tool for styling.

Be sure to save `index.html` before moving on.

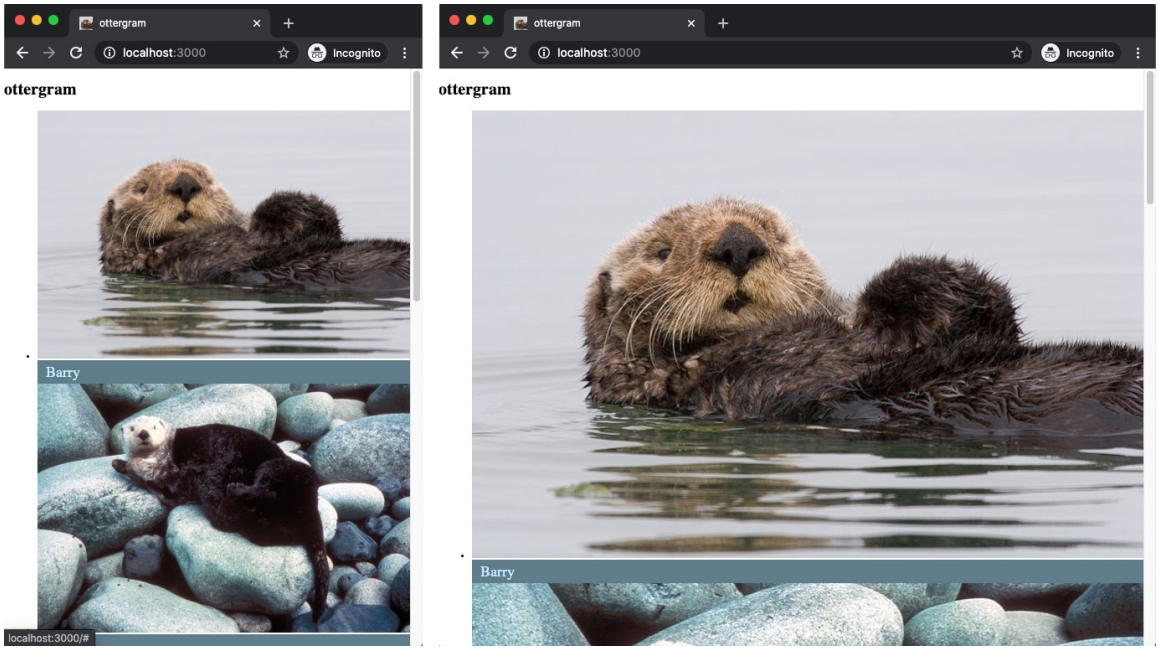
Making Images Fit the Window

Following the atomic styling pattern, the images are next in line for styling. They are so large that they are cut off unless the browser window is also large. Add a styling rule for `.thumbnail-image` in `styles.css` to make the thumbnails fit in the window:

```
...
a {
  text-decoration: none;
```

```
  }
+ .thumbnail-image {
+   width: 100%;
+ }
  .thumbnail-title {
    ...
  }
```

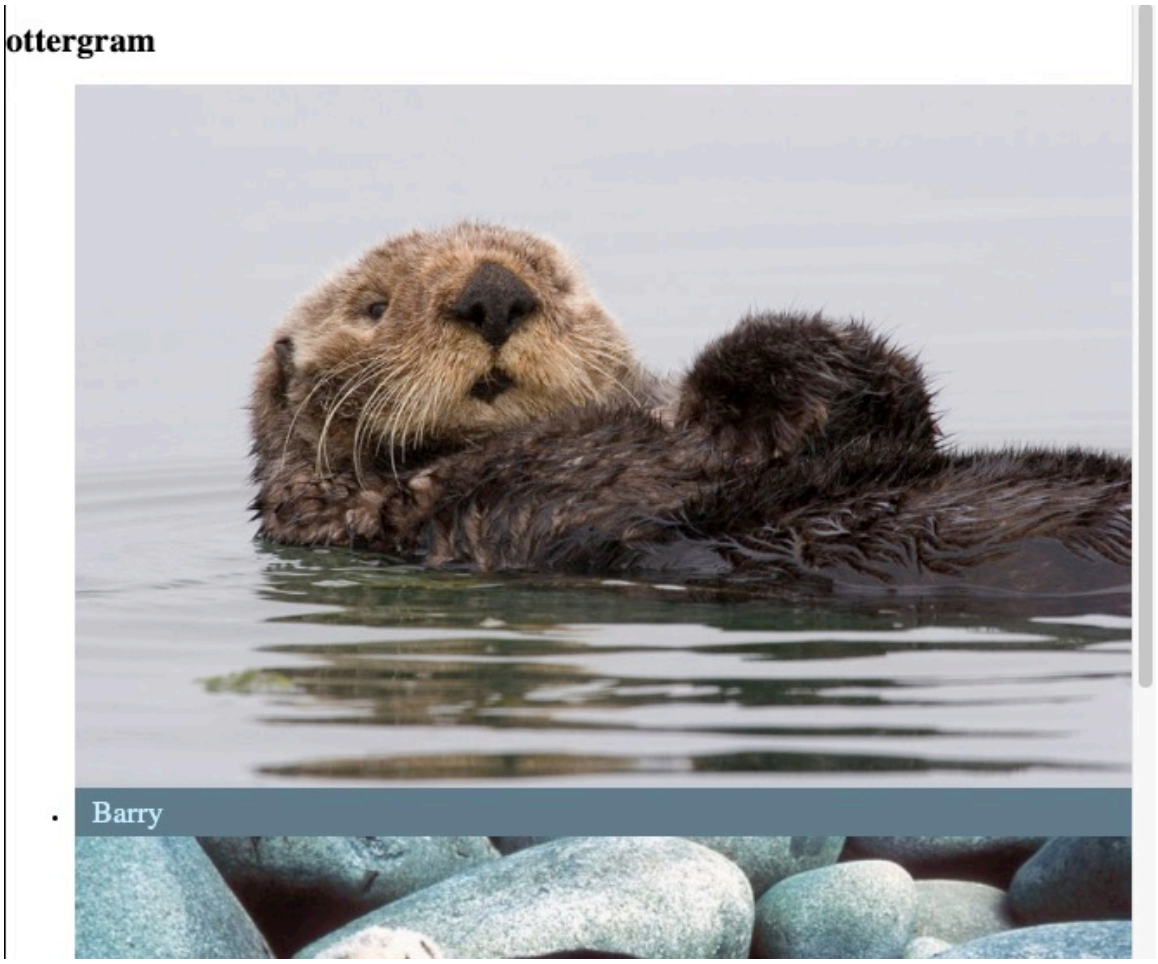
You set the `width` to `100%`, which constrains it to the width of its container. This means that as you widen the browser window, the images get proportionally larger. Check it out: Save `styles.css`, switch to your browser, and make your browser window larger and smaller. The images grow and shrink along with the browser window, always keeping their proportions. The image below shows Ottergram in one narrow and one wider browser window.



If you look closely, the spacing around each `.thumbnail-title` is off, so that it appears that the titles go with the images below them. Fix that in `styles.css` by setting the `.thumbnail-image`'s `display` property to `block`.

```
  ...
  .thumbnail-image {
+   display: block;
    width: 100%;
  }
  ...
```

Now the space between the image and its title is gone.



Why does this work? Images are `display: inline` by default. They are subject to similar rendering rules as text. When text is rendered, the letters are drawn along a common baseline. Some characters, such as **p**, **q**, and **y**, have a *descender* - the tail that drops below this baseline. To accommodate them, there is some whitespace included below the baseline.

Setting the `display` property to `block` removes the whitespace because there is no need to accommodate any text (or any other `display: inline` elements that might be rendered alongside the image).

Color

It is time to explore color a little more deeply. Add the following color styles for the `body` element and the `.thumbnail-item` class in `styles.css`.

```
  body {
    font-size: 10px;
+   background: rgb(149, 194, 215);
  }
```

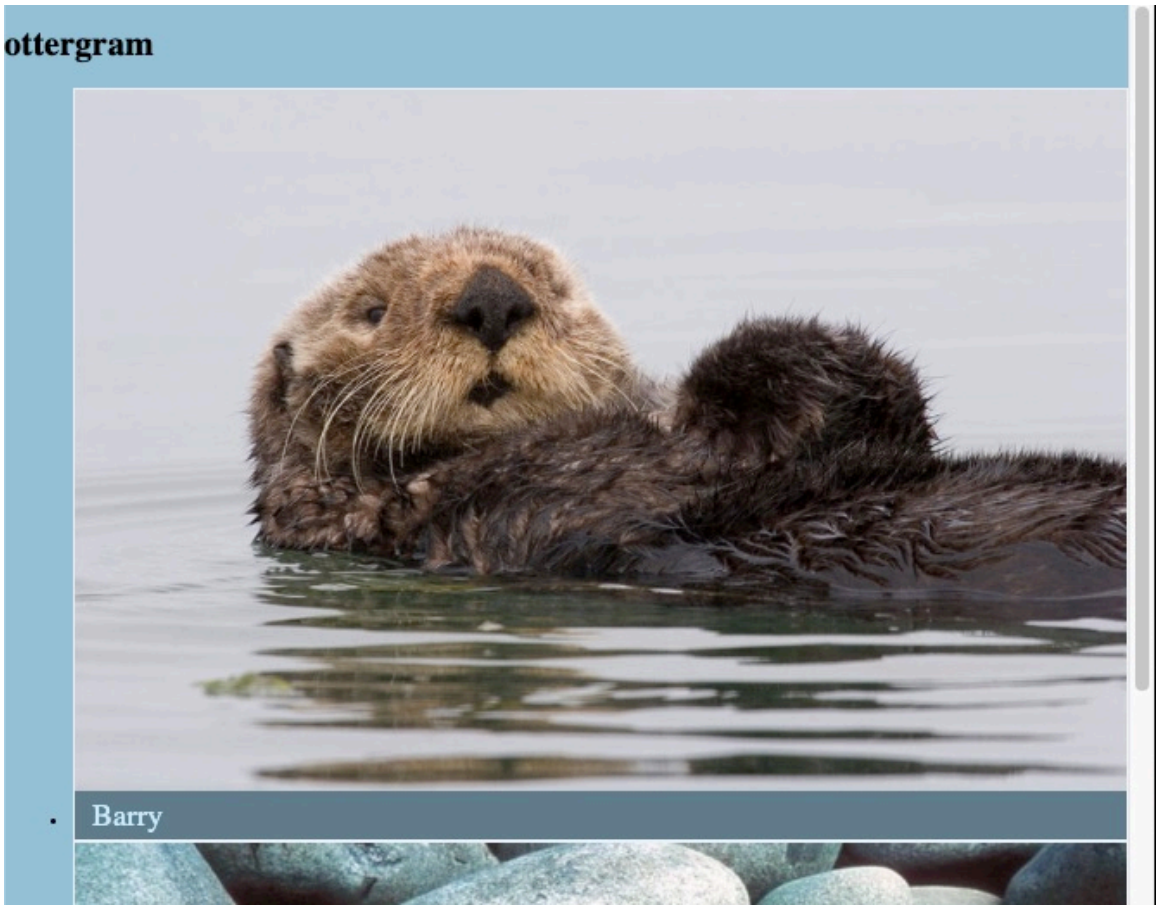
```
  a {
    text-decoration: none;
  }
+ .thumbnail-item {
+   border: 1px solid rgb(100%, 100%, 100%);
+   border: 1px solid rgba(100%, 100%, 100%, 0.8);
+ }
  ...
```

You have declared values for the `.thumbnail-item`'s border twice. Why? Notice that the two declarations use slightly different color functions: `rgb` and `rgba`. The `rgba` color function accepts a fourth argument, which is the opacity. However, some browsers do not support `rgba`, so providing both declarations is a technique that provides a *fallback* value.

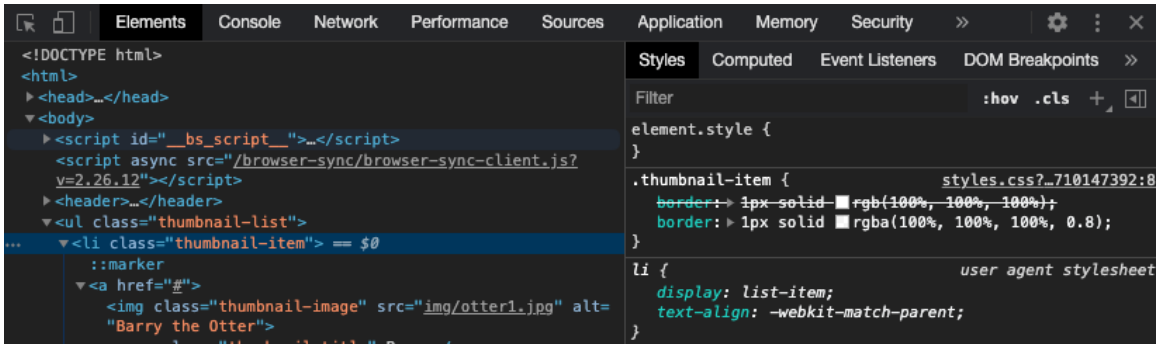
All browsers will see the first declaration (`rgb`) and register its value for the `border` property. When browsers that do not support `rgba` see the second declaration, they will not understand it and will simply ignore it, using the value from the first declaration. Browsers that *do* support `rgba` will use the value in the second declaration and discard the value from the first declaration.

(Wondering why the `body`'s background color is defined with integers and the `.thumbnail-item`'s border color is defined with percentages? We will come back to that in just a moment.)

Save `styles.css` and switch to your browser.

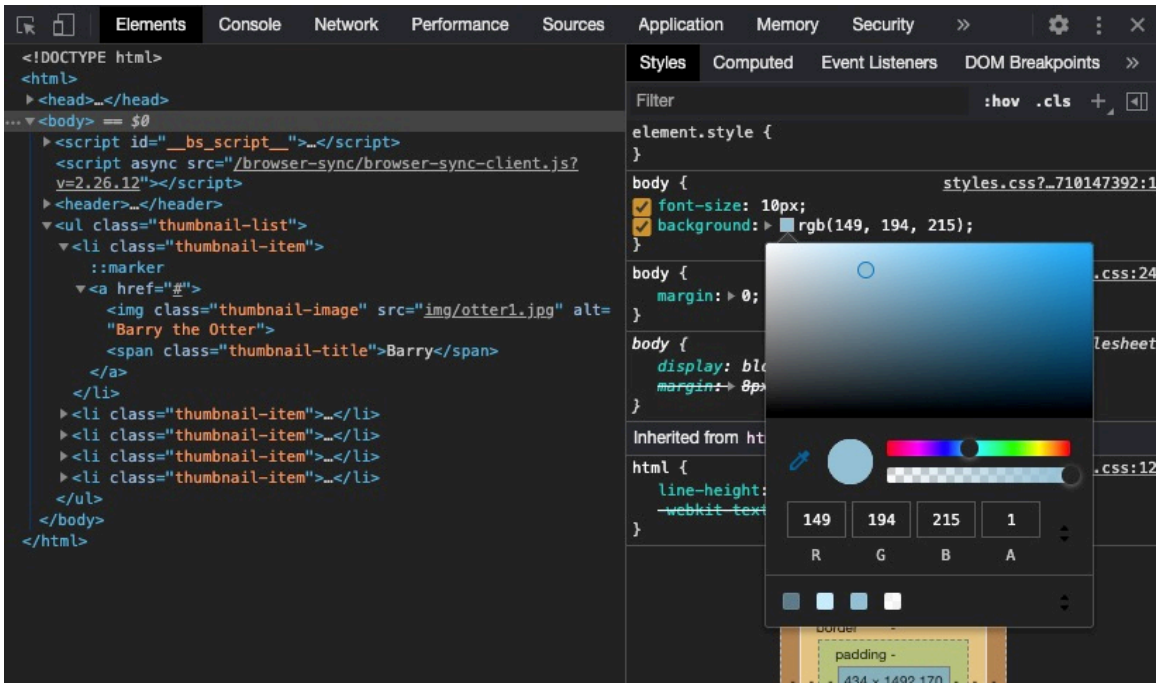


In the DevTools, you can see that `Chrome` supports `rgba`. It denotes that the `rgb` color is not used by striking through the style.



Now, still in the DevTools, select the `body`. In the styles pane, notice the declaration for the background color that you just added. To the left of the RGB value is a small square showing you what the color will look like.

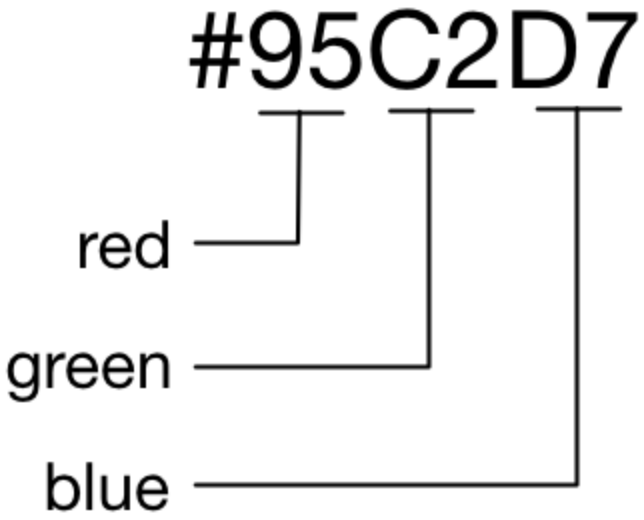
Click that square, and a color picker opens. The color picker lets you choose a color and will give you the CSS color value in a variety of different formats.



To see the background color in different color formats, click the up and down arrows to the right of the RGBA values. You can cycle through HSLA, HEX, and RGBA formats.

The HSLA format (which stands for "hue saturation lightness alpha") is used less frequently than the others, partly because some of the most popular design tools do not provide HSLA values that are accurate for CSS. If you are curious about HSLA, visit the HSLA Explorer at css-tricks.com/examples/HSLaExplorer .

Take a look at the HEX value for the background color: `#95C2D7`. HEX, or hexadecimal, is the oldest color specification format. Each digit represents a value from 0 to 15. (If you are not familiar with hexadecimal numbers, this is done by including the characters A through F as digits.) Each pair of digits, then, can represent a value from 0 to 255. From left to right, the pairs of digits correspond to the intensity of red, green, and blue in the color being specified.



Many find HEX colors unintuitive. A modern alternative is to use RGB (red, green, and blue) values. In this model, each color is also assigned a value from 0 to 255, but the values are represented in more familiar decimal numbers and separated by color. As mentioned earlier, for more capable browsers a fourth value can specify the opacity or transparency of the specified color, from 0.0 (fully transparent) to 1.0 (fully opaque). The opacity is officially known as the *alpha* value – hence the A in RGBA. The RGBA value of the body's background color is `(149, 194, 215, 1)`.

As an alternative to declaring integer values for red, green, and blue, you can also use percentages, as you did for the `.thumbnail-item` borders. There is no functional difference between the two options. Just do not mix percentages and integers in the same declaration.

By the way, for help selecting pleasing color palettes, Adobe provides a free online tool at color.adobe.com .

Adjusting the Space Between Items

Ottergram now has some nice colors reminiscent of otters' ocean home. But adding the colors has revealed some unwanted whitespace inside the border of the `.thumbnail-item` elements. Also, those pesky bullets are drawing attention away from the glory of the otters.

To get rid of the bullets, set the `.thumbnail-list`'s `list-style` property to `none` in `styles.css`:

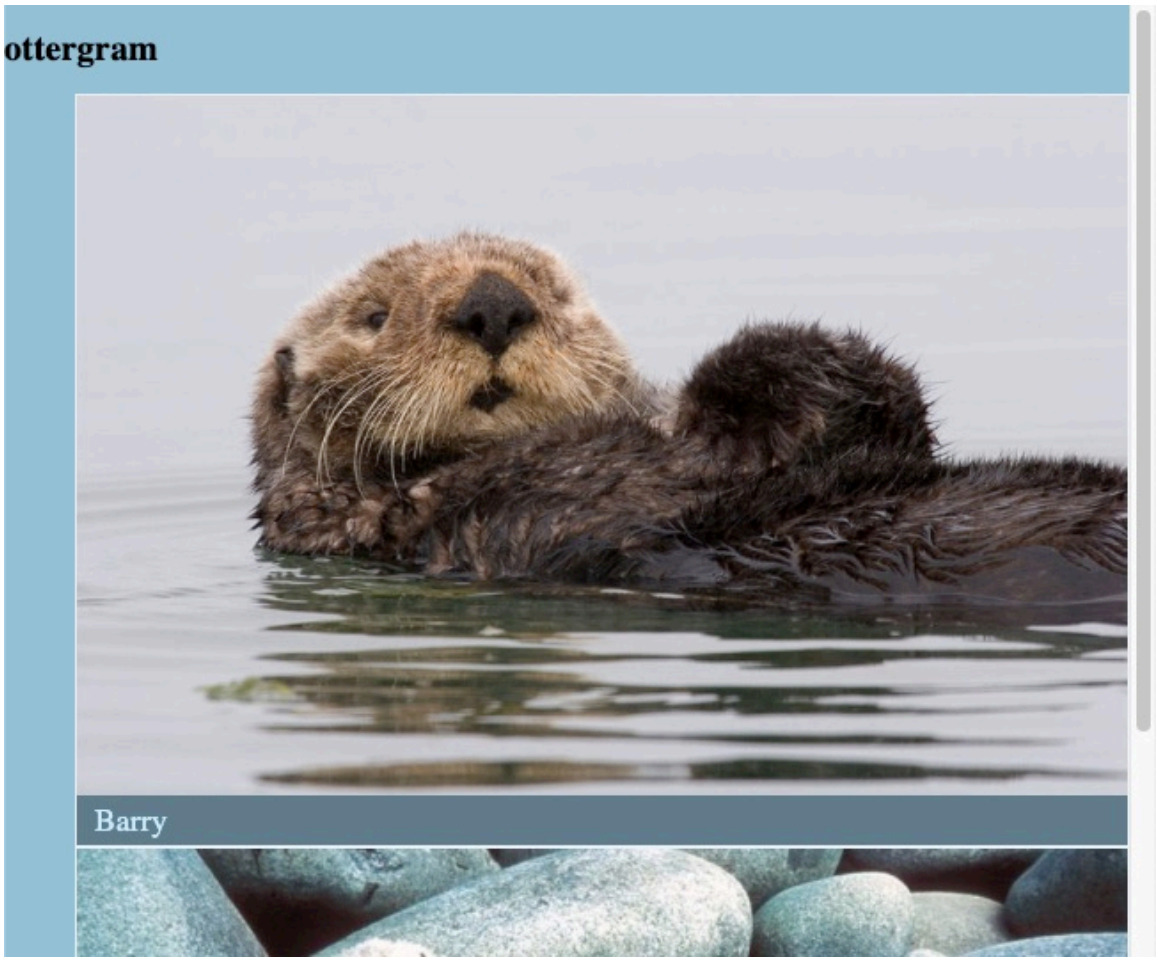
```
...
.thumbnail-item {
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
```

```
  }
+ .thumbnail-list {
+   list-style: none;
+ }
  .thumbnail-image {
    ...
```

To get rid of the whitespace, you will use the same technique you used with the `.thumbnail-image`. Each `.thumbnail-item` has that whitespace by default to accommodate items in a list, just as the `.thumbnail-image` elements had whitespace to accommodate neighboring text. Add a `display: block` declaration for `.thumbnail-item` to remove it.

```
  ...
  .thumbnail-item {
+   display: block;
    border: 1px solid rgb(100%, 100%, 100%);
    border: 1px solid rgba(100%, 100%, 100%, 0.8);
  }
  ...
```

With those additions, the bullets and the excess space above the images disappear, resulting in the more polished layout shown below.



Why use a bullet list if you do not want bullets? It is best to choose HTML tags based on what they are and not how the browser will style them by default. In this case, you want an unordered list of images, so a `ul` is the way to go. The `ul` container for your images will let you style them as a scrolling list when you add a detail image to your project in the later chapter on flexbox. The fact that the browser represents each `li` with a bullet by default is not important, as they are easily removed.

Next, you are going to adjust the spacing of the items in the list. The individual `.thumbnail-item` elements currently have no space between them. You are going to add margins between adjacent thumbnails.

However, you do not want to add a margin to *all* of the list items. Why not? Because the heading already has a margin, so the first list item does not need one. This means that you cannot use the `.thumbnail-item` class selector, at least not on its own. Instead, you will use selector syntax that targets elements based on their relationship to other elements.

Relationship Selectors

Look again at the diagram of your project in the DOM. It looks much like a family tree, doesn't it? This similarity gives the set of relationship selectors their names: *descendent selectors*, *child selectors*,

sibling selectors, and *adjacent sibling selectors*.

Relationship selector syntax includes two selectors (like class or element selectors) joined by a symbol called a *combinator* that determines the targeted relationship between them. To understand how relationship selectors work, it is important to keep in mind that the browser reads selector syntax from *right to left*. Let's look at some examples.

A descendent selector targets any element of one specified type that is the descendent of another specified element. For example, to select any `span` element that is the descendent of the `body` element, the syntax would be:

```css
body span {
  /* style declarations */
}
```

This syntax uses no combinator. Because it is read from right to left, it targets any `span` descended from a `body`, which in the current code means the thumbnail titles. It would also affect any `span`s that might be added within the `header` or elsewhere within the `body`.

Note that you can also use a class selector (or attribute selector, or indeed any type of selector) within a relationship selector, so the selector above could also be written as:

```css
body .thumbnail-title {
  /* style declarations */
}
```

Child selectors target elements of a specified type that are the immediate children of another specified element. Child selector syntax uses the combinator `>`. To use child selector syntax to target each `span` currently in Ottergram, the syntax would be:

```css
a > span {
  /* style declarations */
}
```

Reading from right to left, this selector targets any `span` that is the immediate child of an `a` element – again, the thumbnail titles. (Note: `li > span` would not target them because the `span` is not an immediate child of `li`. `li > a > span` would though.)

Sibling selector syntax uses the combinator `~`. As you might expect, this syntax targets elements with the same parent. However, because of the directional nature of relationship selectors, the results might not be exactly as you expect. Take this example:

```css
header ~ ul {
  /* style declarations */
```

```
}
```

This selector targets any `ul` that is *preceded* by a `header` with the same parent element. This selector would effectively target Ottergram's `ul`, because it has a sibling `header` that precedes it in the code. However, reversing the syntax (`ul ~ header`) would not match any elements, because there is no `header` preceded by a sibling `ul` .

The final relationship selector type is the adjacent sibling selector, which targets elements that are *immediately* preceded by a sibling of the specified type. The adjacent sibling combinator is `+` :

```css
li + li {
  /* style declarations */
}
```

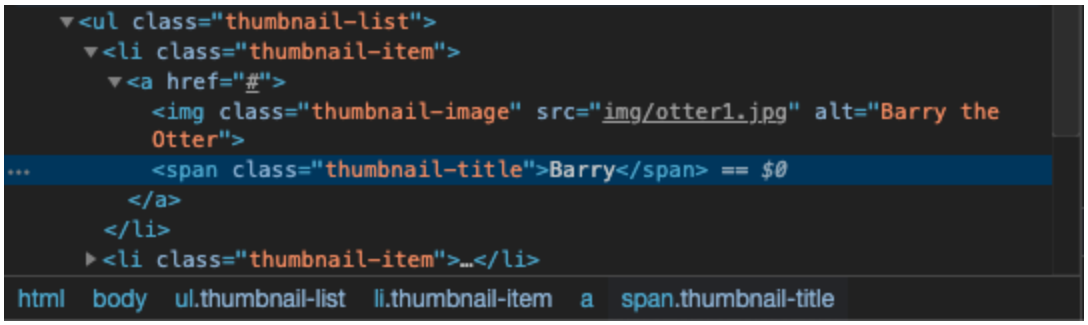
This syntax would select all `li` elements immediately preceded by a sibling `li` . The result is that the declared styles would be applied to the second through fifth `li` – but not the first, because it is not immediately preceded by another `li` . (Note that the general sibling selector and the adjacent sibling selector would work the same way at the moment, due to Ottergram's relatively simple structure.)

Back to the task at hand: adding a margin to the top of each list item except the first. If you used a descendent or child selector to target the `.thumbnail-item` class or the `span` or `li` elements, the margin would be applied to all five thumbnails. Because you want to style all but the first, use the adjacent sibling syntax in `styles.css` to add a top margin to only those thumbnails that are immediately preceded by another thumbnail.

```
  ...
  a {
    text-decoration: none;
  }
+ .thumbnail-item + .thumbnail-item {
+   margin-top: 10px;
+ }
  .thumbnail-item {
    ...
```

Save your file and check out the results in your browser.

Note that the DevTools give you an easy way to find out the nesting path of an element, which can help with writing relationship selectors. If you click one of the `span` elements inside one of the `li` elements, you can see its path at the bottom of the elements panel.



For one final tweak to the thumbnail list's appearance, return to `styles.css` and override the padding that the `ul` inherits from the user agent stylesheet so that the images are no longer indented.

```
  ...
  .thumbnail-list {
    list-style: none;
+   padding: 0;
  }
  ...
```

As usual, save your file and switch to your browser to see your results.

Ottergram is starting to look polished. With some styling for the header, you will have a nice static web page.

Adding a Font

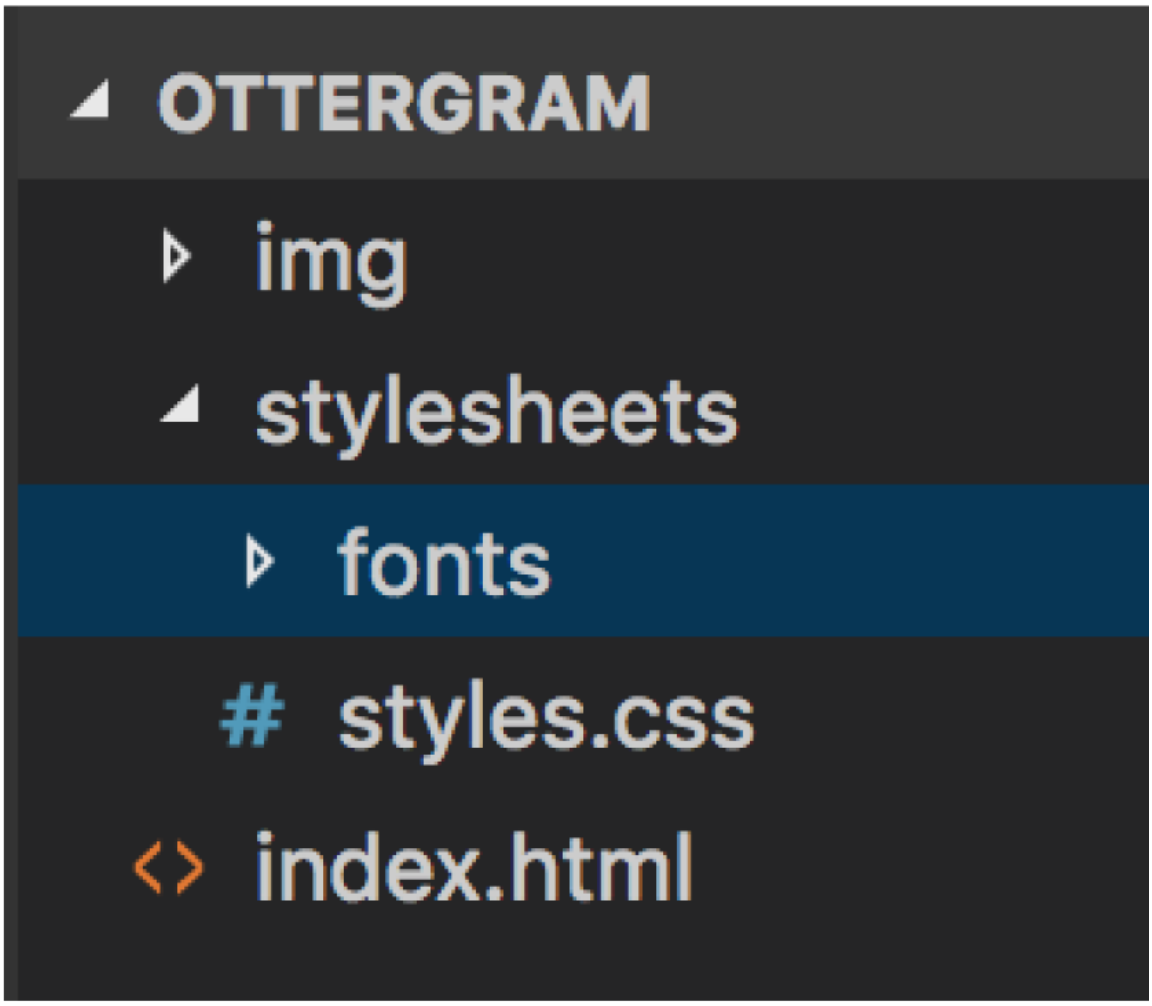
Earlier, you added the `.logo-text` class to the `h1` element. Use that class as the selector for a new styling rule in `styles.css`. Insert it after the styles for the anchor tag. (In general, the order of your styles only matters when you have multiple rule sets for the same selector. In Ottergram, the styles are arranged in roughly the same order as they appear in the code. This is a matter of preference, and you are free to organize your styles as you see fit.)

```
  ...
  a {
    text-decoration: none;
  }
+ .logo-text {
+   background: white;
+   text-align: center;
+   text-transform: uppercase;
+   font-size: 37px;
+ }
  .thumbnail-item + .thumbnail-item {
    ...
```

First, you gave the header a white background. Then, you centered the text inside the `.logo-text` element and used the `text-transform` property to format it as uppercase. Finally, you set the font size. Your results will look like this:

Ottergram looks great. Great... but a little plain for a website with *otters*. To add some pizzazz, you can use a font for the header other than the default provided by the user agent stylesheet.

We included some fonts in the resource files you already downloaded and added to your project directory. To use them, you need to copy the `fonts` folder into your project. Place it *inside* your `stylesheets` folder. (You may need to tell VS Code to "Copy Folder".)



Now you only need to point some styles to those fonts.

The resource files include many formats of each font. As usual, different browser vendors support different kinds of fonts. To support the widest array of browsers, you need to include all of them in your project. Yes, all of them.

To help you out, the *@font-face* syntax lets you give a custom name to a family of fonts that you can use in the rest of your styles.

An `@font-face` block is a little different from the declaration blocks you have been using. Inside of the `@font-face` block are three main parts:

- First, the `font-family` property, whose value is a string identifying the custom font name you can use throughout your CSS file.
- Next, several `src` declarations specifying different font files. (Take note – the order is important!)
- Last, declarations that modify the font's presentation, such as the `font-weight` and the `font-style`.

Add an `@font-face` declaration for the `lakeshore` font family to the top of `styles.css` and a style declaration to use the new font for the `.logo-text` class.

```
+ @font-face {
+     font-family: 'lakeshore';
+     src: url('fonts/LAKESHOR-webfont.eot');
+     src: url('fonts/LAKESHOR-webfont.eot?#iefix') format('embedded-opentype'),
+          url('fonts/LAKESHOR-webfont.woff') format('woff'),
+          url('fonts/LAKESHOR-webfont.ttf') format('truetype'),
+          url('fonts/LAKESHOR-webfont.svg#lakeshore') format('svg');
+     font-weight: normal;
+     font-style: normal;
+ }
  body {
    font-size: 10px;
    background: rgb(149, 194, 215);
  }
  a {
    text-decoration: none;
  }
  .logo-text {
   background: white;
   text-align: center;
   text-transform: uppercase;
+  font-family: lakeshore;
   font-size: 37px;
  }
  ...
```

Admittedly, getting the `@font-face` declaration just right can be tricky, because the order of the individual `url` values is important. It is a good idea to keep a copy of the declaration for reference. You can also look into Visual Studio Code's snippets documentation at code.visualstudio.com/docs/editor/userdefinedsnippets to see how to create your own code "snippet," or template.

After declaring the custom `@font-face`, the rest of your CSS has access to the new `lakeshore` value for the `font-family` property. In the `.logo-text` declaration, you set `font-family: lakeshore` to apply the new font.

You'll also edit the thumbnail titles to use a sans-serif font. (Serifs are the little ticks on the font.)

```
  .thumbnail-title {
    display: block;
    margin: 0;
    padding: 4px 10px;
    background: rgb(96, 125, 139);
    color: rgb(202, 238, 255);
+   font-family: Arial, sans-serif;
    font-size: 18px;
  }
```

When providing a font, you can provide a list of fallback fonts. In this case, you told the system to use Arial, but if it doesn't have Arial, it falls back to `sans-serif`, which is a generic font name for any system font without serifs.

Save `styles.css`, switch to `Chrome`, and see how good it feels to have a web page as stylish as an otter.

You did a lot of styling work in this chapter, and Ottergram looks great! In the next chapter you will make it even better by adding interactive functionality.

🥉 Bronze Challenge: Color Change

Change the background color styles for `body`. Use the color picker in the DevTools to help you choose one.

For a more sophisticated color palette, go to color.adobe.com and create your own scheme for the `body` and `.thumbnail-title` background colors.

🔎 For the More Curious: Specificity! When Selectors Collide

You have already seen how you can override styles. You included the link for `normalize.css` before the one for `styles.css`, for example. This made the browser use the styles from `normalize.css` as a baseline, with your styles taking precedence over the baseline styles.

This is the first basic concept of how the browser chooses which styles to apply to the elements on the page, known to front-end developers as *recency*: As the browser processes CSS rules, they can override rules that were processed earlier. You can control the order in which the browser processes CSS by changing the order of the `<link>` tags.

This is simple enough when the rules have the same selector (for example, if your CSS and `normalize.css` were to declare a different `margin` for the `body` element). In this case, the browser chooses the more recent declaration. But what about elements that are matched by more than one selector?

Say you had these two rules in your Ottergram CSS:

```css
.thumbnail-item {
  background: blue;
}

li {
  background: red;
}
```

Both of these match your `<li>` elements. What background color will your `<li>` elements have? Even though the `li { background: red; }` rule is more recent, `.thumbnail-item { background: blue; }` will be used. Why? Because it uses a class selector, which is more specific (i.e., assigned a higher specificity value) than the element selector.

Class selectors and attribute selectors have the same degree of specificity, and both have a higher specificity than element selectors. The highest degree of specificity goes to *ID selectors*, which you have not seen yet. If you give an element an `id` attribute, you can write an ID selector that is more specific than any other selector.

ID attributes look like other attributes. For example:

```html
<li class="thumbnail-item" id="barry-otter">
```

To use the ID in a selector, you prefix it with `#`:

```css
.thumbnail-item {
  background: blue;
}

#barry-otter {
  background: green;
}

li {
  background: red;
}
```

In this example, the `<li>` is matched by all three selectors, but it will have a green background because the ID selector has the highest specificity. The order of the rules makes no difference here, because each has a different specificity.

In general, ID selectors should be avoided for styling. ID values must be unique in the document, so you cannot use the `id="barry-otter"` attribute for any other element in your document. Even though ID selectors have the highest specificity, their associated styles cannot be reused, making them a maintenance "worst practice."

To learn more about specificity, go to the MDN page developer.mozilla.org/en-US/docs/Web/CSS/Specificity .

The Specificity Calculator at specificity.keegan.st is a great tool for comparing the specificity of different selectors. Check it out to get a more precise understanding of how specificity is computed.

Coffee Shop

Setup

This chapter sets up your workspace.

As you build this section's project, you will learn the fundamentals and patterns for creating a web application with React.

What is React?

React is an open-source JavaScript library for building user interfaces. It doesn't handle everything like larger MVC frameworks, but it does push you in some very good directions. You will find it influences all the other layers of an application!

React's declarative approach allows developers to have more control over data flow in an application. This sets up a flow of predictable and testable changes to the UI. Instead of focusing on how to accomplish a change to the UI, developers can focus on what the UI should look like in a particular state. This also allows you to write code as reusable and composable components.

Utilizing React in the applications you build will make complex changes to data simple, fast, and responsive. To help with efficiency and performance, React uses a virtual DOM and diffing algorithms to batch updates.

React is also agnostic to your tech stack including how you manage data. You can also incrementally add it to an existing project rather than needing to rewrite the whole application at once.

One of the best parts of React is its leverage of standard JavaScript features, which means the API surface is small and everything is delegated to standard JavaScript patterns. So much of what you learn when developing with React can be used anywhere. Even outside React!

Installation and Tooling

To get started, you will need to install a few tools.

First, make sure you are using the latest LTS version of Node.js (>=16.0.0). You can check your version with the terminal command:

```sh
node --version
```

Installing React Developer Tools

Next, install the React Developer Tools extension for Chrome.

> **Install**
>
> Visit the Chrome Webstore https://chrome.google.com/webstore/category/extensions . Search for "React Developer Tools". It should be offered by Facebook. Click Add to Chrome. Follow the prompts to install the extension.

Node and npm

When you installed Node.js, you got access to two command-line programs: `node` and the Node package manager, `npm`. You may recall that `npm` allows you to install open-source development tools, like `browser-sync`. The `node` program does the work of running programs written in JavaScript.

The `npm` command-line tool can perform a variety of tasks, like installing third-party code that you can incorporate into your project and managing your project's workflow and external dependencies.

Creating a React Application

Installing Node also gave you a third tool `npx`. `npx` gives us a command-line utility for running commands from a central cache. `create-react-app` is a tool for creating a new React application with the latest configuration features. You could install `create-react-app` globally with `npm install -g` like you did with `browser-sync`. However, you likely will not be using `create-react-app` very often and want to get the latest version every time you use it so it's easiest to use `npx`.

In your terminal, navigate to your projects folder. Then, create a new React application called `coffee-shop`:

```sh
npx create-react-app coffee-shop
```

Note: You may see a prompt to confirm installing `create-react-app`. You can confirm by pressing "y".

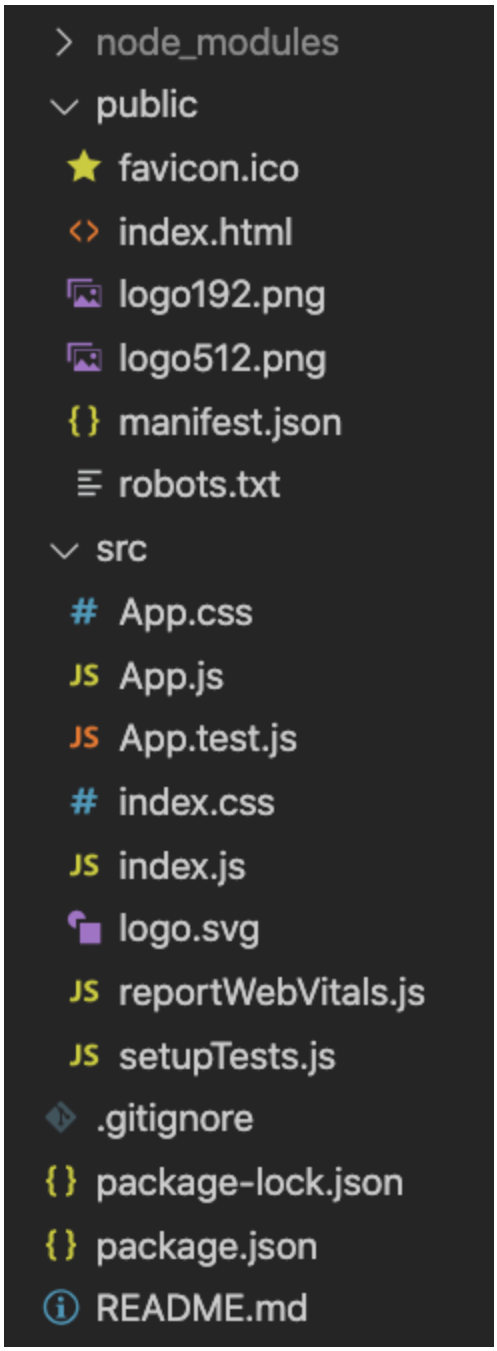
Creating a new React application may take a few minutes. You can see in the terminal output a list of dependencies that have been installed and available commands you can run in your new application.

```
Success! Created coffee-shop at /Users/loren/react-course/coffee-shop
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd coffee-shop
  npm start

Happy hacking!
```

Your React application was created in a sub-directory named `coffee-shop`. Change your terminal to that directory by running:

```sh
cd coffee-shop
```

Let's explore the directory structure of the example app that was generated for you. Open the project in Visual Studio Code. Your project should look like this:



A few files were generated for you, including a sample `README.md`, `package.json`, and `package-lock.json`. There are also three directories: `node_modules/`, `public/`, and `src/`.

The `package.json` file acts as your Node project's manifest. It holds your project's name, version number, description, and other information. More important, it is where you can store configuration settings and commands for npm to use when testing and building your application.

The `package-lock.json` file contains the exact version numbers of every dependency installed by `npm`. In `package.json`, you can specify a range of acceptable versions. `package-lock.json`

contains the exact version installed. This is important for testing and continuous delivery. You want to be sure the server installs the exact same version that was used for testing not an upgraded version. As a result, `package-lock.json` should be committed into version control.

The `node_modules/` directory contains all the project dependencies. These were installed by `npm`, Node Package Manager. This directory is large and exactly matches the specifications from `package-lock.json` as a result it is usually not committed into version control.

The `src/` directory contains the example React app. Most of the files in this folder are sample files to get us started.

The `src/index.js` file is the main JavaScript entry point. This is where your React app will bind to a DOM node in the main HTML template page. In this case, the DOM node React should bind to has an ID of `'root'`.

The `public/` directory contains the main HTML template page - `index.html`. This file includes `<div id='root'></div>`. This is the DOM node that will be used to render your React code.

You may notice that `public/index.html` does not contain any script tags, which means this file is not loading any external JavaScript files. The project also does not have any config files. Where are the build tools?

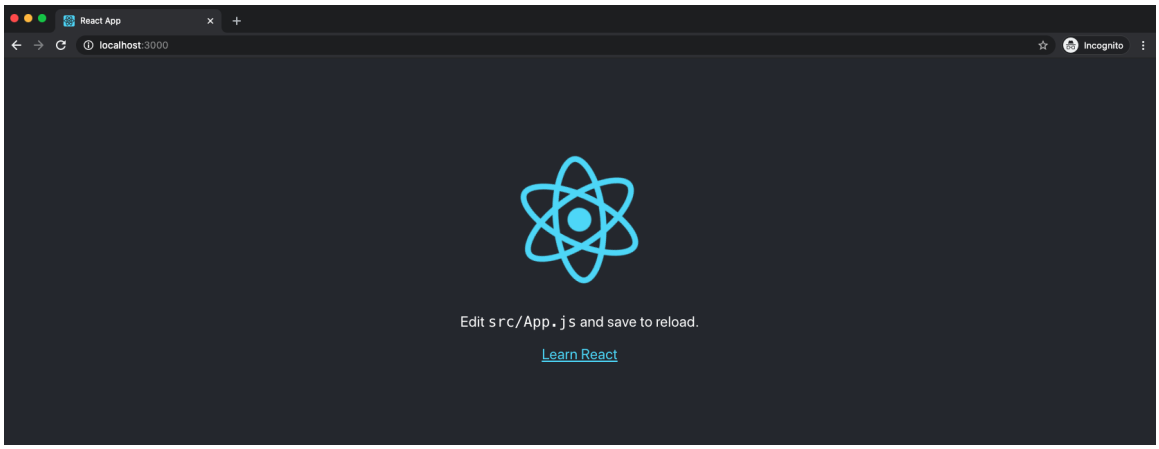
Create React App only creates the files you need to build a basic application and hides the preconfigured build settings elsewhere. This allows you to focus on writing code instead of configuration details! All of this is done through a single dependency found in the `package.json` file: `react-scripts`. This dependency includes all scripts and configuration needed to get your React application up and running. It uses several tools, including Webpack, Babel, autoprefixer, and ESLint.

Running the Dev Server

In the terminal, make sure you are in the project directory. Then, run:

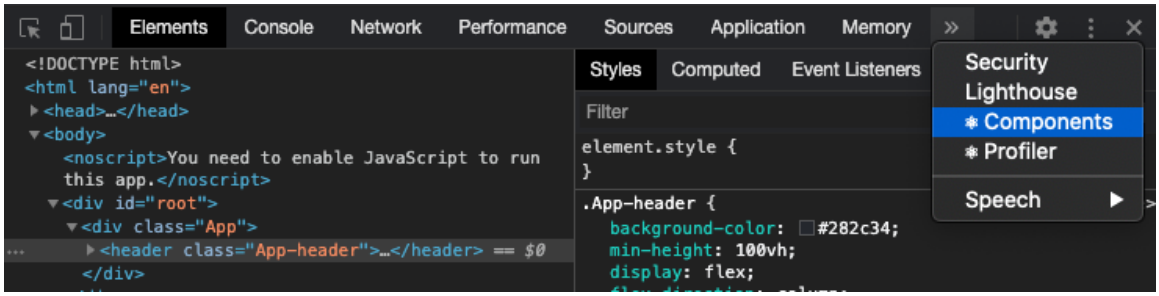
```sh
npm start
```

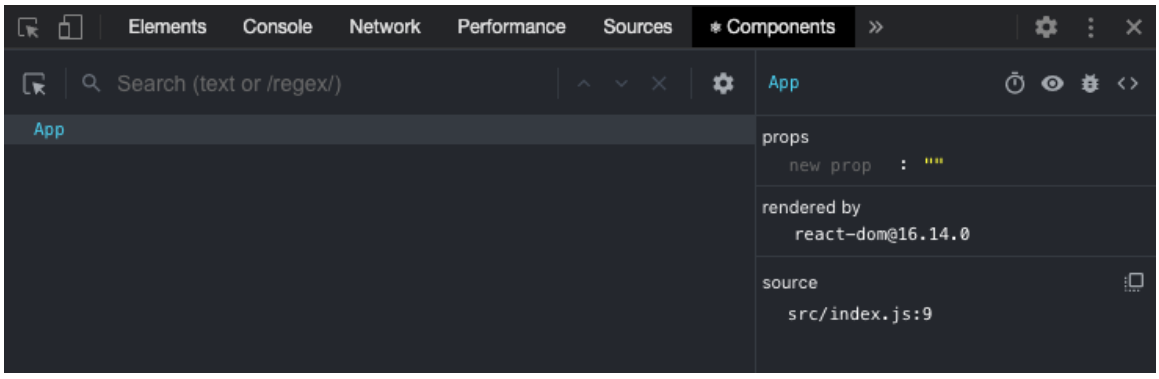
This command will build your app and start a server so you can access it locally. In Chrome, open a new browser window and go to `http://localhost:3000` to view your React application.



As an added convenience, Create React App includes a config to watch your files for changes. If you change a file while your application is running, the page will automatically reload and reflect the changes. This makes development much faster than if you had to restart the server every time you wanted to make a change!

When viewing the app in Chrome, you will also want to open the new React Components tab in the DevTools panel. This tab may be under the double arrow depending on the width of your browser.





Once there are more components, you will be able to click on a component in the tree view to see more details about it in the side pane. You can also use the search bar to find components.

🔎 npx

> **For the More Curious**
>
> This section and all sections with 🔎 are more details on a particular topic.
>
> You do not need to do anything with the code displayed in these sections. They simply provide more details on a topic.

`npx` is a tool for executing npm package binaries. It starts with binaries within the local project and $PATH. If it doesn't find the package there, it will download a temporary copy of the latest version and execute the binary from that download.

Without `npx`, several commands would have been needed to create the `coffee-shop` app:

```sh
npm install --global create-react-app
create-react-app coffee-shop
npm uninstall --global create-react-app
```

With `npx`, you only needed one line!

```sh
npx create-react-app coffee-shop
```

`npx` should be used with care. Ultimately, you're executing code that has been downloaded from the internet on your local machine. Take care vetting the packages that get run or reading the source code yourself.

For more info, check out https://www.npmjs.com/package/npx .

Linting

A linter is a tool that is used to enforce code styles as well as help detect potential errors in code in javascript. In statically typed languages like Java, the compiler can check for many more code errors that create runtime errors in javascript. The ESLint tool can perform inspections for unused variables (which can indicate a misspelled variable later) and missing variables. It also enforces code patterns that can easily cause bugs such as the following code:

```js
return
{
    name: "ESLint"
};
```

This code actually evaluates as

```js
return;
{
    name: "ESLint"
};
```

Which is one of the reasons the `semi` rule exists which forces developers to type all the semicolons. The full documentation of this rule is available at https://eslint.org/docs/rules/semi .

When you install ESLint, you also have to pick the rules that you want to use. For this application, you will use AirBnB's popular ruleset. Their ruleset is opinionated, but the preset lint rules will result in code that is easy to read and will eliminate formatting discrepancies throughout the rest of this course. Also, it's better to have coders angry at the lint rules than co-worker's who are manually enforcing on code style in a PR.

Install and Configure AirBnB ESLint

ESLint ships with create-react app, but you need to install the AirBnB ruleset and its peer dependencies and configure it to run.

package.json CHANGED

```
@@ -6,6 +6,12 @@
      "@testing-library/jest-dom": "^5.16.1",
      "@testing-library/react": "^12.1.2",
      "@testing-library/user-event": "^13.5.0",
+     "eslint": "8.8.0",
+     "eslint-config-airbnb": "19.0.4",
+     "eslint-plugin-import": "2.25.4",
+     "eslint-plugin-jsx-a11y": "6.5.1",
+     "eslint-plugin-react": "7.28.0",
+     "eslint-plugin-react-hooks": "4.3.0",
      "react": "^17.0.2",
      "react-dom": "^17.0.2",
      "react-scripts": "5.0.0",
@@ -15,13 +21,29 @@
      "start": "react-scripts start",
      "build": "react-scripts build",
      "test": "react-scripts test",
+     "lint": "eslint src --max-warnings=0",
      "eject": "react-scripts eject"
    },
    "eslintConfig": {
      "extends": [
        "react-app",
-       "react-app/jest"
-     ]
+       "react-app/jest",
+       "airbnb",
+       "airbnb/hooks"
+     ],
+     "rules": {
+       "react/react-in-jsx-scope": "off",
+       "react/jsx-filename-extension": [
+         "warn",
+         {
+           "extensions": [
+             ".js",
+             ".jsx"
+           ]
```

```
+         }
+       ],
+       "no-console": "off"
+     }
    },
    "browserslist": {
      "production": [
```

After updating the dependencies, you need to install them by running `npm install`. You are likely still running the React app from above. You will want to stop that with `Ctrl + C`. (Note: It is `Ctrl` on both Windows and Mac. This does not use `Cmd`.)

Running the linter

Now, you can use `npm run lint` to run the linter to check code style.

Oops, you already have lint errors! Why? Create React App, doesn't adhere to the lint rules from AirBnB that you configured.

Let's fix them. All the problems should be auto-fixable, you can do that by running the command below.

```sh
npm run lint -- --fix
```

What's the extra `--` for? `--fix` is a param that needs to be passed to `eslint`. The extra `--` tells `npm` to pass all the arguments after that on to the command being executed.

Why not just always autofix? Most of the time this is okay, but sometimes autofixing changes code that you didn't want changed. Usually this is because you made a code mistake which produced working code and happens to look like a lint error, so the linter will happily reformat the code for you. Therefore, it is good to take a quick skim over the list of errors before autofixing them. (Just like it's good to take a quick look over the files you are about to commit before committing everything in `git`.)

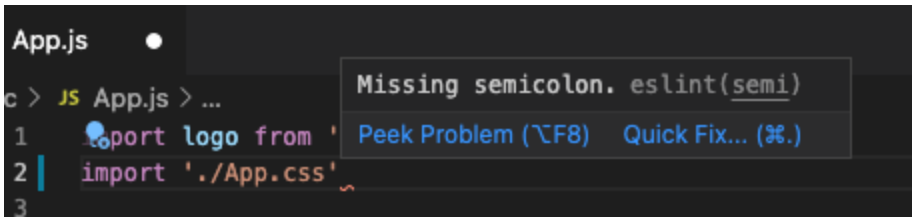
Now, `npm run lint` should not output any errors or warnings!

Installing ESLint Extension in VS Code

If you haven't already, you'll want to install this extension for ESLint in VS Code.
https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

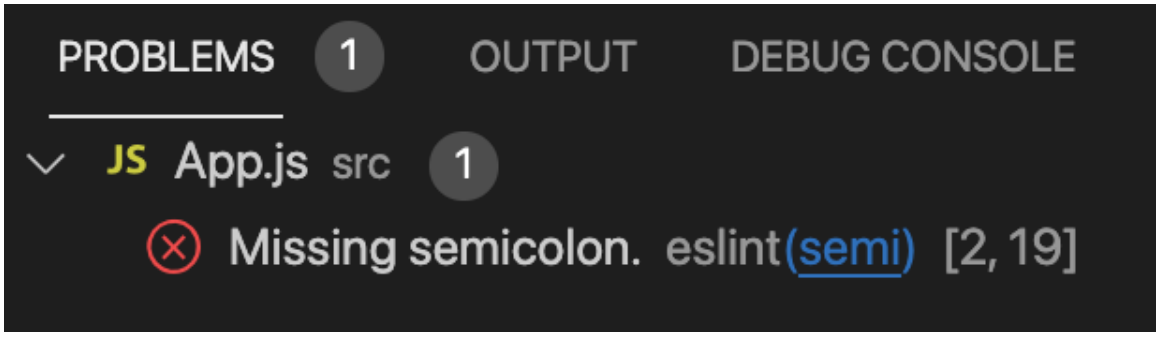
Once you've installed, it may take a few steps to enable the extension. First, restart VS Code. Then, open `src/App.js`, and delete the semi-colon at the end of the second line.

If everything is working, after a few seconds, your line without the semi-colon should show a little red squiggle under it. If this does not appear, scroll down to continue installing. If you mouse over it, you should get a popup with the eslint error.



You should also see one problem in the bottom left of the taskbar.

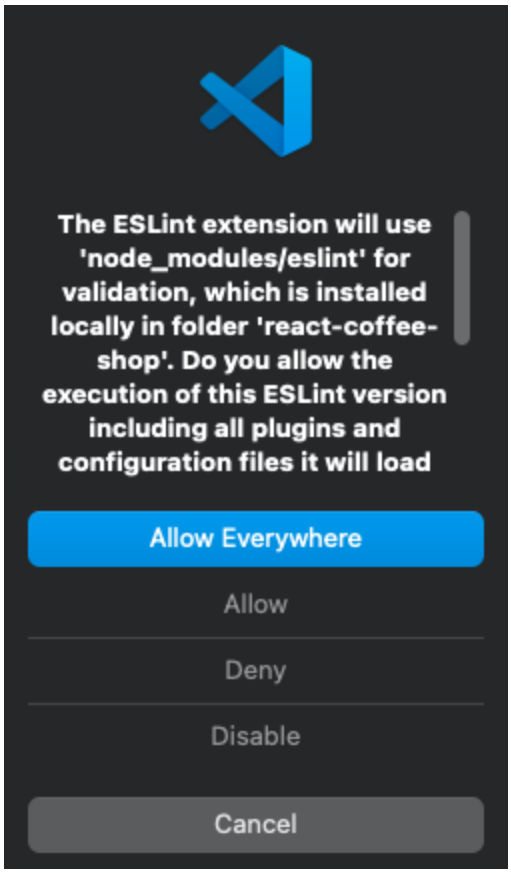
If you click that, it will open the Problems pane.



Still Not Working?

You should see `ESLINT` in the bottom right of the screen on the status bar.

Click "ESLINT". This will open a dialog box. Click "Allow Everywhere".



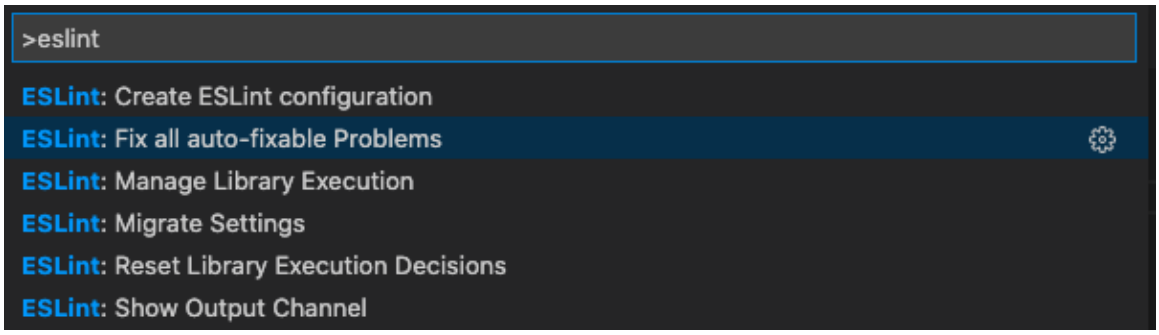
After a few seconds, ESLint should start working as described above.

ESLint Shortcuts

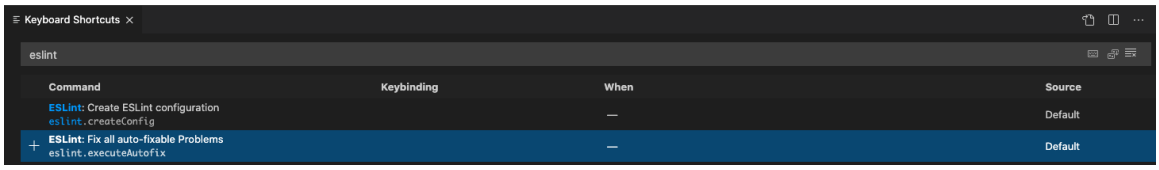
With this plugin installed, you can autofix the document from the actions menu. You can get to the actions search with:

- Windows: `Ctrl + Shift + P` or `F1`
- Mac: `Cmd + Shift + P`
- Ubuntu: `Ctrl + Shift + P`

Type `Eslint`. Then, select "ESLint: Fix all auto-fixable problems".



You can add a shortcut to this by going to the Keyboard Shortcuts preferences (from the main menu bar: `Code -> Preferences -> Keyboard Shortcuts`). Search for `eslint`. Then, hover over the row for "ESLint: Fix all auto-fixable problems" and click the plus to the left of it.



I use `Ctrl + Shift + L`, but it may take you a few tries to find a combination that is not already assigned. Note: It will take two combinations and allow you to make a chord where you must press the first and then the second. When you enter a third combination, it will jump to just that combination. So if you try `Cmd + Shift + L` and find it already has commands, you will need to press `Ctrl + Shift + L` twice to try it. The first time will create a chord with `Cmd + Shift + L`. The second will create the single shortcut `Ctrl + Shift + L`.

It is also possible to set ESLint to autofix files on save. You can find more details on how to enable that on the page where you installed the extension. I personally prefer that code not auto-format because I sometimes intentionally write code with a lint error if I'm just testing something or making a quick fix. Assuming the new code works, the lint error forces me to go back and refactor the code (unless it's auto-fixed away on save). I usually do this by leaving off semicolons or using the wrong quotes (double vs single quotes). Both of those options will autofix.

🔎 Linting Pre-Commit

It can be helpful to have the linter always run before committing. Here are some npm packages that can help with ensuring everyone lints before committing code.

`husky` helps ensure git hooks like `pre-commit` are installed for everyone on the project. You can read more about it at https://www.npmjs.com/package/husky .

Linting the whole project works for small projects, but when projects become large, linting the whole project can be slow.

`lint-staged` is another npm package you can use so only the files that are staged to be committed are run through the linter. This causes the pre-commit hook to be much faster. You can find out more about lint staged at https://www.npmjs.com/package/lint-staged .

Initial HTML

Now, you're ready to start your Home page.

If you open `src/App.js`, you will notice the contents in the `return` statement look a lot like a normal HTML file. They are actually a slightly different language called `JSX` which we will talk more about in the next chapter but for now, you can think of it as HTML.

Let's replace the current React default page with a header for the coffee shop.

src/App.js CHANGED

```
@@ -1,26 +1,10 @@
- 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>
+     <div>
+       <header>
+         <h1>Coffee Shop</h1>
        </header>
      </div>
    );
```

It may be plain, but now your customers know you are a coffee shop.

Coffee Shop

Title

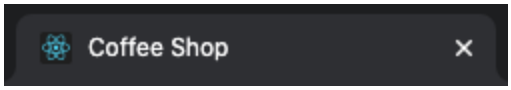
There is a bit of a problem. The page text says coffee shop, but the title on the tab does not.

Let's fix that.

public/index.html CHANGED

```
@@ -24,7 +24,7 @@
24 24          work correctly both with client-side routing and a non-root public URL.
25 25          Learn how to configure a non-root public URL by running `npm run build`.
26 26        -->
27    -      <title>React App</title>
   27 +      <title>Coffee Shop</title>
28 28      </head>
29 29      <body>
30 30        <noscript>You need to enable JavaScript to run this app.</noscript>
```

Nice, now the tab has a title of "Coffee Shop." (Note: You may need to manually refresh the page in your browser.)



Favicon

One more problem with the tab is that it shows the React logo. Wouldn't it be nice to show a picture of some coffee instead?

To help us get started, we've supplied some images for the coffee shop. You can download them from https://reactbook.bignerdranch.com/resources.zip .

The icons are from FontAwesome Free https://fontawesome.com/icons?d=gallery&m=free .

> **Copy Files**
>
> Copy the `favicon.png` from your resources zip to `public/favicon.png`. You can also delete, `public/favicon.ico`, `public/logo192.png`, and `public/logo512.png`.

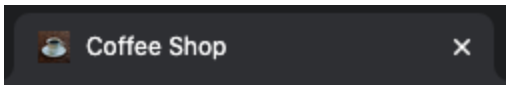
Now, let's swap out the favicon to point to the new one.

public/index.html CHANGED

```
@@ -2,14 +2,14 @@
2 2    <html lang="en">
3 3      <head>
4 4        <meta charset="utf-8" />
5    -      <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
   5 +      <link rel="shortcut icon" type="image/png" href="%PUBLIC_URL%/favicon.png" />
6 6        <meta name="viewport" content="width=device-width, initial-scale=1" />
```

```
 7   7       <meta name="theme-color" content="#000000" />
 8   8       <meta
 9   9         name="description"
10  10         content="Web site created using create-react-app"
11  11       />
12      -     <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
        12 +  <link rel="apple-touch-icon" href="%PUBLIC_URL%/favicon.png" />
13  13       <!--
14  14         manifest.json provides metadata used when your web app is installed on a
15  15         user's mobile device or desktop. See https://developers.google.com/web/fundame
```

Yay! You have a coffee cup. (Note: This change will also require a manual refresh in the browser.)



🔎 Favicon Formats

All browsers have historically supported `ico` as the favicon format, but it can be hard to produce those files. Most browsers now support PNG images as favicons as well.

You can check browser support here https://caniuse.com/link-icon-png .

Many browsers support `svg` files now as well.

🔎 %PUBLIC_URL%

Your React app does not have to be the root of the URL when deployed. The home page could be https://bignerdranch.com/react/. This creates a problem for images because the image src `/favicon.png` would resolve to `https://bignerdranch.com/favicon.png` not `https://bignerdranch.com/react/favicon.png`.

Leaving off the slash at the front would fix that, but react also has "fake" folders so another part of the site could be `https://bignerdranch.com/react/details/course` in this case without the slash the browser would now look for the favicon at `https://bignerdranch.com/react/details/favicon.png`.

`create-react-app` helps us solve both of these problems by providing `%PUBLIC_URL%`. Assuming you properly configure your homepage in `package.json`, React will fill in the correct beginning so both `https://bignerdranch.com/react/` and `https://bignerdranch.com/react/details/course` resolve the favicon to `https://bignerdranch.com/react/favicon.png`.

Manifest

Let's also update `manifest.json` to reflect the new icon.

public/manifest.json CHANGED

```
          @@ -1,25 +1,15 @@
 1   1     {
```

```
2        -  "short_name": "React App",
3        -  "name": "Create React App Sample",
         2  +  "short_name": "Coffee Shop",
         3  +  "name": "Coffee Shop",
4        4     "icons": [
5        5       {
6        -       "src": "favicon.ico",
7        -       "sizes": "64x64 32x32 24x24 16x16",
8        -       "type": "image/x-icon"
9        -     },
10       -     {
11       -       "src": "logo192.png",
         6  +       "src": "favicon.png",
12       7         "type": "image/png",
13       8         "sizes": "192x192"
14       -     },
15       -     {
16       -       "src": "logo512.png",
17       -       "type": "image/png",
18       -       "sizes": "512x512"
19       9       }
20       10    ],
21       11    "start_url": ".",
22       12    "display": "standalone",
23       -     "theme_color": "#000000",
24       -     "background_color": "#ffffff"
         13 +  "theme_color": "#080c14",
         14 +  "background_color": "#282c34"
25       15  }
```

The manifest provides information for allowing the webpage to be installed to the home screen of a mobile device without having to release an app to the app store. Normally, you should provide all the image sizes that it previously provided, but you will just provide the one for now.

The coffee shop name was already short, so the same name could be used in both fields, but you could have named it "The Big Nerd Ranch Coffee Shop" and left the short name as "Coffee Shop". Feel free to get creative with the name for your coffee shop.

The background color will show before the app loads the style sheet. The color you've provided will be the future background color for the website.

Theme color is sometimes used by the OS to adjust the display. Chrome on Android uses it to color the browser UI elements such as the address bar.

You can read more about the manifest at: https://developer.mozilla.org/en-US/docs/Web/Manifest

Show Available Items

You are ready to add some items to the home page so people know what they can order.

> **Copy Files**
>
> Copy the items folder from your resources zip to `src/items`.

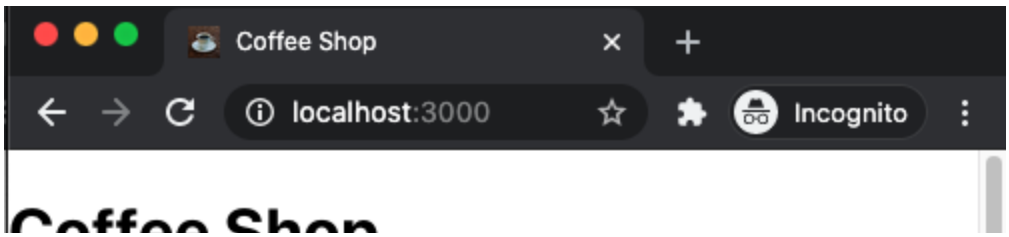
The final app will have more items available and will take advantage of React tools to avoid some of the repetitive typing. For now, you will add 5 items to the home page.

src/App.js CHANGED

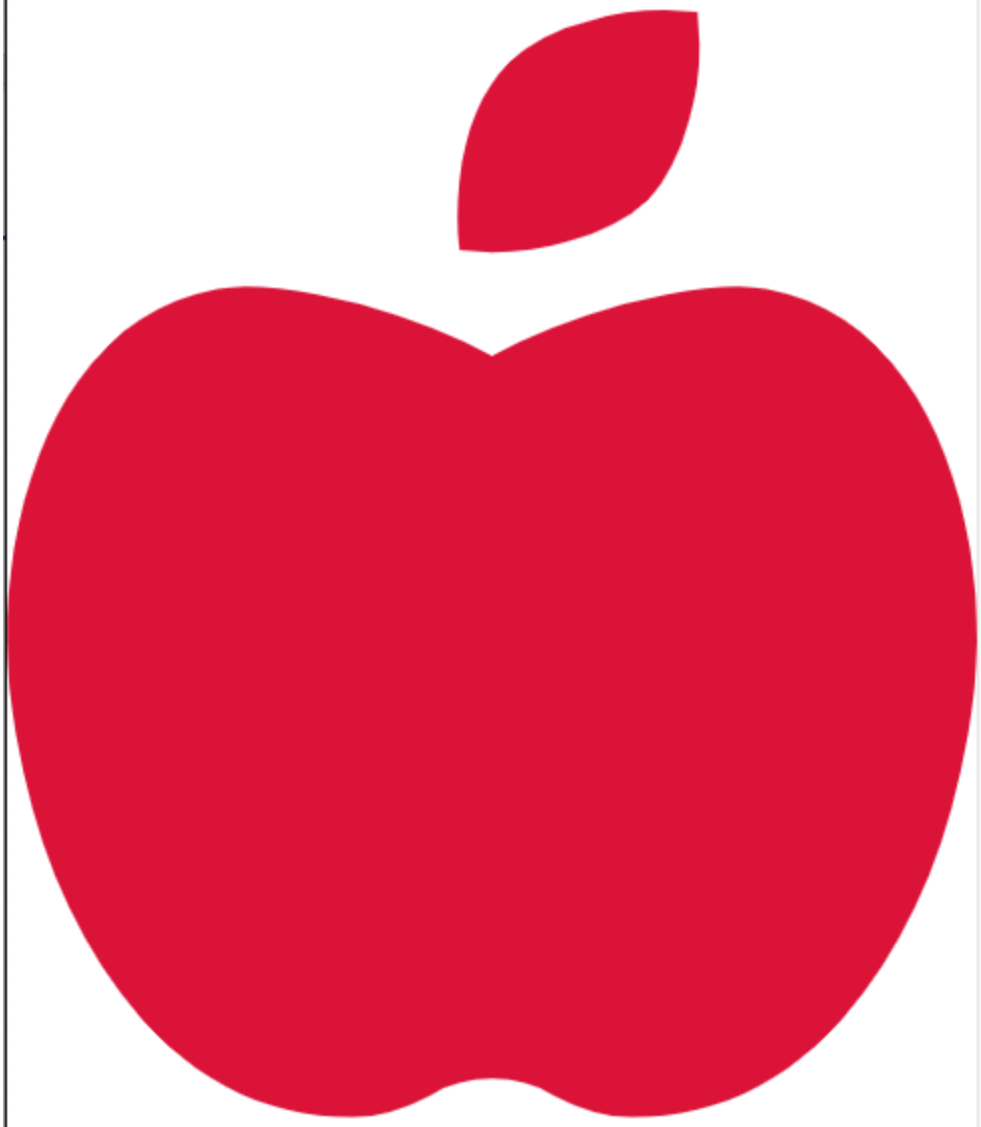
```
@@ -1,4 +1,9 @@
 import './App.css';
+import Apple from './items/apple.svg';
+import Coffee from './items/coffee.svg';
+import Cookie from './items/cookie.svg';
+import Tea from './items/tea.svg';
+import Wine from './items/wine.svg';

 function App() {
   return (
@@ -6,6 +11,28 @@ function App() {
       <header>
         <h1>Coffee Shop</h1>
       </header>
+      <div>
+        <a href="#todo">
+          <img src={Apple} alt="Apple" />
+          <div>Apple</div>
+        </a>
+        <a href="#todo">
+          <img src={Coffee} alt="Coffee" />
+          <div>Coffee</div>
+        </a>
+        <a href="#todo">
+          <img src={Cookie} alt="Cookie" />
+          <div>Cookie</div>
+        </a>
+        <a href="#todo">
+          <img src={Tea} alt="Tea" />
+          <div>Tea</div>
+        </a>
+        <a href="#todo">
+          <img src={Wine} alt="Wine" />
+          <div>Wine</div>
+        </a>
+      </div>
     </div>
   );
 }
```

The main difference between a simple HTML file and the code above is importing the image files into the JavaScript file and passing those files to the `img` tags as a variable. While you could have put the images in `public` like you did with the favicon, doing this allows several things. First, like `%PUBLIC_URL` above, it means you don't have to worry about where the image actually lives on the site. React will handle getting the URL correct. Second, React will handle cache busting by appending extra characters to the image name to force browsers to reload the image if you change it in the future. Third, it allows for image optimizations. Preprocessors can resize or otherwise alter the images. You can even tell React to use base64 and embed the images in the html itself to avoid an extra trip to the server. One important caveat though is that you cannot import the favicon like you did here because `index.html` is an HTML file not a Javascript file.

Now you can see the items in the browser.



Coffee Shop



Apple

Conclusion

The coffee shop now shows part of the menu on the website. You are ready to continue on to showing the full menu and using more of the power of React.

Components

You have the base app setup but are not using one of the main benefits of a front-end framework, components.

In this chapter, you will move HTML from `App.js` into separate component files to keep the code cleaner and increase its reusability.

Header Component

The component you will create will be for the header. It will be simple for now, but you will make it more complex later. You will create this in a new `components` folder.

> **Line Breaks**
>
> On Windows, you may get the following ESLint error:
>
> Expected linebreaks to be 'LF' but found 'CRLF'. `linebreak-style`
>
> If this happens, the instructions to fix it are below the screenshot.

src/components/Header.js ADDED

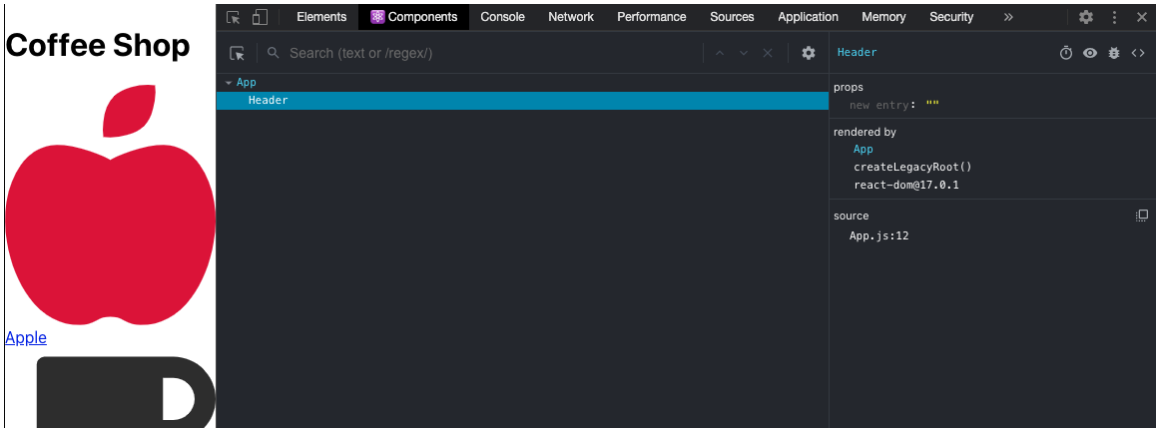
```
@@ -0,0 +1,9 @@
+ function Header() {
+   return (
+     <header>
+       <h1>Coffee Shop</h1>
+     </header>
+   );
+ }
+
+ export default Header;
```

src/App.js CHANGED

```
@@ -1,4 +1,5 @@
  import './App.css';
+ import Header from './components/Header';
  import Apple from './items/apple.svg';
  import Coffee from './items/coffee.svg';
  import Cookie from './items/cookie.svg';
@@ -8,9 +9,7 @@ import Wine from './items/wine.svg';
  function App() {
    return (
      <div>
-       <header>
-         <h1>Coffee Shop</h1>
-       </header>
+       <Header />
        <div>
          <a href="#todo">
```

```
16  15              <img src={Apple} alt="Apple" />
```

The app still looks the same, but the code is more organized. In the React Components tab of devtools, you can see your new component.



CRLF Line Breaks

You may have gotten this ESLint error in your new file.

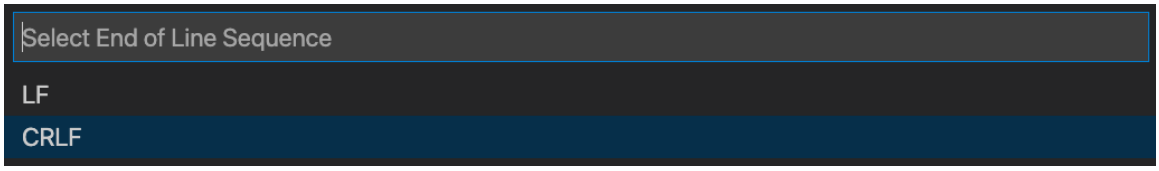
> Expected linebreaks to be 'LF' but found 'CRLF'. `linebreak-style`

Windows uses two characters by default for a new line a carriage return (CR) and a line feed (LF). Unix systems use just a line feed. Historically, the carriage return returned the print head to the left of the page (without advancing the paper), and a line feed advanced the paper one line. Some text communication protocols also would use just a line feed as advancing a line without moving the cursor back to the left so text would end up stair-stepped with just line feeds.

```
stair
     stepped
            text
```

Ok, so how do you fix it? VS Code lets you choose the newline character as well as file spacing. It's show on the status bar in the bottom right.

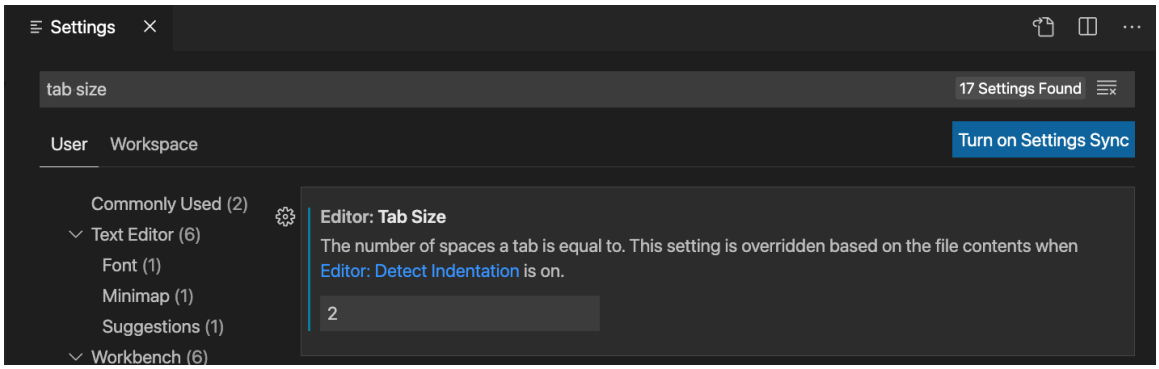
Click where it says "CRLF". In the popup window, select "LF".



Indentation Size

Similar to the above, if VS Code is using the wrong number of spaces as tabs by default, you can click where it says "Spaces: 2" (or "Spaces: 4") on the status bar in the bottom right. Then, click "Indent Using Spaces". Then, click "2". This will match the configuration you're using with ESLint.

You can change this for all files by going to the settings (Code -> Preferences -> Settings). Then, search for "tab size" and changing that to 2.



🔎 Why name the function?

For those already familiar with Javascript, you may be wondering why the code creates and exports the function separately.

This would have saved several lines of code:

```
export default () => (
 <header>
   <h1>Coffee Shop</h1>
 </header>
);
```

That would still render fine, but React devtools would no longer be able to tell you the name of the Component because it assumes that from the function name. In a large application, knowing the name of the component in devtools is very helpful for debugging. We promise you'll be thankful you took the time to write those extra lines later.

🔎 JSX

The contents of this file are `JSX` which is a way to write code in React that looks like HTML. In reality, Babel compiles the JSX to Javascript. Previously this was with `React.createElement` but that recently changed. You can read more about the changes here https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html . The new changes are only for code transformed by the compiler so manually written code still uses `React.createElement`.

To see the compiled of a given component, visit http://babeljs.io/repl . Then, paste in the source of the javascript file.

Here's a manually written version of `Header.js` with `React.createElement`:

```
import React from 'react';

function Header() {
  return React.createElement(
    'header',
    null,
    React.createElement(
      'h1',
      null,
      'Coffee Shop',
    ),
  );
}

export default Header;
```

You'll notice this code no longer uses JSX. Instead, it calls `React.createElement`, and the children of each element are passed as parameters. The 2nd parameter is for attributes like class, style, etc. The code has null since are not any attributes.

Thankfully, you don't have to type code like that because Babel compiles it for you!

Thumbnail Component

Next, you will make a Thumbnail component.

src/components/Thumbnail.js ADDED

```
@@ -0,0 +1,13 @@
+ // eslint-disable-next-line react/prop-types
+ function Thumbnail({ image, title }) {
+   return (
+     <a
+       href="#todo"
+     >
+       <img src={image} alt={title} />
+       <div>{title}</div>
+     </a>
+   );
+ }
```

```
+
+ export default Thumbnail;
```

src/App.js CHANGED

```
@@ -1,5 +1,6 @@
  import './App.css';
  import Header from './components/Header';
+ import Thumbnail from './components/Thumbnail';
  import Apple from './items/apple.svg';
  import Coffee from './items/coffee.svg';
  import Cookie from './items/cookie.svg';
@@ -11,26 +12,11 @@ function App() {
       <div>
         <Header />
         <div>
-          <a href="#todo">
-            <img src={Apple} alt="Apple" />
-            <div>Apple</div>
-          </a>
-          <a href="#todo">
-            <img src={Coffee} alt="Coffee" />
-            <div>Coffee</div>
-          </a>
-          <a href="#todo">
-            <img src={Cookie} alt="Cookie" />
-            <div>Cookie</div>
-          </a>
-          <a href="#todo">
-            <img src={Tea} alt="Tea" />
-            <div>Tea</div>
-          </a>
-          <a href="#todo">
-            <img src={Wine} alt="Wine" />
-            <div>Wine</div>
-          </a>
+          <Thumbnail image={Apple} title="Apple" />
+          <Thumbnail image={Coffee} title="Coffee" />
+          <Thumbnail image={Cookie} title="Cookie" />
+          <Thumbnail image={Tea} title="Tea" />
+          <Thumbnail image={Wine} title="Wine" />
         </div>
       </div>
     );
```

You've removed a bunch of duplicated code, and if you need to change a class, you can do that in just one place! This component looks a little different because you are using props.

You've temporarily disabled a lint rule `react/prop-types`. You will fix that shortly.

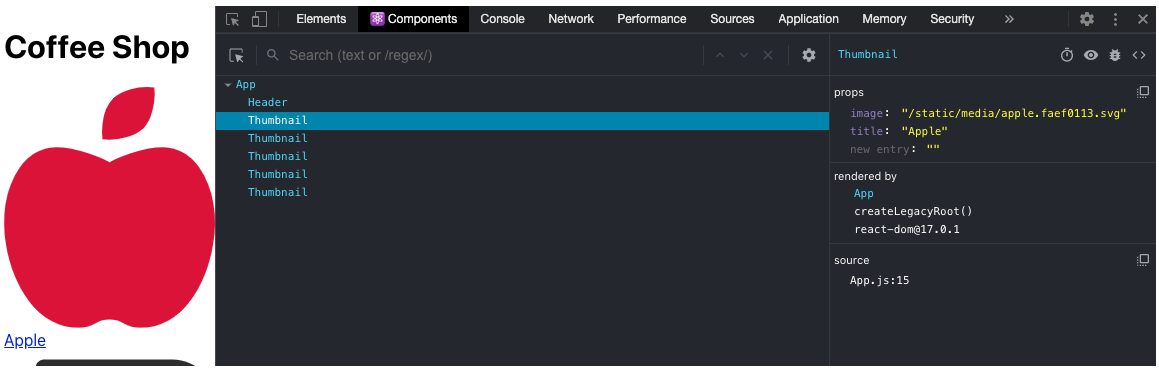
Props

React components are JavaScript functions and have parameters like any other function. React provides props as the first argument when it calls the function. A prop can be a string literal or a JavaScript expression.

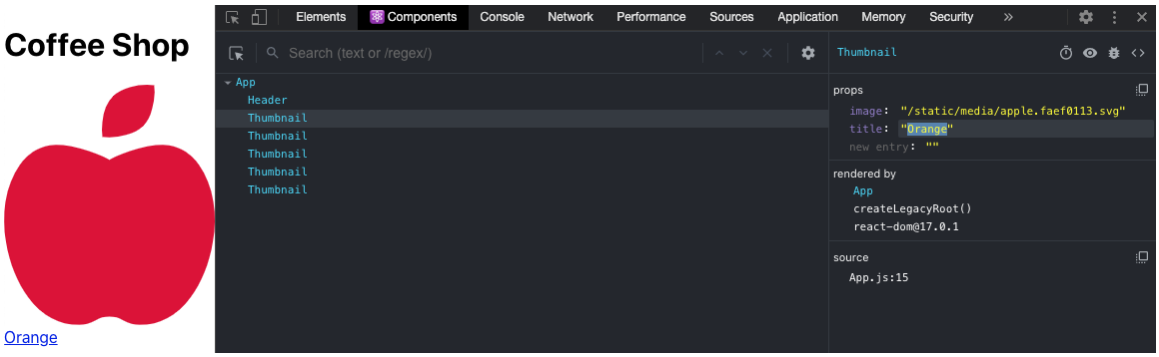
You can use props to pass data from parent components to child components, similar to how you would pass an argument to a function. Functional React components should be "pure"/deterministic functions meaning they will always return the same UI when given the same input.

You can send props to a component by using HTML attributes. So the `image` and `title` attribute on a `Thumbnail` become the props that you extract from the first argument of the function.

Look in the devtools, and you will be able to see the new props with the thumbnail element.



You can also edit props in devtools.



Just like edits in the Elements tab, these changes are only temporary and go away as soon as you refresh the page. Refresh the page to call it on Apple again.

🔎 Object Destructing

The first argument is actually an object which were are destructing to get the props.

Without the ESLint rules, this is equivalent code:

```
function Thumbnail(props) {
    return (
        <a
            href="#todo"
        >
            <img src={props.image} alt={props.title} />
            <div>{props.title}</div>
        </a>
    );
}
```

Having the props in the signature is useful for future developers using the component, which is why the AirBnB ESLint rules require you to destructure the props.

You can read more about destructuring on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#object_destructuring

Prop Types

You previously disabled the `react/prop-types` lint rule. You will enable that now.

> **Install**
>
> In a separate terminal from your app, CD into your `coffee-shop` directory.
>
> Then, install the new library with `npm install --save prop-types`

Now, you can add documentation to the Thumbnail component about what each prop type should be.

src/components/Thumbnail.js CHANGED

```
@@ -1,4 +1,5 @@
- // eslint-disable-next-line react/prop-types
+ import PropTypes from 'prop-types';
+
  function Thumbnail({ image, title }) {
    return (
      <a
@@ -10,4 +11,9 @@ function Thumbnail({ image, title }) {
    );
  }

+ Thumbnail.propTypes = {
+   image: PropTypes.string.isRequired,
+   title: PropTypes.string.isRequired,
+ };
+
  export default Thumbnail;
```

Now future developers know what they are expected to provide for each prop for the Thumbnail.

Item Details

React can help display a list of items without having to type out the `Thumbnail` component individually for each of them. To get ready to do that, you will first add a helper file in the `items` directory which lists all the available items.

src/items/index.js ADDED

```
@@ -0,0 +1,86 @@
+ import Apple from './apple.svg';
+ import Coffee from './coffee.svg';
+ import Cookie from './cookie.svg';
+ import Hamburger from './hamburger.svg';
```

```
+ import IceCream from './ice-cream.svg';
+ import Pizza from './pizza.svg';
+ import Stroopwafel from './stroopwafel.svg';
+ import Tea from './tea.svg';
+ import Toast from './bread.svg';
+ import Wine from './wine.svg';
+
+ export const itemImages = {
+   apple: Apple,
+   coffee: Coffee,
+   cookie: Cookie,
+   hamburger: Hamburger,
+   'ice-cream': IceCream,
+   pizza: Pizza,
+   stroopwafel: Stroopwafel,
+   tea: Tea,
+   toast: Toast,
+   wine: Wine,
+ };
+
+ export const items = [
+   {
+     id: 'apple',
+     imageId: 'apple',
+     title: 'Apple',
+     price: 1,
+   },
+   {
+     id: 'coffee',
+     imageId: 'coffee',
+     title: 'Coffee',
+     price: 2,
+   },
+   {
+     id: 'cookie',
+     imageId: 'cookie',
+     title: 'Cookie',
+     price: 1,
+   },
+   {
+     id: 'hamburger',
+     imageId: 'hamburger',
+     title: 'Hamburger',
+     price: 2.50,
+   },
+   {
+     id: 'ice-cream',
+     imageId: 'ice-cream',
+     title: 'Ice Cream',
+     price: 0.99,
+   },
+   {
+     id: 'pizza',
+     imageId: 'pizza',
+     title: 'Pizza',
+     price: 1,
+   },
+   {
+     id: 'stroopwafel',
+     imageId: 'stroopwafel',
+     title: 'Stroopwafel',
+     price: 0.50,
+   },
+   {
+     id: 'tea',
+     imageId: 'tea',
+     title: 'Tea',
+     price: 1,
+   },
+   {
+     id: 'toast',
+     imageId: 'toast',
```

```
+     title: 'Toast',
+     price: 1,
+   },
+   {
+     id: 'wine',
+     imageId: 'wine',
+     title: 'Wine',
+     price: 10,
+   },
+ ];
```

This file is long, but it provides all the details (including the future price) for each item in the database.

Eventually, you'll replace part of this file with a call to the Coffee Shop's API, but for now, let's focus on React.

Rendering Lists

Now that you have the details saved for the items, you can import them and render them all at once.

src/App.js CHANGED

```
@@ -1,22 +1,20 @@
 import './App.css';
 import Header from './components/Header';
 import Thumbnail from './components/Thumbnail';
-import Apple from './items/apple.svg';
-import Coffee from './items/coffee.svg';
-import Cookie from './items/cookie.svg';
-import Tea from './items/tea.svg';
-import Wine from './items/wine.svg';
+import { items, itemImages } from './items';

 function App() {
   return (
     <div>
       <Header />
       <div>
-        <Thumbnail image={Apple} title="Apple" />
-        <Thumbnail image={Coffee} title="Coffee" />
-        <Thumbnail image={Cookie} title="Cookie" />
-        <Thumbnail image={Tea} title="Tea" />
-        <Thumbnail image={Wine} title="Wine" />
+        {items.map((item) => (
+          <Thumbnail
+            key={item.id}
+            image={itemImages[item.imageId]}
+            title={item.title}
+          />
+        ))}
       </div>
     </div>
   );
```

That was easy. Now all the items display in the browser! (Provided you scroll down.)

You added a new import to import the `items` and `itemImages` previously setup in the details file.

Then, the javascript `map` function to iterate over all the items in the array and return a `Thumbnail` for each one. React knows to use the array as children of the parent div.

You can read more about Array Map on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Inside the map is an arrow function. You can read more about those here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Keys

An advantage of using React is that it tries to update only things that have been added, removed, or changed. One way React does this is by using keys. You can think of keys as a way to uniquely identify a component. They are helpful when using dynamic components, like the list of players in a game. For example, if an application rendered a list of thousands of items, and you wanted to remove a single item, React would use the key attribute to determine which item needs to be updated instead of re-rendering the entire list. One of the most common values used for keys is the id from the data.

The most important thing about keys is that each data item must have a unique key. Each of the items has a unique id, so you used `id` as the key in the code above. In many lists (like employees in a company), there might be two employees with the same name. Using the database key/ID is best if it's available. You could add a random value for the key, but unless you always keep the same random value with each item, it defeats the purpose of the key. If React gets all new keys, it thinks all the list items changed.

Home Component

Generally the `App.js` file only deals with top level data and routing to other components. Let's move the Thumbnails to a new `Home` component.

src/components/Home.js ADDED

```
@@ -0,0 +1,27 @@
+ import PropTypes from 'prop-types';
+ import Thumbnail from './Thumbnail';
+ import { itemImages } from '../items';
+
+ function Home({ items }) {
+   return (
+     <div>
+       {items.map((item) => (
+         <Thumbnail
+           key={item.id}
+           image={itemImages[item.imageId]}
+           title={item.title}
+         />
+       ))}
+     </div>
+   );
+ }
+
+ Home.propTypes = {
+   items: PropTypes.arrayOf(PropTypes.shape({
+     id: PropTypes.string.isRequired,
+     title: PropTypes.string.isRequired,
+     price: PropTypes.number.isRequired,
+
```

```
+   })).isRequired,
+ };
+
+ export default Home;
```

src/App.js CHANGED

```
@@ -1,21 +1,13 @@
  import './App.css';
  import Header from './components/Header';
- import Thumbnail from './components/Thumbnail';
- import { items, itemImages } from './items';
+ import Home from './components/Home';
+ import { items } from './items';

  function App() {
    return (
      <div>
        <Header />
-       <div>
-         {items.map((item) => (
-           <Thumbnail
-             key={item.id}
-             image={itemImages[item.imageId]}
-             title={item.title}
-           />
-         ))}
-       </div>
+       <Home items={items} />
      </div>
    );
  }
```

Most of this code will look familiar, but you will notice some new syntax for defining a prop which is an array of objects. `PropTypes.arrayOf` defines an array of objects. `PropTypes.shape` defines an object and the expected keys that the object should have. The Home component expects an array of objects with an id, title, and price where the id and title are strings and the price is a number and all of these keys are required.

If you edit `items/index` and remove the price for the Apple, you'll see an error in the console warning you of the type mismatch.

The app still renders, but this warning in the console can help avoid hours of debugging because you forgot to pass a required prop or passed a value of the wrong type. If you change the price of an apple to `price: '1'`, you'll see another helpful message: "Warning: Failed prop type: Invalid prop `items[0].price` of type `string` supplied to `Home`, expected `number`."

Conclusion

Now that the coffee shop app is broken down into components. You are ready to make it look prettier with styling.

🔎 Alternate file naming conventions

You are naming your files using `PascalCase`. This is the most common file naming convention you may see. However, React allows you to find a style that is best suited for your project and team. You can also use `camelCase`, `snake_case`, or even `kebab-case`.

🔎 The context variable: `this`

You might see `this` in older, class based React components or in code that doesn't use React. `this` is a special reserved word in JavaScript. It is used to contain contextual data, and varies quite a bit depending on where it appears. This list is more useful to scan than understand in detail.

- In the global execution context, `this` is the global object.
- In functions, `this` depends on how the function is called (e.g. with `fn.apply` vs `fn()`) and if the function is in `strict mode`.
- In `constructor` functions in classes, the value of `this` will be the initial value of the object, unless an object is returned
- In classes, `this` is the value of the object
- In arrow functions, the value of `this` is same as the value of `this` in the containing lexical scope
- In functions on objects, `this` will be the object being called, even in methods in the prototype chain

That list isn't even complete, but the point was to give you a taste of how many different things could be going when using `this`. Given that, `this` must be used with caution. The complexity of `this` has led to many difficult to find bugs. Avoiding `this` as much as possible is the best rule of thumb. `this` does, however, have its uses, particularly in classes, where it behaves more predictably. Even within that context, you would be wise to minimize using `this`.

Finally, even in a non-technical sense, `this` is difficult to talk about. "What's this in this function? Maybe the this here is the same as that this." You get the idea.

Styles

It's time to make the coffee shop look better. To do that, you will add CSS styling. There are a variety of ways to add styles to React. We have chosen one of the simplest ways, importing CSS files.

Background Color

Let's start by getting the Coffee Shop into "dark mode". You'll change the background color of the application.

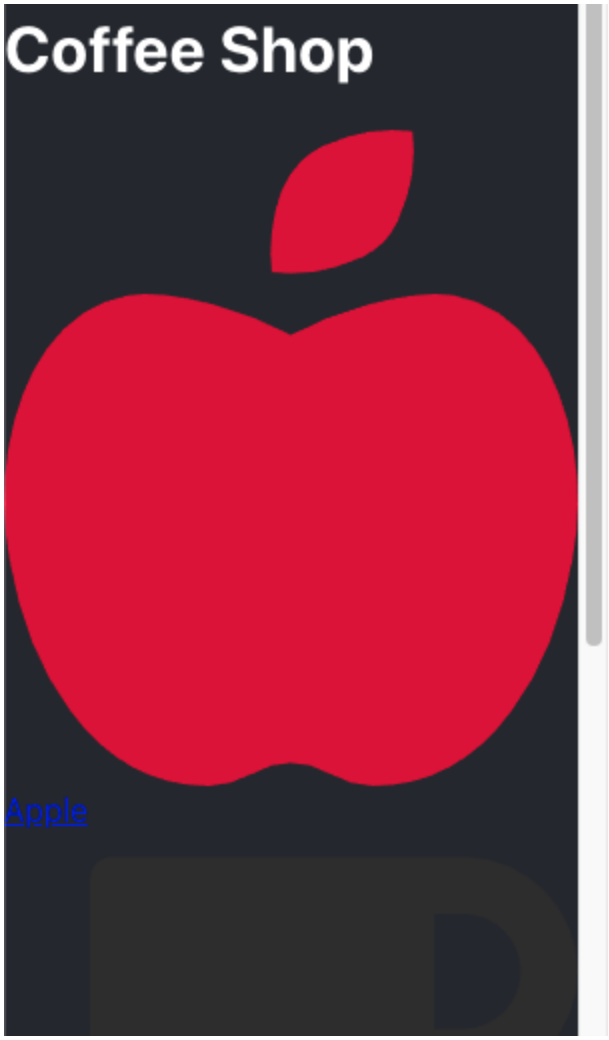
src/App.css CHANGED

```
@@ -1,38 +1,10 @@
- .App {
+ :root {
+   --dark-bg: #080c14;
-   text-align: center;
+   --normal-bg: #282c34;
  }

- .App-logo {
+ body {
-   height: 40vmin;
-   pointer-events: none;
- }
-
- @media (prefers-reduced-motion: no-preference) {
-   .App-logo {
-     animation: App-logo-spin infinite 20s linear;
-   }
- }
-
- .App-header {
-   background-color: #282c34;
+   background-color: var(--normal-bg);
-   min-height: 100vh;
-   display: flex;
-   flex-direction: column;
-   align-items: center;
-   justify-content: center;
-   font-size: calc(10px + 2vmin);
    color: white;
  }

- .App-link {
-   color: #61dafb;
- }
-
- @keyframes App-logo-spin {
-   from {
-     transform: rotate(0deg);
-   }
-   to {
-     transform: rotate(360deg);
-   }
- }
```

All CSS becomes global, so the CSS variables you've created in the CSS above will be available to the entire application.

You can read more on CSS custom properties/variables on MDN https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties .

Check your browser, and you'll see the app has a nice dark background. (If you need to start the application again, you can do that with `npm start`.)



Header

It would be nice if the header was darker and had some padding around it. Let's do that now.

src/components/Header.css ADDED

```
@@ -0,0 +1,10 @@
+ header {
+   padding: 10px 20px;
+   margin-bottom: 10px;
```

```
+   background-color: var(--dark-bg);
+ }
+
+ header h1 {
+   padding: 0;
+   margin: 0;
+ }
```

src/components/Header.js CHANGED

```
@@ -1,3 +1,5 @@
+ import './Header.css';
+
  function Header() {
    return (
      <header>
```

Check the browser. That's better.

Thumbnail

Now, let's make the buttons look prettier.

src/components/Thumbnail.css ADDED

```
@@ -0,0 +1,33 @@
+ .thumbnail-component {
+   background: var(--dark-bg);
+   border-radius: 15px;
+   box-sizing: border-box;
+   display: block;
+   padding: 20px;
+   position: relative;
+   width: 100%;
+   border: none;
+   cursor: pointer;
+ }
+
+ .thumbnail-component:focus {
+   outline: none;
+   border: solid #FFF 1px;
+ }
+
+ .thumbnail-component img {
+   max-width: 150px;
+   width: 100%;
+ }
+
+ .thumbnail-component div.title {
+   position: absolute;
+   bottom: 0;
+   left: 0;
+   right: 0;
+   padding: 5px;
+   background: rgba(255, 255, 255, 0.8);
+   color: #000;
+   border-radius: 0 0 14px 14px;
+   text-align: center;
+ }
```

src/components/Thumbnail.js CHANGED

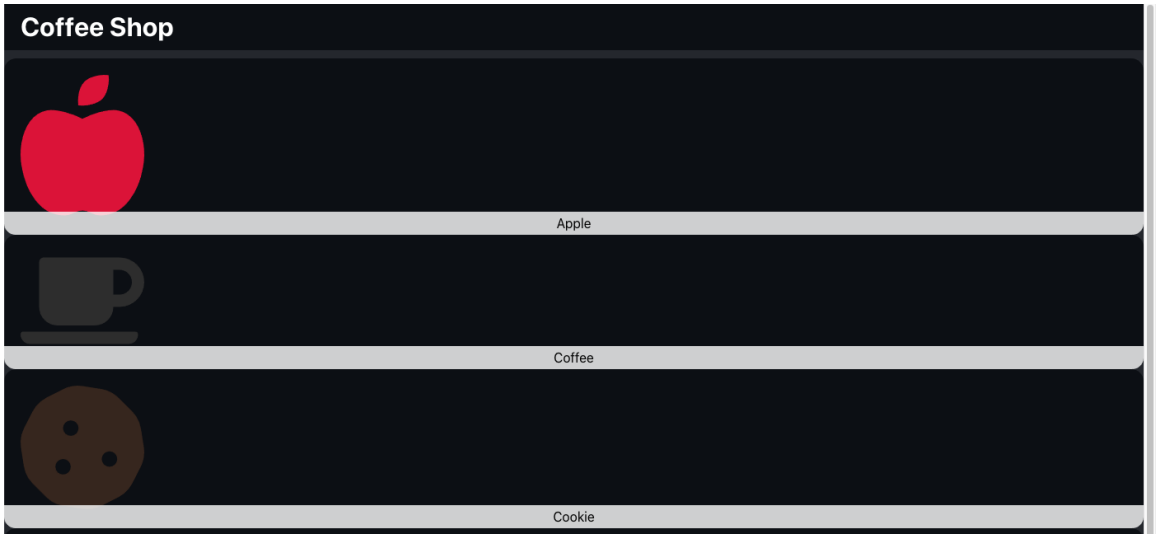
```
@@ -1,12 +1,14 @@
  import PropTypes from 'prop-types';
+ import './Thumbnail.css';

  function Thumbnail({ image, title }) {
    return (
      <a
+       className="thumbnail-component"
        href="#todo"
      >
        <img src={image} alt={title} />
-       <div>{title}</div>
+       <div className="title">{title}</div>
      </a>
    );
  }
```

In this code, you will notice a new attribute `className`. Since `class` is a reserved word in JavaScript. React uses `className` to apply classes to elements. In the rendered page, you will see `class` instead.

You will also notice the CSS applied only to a child of `.thumbnail-component`. This is because all the CSS becomes global even though it's imported into this component. As long as you don't have components with duplicate names, using the components name as the prefix class usually works well. (Note just `.thumbnail` still would have been a pretty generic name that something else could inadvertently use.)

Having some sort of highlight on focus is important, but the default blue outline did not look great, so you changed it to a white border.

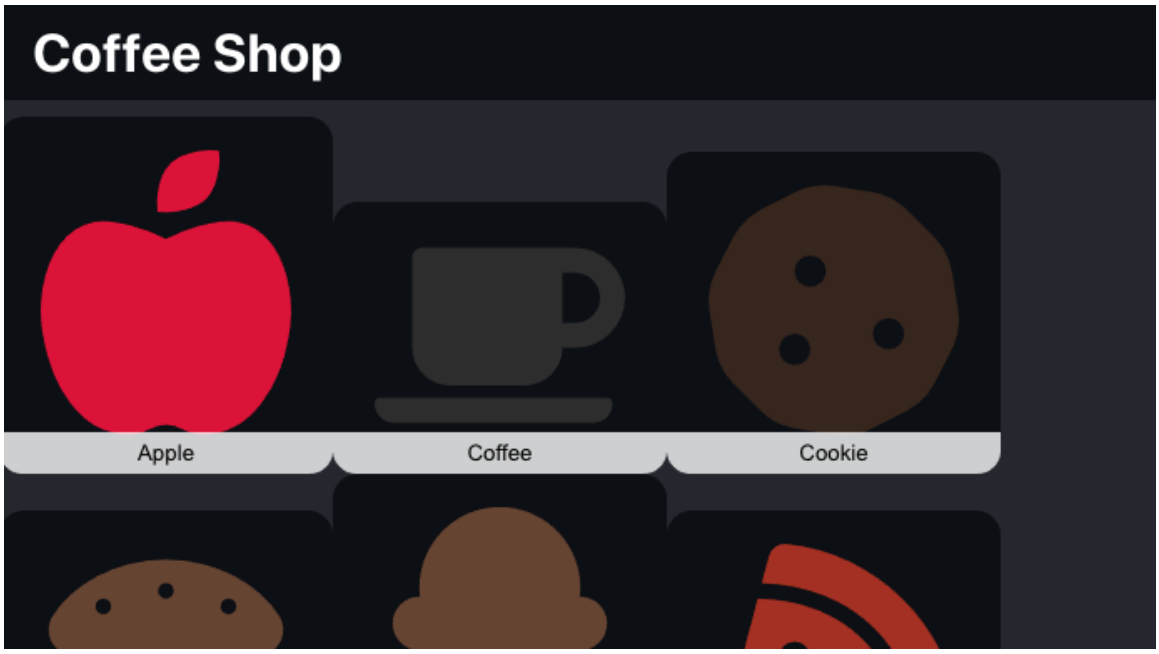
The thumbnails are rather large, but they work.



Adjusting CSS in Devtools

When initially defining CSS, it's often helpful to get instant feedback. You can edit CSS in the devtools in the browser.

Right click one of the thumbnails and select "Inspect Element" in the menu. This will take you to the elements panel in devtools. Make sure you have the `a` element selected. Then, under `.thumbnail-component` click beside one of the styles. This should open a new line where you can add `max-width: 200px;`. Then, on the line with `display: block;`, click `block` and change it to `inline-block`. Now, you have several buttons in a line.



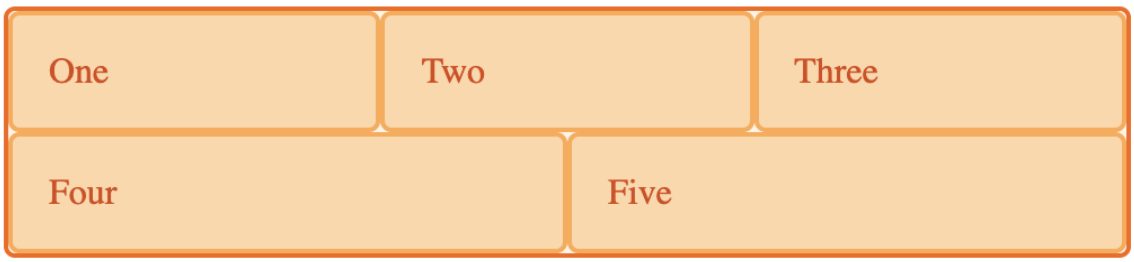
Like changes to props, these go away when you refresh the page. Refresh, and you will be back to the large thumbnails. You are not going to save this change into `Thumbnail.css` because you are going to style the home page with a grid instead to get even height and width thumbnails.

Home

The home page should display the items in nice even cells. The (imaginary) designer wants the site to be usable on smaller devices and requests you show 4 items across on desktops, 3 items on smaller desktops/landscape tablets, 2 across for landscape phones, and 1 across for small portrait phones.

Because the design is primarily focused on the horizontal orientation of the items, you can build this using flexbox or grid layout. Using a table would be difficult because the number of columns on a row cannot be easily changed with CSS in a table.

The main difference between flexbox and grid in this case is what happens to a partial row. In a flexbox the items expand to fill the full row.



In grid layout, the items stay in their cell.

MDN has an article outlining more of the differences: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Relationship_of_Grid_Layout .

For the coffee shop, the designer wants to keep nice clean lines and even cells, so you will use the grid layout.

Using Grid Layout for the Home Page

Now that you've learned about grid layout. You will make a simple 4 column grid for the home page.

src/components/Home.css ADDED

```
@@ -0,0 +1,8 @@
+ div.home-component {
+     display: grid;
+     grid-template-columns: repeat(4, 1fr);
+     grid-gap: 10px;
+     max-width: 800px;
+     margin: 0 auto;
+     padding: 10px;
+   }
```

src/components/Home.js CHANGED

```
@@ -1,10 +1,11 @@
  import PropTypes from 'prop-types';
  import Thumbnail from './Thumbnail';
+ import './Home.css';
  import { itemImages } from '../items';

  function Home({ items }) {
    return (
-     <div>
+     <div className="home-component">
        {items.map((item) => (
          <Thumbnail
            key={item.id}
```

`grid-gap` adds some space between the grid cells such that each Thumbnail takes up a full cell, but there is still space between the Thumbnails.

`max-width` keeps the grid from getting too big on really wide monitors.

`margin: 0 auto` centers the grid on the screen.

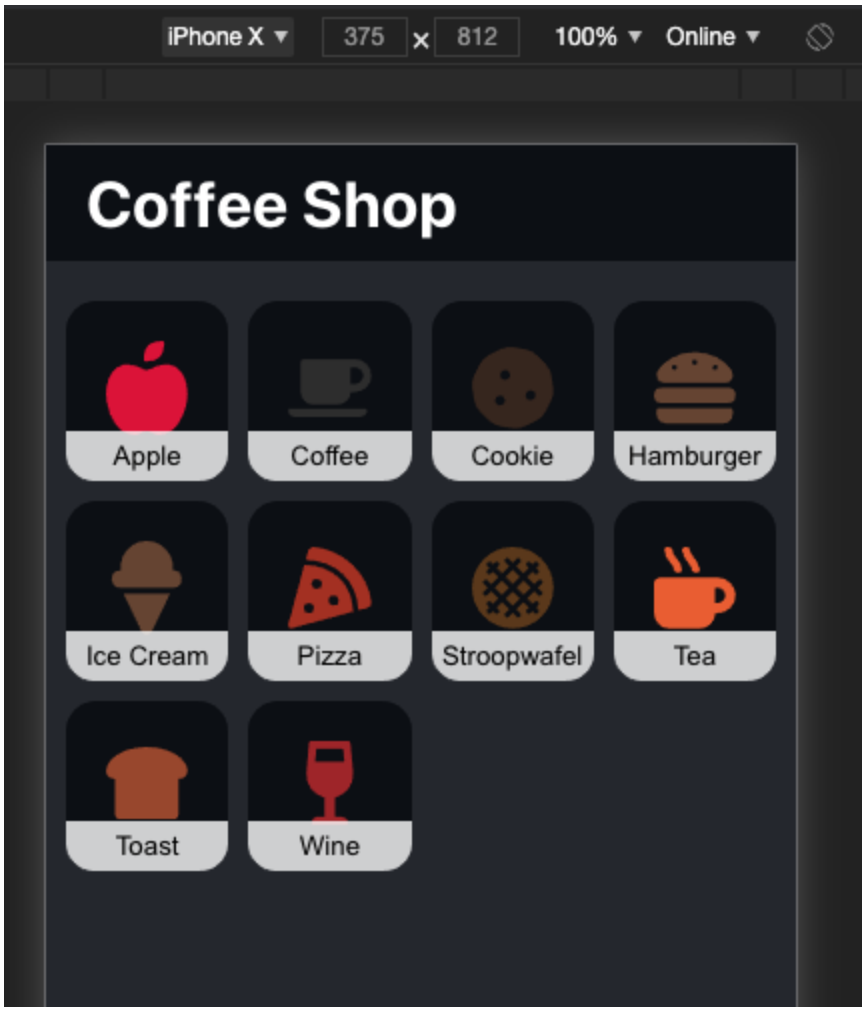
🔎 Grid Layout

If you are not familiar with grid layout, this game is a good introduction to what's possible: https://cssgridgarden.com/

Mobile View

The browser allows you to preview the website in a phone. Open devtools and click the phone/tablet button to toggle device mode.

In the device, dropdown select `iPhone X`.



The thumbnails are viewable, but they are pretty small.

Media Queries

Media queries will make the site responsive as the designer requested above.

Media queries can tell you a lot about the device viewing your application including the device orientation (portrait vs landscape). For the coffee shop, the important part is the width of the screen. This way, if a really wide phone is released, it can use two columns instead of being stuck with one just because it is in portrait orientation. Alternatively, there might be a really narrow tablet that should show fewer items across the page. By defining media queries based on the width, they will focus on the amount of horizontal space available to make each item readable.

src/components/Home.css CHANGED

```
@@ -5,4 +5,22 @@ div.home-component {
5  5      max-width: 800px;
6  6      margin: 0 auto;
7  7      padding: 10px;
   8  +   }
```

```
+
+   @media only screen and (max-width: 600px) {
+     div.home-component {
+       grid-template-columns: repeat(3, 1fr);
+     }
+   }
+
+   @media only screen and (max-width: 450px) {
+     div.home-component {
+       grid-template-columns: repeat(2, 1fr);
+     }
+   }
+
+   @media only screen and (max-width: 300px) {
+     div.home-component {
+       grid-template-columns: repeat(1, 1fr);
+     }
    }
```

Checkout your browser.

The iPhone (and most of the phones in the list) are wide enough for 2 images. Try out landscape view with the rotation button on the far right. The iPhone X gets 4 images across. Try out some other phones. The iPhone SE only fits 3 images in landscape since it's smaller. The Galaxy Fold drops to 1 image in portrait mode.

The media queries will also work outside of device mode if you just drag the browser window to be different widths. (This is handy since desktop users might be open several windows side-by-side.)

CSS Transformations

For desktop users, the designer would like to add a hover effect to the item, which makes the icon a bit larger while the mouse is hovered over it.

Your transformation will increase the size of a thumbnail when you hover over it with the cursor. However, you will not directly change the `width` or `height` styles. You will use the *transform* property, which can alter the shape, size, rotation, and location of an element without interrupting the flow of the elements around it.

The target element for this transition is the `.thumbnail-component img`. You will begin by adding a `transform` declaration directly to the `.thumbnail-component img` element.

After you have tested it and determined that it is working the way you want, you will move the transformation to a new `.thumbnail-component:hover img` declaration block. Finally, you will add a `transition` declaration to `.thumbnail-component img`.

In `styles.css`, begin by adding a `transform` declaration to `.thumbnail-item`.

src/components/Thumbnail.css CHANGED

```
@@ -18,6 +18,7 @@
  .thumbnail-component img {
    max-width: 150px;
    width: 100%;
+   transform: scale(2.2);
  }

  .thumbnail-component div.title {
```

`transform: scale(2.2)` tells the browser that the element should be drawn at 220% of its original size. There are many values that can be used with `transform`, including advanced 3D effects, and MDN https://developer.mozilla.org/en-US/docs/Web/CSS/transform has good explanations for each of them.

Save and view the changes in your browser.

You can see that the thumbnails are now much larger than before. In fact, they are too large. Change the value so that they are only a little larger:

src/components/Thumbnail.css CHANGED

```
@@ -18,7 +18,7 @@
 .thumbnail-component img {
   max-width: 150px;
   width: 100%;
-  transform: scale(2.2);
+  transform: scale(1.2);
 }

 .thumbnail-component div.title {
```

After you save, you should see that the thumbnails are only slightly larger than their original size.

This scale for the thumbnails looks good, so you can move on to the next step.

Scale on Hover

Now it is time to apply the scaling only when the user hovers over the thumbnail.

src/components/Thumbnail.css CHANGED

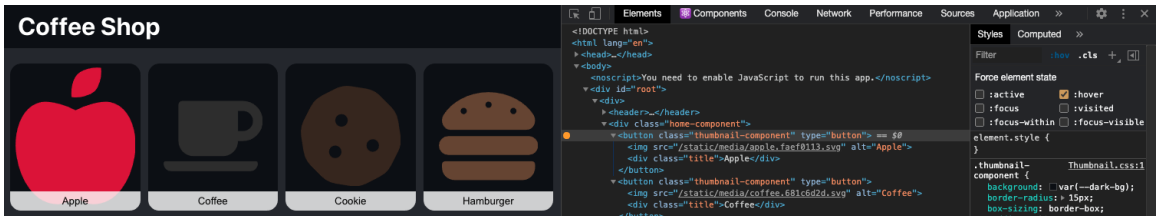
```
@@ -18,6 +18,9 @@
```

```
18  18    .thumbnail-component img {
19  19      max-width: 150px;
20  20      width: 100%;
    21  + }
    22  +
    23  + .thumbnail-component:hover img {
21  24      transform: scale(1.2);
22  25    }
23  26
```

The proper name for this modifier is *pseudo-class*. The psuedo-class `:hover` matches an element when the user holds the mouse cursor over it. There are a number of pseudo-class keywords that describe the various states an element can be in (such as `:focus`). You can search the MDN to learn more.

`:hover` was applied to `.thumbnail-component` rather than `img` so the image scales if the mouse is anywhere in the thumbnail, including the text, rather than only if it is over the image.

DevTools give you a handy way to test pseudo-class states. Go to the elements panel and expand the tags until one of the `<button>` tags is displayed. Click the tag so that it is highlighted. Then, in the styles panel, click the button that says `:hov` to the right of filter. Now check the box next to `:hover`.



An orange circle appears to the left of the `<button>` tag in the elements panel, telling you that one of the pseudo-classes has been activated via the DevTools. The corresponding thumbnail will remain in the `:hover` state, even if you mouse over it and then mouse away from it.

Uncheck the `:hover` checkbox before you continue.

State Changes with CSS Transitions

CSS transitions create a gradual change from one visual state to another, which is just what you need to make the scaling more polished.

When you create a CSS transition, you are telling the browser, "I would like this element's styles to change to these new properties, and I would like for that change to take exactly as long as I tell you."

One common example is the fly-out menu seen on many sites. In a browser with a narrow viewport, clicking the menu icon makes the navigation menu appear from the top – but it does not appear all at once. Instead, it slides down from the header, visually animating from the initial state (hidden) to the end state (visible). Clicking the menu icon again causes the navigation menu to slide back up until it is hidden again.

In general, you should create transitions in three steps:

1. Decide what the end state should be. One good approach is to add the CSS declarations for the end state to the target element. This allows you to see them in the browser and make sure they look the way you intend.
2. Move the declarations from the target element's existing declaration block to a new CSS declaration block. You may want to use a new class for the selector for the new block.
3. Add a `transition` declaration to the target element. The `transition` property tells the browser it will need to visually animate the changes from the current CSS values to the end-state CSS values and that the transition should take place over a specific period of time.

Next, make this change happen as a transition by adding a `transition` declaration to `.thumbnail-component img`. You need to specify the property to animate and how long the animation should take.

src/components/Thumbnail.css CHANGED

```
@@ -18,6 +18,7 @@
  .thumbnail-component img {
    max-width: 150px;
    width: 100%;
+   transition: transform 333ms;
  }

  .thumbnail-component:hover img {
```

You set a `transition` for the `transform` property. This tells the browser that it will need to animate the change, but only for the `transform` property. You also specified that the transition should take place over a period of 333 milliseconds.

Note: you can also transition for other properties like `background-color` or `margin` or `all` to apply to everything.

Save and give your new transition a try. You should see that each thumbnail enlarges when you hover over it. When you move your mouse away, the transition runs in reverse, and the thumbnail returns to its original size.

With the Apple in particular, you can see the nice effect provided by the partially transparent background on the title of the thumbnail. This was set with an `rgba` value.

This produces a nicer effect for the thumbnails.

Learn more about transitions on MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/transition

Using a timing function

Your hover effect is looking good! But it lacks that visual *pop* that would make it really special. With CSS transitions, you can not only specify how much time a transition should take, but also make it transition at different speeds during that time.

There are several timing functions that you can use with transitions. By default, the *ease* timing function is used, which makes the transition faster in the middle than at the start or end. This is a good default though others give you control of the speeds.

Update your transition in `styles.css` so that it uses the `linear` timing function. This will make the rate of the transition slower at the beginning and end and faster in the middle.

src/components/Thumbnail.css CHANGED

```
@@ -18,7 +18,7 @@
.thumbnail-component img {
  max-width: 150px;
  width: 100%;
-   transition: transform 333ms;
+   transition: transform 333ms linear;
}

.thumbnail-component:hover img {
```

Save and hover over one of your thumbnails. The effect is subtle, but noticeable.

There are a number of timing functions available. See the list on the MDN at developer.mozilla.org/en-US/docs/Web/CSS/transition-timing-function .

Your `transition` uses the same duration value and timing function for both the transition *to* the end state and the transition *from* the end state. That does not have to be the case – you can use different values depending on the direction of the transition. If you specify a `transition` property on both the beginning-state declaration and the end-state declaration, the browser uses the value of the declaration it is moving toward.

It might be easier to see this in action. For a quick demonstration, add a `transition` declaration to `.thumbnail-component:hover img` in `styles.css`. (You will delete it after trying it out in the browser.)

src/components/Thumbnail.css CHANGED

```
@@ -23,6 +23,7 @@

.thumbnail-component:hover img {
  transform: scale(1.2);
+   transition: transform 1000ms ease-in;
}

.thumbnail-component div.title {
```

Save and again hover over one of the thumbnails in the browser. The scaling effect will be very slow, taking a full second to complete. This is because it is using the value declared for `.thumbnail-component:hover img`. Now, move your mouse off of the thumbnail. This time, the transition takes 333 milliseconds, the value declared for `.thumbnail-component img`.

Remove the `transition` declaration from `.thumbnail-item:hover` and swap to than `ease` transition function before you continue.

src/components/Thumbnail.css CHANGED

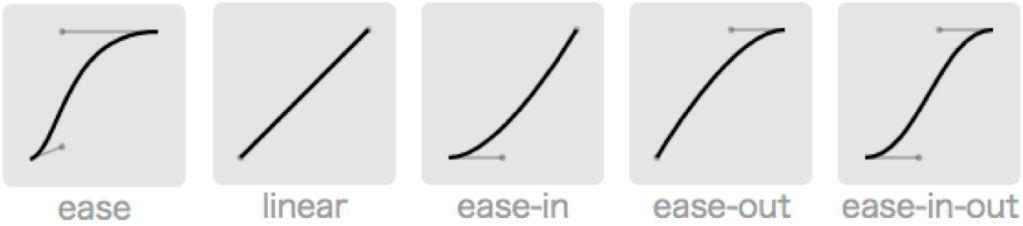
```
@@ -18,12 +18,11 @@
```

```
18  18    .thumbnail-component img {
19  19      max-width: 150px;
20  20      width: 100%;
21      -   transition: transform 333ms linear;
    21  +   transition: transform 333ms ease;
22  22    }
23  23
24  24    .thumbnail-component:hover img {
25  25      transform: scale(1.2);
26      -   transition: transform 1000ms ease-in;
27  26    }
28  27
29  28    .thumbnail-component div.title {
```

Custom Timing Functions

Now, for some icing on the cake, you can create custom timing functions for your transitions instead of limiting yourself to the built-in ones.

Timing functions can be graphed to show the transition's progress over time. Graphs of the built-in timing functions (from the site cubic-bezier.com) are shown below.



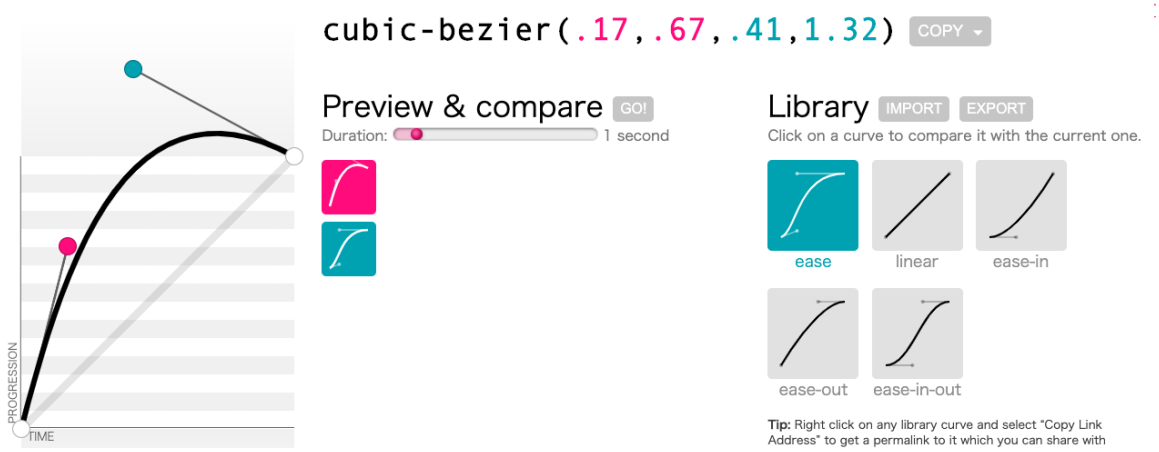
The shapes in these graphs are known as *cubic Bezier curves*. The lines in the graphs describe the behavior of the animation over time. They are defined by four points. You can create custom transitions by specifying the four points that define a curve. Try the following `cubic-bezier` as part of your `transition` declaration.

src/components/Thumbnail.css CHANGED

```
    @@ -18,7 +18,7 @@
18  18    .thumbnail-component img {
19  19      max-width: 150px;
20  20      width: 100%;
21      -   transition: transform 333ms ease;
    21  +   transition: transform 333ms cubic-bezier(.17,.67,.41,1.32);
22  22    }
23  23
24  24    .thumbnail-component:hover img {
```

Save it and hover on some thumbnails in the browser to see the difference in the transition. (It makes the image scale a little too large and then scales it back.)

Thanks to developer Lea Verou and her site cubic-bezier.com , creating custom timing functions is painless.



On the left side is a curve with red and blue drag handles. The curve is a graph of how much of the transition has occurred over the duration. Click and drag the handles to change the curve. As it changes, the decimal values at the top of the page change, too.

On the right side are the built-in timing functions: `ease`, `linear`, `ease-in`, `ease-out`, and `ease-in-out`. Click one of the functions. Then, click the `GO!` button next to **Preview & compare**. The icons representing the two timing functions – the custom `cubic-bezier` and the built-in function – will animate, allowing you to see your custom timing in action and compare it to a built-in option.

Create a custom timing function and, when you are happy with it, copy and paste the values from the website to your code.

```css
transition: transform 333ms cubic-bezier(your values here);
```

Conclusion

The home page is ready to go. Now you're ready to make clicking on an item do something!

🔎 Styled Components

Another way to style is to use Styled Components. This requires an external library but scopes your CSS to the component for you rather than having to do it manually. You can read more about them at https://styled-components.com/ .

🔎 Compiled CSS

Instead of styled components, you can also use a CSS pre-processor like SCSS, SASS, Less, or Stylus to make scoping the CSS to a single component easier. These also provide variables, loops, and functions to generate CSS.

Router

The home page looks good. Now, you are ready to make the thumbnails send users to the detail page. People should also be able to go directly to the detail page, so it will need to have a URL like: `http://localhost:3000/details/apple`.

React Router is a tool to create multiple routes inside a React application.

Install React Router

First, you will need to add a dependency to install React Router.

> **Install**
>
> In a separate terminal from your app, CD into your `coffee-shop` directory.
>
> Then, install the new library with `npm install --save react-router-dom@6.2.1`

Enable React Router

You will make several changes in `App.js` to use React Router.

src/App.js CHANGED

```
@@ -1,3 +1,8 @@
+ import {
+   BrowserRouter as Router,
+   Routes,
+   Route,
+ } from 'react-router-dom';
  import './App.css';
  import Header from './components/Header';
  import Home from './components/Home';
@@ -5,10 +10,12 @@ import { items } from './items';

  function App() {
    return (
-     <div>
+     <Router>
        <Header />
-       <Home items={items} />
+       <Routes>
+         <Route path="/" element={<Home items={items} />} />
-     </div>
+       </Routes>
+     </Router>
    );
  }
```

Visit the app in the browser. You will still see the same thing.

Let's look at the code. First, you'll notice a different part of an import statement `BrowserRouter as Router`. This statement allows you to rename an import. React Router exposes a few different routers that the user can choose between they have the same interface so the code is the same, but it changes how the routes display in the browser. By renaming the chosen router, you can just see Router in your code.

To work properly, the router has to be the root of the React DOM. The very base element in App. Then, the `Routes` component picks which content to display. `Routes` will pick the most specific `Route` to display. It is possible to nest `Route` components inside each other. Then, you can use the `Outlet` component from React Router to choose where in the parent to display the details. However, you will not use `Outlet` as a part of this course. The Route component lists a path and contains what will be displayed on that route so the `/` route contains the `Home` component.

You can learn more about the components of React Router at https://reactrouter.com/docs/en/v6/api .

Not Found

Because React Router uses exact matching by default, visiting http://localhost:3000/unknown returns a blank page. Let's build a page to show when the desired page is not found.

src/components/NotFound.css ADDED

```
@@ -0,0 +1,11 @@
+ .not-found-component {
+     padding: 20px;
+ }
+
+ .not-found-component h2 {
+     margin-top: 0;
+ }
+
+ .not-found-component a {
+     color: #FFF;
+ }
```

src/components/NotFound.js ADDED

```
@@ -0,0 +1,13 @@
+ import { Link } from 'react-router-dom';
+ import './NotFound.css';
+
+ function NotFound() {
+   return (
+     <div className="not-found-component">
+       <h2>Page Not Found</h2>
+       <Link to="/">Return Home</Link>
+     </div>
+   );
+ }
+
+ export default NotFound;
```

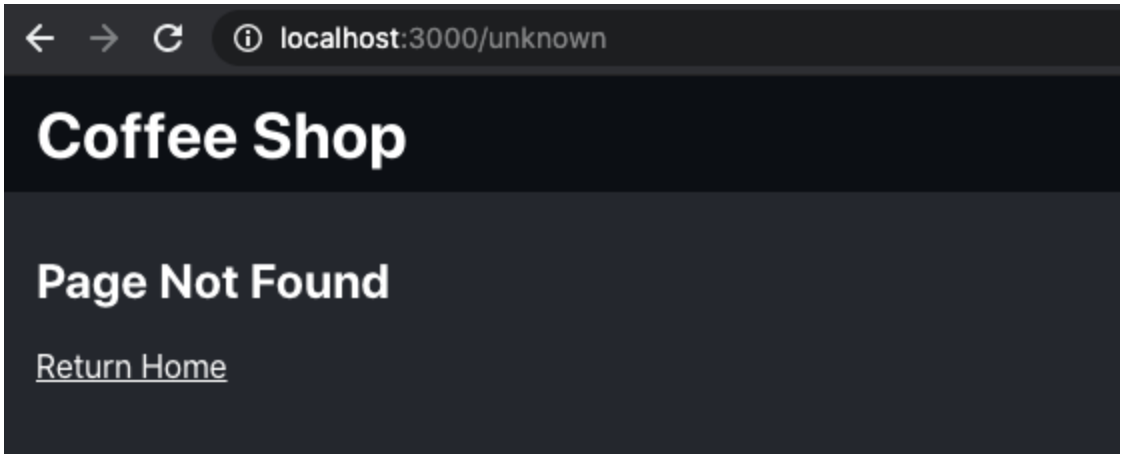
src/App.js CHANGED

```
@@ -6,6 +6,7 @@ import {
  import './App.css';
  import Header from './components/Header';
  import Home from './components/Home';
+ import NotFound from './components/NotFound';
  import { items } from './items';

  function App() {
@@ -14,6 +15,7 @@ function App() {
        <Header />
        <Routes>
          <Route path="/" element={<Home items={items} />} />
+         <Route path="*" element={<NotFound />} />
        </Routes>
      </Router>
    );
```

In `App.js`, the not found component is the last route.

In `NotFound.js`, you will notice a new import `Link`. `Link` is provided by React Router to provide a way to use JavaScript to change what is displayed on the page and the HTML5 History API to change the URL shown in the browser. This way there is no reloading of scripts. You could have used a normal `a` tag, but if you do that and watch the network panel, you will see the browser reloads all the scripts which is not ideal. (Feel free to try this out.)

Visit http://localhost:3000/unknown , you will see the new 404 page.



Click the link to return home, and you are back at the home page. (If you open the network tab while you do this, you will notice only the SVG files load not any JavaScript files because React is handling the routing.)

Details

Let's create a page that shows the details for an item in the coffee shop.

src/components/Details.css ADDED

```
@@ -0,0 +1,13 @@
+ .details-component {
+   padding: 10px;
+ }
```

```
+ .details-component .details-box {
+   padding: 5px 20px;
+ }
+ .details-component h2 {
+   margin: 0 0 10px 0;
+ }
+ .details-component img {
+   max-width: 400px;
+   max-height: 400px;
+ }
```

src/components/Details.js ADDED

```
@@ -0,0 +1,37 @@
+ import { useParams } from 'react-router-dom';
+ import PropTypes from 'prop-types';
+ import { itemImages } from '../items';
+ import './Details.css';
+
+ function Details({ items }) {
+   const { id } = useParams();
+   const detailItem = items.find((item) => item.id === id);
+
+   return (
+     <div className="details-component">
+       <div className="details-box">
+         <h2>{detailItem.title}</h2>
+         <img
+           className="details-image"
+           src={itemImages[detailItem.imageId]}
+           alt={detailItem.title}
+         />
+         <div>
+           Price: $
+           {detailItem.price.toFixed(2)}
+         </div>
+       </div>
+     </div>
+   );
+ }
+
+ Details.propTypes = {
+   items: PropTypes.arrayOf(PropTypes.shape({
+     id: PropTypes.string.isRequired,
+     imageId: PropTypes.string.isRequired,
+     title: PropTypes.string.isRequired,
+     price: PropTypes.number.isRequired,
+   })).isRequired,
+ };
+
+ export default Details;
```

src/App.js CHANGED

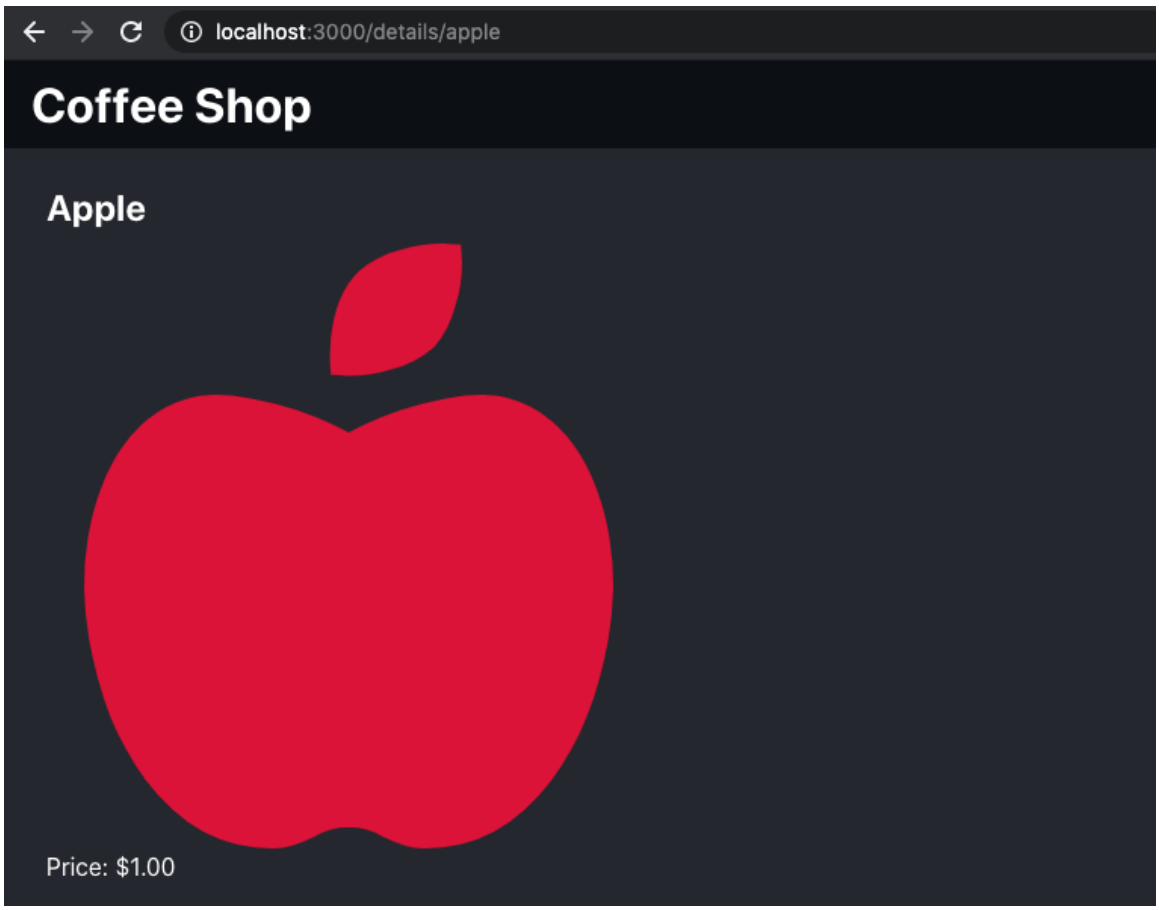
```
@@ -4,6 +4,7 @@ import {
     Route,
   } from 'react-router-dom';
   import './App.css';
+ import Details from './components/Details';
   import Header from './components/Header';
   import Home from './components/Home';
   import NotFound from './components/NotFound';
@@ -14,6 +15,7 @@ function App() {
       <Router>
         <Header />
         <Routes>
+          <Route path="/details/:id" element={<Details items={items} />} />
           <Route path="/" element={<Home items={items} />} />
           <Route path="*" element={<NotFound />} />
         </Routes>
```

You will notice a new part of the route `/details/:id`. `:id` is a router parameter. The colon tells the router it is a parameter and `id` is the name. In the detail component, you can read the value of the parameter using `const { id } = useParams();`.

Then, `items.find` finds the particular item in the `items` array to use to display the details. If you'd like to know more about how `Array.find` works, check out the page on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find.

Finally, `.toFixed` is a great method for display numbers. It takes a number and returns a string with a fixed number of decimal places. `detailItem.price.toFixed(2)` always has two decimal places. Direct conversion of the number `7.50` to a string results in `7.5` (without the last zero). Alternatively, because of round off error, sometimes a number like `7.50` ends up displaying as `7.4999999999998`. Using `.toFixed` fixes both problems by rounding and making sure the result has exactly 2 decimal places. The return value is a string because numbers cannot store an exact number of decimal places and are subject to internal round off errors.

Great! Now, you can visit http://localhost:3000/details/apple and see the details.



🔎 React Router `useParams` Hook

`useParams` is the first React Hook in the course, but it is provided by React Router rather than React. React Hooks were introduced in React 16.8 and provide a way to get to features without the additional boilerplate and complexity of writing a class based component. You will use several React Hooks in later chapters, and we will discuss them in more detail at that time.

You can read more about this specific hook at: https://reactrouter.com/docs/en/v6/api#useparams

Unknown Items

What happens if you visit http://localhost:3000/details/unknown ? Oops! The app crashed `TypeError: Cannot read property 'title' of undefined`.

`items.find` returns null when there is not a matching item. Let's update the page to show a message instead of crashing.

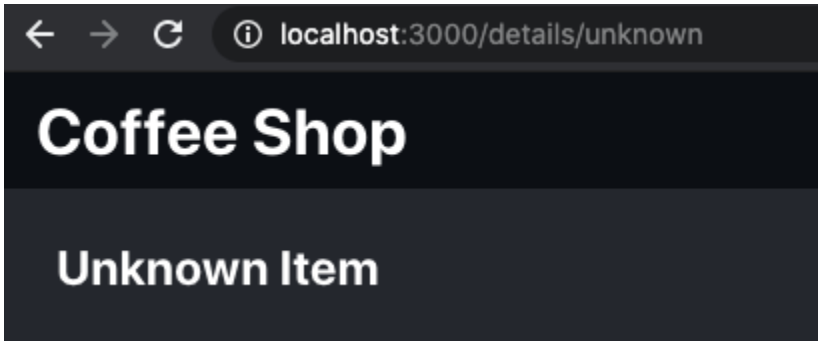
> **Tip**
>
> In the diff below, a chunk of `Details.js` is tabbed in further.
>
> You can do this easily in VS Code by highlighting the lines which need to be indented and pressing `Tab` three times. (If you tab in too far, you can press `Shift + Tab` to un-indent.)
>
> Alternatively, you can add the 3 new lines before and after the existing code without adjusting the tabs and run ESLint autofix which will correct the tabs for you.

src/components/Details.js CHANGED

```
@@ -10,16 +10,22 @@ function Details({ items }) {
    return (
      <div className="details-component">
        <div className="details-box">
+         { detailItem
+           ? (
+             <>
-           <h2>{detailItem.title}</h2>
+             <h2>{detailItem.title}</h2>
-           <img
+             <img
-             className="details-image"
+               className="details-image"
-             src={itemImages[detailItem.imageId]}
+               src={itemImages[detailItem.imageId]}
-             alt={detailItem.title}
+               alt={detailItem.title}
-           />
+             />
-           <div>
+             <div>
-             Price: $
+               Price: $
-             {detailItem.price.toFixed(2)}
+               {detailItem.price.toFixed(2)}
-           </div>
+             </div>
+             </>
+           )
+           : (<h2>Unknown Item</h2>)}
```

```
23  29          </div>
24  30        </div>
25  31      );
```

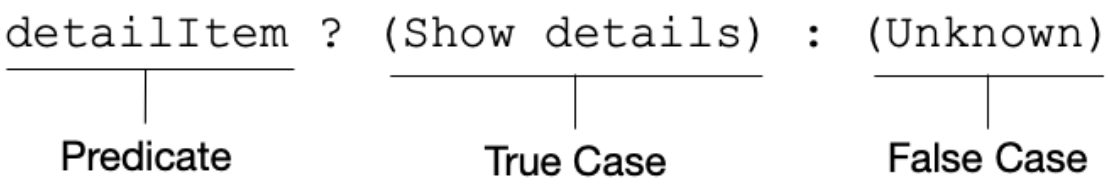
Visit http://localhost:3000/details/unknown again.



Yay! You have a details page.

🔎 Ternary

In React, you can use ternaries to decide what to render. In a ternary, there is a condition that evaluates to a truthy or falsy value, then a question mark, next the result if the condition is truthy, then a colon, and finally the result if the condition is falsy.



🔎 React Fragments

You will notice another new symbol in the code `<>`. This is called a React Fragment. React allows each component to return only one root element. Sometimes it will make sense to have more than one root element, but you don't want to add a top level element in your component's JSX which would add an unnecessary element to the DOM. Fragments allow you to group several elements in the JSX, but React will render them at the root level of that component. You can read more about them here: https://reactjs.org/docs/fragments.html .

Getting Back Home

From the details page, it would be nice if there was an easy way to get back home. On many websites clicking the title or logo in the header sends you back to the home page. Let's edit the header to add a link to the home page. (See if you can do this without the sample code.)

src/components/Header.css CHANGED

```
@@ -7,4 +7,9 @@ header {
 header h1 {
   padding: 0;
   margin: 0;
+ }
+
+ header h1 a {
+   color: #FFF;
+   text-decoration: none;
 }
```

src/components/Header.js CHANGED

```
@@ -1,9 +1,10 @@
+ import { Link } from 'react-router-dom';
 import './Header.css';

 function Header() {
   return (
     <header>
-      <h1>Coffee Shop</h1>
+      <h1><Link to="/">Coffee Shop</Link></h1>
     </header>
   );
 }
```

Here is React Router Link again. This time with a little added style, so the title is not underlined and does not change color.

Thumbnails to Details

The last thing you need to do is to link the thumbnails on the home page to the details page.

src/components/Home.js CHANGED

```
@@ -9,6 +9,7 @@ function Home({ items }) {
         {items.map((item) => (
           <Thumbnail
             key={item.id}
+            id={item.id}
             image={itemImages[item.imageId]}
             title={item.title}
           />
```

src/components/Thumbnail.js CHANGED

```
@@ -1,19 +1,21 @@
 import PropTypes from 'prop-types';
+ import { Link } from 'react-router-dom';
 import './Thumbnail.css';

- function Thumbnail({ image, title }) {
+ function Thumbnail({ id, image, title }) {
   return (
-    <a
+    <Link
       className="thumbnail-component"
-      href="#todo"
+      to={`/details/${id}`}
     >
       <img src={image} alt={title} />
       <div className="title">{title}</div>
-    </a>
+    </Link>
```

```
13  14      );
14  15    }
15  16
16  17    Thumbnail.propTypes = {
    18  +   id: PropTypes.string.isRequired,
17  19      image: PropTypes.string.isRequired,
18  20      title: PropTypes.string.isRequired,
19  21    };
```

Check out your home page and click one of the items. You go to the detail page. Click the header title "Coffee Shop" to go back home and try another item. Yay!

🔎 Template Literals

What is `` `/details/${id}` `` ? The backticks indicate a template literal. They are a standard feature of JavaScript introduced in ES2015. Template literals can contain expressions which are inside `${ }` . This template literal concatenates the id onto the end of the url. It is the same result as `'/details/' + id` .

Template literals can also span multiple lines. They can also be tagged by a processing function. Tagging is seldom used, but it looks like this: `` processorName`This ${variable} is in a tagged template literal.` `` and is valid JavaScript.

You can read more about template literals on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals .

Conclusion

Nice! Now, you have routes for each detail item. In the next chapter, you will enhance the details page with flexbox.

Flexbox

The designer wants you to format the details page to have a list of all the other items in a sidebar on the left side of the page and the main detail image to the right of that. On mobile, the other items should show below the main detail item. Because the design requires the other items to come before the main item on desktop and after the main item on mobile, historically, you would have needed to have two copies of the thumbnails in the source and hide one with media queries. With flexbox, items can be reordered, so you no longer need two copies of the HTML.

Add The Thumbnails

Let's add the sidebar with thumbnails. The sidebar will be a fixed 200px wide and the details of the selected item will take up the rest of the page.

src/components/Details.css CHANGED

```
@@ -1,7 +1,21 @@
 .details-component {
+  display: flex;
+  flex-direction: row;
   padding: 10px;
+  height: calc(100vh - 100px);
+ }
+ .details-sidebar {
+  flex: 0 0 200px;
+  height: 100%;
+  overflow: auto;
+  padding: 5px;
+  box-sizing: border-box;
+ }
+ .details-sidebar > * {
+  margin: 10px 0;
 }
 .details-component .details-box {
+  flex: 1 1 auto;
   padding: 5px 20px;
 }
 .details-component h2 {
```

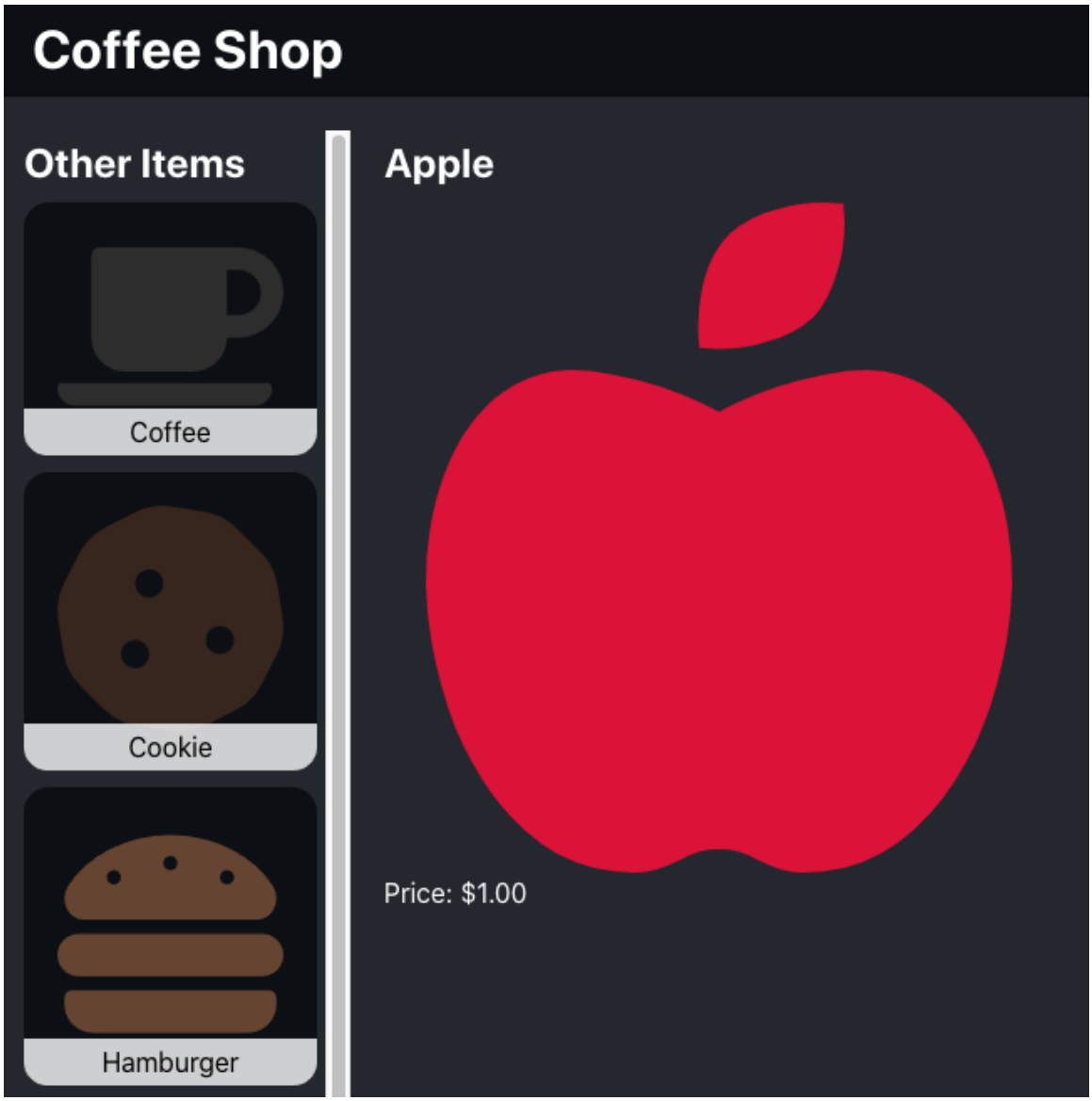
src/components/Details.js CHANGED

```
@@ -2,13 +2,26 @@ import { useParams } from 'react-router-dom';
 import PropTypes from 'prop-types';
 import { itemImages } from '../items';
 import './Details.css';
+ import Thumbnail from './Thumbnail';

 function Details({ items }) {
   const { id } = useParams();
   const detailItem = items.find((item) => item.id === id);
+  const otherItems = items.filter((item) => item.id !== id);

   return (
     <div className="details-component">
+      <div className="details-sidebar">
```

```
15 +         <h2>Other Items</h2>
16 +         {otherItems.map((item) => (
17 +           <Thumbnail
18 +             id={item.id}
19 +             image={itemImages[item.imageId]}
20 +             title={item.title}
21 +             key={item.id}
22 +           />
23 +         ))}
24 +       </div>
12 25        <div className="details-box">
13 26          { detailItem
14 27            ? (
```

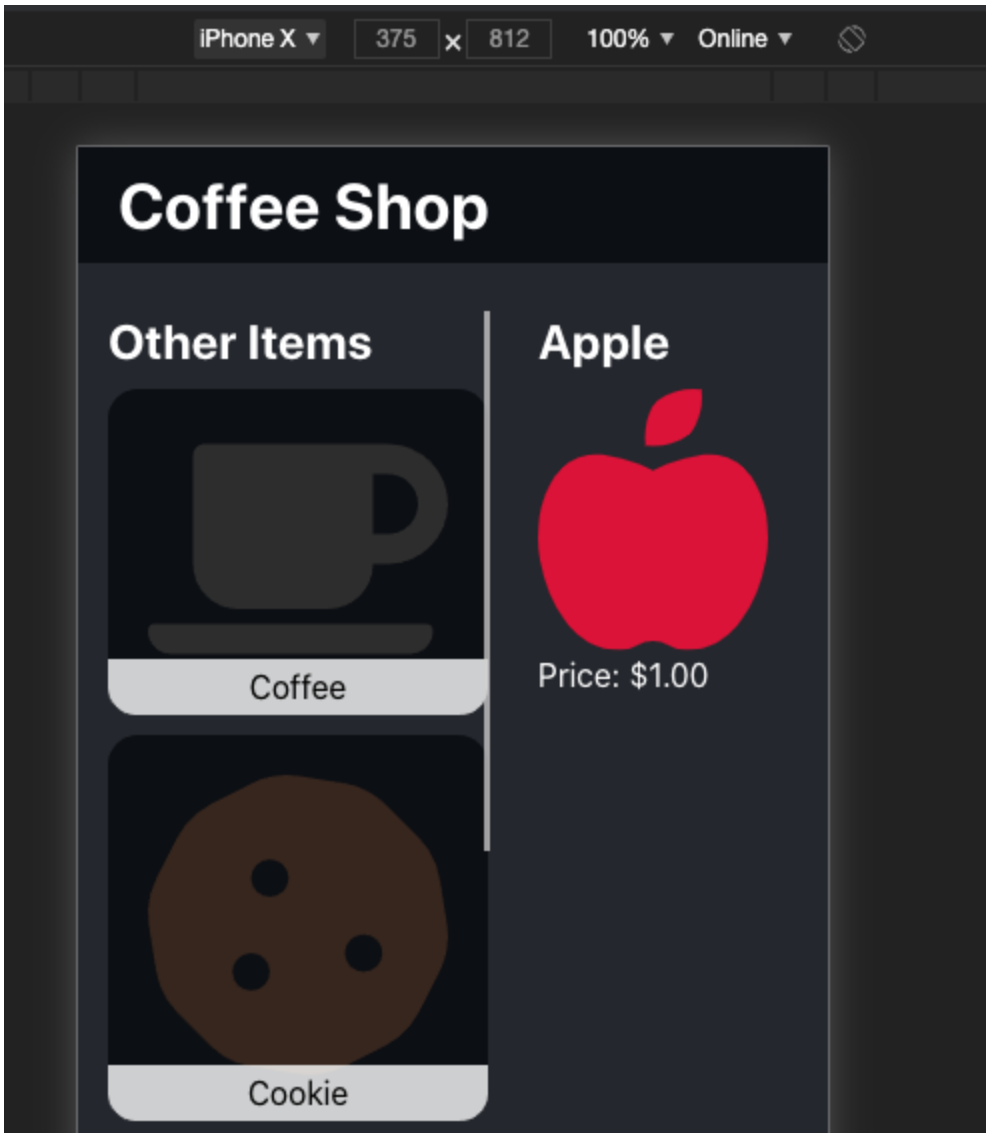
CSS `calc` forces `details-component` to be the height of the browser while also leaving space for the page header. If you wanted the header to never scroll away for the entire site, you could use a flexbox as the root of the page with a fixed height for the header and allow the body to consume the rest of the page.

Nice, now you have a sidebar.



Mobile View

If you switch over to the device mode, the website looks pretty ugly right now.



You can fix that with media queries.

src/components/Details.css CHANGED

```
@@ -25,3 +25,14 @@
    max-width: 400px;
    max-height: 400px;
  }
+ @media only screen and (max-width: 600px) {
+   .details-component {
+     flex-direction: column;
+     height: auto;
+   }
+   .details-component .details-sidebar {
+     flex: 1 0 auto;
+     height: auto;
+     overflow: visible;
+   }
+ }
```

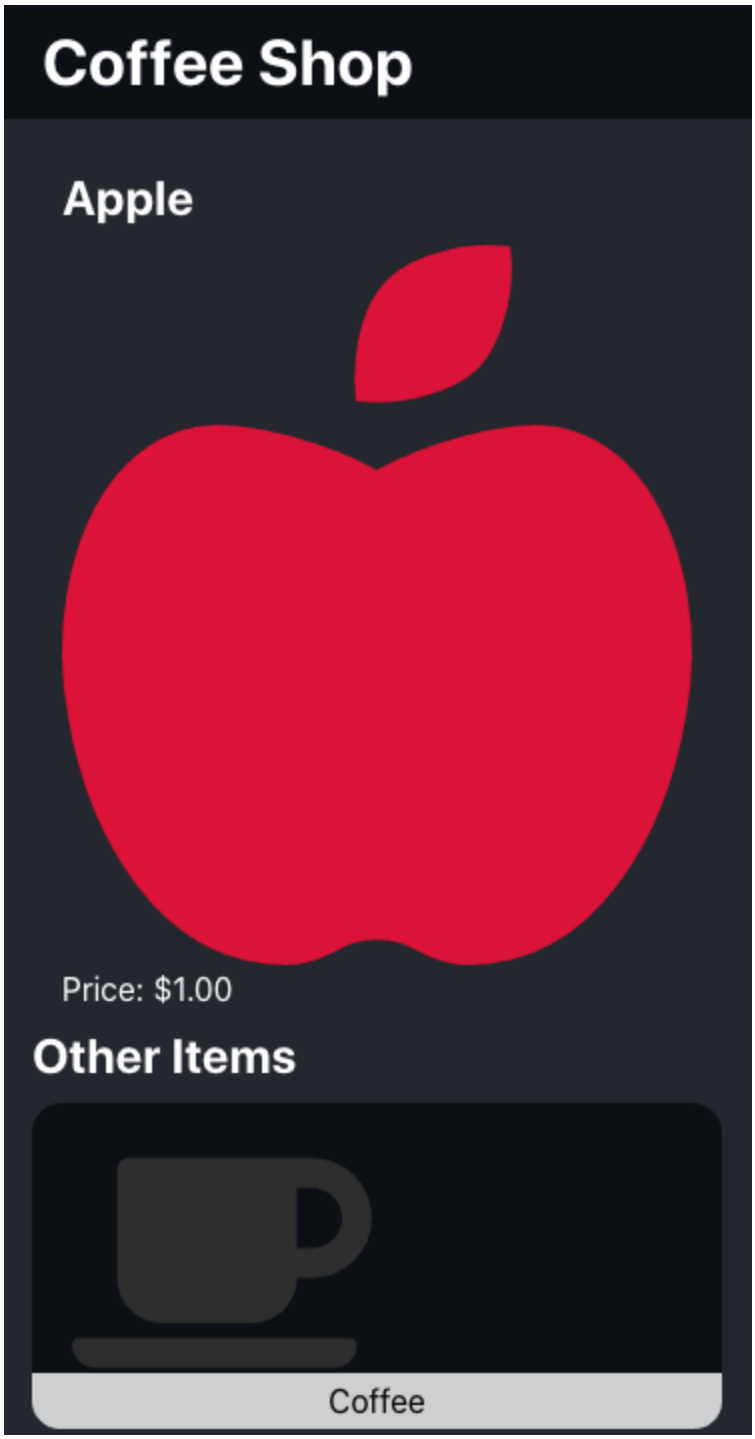
The code resets the sidebar to the full height and swapped the flex direction to column so they show one above the other on mobile.

Correct Order

Unfortunately, now all the other items are above the detail item. You can fix that with the `order` property.

src/components/Details.css CHANGED

```
@@ -32,6 +32,7 @@
  }
  .details-component .details-sidebar {
    flex: 1 0 auto;
+   order: 1;
    height: auto;
    overflow: visible;
  }
```



Flexbox to Fix the Thumbnails

In the thumbnails, the coffee cup is over to the left side. On the home page, it's also at the top of the box instead of in the middle. Using flexbox, you can fix both of those.

src/components/Thumbnail.css CHANGED

```
@@ -2,7 +2,9 @@
    background: var(--dark-bg);
    border-radius: 15px;
    box-sizing: border-box;
-   display: block;
+   display: flex;
+   justify-content: center;
+   align-items: center;
    padding: 20px;
    position: relative;
    width: 100%;
```

Check back in the browser, the coffee cup is centered both on the home page and in the other items section.

🔎 What is flexbox?

If you're not familiar with flexbox, here are two games to introduce you to it:

- https://flexboxfroggy.com/
- http://www.flexboxdefense.com/

Conclusion

The detail page looks good on desktop and mobile now. You're ready to continue on to build out a cart and allow users to add items to their cart.

Add to Cart

Now, you are ready to build out the cart.

Reducer

For the cart, you will use React's `useReducer` hook. This hook helps handle more complicated situations like a cart where you need to add and remove single items and empty the entire cart where each change to the cart needs to happen in sequence and update the full cart object. Then, the next change can take that cart object and update it again.

Understanding React's reducers will also help you if you use Redux later because redux is based on reducers.

🔎 Hooks

`useReducer` is the first native React hook you are using. Hooks were released in react 16.8.0 and allow functional components the ability to use state, reducers, and other react features without the complexity associated with a class based component.

You can read more on hooks in general at https://reactjs.org/docs/hooks-intro.html . You will be using several more of React's built-in hooks in later chapters.

Cart Reducer

Now, let's see a reducer in action.

src/reducers/cartReducer.js ADDED

```
@@ -0,0 +1,32 @@
+ import { useReducer } from 'react';
+
+ const initialCart = [];
+
+ export const CartTypes = {
+   ADD: 'ADD',
+ };
+
+ const findItem = (cart, itemId) => cart.find((item) => item.id === itemId);
+
+ const cartReducer = (state, action) => {
+   switch (action.type) {
+     case CartTypes.ADD:
+       if (findItem(state, action.itemId)) {
+         return state.map((item) => {
+           if (item.id === action.itemId) {
```

```
+              return { ...item, quantity: item.quantity + 1 };
+            }
+            return item;
+          });
+        }
+
+        return [
+          ...state,
+          { id: action.itemId, quantity: 1 },
+        ];
+      default:
+        throw new Error(`Invalid action type ${action.type}`);
+    }
+ };
+
+ export const useCartReducer = () => useReducer(cartReducer, initialCart);
```

The reducer is in a separate file to help keep the code clean and provide a way to import the various action types from the const.

The `useReducer` function takes a reducer function (`cartReducer`) and the initial state (`initialCart`) and returns the current state (`cart`) and a dispatch function. The `dispatch` function calls the reducer to update the state.

The `dispatch` function takes an action which will be passed to the reducer function. Right now, the reducer supports only one action type `ADD` . In the `ADD` case, the reducer checks to see if the item is already in the cart. If the item is in the cart, it increases the quantity by one. Otherwise, it adds the item to the cart with a quantity of 1.

`useCartReducer` creates a simple custom hook that creates a cart reducer.

🔎 Spread Syntax

The `...` is spread syntax from Javascript. You can learn more about it on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax .

🔎 Array.reduce

You're also using a reducer from Javascript. You can learn more about it on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Display Cart Quantity in Header

Let's create the cart and show the quantity in the header.

> **Copy Files**
>
> Copy the `images` folder which contains `cart.svg` from your resources zip to `src/images` .

src/App.js CHANGED

```
@@ -9,11 +9,15 @@ import Header from './components/Header';
 import Home from './components/Home';
 import NotFound from './components/NotFound';
 import { items } from './items';
+import { useCartReducer } from './reducers/cartReducer';

 function App() {
+  // eslint-disable-next-line no-unused-vars
+  const [cart, dispatch] = useCartReducer();
+
   return (
     <Router>
-      <Header />
+      <Header cart={cart} />
       <Routes>
         <Route path="/details/:id" element={<Details items={items} />} />
         <Route path="/" element={<Home items={items} />} />
```

src/components/Header.css CHANGED

```
@@ -12,4 +12,29 @@ header h1 a {
   color: #FFF;
   text-decoration: none;
 }
+
+header .cart {
+  display: inline-block;
+  position: relative;
+  width: 50px;
+}
+
+header .cart img {
+  width: 100%;
+}
+
+header .cart .badge {
+  min-width: 16px;
+  height: 16px;
+  padding: 3px;
+  border-radius: 10px;
+  font-size: 14px;
+  background-color: #FF0000;
+  color: #FFF;
+  font-weight: bold;
+  position: absolute;
+  bottom: -5px;
+  right: 0;
+  text-align: center;
+}
```

src/components/Header.js CHANGED

```
@@ -1,12 +1,29 @@
 import { Link } from 'react-router-dom';
+import PropTypes from 'prop-types';
+import CartIcon from '../images/cart.svg';
 import './Header.css';

-function Header() {
+function Header({ cart }) {
+  const cartQuantity = cart.reduce((acc, item) => acc + item.quantity, 0);
   return (
     <header>
       <h1><Link to="/">Coffee Shop</Link></h1>
+      <a
+        className="cart"
+        href="#todo"
```

```
+       >
+         <img src={CartIcon} alt="Cart" />
+         <div className="badge">{cartQuantity}</div>
+       </a>
      </header>
    );
  }

+ Header.propTypes = {
+   cart: PropTypes.arrayOf(PropTypes.shape({
+     id: PropTypes.string.isRequired,
+     quantity: PropTypes.number.isRequired,
+   })).isRequired,
+ };
+
  export default Header;
```

If called from multiple components, the `useCartReducer` hook would create multiple separate carts, so you have to create the cart from `App.js` and pass it to the children that way there is just one cart in the application.

Nice, now you have a cart icon.

🔎 Array Destructuring

`const [cart, dispatch]` is using array destructuring to pull items out of specific array indexes. This is very similar to the object destructuring you used earlier in the book.

In fact, they are both covered on the same MDN page: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Header Alignment

It would be nice if the header was all on one line where the right was on the far right. There are a number of ways to accomplish this, but flexbox is a great choice for allowing the title to expand to the full available space and fixing the cart icon to 50px on the right.

src/components/Header.css CHANGED

```
@@ -1,10 +1,12 @@
  header {
+   display: flex;
    padding: 10px 20px;
    margin-bottom: 10px;
    background-color: var(--dark-bg);
  }

  header h1 {
```

```
         9  +   flex: 1 1 auto;
  8     10      padding: 0;
  9     11      margin: 0;
 10     12    }
              @@ -15,9 +17,9 @@ header h1 a {
 15     17    }
 16     18
 17     19    header .cart {
         20  +   flex: 0 0 50px;
 18     21      display: inline-block;
 19     22      position: relative;
 20        -   width: 50px;
 21     23    }
 22     24
 23     25    header .cart img {
```

That's much better.

Add to Cart

Now, let's create a button, so the user can add items to the cart.

src/App.js CHANGED

```
              @@ -9,17 +9,20 @@ import Header from './components/Header';
  9      9    import Home from './components/Home';
 10     10    import NotFound from './components/NotFound';
 11     11    import { items } from './items';
 12        - import { useCartReducer } from './reducers/cartReducer';
         12 + import { CartTypes, useCartReducer } from './reducers/cartReducer';
 13     13
 14     14    function App() {
 15        -   // eslint-disable-next-line no-unused-vars
 16     15      const [cart, dispatch] = useCartReducer();
         16 +   const addToCart = (itemId) => dispatch({ type: CartTypes.ADD, itemId });
 17     17
 18     18      return (
 19     19        <Router>
 20     20          <Header cart={cart} />
 21     21          <Routes>
 22        -         <Route path="/details/:id" element={<Details items={items} />} />
         22 +         <Route
         23 +           path="/details/:id"
         24 +           element={<Details addToCart={addToCart} items={items} />}
         25 +         />
 23     26            <Route path="/" element={<Home items={items} />} />
 24     27            <Route path="*" element={<NotFound />} />
 25     28          </Routes>
```

src/components/Details.js CHANGED

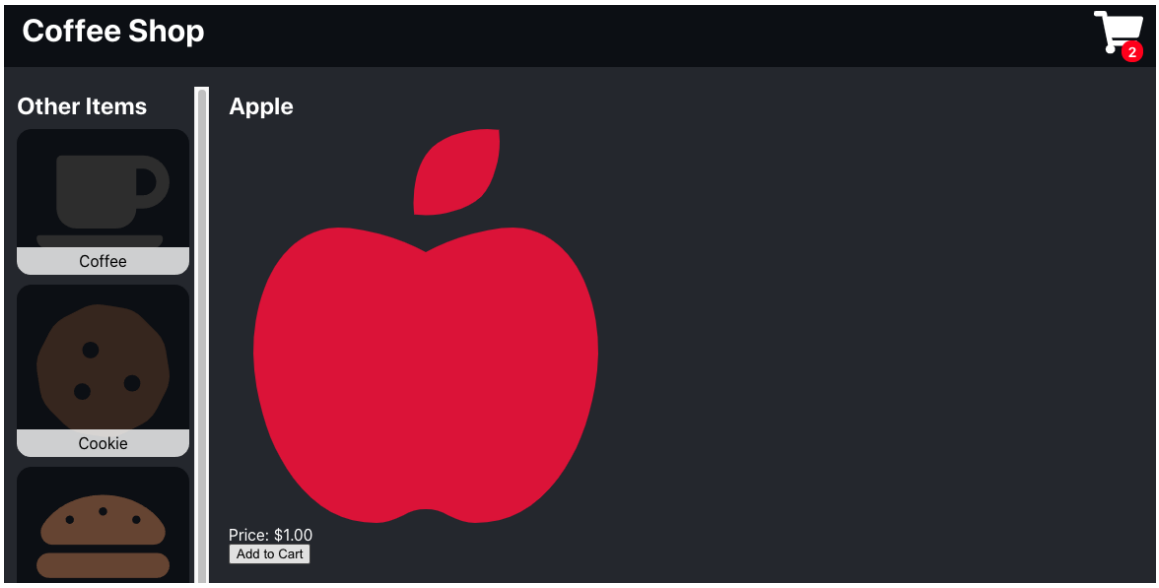
```
              @@ -4,7 +4,7 @@ import { itemImages } from '../items';
  4      4    import './Details.css';
  5      5    import Thumbnail from './Thumbnail';
  6      6
  7        - function Details({ items }) {
          7 + function Details({ addToCart, items }) {
  8      8      const { id } = useParams();
  9      9      const detailItem = items.find((item) => item.id === id);
 10     10      const otherItems = items.filter((item) => item.id !== id);
              @@ -36,6 +36,14 @@ function Details({ items }) {
 36     36                    Price: $
 37     37                    {detailItem.price.toFixed(2)}
```

```
38  38                </div>
    39  +             <div>
    40  +               <button
    41  +                 type="button"
    42  +                 onClick={() => addToCart(detailItem.id)}
    43  +               >
    44  +                 Add to Cart
    45  +               </button>
    46  +             </div>
39  47              </>
40  48            )
41  49            : (<h2>Unknown Item</h2>)}
    @@ -45,6 +53,7 @@ function Details({ items }) {
45  53    }
46  54
47  55    Details.propTypes = {
    56  +   addToCart: PropTypes.func.isRequired,
48  57      items: PropTypes.arrayOf(PropTypes.shape({
49  58        id: PropTypes.string.isRequired,
50  59        imageId: PropTypes.string.isRequired,
```

To make it easier on the Details page, there is a helper function that adds items to the cart based on the `itemId`. This function calls the dispatch function from the reducer with the type `ADD` and passes the item ID.

In the details page, you created a button with an `onClick` property. This attribute is how you instruct React to listen for click events. When the user clicks the button, it calls the `addToCart` function with the ID of the item. This in turn updates the cart to have the item in it.

Head over to your browser and visit the detail page for an item. Click add to cart. You will see the cart quantity update to one. Click add to cart again, the quantity is now 2.



🔎 Click Events `onClick`

`onClick` is how you tell React to listen to click events. Any click automatically calls that function. The click passes the `event` as a parameter, but since this function did not need the event, it is not in

the signature. One of the most common uses for the event is to call `event.preventDefault()` to prevent a click on a link from navigating to the new page.

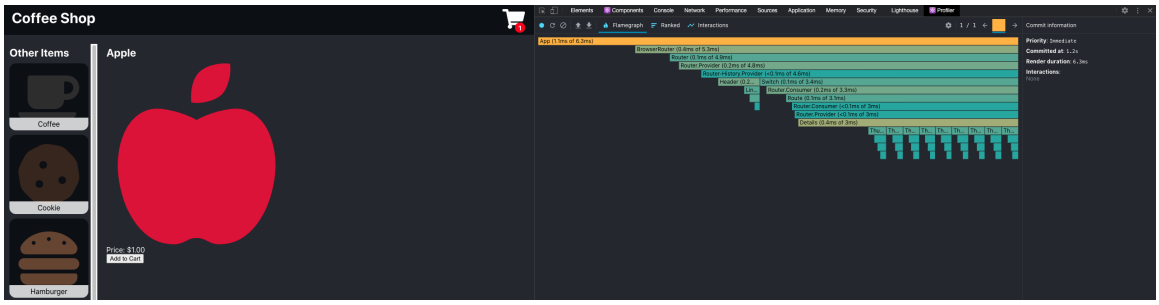
Prevent Changes

You are almost done, but there is one small problem. Unnecessary re-renders.

These are hard to see from the front end, but you can see them with the React Profiler tools.

1. Navigate to the detail page for an Apple
2. Open the React Profiler tab in devtools
3. Click the Flamegraph tab within the React Profiler.
4. Click the record button on the Profiler tab
5. Click the Add to Cart button under the Apple
6. Click the record button on the Profiler tab again to stop recording

You should see something like this:



Towards the bottom of the list, you can see that the details component and all 9 thumbnails re-rendered. Currently, this is expected because functional components always re-render. This component is not terribly expensive at 6.3ms, but it does render all the thumbnails under it so it would be nice if React did not re-render it.

Memo

React provides a `memo` helper function for functional components, so they can behave more like class components and only re-render if state changes. (Note: This is not the same as the `useMemo` hook.)

Let's import and use that.

src/components/Details.js CHANGED

```
@@ -1,4 +1,5 @@
  import { useParams } from 'react-router-dom';
+ import { memo } from 'react';
  import PropTypes from 'prop-types';
  import { itemImages } from '../items';
  import './Details.css';
```

```
@@ -52,6 +53,8 @@ function Details({ addToCart, items }) {
    );
  }

+ const DetailsMemo = memo(Details);
+
  Details.propTypes = {
    addToCart: PropTypes.func.isRequired,
    items: PropTypes.arrayOf(PropTypes.shape({
@@ -62,4 +65,4 @@ Details.propTypes = {
    })).isRequired,
  };

- export default Details;
+ export default DetailsMemo;
```

We keep the Details function to be sure we keep the name of the component. Then, Details is wrapped with memo to memoize the result. The default export is now `DetailsMemo`. Therefore, `import Details from './components/Details';` is now getting the version with the memo.

Unfortunately, if you refresh the page and re-run the experiment above you will still see that the component is re-rendering.

useParams "Bug"

The `useParams` hook is causing a re-render. The reason why is outside the scope of the course. Let's extract a wrapper component to help out with that.

src/components/Details.js CHANGED

```
@@ -5,8 +5,7 @@ import { itemImages } from '../items';
  import './Details.css';
  import Thumbnail from './Thumbnail';

- function Details({ addToCart, items }) {
-   const { id } = useParams();
+ function Details({ addToCart, id, items }) {
    const detailItem = items.find((item) => item.id === id);
    const otherItems = items.filter((item) => item.id !== id);

@@ -55,7 +54,7 @@ function Details({ addToCart, items }) {

  const DetailsMemo = memo(Details);

- Details.propTypes = {
+ const sharedProps = {
    addToCart: PropTypes.func.isRequired,
    items: PropTypes.arrayOf(PropTypes.shape({
      id: PropTypes.string.isRequired,
@@ -65,4 +64,22 @@ Details.propTypes = {
    })).isRequired,
  };

- export default DetailsMemo;
+ Details.propTypes = {
+   ...sharedProps,
+   id: PropTypes.string.isRequired,
+ };
+
+ function DetailsOuter({ addToCart, items }) {
+   const { id } = useParams();
+   return (
+     <DetailsMemo
+       addToCart={addToCart}
```

```
+       id={id}
+       items={items}
+     />
+   );
+ }
+
+ DetailsOuter.propTypes = sharedProps;
+
+ export default DetailsOuter;
```

Try the experiment again, and it still re-renders the `Details` component. Why?

useCallback Reference Equality

React checks reference equality of the props. This means when you pass in a function like `addToCart` React doesn't try to look inside the function it just looks at the reference. The main `App` component re-renders because the state of the cart changes. A part of that render loop creates the `addToCart` function which means every time `App` renders a new `addToCart` function is created which causes `Details` to re-render.

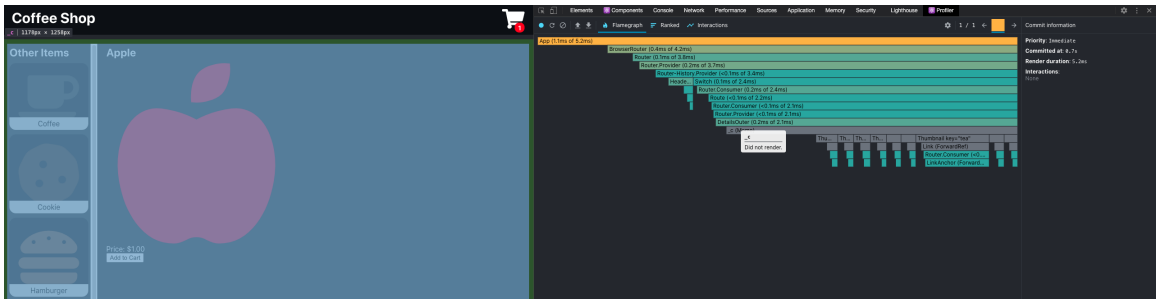
A property of the `useReducer` hook is that it always returns the same (reference equality) `dispatch` function to help prevent re-renders. React provides a hook called `useCallback` so helper functions can also have the same property of reference equality.

src/App.js CHANGED

```
@@ -3,6 +3,7 @@ import {
    Routes,
    Route,
  } from 'react-router-dom';
+ import { useCallback } from 'react';
  import './App.css';
  import Details from './components/Details';
  import Header from './components/Header';
@@ -13,7 +14,10 @@ import { CartTypes, useCartReducer } from './reducers/cartReducer';

  function App() {
    const [cart, dispatch] = useCartReducer();
-   const addToCart = (itemId) => dispatch({ type: CartTypes.ADD, itemId });
+   const addToCart = useCallback(
+     (itemId) => dispatch({ type: CartTypes.ADD, itemId }),
+     [dispatch],
+   );

    return (
      <Router>
```

Here `useCallback` creates the `addToCart` function. `useCallback` takes as the first parameter the function that the callback should return. The second parameter is the dependencies that should cause the callback to be updated. In this case, the only dependency is the `dispatch` function. As mentioned above, React guarantees the dispatch function will not change as a result `addToCart` will only be created once and will not change.

Try the experiment one last time.



At first, you might think it still did not work, but the gray bars mean that the component did not re-render. The `Link` component in the Thumbnails re-renders, but the details page and the main part of the thumbnails did not re-render which was the goal.

Similar to `useParams`, the reason why `Link` re-renders is outside the scope of this course.

Conclusion

In general, the Details component is not complicated enough to warrant this much effort for optimization, but it is the most complicated component in the app so it makes a good demo of how to optimize components and use the React Profiler tools.

Now, you are ready to build out a page to view and edit the cart.

Edit Cart

Now that you can add items to the cart, you are ready to view and edit the cart.

This chapter will expand the coffee shop by reusing concepts you already know. Try to do all the exercises without looking at the solutions first.

Create the Cart Page

First, you need to create a page that displays all the items currently in the cart. The route for this page should be `/cart`.

> **Practice**
>
> Without looking at the code below, can you setup a new cart component with the tile "Your Cart" and the route "/cart"?

src/components/Cart.css ADDED

```
@@ -0,0 +1,7 @@
+ .cart-component {
+   padding: 10px;
+ }
+
+ .cart-component h2 {
+   margin-top: 0;
+ }
```

src/components/Cart.js ADDED

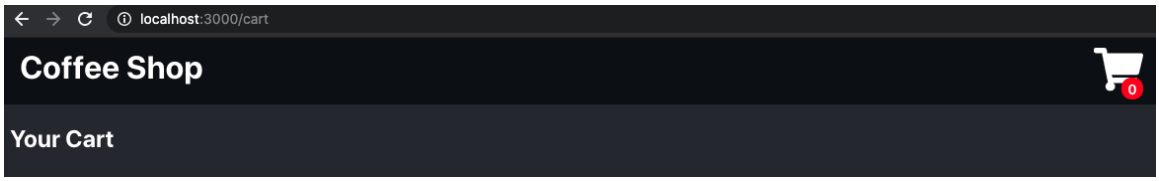
```
@@ -0,0 +1,11 @@
+ import './Cart.css';
+
+ function Cart() {
+   return (
+     <div className="cart-component">
+       <h2>Your Cart</h2>
+     </div>
+   );
+ }
+
+ export default Cart;
```

src/App.js CHANGED

```
@@ -5,6 +5,7 @@ import {
  } from 'react-router-dom';
  import { useCallback } from 'react';
  import './App.css';
+ import Cart from './components/Cart';
  import Details from './components/Details';
  import Header from './components/Header';
  import Home from './components/Home';
```

```
@@ -23,6 +24,9 @@ function App() {
      <Router>
        <Header cart={cart} />
        <Switch>
+         <Route path="/cart">
+           <Cart />
+         </Route>
          <Route path="/details/:id">
            <Details addToCart={addToCart} items={items} />
          </Route>
```

You probably did not use the same styles as us and might not have made styles at all. That's totally fine! Visit http://localhost:3000/cart , you should see your new cart page.



Link to the Cart Page

Update the cart icon in the `Header` so it Links to the new Cart page.

> **Practice**
>
> Try to do this on your own.

src/components/Header.js CHANGED

```
@@ -8,13 +8,13 @@ function Header({ cart }) {
    return (
      <header>
        <h1><Link to="/">Coffee Shop</Link></h1>
-       <a
+       <Link
          className="cart"
-         href="#todo"
+         to="/cart"
        >
          <img src={CartIcon} alt="Cart" />
          <div className="badge">{cartQuantity}</div>
-       </a>
+       </Link>
      </header>
    );
  }
```

Try out your cart button in the header. It should link to the cart. (Note: You will not yet see the contents of the cart.)

Pass the Cart Contents

To display the cart, you will need to get the cart contents from the `App` component to the cart.

Note: You will have to disable the `no-unused-vars` lint rule (or just ignore the error) because you are not using the cart contents yet. You will use them in the next section.

> **Practice**
>
> You can do it!

src/App.js CHANGED

```
@@ -25,7 +25,7 @@ function App() {
25  25          <Header cart={cart} />
26  26          <Switch>
27  27            <Route path="/cart">
28      -           <Cart />
    28  +           <Cart cart={cart} />
29  29            </Route>
30  30            <Route path="/details/:id">
31  31              <Details addToCart={addToCart} items={items} />
```

src/components/Cart.js CHANGED

```
@@ -1,6 +1,8 @@
    1  + import PropTypes from 'prop-types';
1   2    import './Cart.css';
2   3
3       - function Cart() {
    4  + // eslint-disable-next-line no-unused-vars
    5  + function Cart({ cart }) {
4   6      return (
5   7        <div className="cart-component">
6   8          <h2>Your Cart</h2>
@@ -8,4 +10,11 @@ function Cart() {
8   10     );
9   11   }
10  12
    13 + Cart.propTypes = {
    14 +   cart: PropTypes.arrayOf(PropTypes.shape({
    15 +     id: PropTypes.string.isRequired,
    16 +     quantity: PropTypes.number.isRequired,
    17 +   })).isRequired,
    18 + };
    19 +
11  20   export default Cart;
```

Nice!

Display the Cart Contents

You have the cart contents now it is time to display them on the page.

Use a table to show the quantity in the first column and the id of the item in the second column. (You can refer to the prop types to verify the shape of the cart object.)

> **Practice**
>
> I believe in you.

src/components/Cart.js CHANGED

```
@@ -1,11 +1,26 @@
 import PropTypes from 'prop-types';
 import './Cart.css';

- // eslint-disable-next-line no-unused-vars
 function Cart({ cart }) {
   return (
     <div className="cart-component">
       <h2>Your Cart</h2>
+      <table>
+        <thead>
+          <tr>
+            <th>Quantity</th>
+            <th>Item</th>
+          </tr>
+        </thead>
+        <tbody>
+          {cart.map((item) => (
+            <tr key={item.id}>
+              <td>{item.quantity}</td>
+              <td>{item.id}</td>
+            </tr>
+          ))}
+        </tbody>
+      </table>
     </div>
   );
 }
```

Add some items to the cart. Then, click the cart icon to see the cart contents. (Note: You must be careful not to refresh the page otherwise your cart contents will be erased. Cart contents are stored as a part of state which only exists in this part of the page. It would be possible to use browser storage, so the cart could persist across refreshes, but that's not part of this app.)

Display Item Title and Price

The items have nicer human readable titles, but only the IDs are stored in the cart object.

Pass the items to the cart and use the `find` method to help you get the title of the item.

Also, add a third column which shows the item's price.

> **Practice**
>
> Practice, Practice, Practice

src/App.js CHANGED

```
@@ -25,7 +25,7 @@ function App() {
25 25         <Header cart={cart} />
26 26         <Switch>
27 27           <Route path="/cart">
28    -           <Cart cart={cart} />
   28 +           <Cart cart={cart} items={items} />
29 29           </Route>
30 30           <Route path="/details/:id">
31 31             <Details addToCart={addToCart} items={items} />
```

src/components/Cart.js CHANGED

```
@@ -1,7 +1,7 @@
1  1  import PropTypes from 'prop-types';
2  2  import './Cart.css';
3  3
4     - function Cart({ cart }) {
   4  + function Cart({ cart, items }) {
5  5    return (
6  6      <div className="cart-component">
7  7        <h2>Your Cart</h2>
@@ -10,13 +10,20 @@ function Cart({ cart }) {
10 10             <tr>
11 11               <th>Quantity</th>
12 12               <th>Item</th>
   13 +             <th>Price</th>
13 14             </tr>
14 15           </thead>
15 16           <tbody>
16 17             {cart.map((item) => (
17 18               <tr key={item.id}>
18 19                 <td>{item.quantity}</td>
19    -                <td>{item.id}</td>
   20 +                <td>{items.find((i) => i.id === item.id).title}</td>
   21 +                <td>
   22 +                  {`$${(
   23 +                    item.quantity
   24 +                    * items.find((i) => i.id === item.id).price
   25 +                  ).toFixed(2)}`}
   26 +                </td>
20 27               </tr>
21 28             ))}
22 29           </tbody>
@@ -30,6 +37,11 @@ Cart.propTypes = {
30 37       id: PropTypes.string.isRequired,
31 38       quantity: PropTypes.number.isRequired,
32 39     })).isRequired,
   40 +   items: PropTypes.arrayOf(PropTypes.shape({
   41 +     id: PropTypes.string.isRequired,
   42 +     title: PropTypes.string.isRequired,
   43 +     price: PropTypes.number.isRequired,
   44 +   })).isRequired,
33 45   };
34 46
35 47   export default Cart;
```

You do not need to have used as many lines as we did for the price. (We wanted to avoid horizontal scrolling/odd line breaks in the materials.)

Remove Reducer

The user might want to remove an item in their cart.

Update the cart reducer to support a new `REMOVE` action type which removes an item from the cart.

> **Practice**
>
> Improving every day.

src/reducers/cartReducer.js CHANGED

```
@@ -4,6 +4,7 @@ const initialCart = [];

 export const CartTypes = {
   ADD: 'ADD',
+  REMOVE: 'REMOVE',
 };

 const findItem = (cart, itemId) => cart.find((item) => item.id === itemId);
@@ -24,6 +25,8 @@ const cartReducer = (state, action) => {
         ...state,
         { id: action.itemId, quantity: 1 },
       ];
+    case CartTypes.REMOVE:
+      return state.filter((item) => item.id !== action.itemId);
     default:
       throw new Error(`Invalid action type ${action.type}`);
   }
```

Using the `filter` function on an array accomplishes this in a single line, but there are many ways to accomplish this task.

Remove Button

Add a fourth column to the table for the cart which contains a button to remove that item from the cart.

For this component, pass the `dispatch` function as a prop instead of making helpers. While either way is valid, since the cart component will need to use several actions, we choose to pass the dispatch function in the solution code.

> **Practice**
>
> Try, try, again

src/App.js CHANGED

```
@@ -25,7 +25,7 @@ function App() {
         <Header cart={cart} />
         <Switch>
           <Route path="/cart">
-            <Cart cart={cart} items={items} />
+            <Cart cart={cart} dispatch={dispatch} items={items} />
           </Route>
           <Route path="/details/:id">
             <Details addToCart={addToCart} items={items} />
```

src/components/Cart.js CHANGED

```
@@ -1,7 +1,8 @@
 import PropTypes from 'prop-types';
+import { CartTypes } from '../reducers/cartReducer';
```

```
2  3   import './Cart.css';
3  4
4      - function Cart({ cart, items }) {
       5 + function Cart({ cart, dispatch, items }) {
5  6     return (
6  7       <div className="cart-component">
7  8         <h2>Your Cart</h2>
@@ -11,6 +12,7 @@ function Cart({ cart, items }) {
11 12              <th>Quantity</th>
12 13              <th>Item</th>
13 14              <th>Price</th>
   15 +            <th>Remove</th>
14 16            </tr>
15 17          </thead>
16 18          <tbody>
@@ -24,6 +26,17 @@ function Cart({ cart, items }) {
24 26                  * items.find((i) => i.id === item.id).price
25 27                ).toFixed(2)}`}
26 28              </td>
   29 +            <td>
   30 +              <button
   31 +                type="button"
   32 +                onClick={() => dispatch({
   33 +                  type: CartTypes.REMOVE,
   34 +                  itemId: item.id,
   35 +                })}
   36 +              >
   37 +                Remove
   38 +              </button>
   39 +            </td>
27 40            </tr>
28 41          ))}
29 42        </tbody>
@@ -37,6 +50,7 @@ Cart.propTypes = {
37 50      id: PropTypes.string.isRequired,
38 51      quantity: PropTypes.number.isRequired,
39 52    })).isRequired,
   53 +  dispatch: PropTypes.func.isRequired,
40 54    items: PropTypes.arrayOf(PropTypes.shape({
41 55      id: PropTypes.string.isRequired,
42 56      title: PropTypes.string.isRequired,
```

Add some items to your cart. Then, go use the remove button to remove them.

Empty Cart

After you removed all the items, you ended up with just an empty table. It would be nice if the user saw a message that said, "Your cart is empty." instead of a blank page.

Do you remember how to conditionally change what displays in a component?

> **Practice**
>
> Conditioning is good practice!

src/components/Cart.js CHANGED

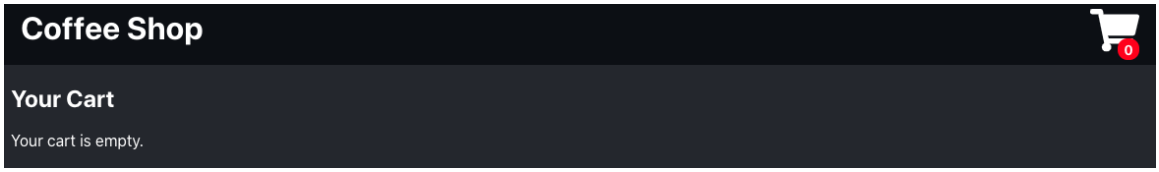
```
@@ -6,41 +6,45 @@ function Cart({ cart, dispatch, items }) {
6  6     return (
7  7       <div className="cart-component">
8  8         <h2>Your Cart</h2>
   9 +       {(cart.length === 0
   10 +         ? <div>Your cart is empty.</div>
```

```
+         : (
-       <table>
+           <table>
-         <thead>
+             <thead>
-           <tr>
+               <tr>
-             <th>Quantity</th>
+                 <th>Quantity</th>
-             <th>Item</th>
+                 <th>Item</th>
-             <th>Price</th>
+                 <th>Price</th>
-             <th>Remove</th>
+                 <th>Remove</th>
-           </tr>
+             </tr>
-         </thead>
+             </thead>
-         <tbody>
+             <tbody>
-           {cart.map((item) => (
+               {cart.map((item) => (
-             <tr key={item.id}>
+                 <tr key={item.id}>
-               <td>{item.quantity}</td>
+                   <td>{item.quantity}</td>
-               <td>{items.find((i) => i.id === item.id).title}</td>
+                   <td>{items.find((i) => i.id === item.id).title}</td>
-               <td>
+                   <td>
-                 {`$${(
+                     {`$${(
-                   item.quantity
+                     item.quantity
                     * items.find((i) => i.id === item.id).price
-                 ).toFixed(2)}`}
+                     ).toFixed(2)}`}
-               </td>
+                   </td>
-               <td>
+                   <td>
-                 <button
+                     <button
-                   type="button"
+                     type="button"
-                   onClick={() => dispatch({
+                     onClick={() => dispatch({
-                     type: CartTypes.REMOVE,
+                     type: CartTypes.REMOVE,
-                     itemId: item.id,
+                     itemId: item.id,
-                   })}
+                     })}
-                 >
+                     >
-                   Remove
+                     Remove
-                 </button>
+                     </button>
-               </td>
+                   </td>
-             </tr>
+                 </tr>
-           ))}
+               ))}
-         </tbody>
+             </tbody>
-       </table>
+           </table>
+         ))}
       </div>
```

```
45  49      );
46  50    }
```

Whew, that's a lot of lines, but almost all of it is an additional tab that eslint required as code style.

Visit your cart and remove all the items. You should see your new message.



Sub-Total

You will add a tax computation in the next chapter, but let's go ahead and compute the users subtotal and display it below the table if the user has items in their cart.

> **Practice**
>
> Hm, thought provoking.

src/components/Cart.js CHANGED

```
@@ -3,47 +3,58 @@ import { CartTypes } from '../reducers/cartReducer';
3   3   import './Cart.css';
4   4
5   5   function Cart({ cart, dispatch, items }) {
6   +     const subTotal = cart.reduce((acc, item) => {
7   +       const details = items.find((i) => i.id === item.id);
8   +       return item.quantity * details.price + acc;
9   +     }, 0);
10  +
6   11    return (
7   12      <div className="cart-component">
8   13        <h2>Your Cart</h2>
9   14        {(cart.length === 0
10  15          ? <div>Your cart is empty.</div>
11  16          : (
12      -         <table>
13      -           <thead>
14      -             <tr>
15      -               <th>Quantity</th>
16      -               <th>Item</th>
17      -               <th>Price</th>
18      -               <th>Remove</th>
19      -             </tr>
20      -           </thead>
21      -           <tbody>
22      -             {cart.map((item) => (
23      -               <tr key={item.id}>
24      -                 <td>{item.quantity}</td>
25      -                 <td>{items.find((i) => i.id === item.id).title}</td>
26      -                 <td>
27      -                   {`$${(
28      -                     item.quantity
29      -                   * items.find((i) => i.id === item.id).price
30      -                   ).toFixed(2)}`}
31      -                 </td>
32      -                 <td>
```

```
33  -                     <button
34  -                       type="button"
35  -                       onClick={() => dispatch({
36  -                         type: CartTypes.REMOVE,
37  -                         itemId: item.id,
38  -                       })}
39  -                     >
40  -                       Remove
41  -                     </button>
42  -                   </td>
    17  +           <>
    18  +             <table>
    19  +               <thead>
    20  +                 <tr>
    21  +                   <th>Quantity</th>
    22  +                   <th>Item</th>
    23  +                   <th>Price</th>
    24  +                   <th>Remove</th>
43  25                    </tr>
44  -                   ))}
45  -                 </tbody>
46  -               </table>
    26  +               </thead>
    27  +               <tbody>
    28  +                 {cart.map((item) => (
    29  +                   <tr key={item.id}>
    30  +                     <td>{item.quantity}</td>
    31  +                     <td>{items.find((i) => i.id === item.id).title}</td>
    32  +                     <td>
    33  +                       {`$${(
    34  +                         item.quantity
    35  +                     * items.find((i) => i.id === item.id).price
    36  +                         ).toFixed(2)}`}
    37  +                     </td>
    38  +                     <td>
    39  +                       <button
    40  +                         type="button"
    41  +                         onClick={() => dispatch({
    42  +                           type: CartTypes.REMOVE,
    43  +                           itemId: item.id,
    44  +                         })}
    45  +                       >
    46  +                         Remove
    47  +                       </button>
    48  +                     </td>
    49  +                   </tr>
    50  +                 ))}
    51  +               </tbody>
    52  +             </table>
    53  +             <div>
    54  +               Sub-total: $
    55  +               { subTotal.toFixed(2) }
    56  +             </div>
    57  +           </>
47  58              ))}
48  59          </div>
49  60      );
```

The javascript `Array.reduce` function provided a nice way to get the subtotal from all the cart items.

The code uses a React fragment again since React requires exactly one root element on either side of a ternary (which caused you to need to tab table the whole table in again).

Add some items to your cart and check out the subtotal.

$4.00 for drinks for two not bad!

Conclusion

The cart is looking good. Hopefully, you are feeling good about your React skills as well.

Next, you will add a form so users can checkout to place their order.

We've supplied some challenges below to continue practicing these skills. Remember to back up your code before starting challenges. You will probably be able continue through the rest of the course with your code from these two challenges, but it's always a good idea to have something to come back to if you get stuck.

🥉 Increase Quantity

What happens if you get to your cart and remember you wanted to add a second coffee for a friend?

Right now, you have to go back home, click coffee, and click add to cart again.

Add a `+` (plus) button so the user can increase the quantity of an item by one.

Note: You can use the existing `ADD` action type with the dispatch function.

🥈 Decrease Quantity

What happens if you accidentally ordered two coffees instead of one?

Right now, the user has to remove both and go back to the coffee detail page to add one. That's not very nice.

Add a `-` (minus) button so the user can decrease the quantity of an item by one.

Note: You will need to add a new `DECREASE` reducer type.

Also, there is a special case. If the user only has one of the item in the cart, decreasing the quantity should remove it from the cart rather than having an item with a quantity of zero in the cart.

Checkout

You are ready to add a checkout form to the cart page. To keep things simple, you will add the form directly under the cart.

If you completed the challenges in the previous chapter, your code diffs will look different than ours. Feel free to continue with your code from the challenges, but if you get stuck, switch back to the backup copy from before the challenges. If you forgot to take a backup, you can also start from the solution file from the previous chapter.

Creating the Form

The user will be able to enter their name, an optional phone number, and their zip code. (The zip code will be used to compute tax.)

> **WARNING**
>
> Clicking submit on the checkout form will cause the page to refresh erasing your cart. You will fix this later in the chapter.

src/components/Cart.css CHANGED

```
@@ -4,4 +4,13 @@

  .cart-component h2 {
    margin-top: 0;
- }
+ }
+
+ .cart-component label {
+   display: block;
+ }
+
+ .cart-component form input {
+   margin: 5px;
+   padding: 2px;
+ }
```

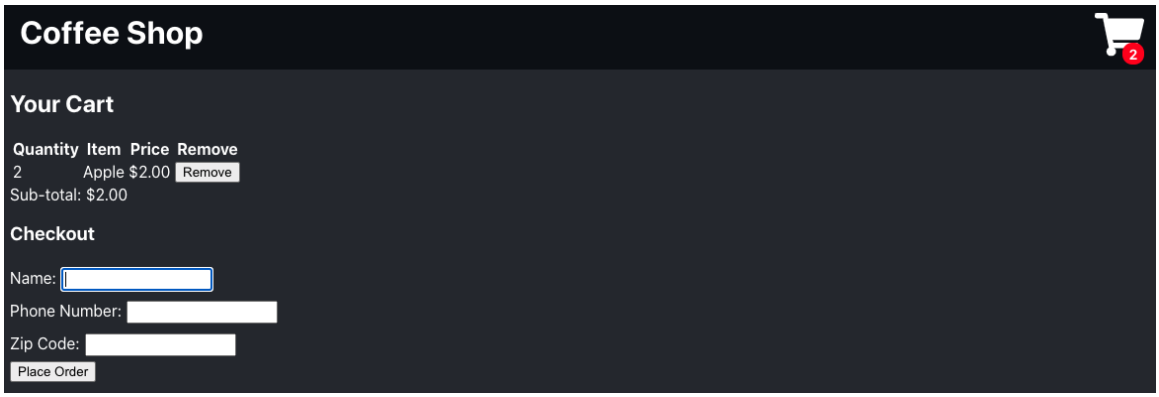
src/components/Cart.js CHANGED

```
@@ -54,6 +54,36 @@ function Cart({ cart, dispatch, items }) {
                  Sub-total: $
                  { subTotal.toFixed(2) }
                </div>
+               <h3>Checkout</h3>
+               <form>
+                 <label htmlFor="name">
+                   Name:
+                   <input
+                     id="name"
+                     type="text"
```

```
+                    required
+                  />
+                </label>
+                <label htmlFor="phone">
+                  Phone Number:
+                  <input
+                    id="phone"
+                    type="tel"
+                  />
+                </label>
+                <label htmlFor="zipcode">
+                  Zip Code:
+                  <input
+                    id="zipcode"
+                    required
+                    maxLength="5"
+                    type="number"
+                  />
+                </label>
+                <button type="submit">
+                  Place Order
+                </button>
+              </form>
            </>
          ))}
      </div>
```

In this code, you'll notice another JSX change `htmlFor`. `for` (like `class`) is a reserved word so you need to use `htmlFor` to represent `for` on labels. Now, the labels are nicely associated with the inputs.

Add at least one item to the cart and checkout the new form.



🔎 Controlled Components

The checkout form will need to be able to get the value for each of the inputs. There are a variety of ways to do this, but controlled components are the recommended way.

React uses one way data binding where what is contained in `state`/JSX should exactly match what is on the page. Controlled components make sure the component value matches state.

https://reactjs.org/docs/forms.html#controlled-components

To experiment with this set the value for the `name` input to `hi`.

src/components/Cart.js CHANGED

```
@@ -62,6 +62,7 @@ function Cart({ cart, dispatch, items }) {
                     id="name"
                     type="text"
                     required
+                    value="hi"
                   />
                 </label>
                 <label htmlFor="phone">
```

Now, visit your page and try to edit the value for `name`. You will notice that you cannot edit it. This is because of one way data binding. React makes sure the value in the input matches what is defined in the JSX.

If you open the console, you will notice a helpful warning from React:

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

Capturing Input and `useState`

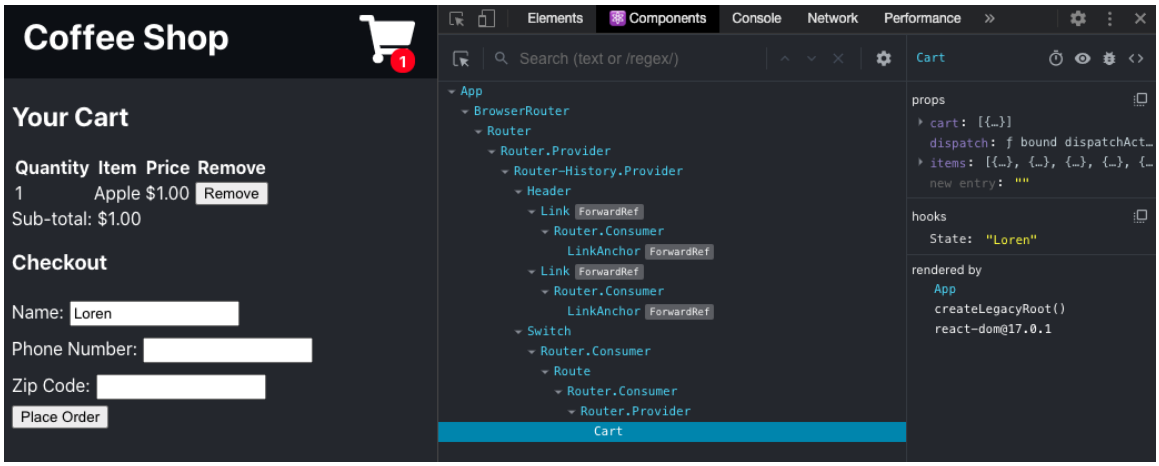
The user needs to be able to edit the value of the name, which means you need on `onChange` handler and need a state value to track the `name`. React provides as `useState` hook which allows you to get state values in a functional component rather than having to use a class. `useState` is easier to use than `useReducer`. It takes the initial state value as an argument and returns the current state value and a function to set a new state value.

src/components/Cart.js CHANGED

```
@@ -1,8 +1,10 @@
 import PropTypes from 'prop-types';
+import { useState } from 'react';
 import { CartTypes } from '../reducers/cartReducer';
 import './Cart.css';

 function Cart({ cart, dispatch, items }) {
+  const [name, setName] = useState('');
   const subTotal = cart.reduce((acc, item) => {
     const details = items.find((i) => i.id === item.id);
     return item.quantity * details.price + acc;
@@ -62,7 +64,8 @@ function Cart({ cart, dispatch, items }) {
                     id="name"
                     type="text"
                     required
-                    value="hi"
+                    value={name}
+                    onChange={(event) => setName(event.target.value)}
                   />
                 </label>
                 <label htmlFor="phone">
```

On change passes the event which contains the new value in `event.target.value`. The code uses that to update the state.

Nice, visit your form and enter a name into the name field. Then, open the React component window in devtools and select the `Cart` component. In the sidebar, under hooks, you will see the name you typed in state.



Phone Number and Zip Code

Let's use this same skill again to store the values for phone number and zip code in state.

> **Practice**
>
> See if you can do this without the solution.

src/components/Cart.js CHANGED

```
@@ -5,6 +5,8 @@ import './Cart.css';

 function Cart({ cart, dispatch, items }) {
   const [name, setName] = useState('');
+  const [phone, setPhone] = useState('');
+  const [zipCode, setZipCode] = useState('');
   const subTotal = cart.reduce((acc, item) => {
     const details = items.find((i) => i.id === item.id);
     return item.quantity * details.price + acc;
@@ -73,6 +75,8 @@ function Cart({ cart, dispatch, items }) {
                 <input
                   id="phone"
                   type="tel"
+                  value={phone}
+                  onChange={(event) => setPhone(event.target.value)}
                 />
               </label>
               <label htmlFor="zipcode">
@@ -82,6 +86,8 @@ function Cart({ cart, dispatch, items }) {
                   required
                   maxLength="5"
                   type="number"
+                  value={zipCode}
+                  onChange={(event) => setZipCode(event.target.value)}
                 />
               </label>
               <button type="submit">
```

Now, you can enter name, phone number, and zip code and see the state values in devtools.

Formatting Input

One of the nice parts about controlled inputs is it enables you to format input for the user. Let's format phone numbers with dashes (`xxx-xxx-xxxx`).

src/components/Cart.js CHANGED

```
@@ -12,6 +12,22 @@ function Cart({ cart, dispatch, items }) {
      return item.quantity * details.price + acc;
    }, 0);

+   const updatePhoneNumber = (newNumber) => {
+     const digits = newNumber.replace(/\D/g, '');
+     let formatted = digits.substring(0, 3);
+     if (digits.length === 3 && newNumber[3] === '-') {
+       formatted = `${formatted}-`;
+     } else if (digits.length > 3) {
+       formatted = `${formatted}-${digits.substring(3, 6)}`;
+     }
+     if (digits.length === 6 && newNumber[7] === '-') {
+       formatted = `${formatted}-`;
+     } else if (digits.length > 6) {
+       formatted = `${formatted}-${digits.substring(6, 10)}`;
+     }
+     setPhone(formatted);
+   };
+
    return (
      <div className="cart-component">
        <h2>Your Cart</h2>
@@ -76,7 +92,7 @@ function Cart({ cart, dispatch, items }) {
                    id="phone"
                    type="tel"
                    value={phone}
-                   onChange={(event) => setPhone(event.target.value)}
+                   onChange={(event) => updatePhoneNumber(event.target.value)}
                  />
                </label>
                <label htmlFor="zipcode">
```

Since this is more complicated, a helper function does the actual formatting. Once the phone number has been formatted, the function calls `setPhone` to store the newly formatted number.

Type in a number using only digits. You will see the dashes automatically inserted for you.

There is, however, a bug in the code. If you try to edit the middle of the phone number, your cursor will continuously jump to the end of the input. React/Javascript allows you to work with the exact cursor position, but that is complicated, so it is generally better to use an npm package to handle number formatting. You will continue on with the current solution.

🔎 String Manipulation

This section uses string replace, substring, and indexes on strings to check the values.

You can find out more about the built-in string methods on MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Autofocus

It might be nice if the cursor was in the name field immediately when the page loaded. React provides an `autoFocus` prop that does just that.

src/components/Cart.js CHANGED

```
@@ -78,7 +78,9 @@ function Cart({ cart, dispatch, items }) {
78  78                <form>
79  79                  <label htmlFor="name">
80  80                    Name:
    81 +                  {/* eslint-disable jsx-a11y/no-autofocus */}
81  82                    <input
    83 +                    autoFocus
82  84                      id="name"
83  85                      type="text"
84  86                      required
```

This prop does not use the `autofocus` HTML attribute. You can check the input in the elements panel of devtools to be sure. React uses a polyfill instead that automatically calls `.focus()` on the input when the component mounts.

There are accessibility concerns with autofocusing fields which is why you needed to disable the eslint rule. You can read more about it on the website https://github.com/jsx-eslint/eslint-plugin-jsx-a11y/blob/master/docs/rules/no-autofocus.md . Be sure to also check out the resources section for more details.

Advancing Focus

It also might be nice if the user's cursor automatically moved from the phone number input to the zip code input. (Note: This also has some accessibility concerns.)

To advance the cursor you need to get the input element from the DOM and call `.focus()`. React provides a `useRef` hook to help with this.

src/components/Cart.js CHANGED

```
@@ -1,5 +1,5 @@
1   1   import PropTypes from 'prop-types';
2       - import { useState } from 'react';
        + import { useState, useRef } from 'react';
3   3   import { CartTypes } from '../reducers/cartReducer';
4   4   import './Cart.css';
5   5
@@ -7,6 +7,8 @@ function Cart({ cart, dispatch, items }) {
7   7     const [name, setName] = useState('');
8   8     const [phone, setPhone] = useState('');
9   9     const [zipCode, setZipCode] = useState('');
    10 +   const zipRef = useRef();
    11 +
10  12    const subTotal = cart.reduce((acc, item) => {
11  13      const details = items.find((i) => i.id === item.id);
```

```
12  14      return item.quantity * details.price + acc;
        @@ -26,6 +28,10 @@ function Cart({ cart, dispatch, items }) {
26  28        formatted = `${formatted}-${digits.substring(6, 10)}`;
27  29      }
28  30      setPhone(formatted);
        31  +
        32  +    if (digits.length === 10) {
        33  +      zipRef.current.focus();
        34  +    }
29  35    };
30  36
31  37    return (
        @@ -101,6 +107,7 @@ function Cart({ cart, dispatch, items }) {
101 107                   Zip Code:
102 108                   <input
103 109                     id="zipcode"
        110 +                 ref={zipRef}
104 111                   required
105 112                   maxLength="5"
106 113                   type="number"
```

🔎 `useRef` hook

The `useRef` hook does not require any arguments and returns a ref object. You can get the current ref with `ref.current` so `ref.current.focus()` focus the input.

Inputs have a `ref` prop which can be set to the ref object. Then, React will set the current ref when the component renders.

You can read more about `useRef` here https://reactjs.org/docs/hooks-reference.html#useref .

Refs can also be used to track other values https://reactjs.org/docs/hooks-faq.html#is-there-something-like-instance-variables .

Tax Calculation

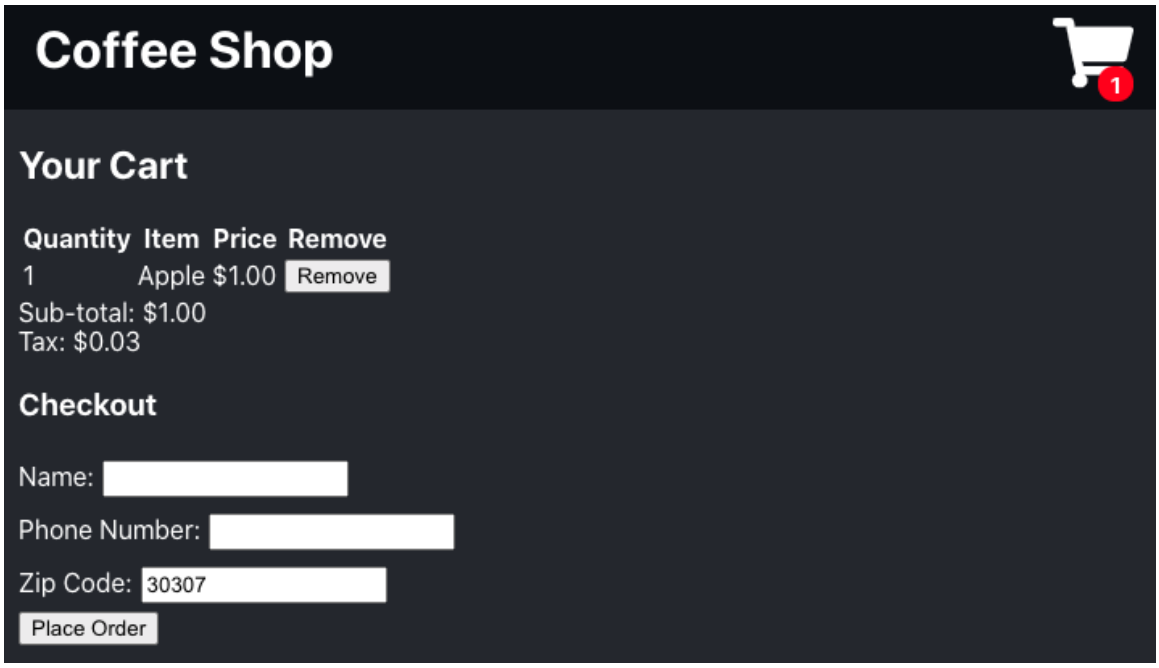
To compute the total for the order, you need to add in the tax. Use the first digit of the zip code plus 1 as the tax rate. Then, compute and display the tax.

src/components/Cart.js CHANGED

```
        @@ -13,6 +13,9 @@ function Cart({ cart, dispatch, items }) {
13  13      const details = items.find((i) => i.id === item.id);
14  14      return item.quantity * details.price + acc;
15  15    }, 0);
        16  +  const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
        17  +  const taxRate = taxPercentage / 100;
        18  +  const tax = subTotal * taxRate;
16  19
17  20    const updatePhoneNumber = (newNumber) => {
18  21      const digits = newNumber.replace(/\D/g, '');
        @@ -80,6 +83,10 @@ function Cart({ cart, dispatch, items }) {
80  83                 Sub-total: $
81  84                 { subTotal.toFixed(2) }
82  85               </div>
        86  +            <div>
        87  +              Tax: $
        88  +              { tax.toFixed(2) }
        89  +            </div>
83  90               <h3>Checkout</h3>
```

```
84  91              <form>
85  92                <label htmlFor="name">
```

Fill in a zip code, and you will see the tax displayed.



Total

Now, let's also show the total, which will be the sum of the subtotal and tax.

> **Practice**
>
> See if you can do this without the solution.

src/components/Cart.js CHANGED

```
        @@ -16,6 +16,7 @@ function Cart({ cart, dispatch, items }) {
16  16      const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
17  17      const taxRate = taxPercentage / 100;
18  18      const tax = subTotal * taxRate;
    19  +   const total = subTotal + tax;
19  20
20  21      const updatePhoneNumber = (newNumber) => {
21  22        const digits = newNumber.replace(/\D/g, '');
        @@ -87,6 +88,10 @@ function Cart({ cart, dispatch, items }) {
87  88                  Tax: $
88  89                  { tax.toFixed(2) }
89  90                </div>
    91  +             <div>
    92  +               Total: $
    93  +               { total.toFixed(2) }
    94  +             </div>
90  95                <h3>Checkout</h3>
91  96                <form>
92  97                  <label htmlFor="name">
```

Conditional Total

Right now, when a user visits the checkout page, they immediately see $0.00 as tax and a total. Since the tax rate is unknown, it should not show the tax or total until the user enters their zip code.

Update the code to tell the user to enter their zip code to see the total if they have not entered a zip code. If they have entered their zipcode, they should see the total.

> **Practice**
>
> See if you can do this without the solution.

src/components/Cart.js CHANGED

```
@@ -84,14 +84,20 @@ function Cart({ cart, dispatch, items }) {
                  Sub-total: $
                  { subTotal.toFixed(2) }
                </div>
+               { zipCode.length === 5
+                 ? (
+                   <>
-               <div>
+                 <div>
-                 Tax: $
+                   Tax: $
-                 { tax.toFixed(2) }
+                   { tax.toFixed(2) }
-               </div>
+                 </div>
-               <div>
+                 <div>
-                 Total: $
+                   Total: $
-                 { total.toFixed(2) }
+                   { total.toFixed(2) }
-               </div>
+                 </div>
+                   </>
+                 )
+                 : <div>Enter Zip Code to get total</div> }
                <h3>Checkout</h3>
                <form>
                  <label htmlFor="name">
```

Test out your changes in the browser. You should only see the total after entering your 5 digit zip code.

When Calculations Run

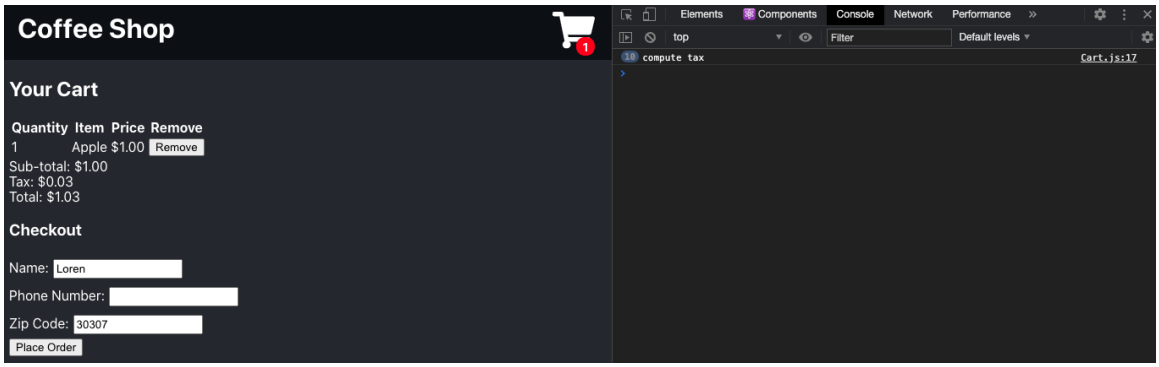
Let's add a console log to see how often tax calculations run.

src/components/Cart.js CHANGED

```
@@ -14,6 +14,7 @@ function Cart({ cart, dispatch, items }) {
```

```
14  14        return item.quantity * details.price + acc;
15  15      }, 0);
16  16      const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
    17  +   console.log('compute tax');
17  18      const taxRate = taxPercentage / 100;
18  19      const tax = subTotal * taxRate;
19  20      const total = subTotal + tax;
```

Visit the checkout page, open and clear the console tab of devtools, and type in your name and zip code.



The `10` next to "compute tax" means that was logged 10 times. Once for each character of my name and zipcode. You might see a higher count depending on when you last cleared devtools and if needed to backspace and retype any characters. This happened even though my name has nothing to do with calculating the tax rate because the component re-renders every time state changes which occurred with each character of my name and zip code.

Reducing Calculations

Computing on every render is not problematic if the computations are very fast, but if they are slow, it can result in slow rendering and slow down users trying to type into a form.

This tax calculation is simple and fast, but tax calculations in general are complicated and could be slow, so you will optimize the tax calculations to only re-run when the zip code changes. You can do this with the `useMemo` hook.

src/components/Cart.js CHANGED

```
@@ -1,5 +1,5 @@
1   1   import PropTypes from 'prop-types';
2       - import { useState, useRef } from 'react';
        2 + import { useMemo, useState, useRef } from 'react';
3   3   import { CartTypes } from '../reducers/cartReducer';
4   4   import './Cart.css';
5   5
@@ -13,9 +13,14 @@ function Cart({ cart, dispatch, items }) {
13  13       const details = items.find((i) => i.id === item.id);
14  14       return item.quantity * details.price + acc;
15  15     }, 0);
        16 +   const taxRate = useMemo(
        17 +     () => {
16      -   const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
        18 +     const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
17      -   console.log('compute tax');
```

```
+       console.log('compute tax');
-     const taxRate = taxPercentage / 100;
+       return taxPercentage / 100;
+     },
+     [zipCode],
+   );
    const tax = subTotal * taxRate;
    const total = subTotal + tax;
```

Visit the checkout page again, open and clear the console tab, and type in your name and zip code. You should now only see "compute tax" logged when you are changing the zip code field.

🔎 `useMemo` Hook

The `useMemo` hook, like `useCallback`, takes two arguments the function to compute the value and the dependencies of the value. The returned value from `useMemo` is the result of the function, which in this case the tax rate. React uses memos to improve efficiency but does not guarantee that they will only ever be called when the values change. It's possible React might clean out some memory or otherwise need to re-run them in the future.

You can read more about `useMemo` here: https://reactjs.org/docs/hooks-reference.html#usememo

Clean Up

Now that the tax calculation is optimized, remove the console log.

src/components/Cart.js CHANGED

```
@@ -16,7 +16,6 @@ function Cart({ cart, dispatch, items }) {
    const taxRate = useMemo(
      () => {
        const taxPercentage = parseInt(zipCode.substring(0, 1) || '0', 10) + 1;
-       console.log('compute tax');
        return taxPercentage / 100;
      },
      [zipCode],
```

Valid Form

The checkout button should only be enabled if the user has supplied their name and zip code.

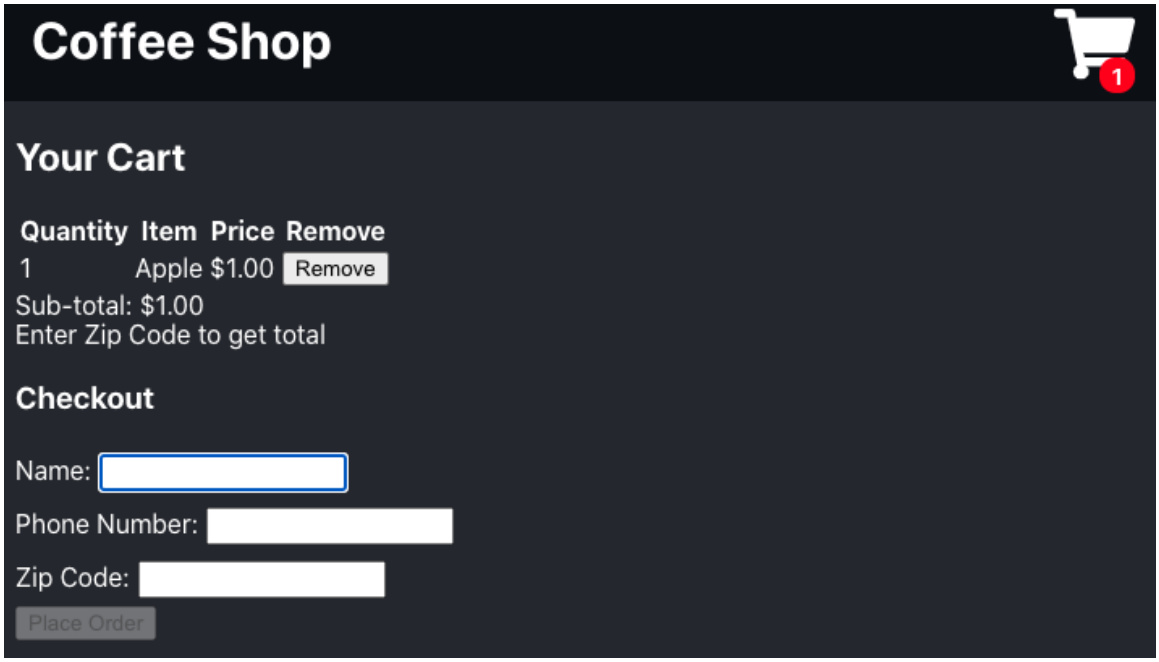
You can do that with the `disabled` HTML attribute, and React can bind a boolean value to the attribute.

src/components/Cart.js CHANGED

```
@@ -22,6 +22,7 @@ function Cart({ cart, dispatch, items }) {
    );
    const tax = subTotal * taxRate;
    const total = subTotal + tax;
+   const formValid = zipCode.length === 5 && name.trim();
```

```
26  27      const updatePhoneNumber = (newNumber) => {
27  28        const digits = newNumber.replace(/\D/g, '');
    @@ -138,7 +139,7 @@ function Cart({ cart, dispatch, items }) {
138 139                       onChange={(event) => setZipCode(event.target.value)}
139 140                     />
140 141                   </label>
141     -                 <button type="submit">
    142 +                 <button type="submit" disabled={!formValid}>
142 143                     Place Order
143 144                   </button>
144 145                 </form>
```

Visit the browser and the "Place Order" button will be disabled until you fill in a name and zip code.



Conclusion

The checkout form is done. In the next chapter, you will hook the coffee shop up to the backend API so users can place orders and get the item list from the coffee shop's server.

🥇 Set Quantity

Plus and minus buttons are nice, but you could also offer the user a way to directly set the quantity.

Replace the text number with a select box where the user can select their desired quantity.

Note: You will need to add a new `SET_QUANTITY` reducer type.

For an added challenge, you can cause the quantity of 0 to remove the item.

Submit Orders

You are ready to start integrating with the API.

Start the Server

Your download includes a server for the coffee shop. Locate the `coffee-backend` folder from your resources download.

In a terminal, navigate to that folder and install the dependencies. Then, start the server.

```
cd YOUR_PATH/resources/coffee-backend
npm install
npm start
```

You should see "Listening on http://localhost:3030" print to the console if the server started correctly. After that, you should also be able to visit http://localhost:3030 and see `{"message":"Hi!"}`.

There is now a REST running API at `http://localhost:3030/`. It has the following endpoints ready to be interacted with:

- GET `/api/items` returns all the items for the coffee shop
- `/api/orders` is a RESTful interface for orders
  - GET `/api/orders` returns all the orders
  - POST `/api/orders` creates an order
  - DELETE `/api/orders` deletes all the orders
  - GET `/api/orders/:id` return one order
  - PUT `/api/orders/:id` updates an order
  - DELETE `/api/orders/:id` deletes one order
- GET `/api/auth/current-user` returns the currently logged-in user or `{}` if no user is logged in
- POST `/api/auth/login` with username and password as the JSON body logs in a user and returns 401 if the login is invalid
- POST `/api/auth/logout` logs out a user

You can refer back to this section as you move along if necessary. If you are feeling overwhelmed, or that the documentation above is not sufficient, do not worry we will provide step by step code to use the API. You can also hit the endpoints from a browser or a tool such as cURL or Postman.

Like React, the server will run until you stop it with `Ctrl + C`. Leave this terminal window open. You will need the server running for the rest of the development of the coffee shop. If you take a break, feel free to stop the coffee shop, but remember to start it again, or you will get network errors. Additionally, if you start seeing network errors, it is good to first check that the server is running and if it has logged any errors to the console.

> **WARNING**
>
> The server does not use any persistent storage so each time you restart your server, all coffee orders will be lost. You will only need to restart the server if your computer restarts or you are taking a long break.

Proxy

Because of cross site script attacks, there are Cross Origin Resource Sharing (CORS) policies that must be set up to allow an application to make API requests to a different domain. These can be complicated to set up properly. If you make them too lax, it could allow your site to be attacked. If you're too strict, your API will not work from your own front end.

Instead of dealing with CORS policies, you will have the front end act as a proxy to the backend server. By doing this, you don't have to open any allowances for cross origin sharing.

Because this is a common pattern, `create-react-app` makes it easy to set this up in their development server.

package.json CHANGED

```
@@ -26,6 +26,7 @@
26  26      "lint": "eslint src --max-warnings=0",
27  27      "eject": "react-scripts eject"
28  28    },
    29  +   "proxy": "http://localhost:3030",
29  30    "eslintConfig": {
30  31      "extends": [
31  32        "react-app",
```

> **Restart React Coffee Shop**
>
> If you are currently running Coffee Shop, you will need to restart for this change to take effect. You can do that by killing the currently running process for `coffee-shop` with `Ctrl + C`, and starting it again `npm start`.
>
> Assuming you are in the terminal where you had just run it, you can get to the most recently run command by pressing the up arrow. Then, press enter to run it. Bash also provides a shortcut to the most recent command without needing the up arrow `!!`.

Note: This `proxy` setup is only for local development when running `npm start`. It will not do anything when running a production build on the server. For production, you would need to make sure whatever web server you have server the static files React builds also running a proxy. If your server is Node.js, `http-proxy-middleware` is a popular package to use for setting up a proxy https://www.npmjs.com/package/http-proxy-middleware .

🔎 Cross Origin Resource Sharing (CORS)

Cross Origin Resource Sharing is when a resource is requested from a domain that does not match the current domain. The browser allows scripts and style sheets to be loaded from other origins, but it does not allow AJAX requests to be made to another origin. When resource request is going to go across origins, the browser sends an `OPTIONS` request as a pre-flight check to see if the server on the other domain wants to allow that request.

One of the reasons this is blocked is to help prevent malicious requests. An example attack might be whenever someone visits my website, I send a request to vote for my opinion in a public pool skewing the results. By default, the browser does not send cookies even when CORS is allowed which prevents users from being logged in. Assuming you were logged in and cookies were sent, I could send a request to Amazon to make a buy it now purchase of this book or share an article on Facebook.

You can read more about CORS here: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS .

Submit the Form

Now that the server is running, you are ready to submit the coffee order form.

First, install `axios`. A library for making HTTP requests.

> **Install**
>
> In a separate terminal from your app, CD into your `coffee-shop` directory.
>
> Then, install Axios `npm install --save axios`

Now, use axios to make a post request.

src/components/Cart.js CHANGED

```
@@ -1,3 +1,4 @@
+ import axios from 'axios';
  import PropTypes from 'prop-types';
  import { useMemo, useState, useRef } from 'react';
  import { CartTypes } from '../reducers/cartReducer';
@@ -44,6 +45,20 @@ function Cart({ cart, dispatch, items }) {
      }
    };
```

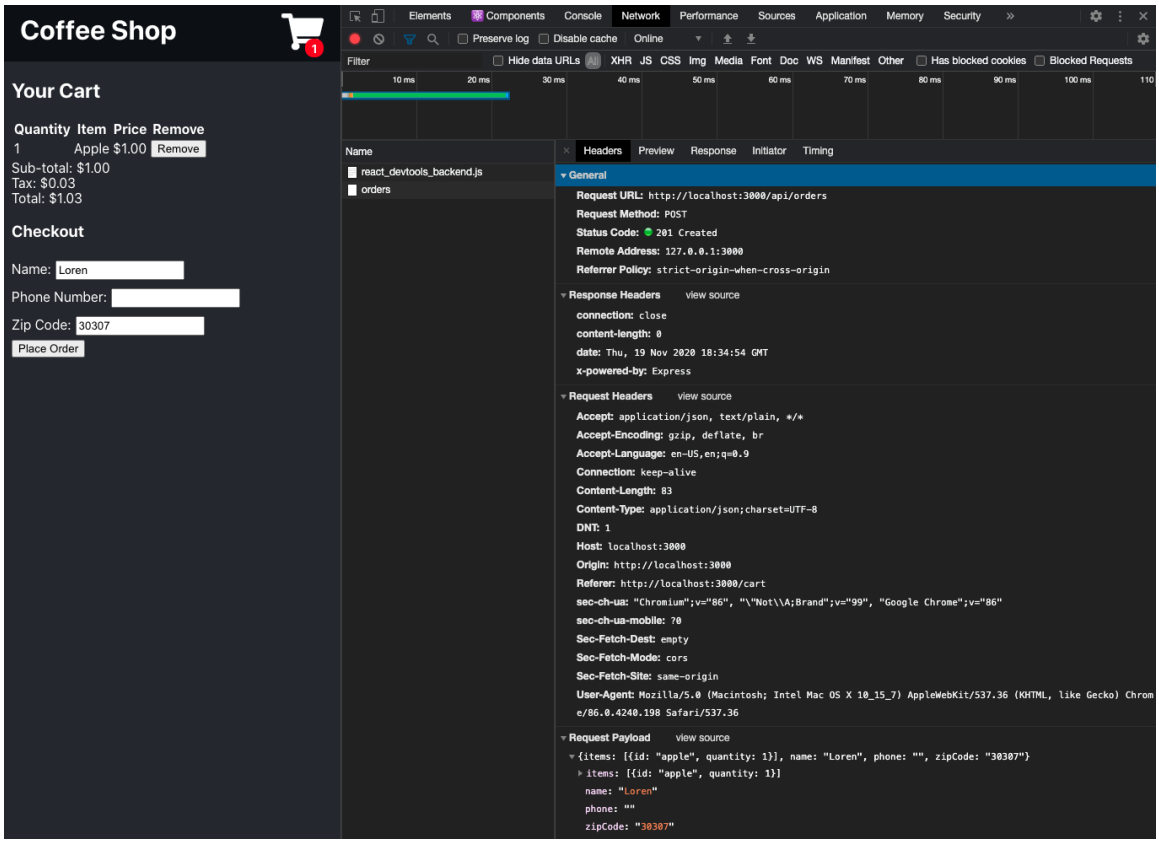
```
+   const submitOrder = (event) => {
+     event.preventDefault();
+     axios.post('/api/orders', {
+       items: cart,
+       name,
+       phone,
+       zipCode,
+     }).then(() => {
+       console.log('Order Submitted');
+     }).catch((error) => {
+       console.error(error);
+     });
+   };
+
    return (
      <div className="cart-component">
        <h2>Your Cart</h2>
@@ -105,7 +120,7 @@ function Cart({ cart, dispatch, items }) {
                  )
                  : <div>Enter Zip Code to get total</div> }
              <h3>Checkout</h3>
-             <form>
+             <form onSubmit={submitOrder}>
                <label htmlFor="name">
                  Name:
                  {/* eslint-disable jsx-a11y/no-autofocus */}
```

React lets you listen for form submission events with the `onSubmit` prop of a form, and passes the event as the first argument to the function. You need to call `.preventDefault()` on the event to prevent native browser form submission, which results in reloading the page. Then, axios makes a POST request to the API with the order.

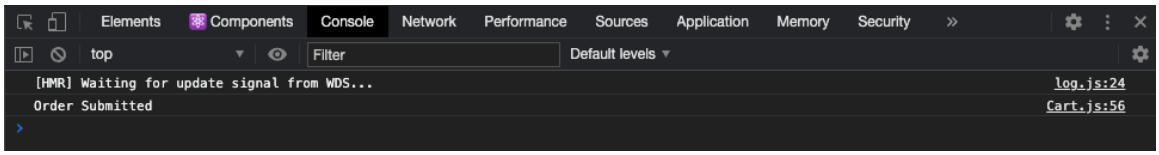
Try it out. Add an item to your cart. Then, open the network tab, and checkout.

> **WARNING**
>
> Do not use the zip code "99999" when checking out. It intentionally causes an error which you will handle later in the chapter.

You should see a network request to `orders`. Click on it and you will see the information about the request including the 201 created status code.



Switch over to the console tab and you will see the console log as well.



You can also view the order list from the server at http://localhost:3030/api/orders .

Empty the Cart

It would be nice if the user's cart was emptied after their order was successfully submitted, so they can feel more confident and do not accidentally place duplicated orders.

Start out by creating a new cart reducer action called `EMPTY` to empty the cart.

> **Practice**
>
> Can you create that on your own?

src/reducers/cartReducer.js CHANGED

```
@@ -4,6 +4,7 @@ const initialCart = [];
```

```
 4   4
 5   5   export const CartTypes = {
 6   6     ADD: 'ADD',
     7 +   EMPTY: 'EMPTY',
 7   8     REMOVE: 'REMOVE',
 8   9   };
 9  10
     @@ -25,6 +26,8 @@ const cartReducer = (state, action) => {
25  26           ...state,
26  27           { id: action.itemId, quantity: 1 },
27  28         ];
     29 +     case CartTypes.EMPTY:
     30 +       return [];
28  31       case CartTypes.REMOVE:
29  32         return state.filter((item) => item.id !== action.itemId);
30  33       default:
```

Now, dispatch that action along with the `console.log`.

src/components/Cart.js CHANGED

```
     @@ -54,6 +54,7 @@ function Cart({ cart, dispatch, items }) {
54  54         zipCode,
55  55       }).then(() => {
56  56         console.log('Order Submitted');
     57 +       dispatch({ type: CartTypes.EMPTY });
57  58       }).catch((error) => {
58  59         console.error(error);
59  60       });
```

Check out again, and you will have an empty cart.

Show a Thank You Modal

This is better, but it would be even nicer if the user received thank you message.

You will use a new package for this `react-modal`.

> **Install**
>
> In a separate terminal from your app, CD into your `coffee-shop` directory.
>
> Then, install React Modal `npm install --save react-modal`

src/components/Cart.js CHANGED

```
     @@ -1,13 +1,29 @@
1   1   import axios from 'axios';
2   2   import PropTypes from 'prop-types';
3   3   import { useMemo, useState, useRef } from 'react';
    4 + import Modal from 'react-modal';
4   5   import { CartTypes } from '../reducers/cartReducer';
5   6   import './Cart.css';
6   7
    8 + const customStyles = {
    9 +   content: {
   10 +     top: '50%',
   11 +     left: '50%',
   12 +     right: 'auto',
   13 +     bottom: 'auto',
   14 +     marginRight: '-50%',
   15 +     transform: 'translate(-50%, -50%)',
```

```
+     color: '#000',
+   },
+ };
+
+ Modal.setAppElement('#root');
+
  function Cart({ cart, dispatch, items }) {
    const [name, setName] = useState('');
    const [phone, setPhone] = useState('');
    const [zipCode, setZipCode] = useState('');
+   const [thankYouOpen, setThankYouOpen] = useState(false);
    const zipRef = useRef();

    const subTotal = cart.reduce((acc, item) => {
@@ -53,15 +69,30 @@ function Cart({ cart, dispatch, items }) {
        phone,
        zipCode,
      }).then(() => {
-       console.log('Order Submitted');
        dispatch({ type: CartTypes.EMPTY });
+       setThankYouOpen(true);
      }).catch((error) => {
        console.error(error);
      });
    };

+   const closeThankYouModal = () => {
+     setThankYouOpen(false);
+   };
+
    return (
      <div className="cart-component">
+       <Modal
+         isOpen={thankYouOpen}
+         onRequestClose={closeThankYouModal}
+         style={customStyles}
+         contentLabel="Thanks for your order"
+       >
+         <p>Thanks for your order!</p>
+         <button onClick={closeThankYouModal} type="button">
+           Close
+         </button>
+       </Modal>
        <h2>Your Cart</h2>
        {(cart.length === 0
          ? <div>Your cart is empty.</div>
```

`react-modal` allows for custom styles. The provided styles adjust the size and placement of the modal. `react-modal` also needs to know the react root element, which was setup as `#root` by create react app.

There is a new boolean value in state named `thankYouOpen` to track if the modal should be open or closed.

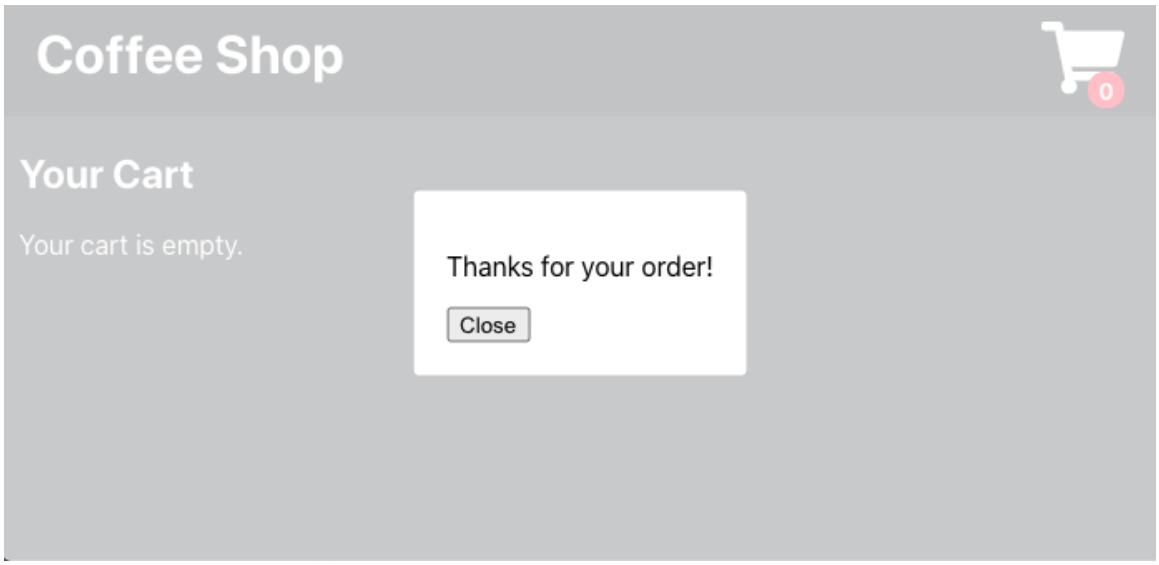
Instead of calling `console.log`, the code updates state to open the modal.

The `Modal` component itself has props for if it is open, the function to call when the user requests it to close by clicking outside the modal, custom styles, and a content label for screen readers.

The content of the modal thanks the user and provides a button to close the modal. This function is the same one used if the user clicks outside the modal.

You can learn more about react-modal here: http://reactcommunity.org/react-modal/ .

Submit a new order to try out your new modal. You should be able to close it by clicking on the translucent background outside the modal, clicking the close button, or by pressing escape.



Redirect Home

Rather than leaving the user on the cart page, let's redirect them to the home page when they close the modal.

Previously, you've used links to change the page the user is on, but you cannot change the background of the modal to a link so you will want to use the `useNavigate` hook from React Router.

src/components/Cart.js CHANGED

```
@@ -2,6 +2,7 @@ import axios from 'axios';
  import PropTypes from 'prop-types';
  import { useMemo, useState, useRef } from 'react';
  import Modal from 'react-modal';
+ import { useNavigate } from 'react-router-dom';
  import { CartTypes } from '../reducers/cartReducer';
  import './Cart.css';

@@ -25,6 +26,7 @@ function Cart({ cart, dispatch, items }) {
    const [zipCode, setZipCode] = useState('');
    const [thankYouOpen, setThankYouOpen] = useState(false);
    const zipRef = useRef();
+   const navigate = useNavigate();

    const subTotal = cart.reduce((acc, item) => {
      const details = items.find((i) => i.id === item.id);
@@ -78,6 +80,7 @@ function Cart({ cart, dispatch, items }) {

    const closeThankYouModal = () => {
      setThankYouOpen(false);
+     navigate('/');
    };

    return (
@@ -90,7 +93,7 @@ function Cart({ cart, dispatch, items }) {
        >
          <p>Thanks for your order!</p>
```

```
92  95            <button onClick={closeThankYouModal} type="button">
93      -           Close
        96  +       Return Home
94  97            </button>
95  98          </Modal>
96  99          <h2>Your Cart</h2>
```

Place another order and close the modal, and you will be back on the home page.

🔎 `useNavigate` Hook

In this case, you are using a hook provided by React Router instead of one that is native to React. This hook returns a function, which allows you to change the user's location. By default, it will `push` a new browser state, which means the user can click back in the browser to go back to the previous page. You can try that out by placing an order and clicking back in your browser. You end up back on the cart page. Also, the URL in the URL bar changes to the home URL while you are on that page and back to the cart page when you click back.

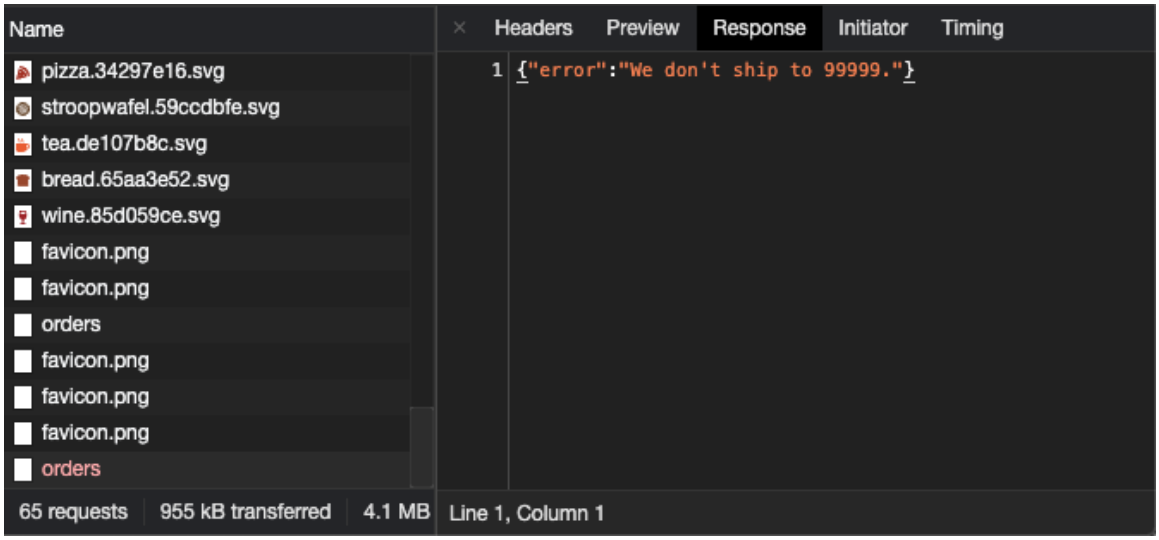
The full API of the `navigate` function is available at:
https://reactrouter.com/docs/en/v6/api#usenavigate

Show an Error Modal

The happy path is working well, but what happens if the user gets an error? Try to check out with the zip code "99999".

You will see an error in the console with 400 Bad request.

If you look at the network tab, you will see a request to `orders` in red. Click that request and click the "response" tab. You will see an error message back from the server.



Most users will not open devtools, so you need a way to show them the error. Show the error message in another modal like the thank you modal, so the user can correct it.

src/components/Cart.js CHANGED

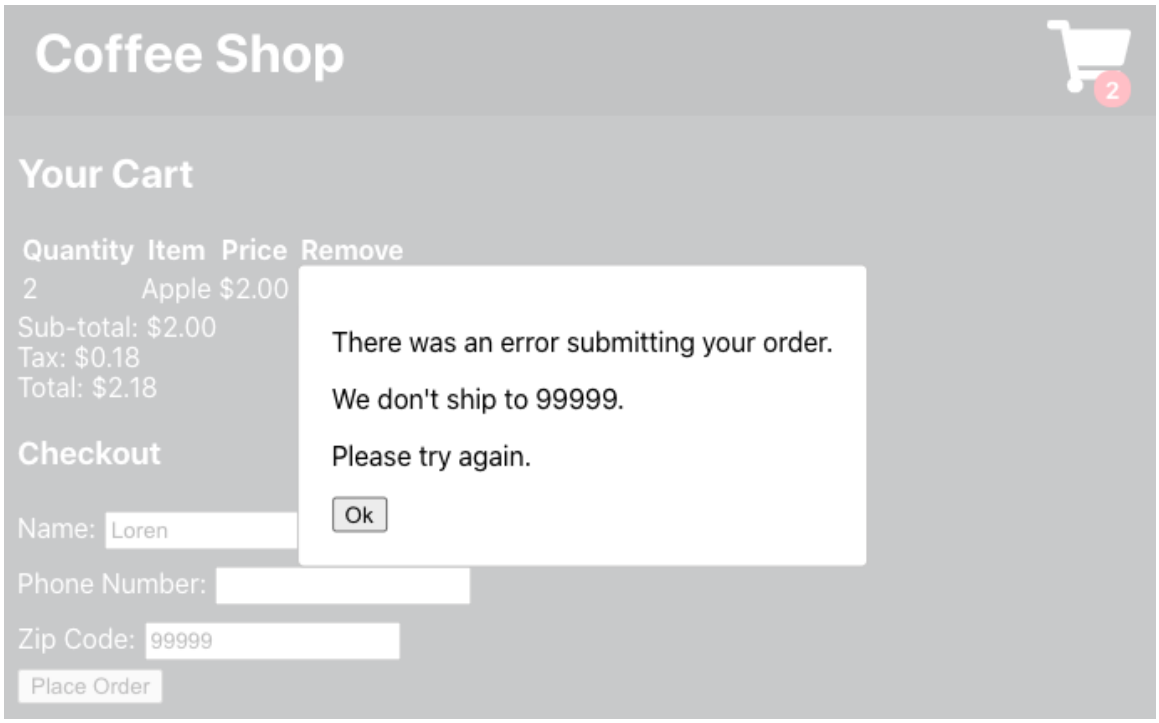
```
@@ -25,6 +25,7 @@ function Cart({ cart, dispatch, items }) {
    const [phone, setPhone] = useState('');
    const [zipCode, setZipCode] = useState('');
    const [thankYouOpen, setThankYouOpen] = useState(false);
+   const [apiError, setApiError] = useState('');
    const zipRef = useRef();
    const navigate = useNavigate();

@@ -75,6 +76,7 @@ function Cart({ cart, dispatch, items }) {
        setThankYouOpen(true);
      }).catch((error) => {
        console.error(error);
+       setApiError(error?.response?.data?.error || 'Unknown Error');
      });
    };

@@ -83,6 +85,10 @@ function Cart({ cart, dispatch, items }) {
      navigate('/');
    };

+   const closeApiErrorModal = () => {
+     setApiError('');
+   };
+
    return (
      <div className="cart-component">
        <Modal
@@ -96,6 +102,17 @@ function Cart({ cart, dispatch, items }) {
            Return Home
          </button>
        </Modal>
+       <Modal
+         isOpen={!!apiError}
+         onRequestClose={closeApiErrorModal}
+         style={customStyles}
+         contentLabel="There was an error"
+       >
+         <p>There was an error submitting your order.</p>
+         <p>{ apiError }</p>
+         <p>Please try again.</p>
+         <button onClick={closeApiErrorModal} type="button">Ok</button>
+       </Modal>
        <h2>Your Cart</h2>
        {(cart.length === 0
          ? <div>Your cart is empty.</div>
```

For this modal, a string to determines if it should open. If there is an API Error in state, show the modal with the error. If not, hide the modal.

When setting the API error, the code uses optional chaining again to try to get the error message from the server. If there is not an error message, it uses the generic message "Unknown Error" instead.

`!!` converts any falsy/truthy value to its respective boolean. Empty string is false, any other string value is true.

Try to submit another order with "99999". You will get a modal with an error.



Change the zip code to something else and submit your order. It worked!

🔎 Boolean Type Conversion

Javascript does unexpected things converting types sometimes. It's always a good idea to verify how things will convert in the console.

For more reading on see:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_NOT
- https://developer.mozilla.org/en-US/docs/Glossary/truthy
- https://developer.mozilla.org/en-US/docs/Glossary/falsy

Conclusion

Users can now submit orders to the API and see error messages if something goes wrong. You are done with the order page. In the next chapter, you will load the items from the API as well.

Fetch Items

You are ready to fetch the items from the API as well, which will make it easier to remove out of stock items and update prices.

Loading Items from the API

To load items from the API, make a GET request to `/api/items`. You will also need to make a state variable to store the items and the `useEffect` hook.

src/App.js CHANGED

```
@@ -1,24 +1,30 @@
+ import axios from 'axios';
  import {
    BrowserRouter as Router,
    Switch,
    Route,
  } from 'react-router-dom';
- import { useCallback } from 'react';
+ import { useCallback, useEffect, useState } from 'react';
  import './App.css';
  import Cart from './components/Cart';
  import Details from './components/Details';
  import Header from './components/Header';
  import Home from './components/Home';
  import NotFound from './components/NotFound';
- import { items } from './items';
  import { CartTypes, useCartReducer } from './reducers/cartReducer';

  function App() {
    const [cart, dispatch] = useCartReducer();
+   const [items, setItems] = useState([]);
    const addToCart = useCallback(
      (itemId) => dispatch({ type: CartTypes.ADD, itemId }),
      [dispatch],
    );
+   useEffect(() => {
+     axios.get('/api/items')
+       .then((result) => setItems(result.data))
+       .catch(console.error);
+   }, []);

    return (
      <Router>
```

Hop back over to your browser. If it worked, "Apple" is now called a "Red Apple" and costs $0.98 instead of $1.00.

The code now uses state to store the items rather than the static items array. In `useEffect`, axios makes a GET request to the API. If the request works, the items are stored in state. If not, an error is logged to the console. Because `console.error` is a function, `.catch(console.error)` is roughly the same as `.catch((error) => console.error(error))`.

🔎 `useEffect` Hook

Reacts effect hook helps replace the previous `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` lifecycle methods, which makes it very powerful.

Some effects require cleanup (like attaching a key listener to the document). Some effects do not (like calling an API).

Some effects should run on every update cycle, but most should only run with certain dependencies have changed or should only run once.

The hook above specifies an empty array of dependencies which means it will run only one time when the component is first mounted (like `componentDidMount`).

You can read more about this hook and see more example uses and code here: https://reactjs.org/docs/hooks-effect.html .

Async/Await

The promises could also have been written with async/await. Let's re-write the code to fetch the items using async/await and try/catch blocks.

src/App.js CHANGED

```
@@ -21,9 +21,15 @@ function App() {
      [dispatch],
    );
    useEffect(() => {
-     axios.get('/api/items')
-       .then((result) => setItems(result.data))
-       .catch(console.error);
+     const fetchData = async () => {
+       try {
+         const result = await axios.get('/api/items');
+         setItems(result.data);
+       } catch (error) {
+         console.error(error);
+       }
+     };
+     fetchData();
    }, []);

    return (
```

The main function for a use effect hook has to be synchronous, so it cannot be `async`. That's why there is a new async `fetchData` function inside the hook.

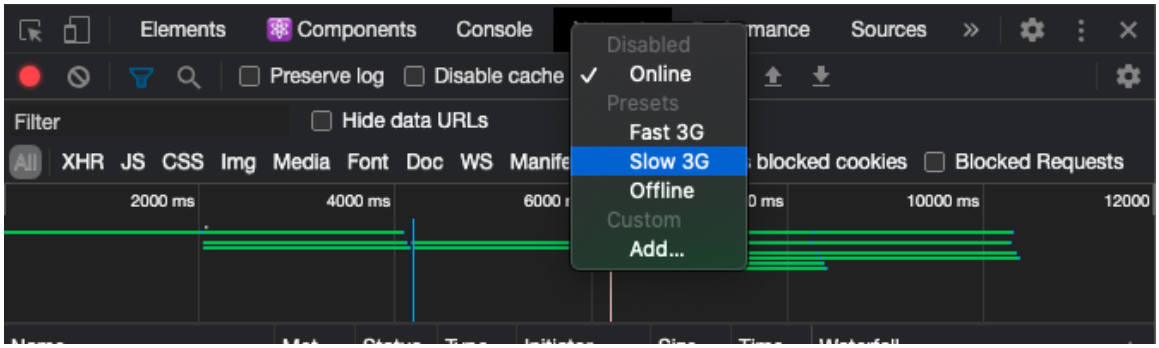
I generally find try/catch blocks easier to read particularly if there are several layers of promises or data processing, but because these calls to fetch data are simple, and the async call requires several more indent levels, so I find them about the same. Which method feels better to you?

We will use promises with `then` and `catch` for simple data fetching in the book, but feel free to implement your code with Async/Await instead.

Show a Loading Message

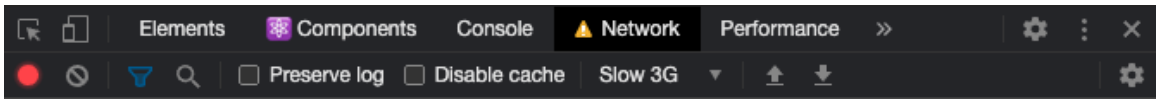
Because every part of the app is running locally on your machine, everything loads really fast, but what if the user has a slow connection?

Chrome devtools let you try this out. Open the network tab of devtools. Click "Online" and select "Slow 3G" in the dropdown.



By clicking Add at the bottom of the menu, you can create other custom profiles to simulate specific user environments.

Once you select Slow 3G, the network tab will show an exclamation point to remind you that you have altered your internet access.



Refresh the page and watch it slowly load everything. For a brief period, you will have a header and a blank area below it while the app fetches the items from the API. It would be nice to show a message while the items are loading instead of just a blank section.

Use a ternary to show a loading message while there are not items.

> **Practice**
>
> See if you can do this without the solution.

src/App.js CHANGED

```
@@ -35,20 +35,24 @@ function App() {
    return (
      <Router>
        <Header cart={cart} />
+       {items.length === 0
+         ? <div>Loading...</div>
```

```
        40  +         : (
  38    -       <Switch>
        41  +           <Switch>
  39    -         <Route path="/cart">
        42  +             <Route path="/cart">
  40    -           <Cart cart={cart} dispatch={dispatch} items={items} />
        43  +               <Cart cart={cart} dispatch={dispatch} items={items} />
  41    -         </Route>
        44  +             </Route>
  42    -         <Route path="/details/:id">
        45  +             <Route path="/details/:id">
  43    -           <Details addToCart={addToCart} items={items} />
        46  +               <Details addToCart={addToCart} items={items} />
  44    -         </Route>
        47  +             </Route>
  45    -         <Route exact path="/">
        48  +             <Route exact path="/">
  46    -           <Home items={items} />
        49  +               <Home items={items} />
  47    -         </Route>
        50  +             </Route>
  48    -         <Route>
        51  +             <Route>
  49    -           <NotFound />
        52  +               <NotFound />
  50    -         </Route>
        53  +             </Route>
  51    -       </Switch>
        54  +           </Switch>
        55  +         )}
  52    56        </Router>
  53    57      );
  54    58    }
```

Refresh the page, and you should see your new loading message briefly before the items load.

> **TIP**
>
> Restore your network to online state. By visiting the network tab and clicking the dropdown with "Slow 3G" and selecting "Online" again.

Conclusion

You are now integrated with the API. In the next section, you will allow users to login to the app. If they are an associate, they will be able to view and complete orders.

🥉 Style the Loading Message

Add some padding around the loading message, so it is not positioned against the side of the browser window.

Alternatively, find a pure CSS loader online and import it into the app. (Pure CSS loaders make lots of cool uses of keyframes and CSS transitions/animations.)

🥈 Items Load Error

Update your App to show an error message if the items fail to load. You could do this with a modal or just replace the content of the app with a message instructing the user to try again later.

> **TIP**
>
> You can create an error to try out your code by stopping the server but leaving the React app running.

Authentication

The next step is allowing users to log in.

The server we provided already contains this functionality, so you just need to build the UI to enable it.

You will build a new route `/login` which displays a form for the user to enter their username and password to login.

You will also add a new section to the header next to the cart which shows the username if they are logged in and a link to the login page if they are not.

Getting the Logged-In User

You may recall from the previous chapter that the API provides a `/api/auth/current-user` endpoint which will return the current user or `{}` if no user is logged in. Let's call that from `App` and save the results into a part of state called `userDetails`.

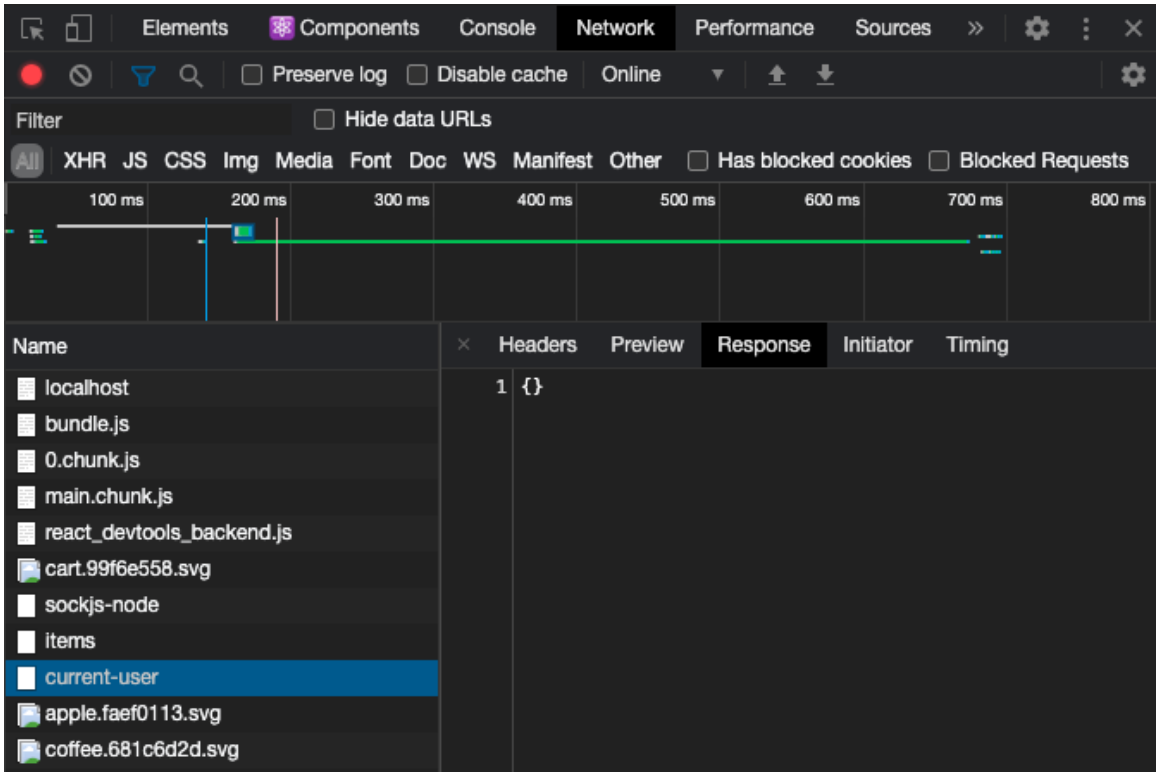
> **Practice**
>
> This one is harder, but try it on your own.
>
> You will need to disable a lint rule (or ignore the lint error).

src/App.js CHANGED

```
@@ -16,6 +16,8 @@ import { CartTypes, useCartReducer } from './reducers/cartReducer';
 function App() {
   const [cart, dispatch] = useCartReducer();
   const [items, setItems] = useState([]);
+  // eslint-disable-next-line no-unused-vars
+  const [userDetails, setUserDetails] = useState({});
   const addToCart = useCallback(
     (itemId) => dispatch({ type: CartTypes.ADD, itemId }),
     [dispatch],
@@ -32,6 +34,12 @@ function App() {
     fetchData();
   }, []);

+  useEffect(() => {
+    axios.get('/api/auth/current-user')
+      .then((result) => setUserDetails(result.data))
+      .catch(console.error);
+  }, []);
+
   return (
     <Router>
       <Header cart={cart} />
```

Initial state for `userDetails` is an empty object `{}`.

Check the network tab of devtools to see the request.



Right now, there is not a user logged in, so the server returned an empty object.

Login Button

You are going to create a new `UserDetails` component to display the user's username or Login link.

> **WARNING**
>
> The login link will currently go to "Page Not Found". You will build the login page later in this chapter.

src/components/UserDetails.css ADDED

```
@@ -0,0 +1,18 @@
+ .user-details-component {
+   display: flex;
+   align-items: center;
+ }
+
+ .user-details-component a {
+   color: #FFF;
+   margin: 0 5px;
+ }
+
```

```
+ .user-details-component button {
+   margin: 0 5px;
+ }
+
+ .user-details-component a + button {
+   margin-left: 0;
+ }
+
```

src/components/UserDetails.js ADDED

```
@@ -0,0 +1,18 @@
+ import { Link } from 'react-router-dom';
+ import './UserDetails.css';
+
+ function UserDetails() {
+   const userDetails = {};
+   return (
+     <div className="user-details-component">
+       { userDetails.username
+         ? (
+           <div>
+             {`Welcome, ${userDetails.username}`}
+           </div>
+         ) : <Link to="/login">Login</Link> }
+     </div>
+   );
+ }
+
+ export default UserDetails;
```

src/components/Header.js CHANGED

```
@@ -1,6 +1,7 @@
  import { Link } from 'react-router-dom';
  import PropTypes from 'prop-types';
  import CartIcon from '../images/cart.svg';
+ import UserDetails from './UserDetails';
  import './Header.css';

  function Header({ cart }) {
@@ -8,6 +9,7 @@ function Header({ cart }) {
    return (
      <header>
        <h1><Link to="/">Coffee Shop</Link></h1>
+       <UserDetails />
        <Link
          className="cart"
          to="/cart"
```

Look at your browser, and you should see a login link in the header.

Some of the CSS above is for features you will add in the future. Also, the `userDetails` object is stubbed out for now.

Passing `userDetails` with Context

The `UserDetails` component needs `userDetails` from the state of `App`. `Header`, however, does not care about the user details. You could still pass the `userDetails` from `App` to `Header` and from `Header` to `UserDetails`, but that starts looking like prop drilling, which is when props are passed through components that do not care about them to get to one that does care about it.

React provides a few ways to help with this, one is component composition and another is context. With the current code, component composition would be more simple, but user information is frequently needed in quite a few components at various levels in an application which is where context really shines so this is a good opportunity to demo context.

src/context/UserContext.js ADDED

```
@@ -0,0 +1,8 @@
+ import { createContext } from 'react';
+
+ const UserContext = createContext({
+   userDetails: { access: '', username: '' },
+   setUserDetails: () => {},
+ });
+
+ export default UserContext;
```

src/App.js CHANGED

```
@@ -4,19 +4,21 @@ import {
    Switch,
    Route,
  } from 'react-router-dom';
+ import {
- import { useCallback, useEffect, useState } from 'react';
+   useCallback, useEffect, useMemo, useState,
+ } from 'react';
  import './App.css';
  import Cart from './components/Cart';
  import Details from './components/Details';
  import Header from './components/Header';
  import Home from './components/Home';
  import NotFound from './components/NotFound';
+ import UserContext from './context/UserContext';
  import { CartTypes, useCartReducer } from './reducers/cartReducer';

  function App() {
    const [cart, dispatch] = useCartReducer();
    const [items, setItems] = useState([]);
-   // eslint-disable-next-line no-unused-vars
    const [userDetails, setUserDetails] = useState({});
    const addToCart = useCallback(
      (itemId) => dispatch({ type: CartTypes.ADD, itemId }),
@@ -40,27 +42,34 @@ function App() {
        .catch(console.error);
    }, []);

+   const userContextValue = useMemo(
+     () => ({ userDetails, setUserDetails }),
+     [userDetails, setUserDetails],
+   );
+
    return (
      <Router>
+       <UserContext.Provider value={userContextValue}>
-       <Header cart={cart} />
+         <Header cart={cart} />
-       {items.length === 0
+         {items.length === 0
-       ? <div>Loading...</div>
+         ? <div>Loading...</div>
```

```
48      -          : (
56      +          : (
49      -          <Switch>
57      +          <Switch>
50      -          <Route path="/cart">
58      +          <Route path="/cart">
51      -          <Cart cart={cart} dispatch={dispatch} items={items} />
59      +          <Cart cart={cart} dispatch={dispatch} items={items} />
52      -          </Route>
60      +          </Route>
53      -          <Route path="/details/:id">
61      +          <Route path="/details/:id">
54      -          <Details addToCart={addToCart} items={items} />
62      +          <Details addToCart={addToCart} items={items} />
55      -          </Route>
63      +          </Route>
56      -          <Route exact path="/">
64      +          <Route exact path="/">
57      -          <Home items={items} />
65      +          <Home items={items} />
58      -          </Route>
66      +          </Route>
59      -          <Route>
67      +          <Route>
60      -          <NotFound />
68      +          <NotFound />
61      -          </Route>
69      +          </Route>
62      -          </Switch>
70      +          </Switch>
63      -          )}
71      +          )}
72      +        </UserContext.Provider>
64      73       </Router>
65      74     );
66      75   }
```

src/components/UserDetails.js CHANGED

```
         @@ -1,8 +1,10 @@
1        + import { useContext } from 'react';
1       2  import { Link } from 'react-router-dom';
3        + import UserContext from '../context/UserContext';
2       4  import './UserDetails.css';
3       5
4       6  function UserDetails() {
5        -   const userDetails = {};
7        +   const { userDetails } = useContext(UserContext);
6       8    return (
7       9      <div className="user-details-component">
8       10       { userDetails.username
```

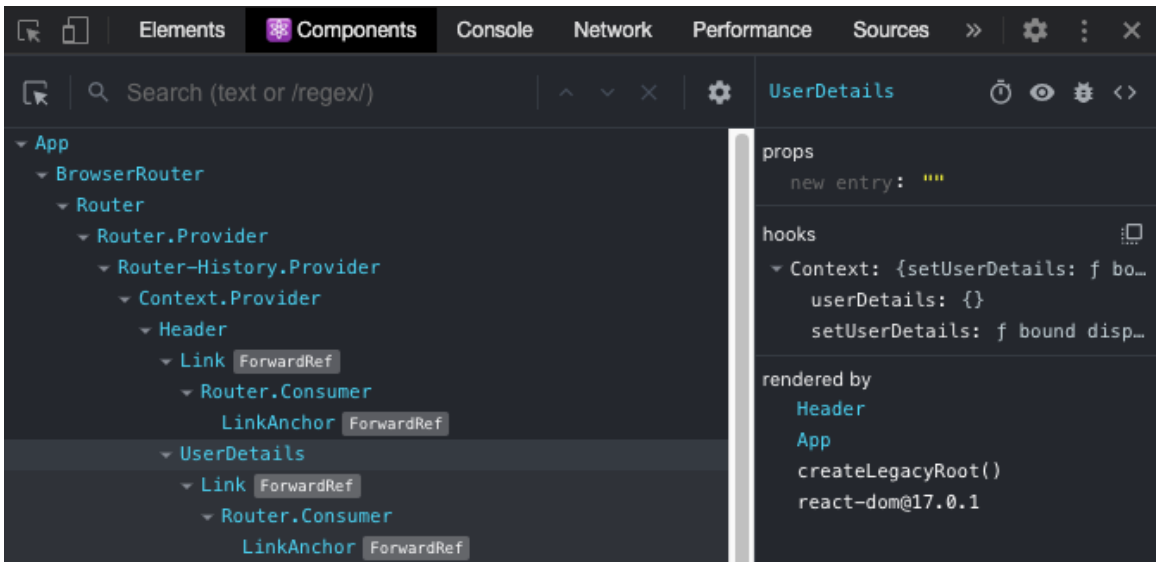
The React context object, created in a new file, will hold the method to set user details and the current user details value. (If user details change frequently, an optimization would be using two providers, one with the current value and the other with the method to set the value, this would avoid re-renders of components which only care about setting user details.)

`App` provides values with a UserContext Provider. Since it wraps the other code, ESLint forced you to indent the contents.

Because React uses object equality `===` rather than deep comparisons, you needed to `useMemo` so that the object containing `{ setUserDetails, userDetails }` is only recreated if a value changes. Otherwise, a new object would be created on each render cycle.

In the `UserDetails` component, the `useContext` hook gets the value of userDetails.

Now, `Header` does not know anything about `userDetails`, and if you check the React Component devtools you can see the values of context in the `UserDetails` component.



You can also see the `Context.Provider` and see the context values there as well.

🔎 React Context

You can read more about context at https://reactjs.org/docs/context.html

🔎 Use Context Hook

The use context hook takes a context object and returns the current value of it from the nearest context provider.

You can read more about the hook at https://reactjs.org/docs/hooks-reference.html#usecontext .

Login Page

Now you are ready to build the login page. It should:

- Be a new component with the route `/login`
- Have a form which requests a username and password and stores the users input in state
- When the user submits the form, make a POST request to `/api/auth/login` with `{ username, password }`
- On success
  - Call `setUserDetails`, which you can get from context, with `result.data`. (Similar to the request for `/api/auth/current-user`.)

    - Then, redirect the user to the home page.
- In the error case, log to console. (The next exercise will be to create a modal.)

You can use any username you want. The password is always `pass`. Username: `BigNerdRanch` and Password: `pass` would work. Username: `BigNerdRanch` and Password: `invalid` would not work.

> **Practice**
>
> That's a lot of changes, but we think you can do it.
>
> Try not to look at the solution and ask your instructor if you need help with something.

Got it? Okay, compare your code to how we built it. It's okay if you built it differently than we did.

If you have questions about differences, ask your instructor.

src/components/Login.css ADDED

```
@@ -0,0 +1,12 @@
+ .login-component {
+   padding: 10px;
+ }
+
+ .login-component label {
+   display: block;
+ }
+
+ .login-component form input {
+   margin: 5px;
+   padding: 2px;
+ }
```

src/components/Login.js ADDED

```
@@ -0,0 +1,59 @@
+ import { useContext, useState } from 'react';
+ import axios from 'axios';
+ import { useHistory } from 'react-router-dom';
+ import UserContext from '../context/UserContext';
+ import './Login.css';
+
+ function Login() {
+   const { setUserDetails } = useContext(UserContext);
+   const [username, setUsername] = useState('');
+   const [password, setPassword] = useState('');
+   const history = useHistory();
+
+   const login = (event) => {
+     event.preventDefault();
+     axios.post('/api/auth/login', {
+       username,
+       password,
+     }).then((result) => {
+       setUserDetails(result.data);
+       history.push('/');
+     }).catch((error) => {
+       console.error(error);
+     });
+   };
+
+   return (
+     <div className="login-component">
+       <form onSubmit={login}>
```

```
+         <div>
+           <label htmlFor="username">
+             Username:
+             <input
+               type="text"
+               id="username"
+               value={username}
+               onChange={(event) => setUsername(event.target.value)}
+               required
+             />
+           </label>
+         </div>
+         <div>
+           <label htmlFor="password">
+             Password:
+             <input
+               type="password"
+               id="password"
+               value={password}
+               onChange={(event) => setPassword(event.target.value)}
+               required
+             />
+           </label>
+         </div>
+         <div><button type="submit">Login</button></div>
+       </form>
+     </div>
+   );
+ }
+
+ export default Login;
```

src/App.js CHANGED

```
@@ -12,6 +12,7 @@ import Cart from './components/Cart';
  import Details from './components/Details';
  import Header from './components/Header';
  import Home from './components/Home';
+ import Login from './components/Login';
  import NotFound from './components/NotFound';
  import UserContext from './context/UserContext';
  import { CartTypes, useCartReducer } from './reducers/cartReducer';
@@ -61,6 +62,9 @@ function App() {
                  <Route path="/details/:id">
                    <Details addToCart={addToCart} items={items} />
                  </Route>
+                 <Route path="/login">
+                   <Login />
+                 </Route>
                  <Route exact path="/">
                    <Home items={items} />
                  </Route>
```

Try logging in with an invalid username and password (ex `BigNerdRanch` and `invalid`). You should see an error in your console.

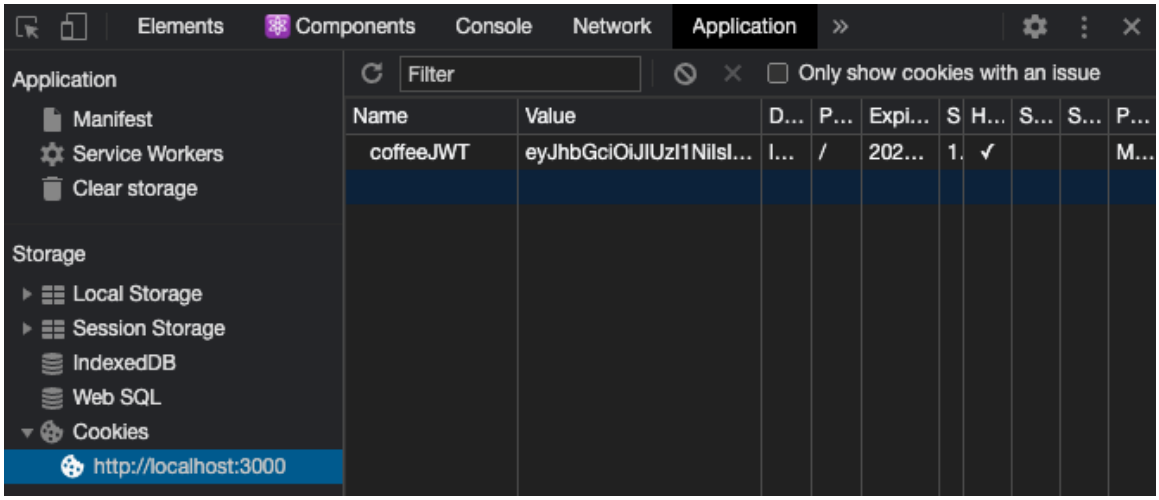
Now, login with a valid username and password (ex `BigNerdRanch` and `pass`). You should get sent to the home page and see your username in the upper right corner of the browser window.

Because the server uses a cookie for the login, you can refresh the page, and you will stay logged in.

🔎 Login Cookie

You can find the login cookie in the application tab of devtools.

Look under storage, then cookies, and then `http://localhost:3000`. You should see a cookie named `coffeeJWT`.



Chrome will let you edit the cookie value, or you can click on it and press delete to delete the cookie.

Once you delete it, refresh the page, and you will be logged out.

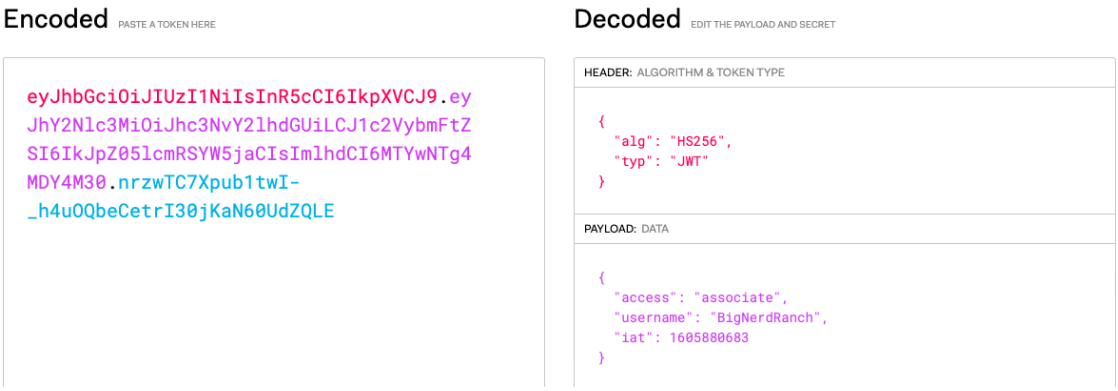
🔎 JSON Web Token (JWT)

The name of the cookie ends with JWT which is a clue that it is a JSON Web Token.

JSON Web Tokens store a chunk of header data, a payload, and a signature. The three sections are separated by a period `.` and the contents of each section is Base64URL encoded. It is easy to edit the header data or payload locally, but you would need to know the signing secret to update the signature. The React app does not know the secret, so it cannot verify the signature. It has to rely on the server to enforce authorization. The token is signed with a string based secret so putting the secret in the front end would publicly expose it in the source code. You can also use public/private keys for secret generation in which case you could safely bundle the private key into the source of the React app. In the end, the server should always be doing authorization checks, and you should never have any secret URLs/etc in the source of the front end, so it does not matter if a user makes themself an admin in React and can see the buttons for admin tasks as long as the server prevents them from actually doing admin tasks.

If you logged out previously, go ahead and log back in. Then, copy the value of the cookie by double clicking the value and pressing `Cmd + C` on Mac or `Ctrl + C` on Windows.

Open https://jwt.io . Scroll slightly down the page to the "Debugger" section, remove the current token in the "encoded" section on the left side and paste your token in place of it.



Here is the decoded JWT token. Do not worry that it says invalid signature at the bottom. This is because the debugger does not know the secret so it cannot verify the signature.

If you want to learn more, the website has a nice introduction section: https://jwt.io/introduction/ .

Logout

For better security, add a logout button to the header when the user is logged in.

The logout button should:

- Make a post request to `/api/auth/logout`
- `setUserDetails` to an empty object
- Redirect the user to the home page

This way even after refreshing the user should still be logged out. If you just call `setUserDetails`, you will be logged in again after a refresh because the cookie will still be there.

> **Practice**
>
> Give this a try on your own

src/components/UserDetails.js CHANGED

```
@@ -1,16 +1,33 @@
+ import axios from 'axios';
  import { useContext } from 'react';
- import { Link } from 'react-router-dom';
+ import { Link, useHistory } from 'react-router-dom';
  import UserContext from '../context/UserContext';
  import './UserDetails.css';
```

```
 6   7  function UserDetails() {
 7     -  const { userDetails } = useContext(UserContext);
       8  +  const history = useHistory();
       9  +  const { userDetails, setUserDetails } = useContext(UserContext);
      10  +
      11  +  const logout = () => {
      12  +    axios.post('/api/auth/logout', {})
      13  +      .then(() => {
      14  +        setUserDetails({});
      15  +        history.push('/');
      16  +      })
      17  +      .catch((error) => {
      18  +        console.error(error);
      19  +      });
      20  +  };
      21  +
 8   22    return (
 9   23      <div className="user-details-component">
10   24        { userDetails.username
11   25          ? (
12   26            <div>
13   27              {`Welcome, ${userDetails.username}`}
      28  +            <button type="button" onClick={logout}>
      29  +              Logout
      30  +            </button>
14   31            </div>
15   32          ) : <Link to="/login">Login</Link> }
16   33      </div>
```

Nice, now there is a logout button. Click that, and you will be logged out.

Login Error Modal

As promised above, your next task is to show the login error to the user in a modal.

The server returns the error in the same place as the other APIs.

> **Practice**
>
> Can you build it without the solution?

src/components/Login.js CHANGED

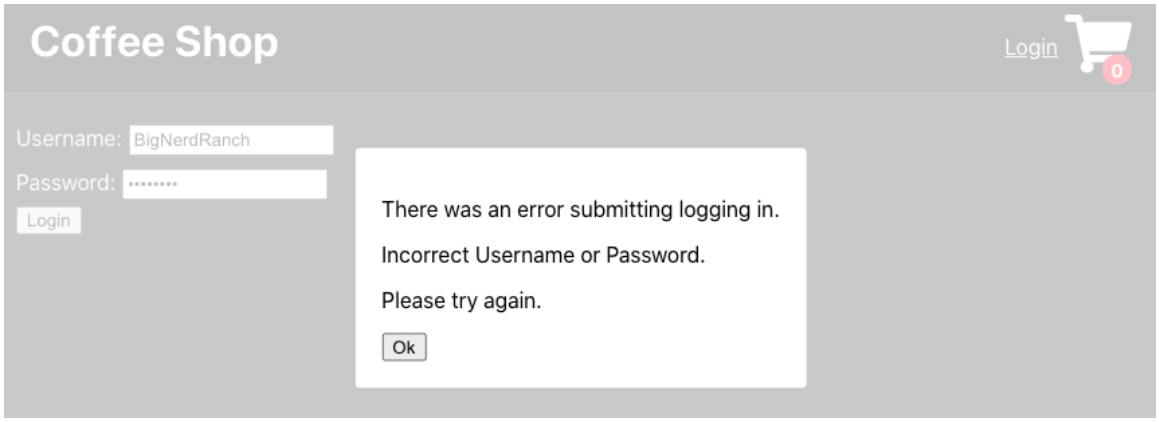
```
@@ -1,13 +1,29 @@
 1   1  import { useContext, useState } from 'react';
 2   2  import axios from 'axios';
     3  + import Modal from 'react-modal';
 3   4  import { useHistory } from 'react-router-dom';
 4   5  import UserContext from '../context/UserContext';
 5   6  import './Login.css';
 6   7
     8  + const customStyles = {
     9  +   content: {
    10  +     top: '50%',
    11  +     left: '50%',
    12  +     right: 'auto',
    13  +     bottom: 'auto',
    14  +     marginRight: '-50%',
    15  +     transform: 'translate(-50%, -50%)',
    16  +     color: '#000',
    17  +   },
    18  + };
    19  +
```

```
+ Modal.setAppElement('#root');
+
  function Login() {
    const { setUserDetails } = useContext(UserContext);
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');
+   const [apiError, setApiError] = useState('');
    const history = useHistory();

    const login = (event) => {
@@ -20,11 +36,28 @@ function Login() {
        history.push('/');
      }).catch((error) => {
        console.error(error);
+       setApiError(error?.response?.data?.error || 'Unknown Error');
      });
    };

+   const closeApiErrorModal = () => {
+     setApiError('');
+     setPassword('');
+   };
+
    return (
      <div className="login-component">
+       <Modal
+         isOpen={!!apiError}
+         onRequestClose={closeApiErrorModal}
+         style={customStyles}
+         contentLabel="Login Error"
+       >
+         <p>There was an error submitting logging in.</p>
+         <p>{ apiError }</p>
+         <p>Please try again.</p>
+         <button onClick={closeApiErrorModal} type="button">Ok</button>
+       </Modal>
        <form onSubmit={login}>
          <div>
            <label htmlFor="username">
```

If you logged in before, you will need to delete the login cookie as explained above or use an incognito window, so you are logged out.

Try to log in with an invalid password.

You will get an error modal.



Conclusion

Yay! Users can log in now. In the next chapter, you will show associates a page to view the orders and complete them.

Fulfill Orders

Now that users can log in, you are ready to show the orders page to associates.

Create the Orders Component

Let's create a new component to hold the orders at the route `/orders`.

src/components/Orders.css ADDED

```
@@ -0,0 +1,9 @@
+ div.orders-component {
+   padding: 5px;
+ }
+
+ div.orders-component div.order {
+   border: 1px solid white;
+   margin: 10px 5px;
+   padding: 5px;
+ }
```

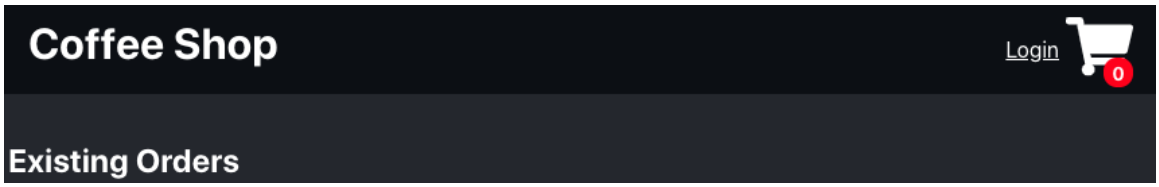
src/components/Orders.js ADDED

```
@@ -0,0 +1,11 @@
+ import './Orders.css';
+
+ function Orders() {
+   return (
+     <div className="orders-component">
+       <h2>Existing Orders</h2>
+     </div>
+   );
+ }
+
+ export default Orders;
```

src/App.js CHANGED

```
@@ -14,6 +14,7 @@ import Header from './components/Header';
  import Home from './components/Home';
  import Login from './components/Login';
  import NotFound from './components/NotFound';
+ import Orders from './components/Orders';
  import UserContext from './context/UserContext';
  import { CartTypes, useCartReducer } from './reducers/cartReducer';

@@ -65,6 +66,9 @@ function App() {
                  <Route path="/login">
                    <Login />
                  </Route>
+                 <Route path="/orders">
+                   <Orders />
+                 </Route>
                  <Route exact path="/">
                    <Home items={items} />
                  </Route>
```

Visit http://localhost:3000/orders . Soon you will see orders here.



Link to Orders

Associates should see a link to the orders page in the header when they are logged in.

The access level is provided in `userDetails.access`.

> **TIP**
>
> There is also a guest user which does not have associate level access. Use the username `Guest` and the password `pass`.
>
> All other users have associate level access.

> **Practice**
>
> Try to do add the link on your own.

src/components/UserDetails.js CHANGED

```
@@ -25,6 +25,9 @@ function UserDetails() {
25 25            ? (
26 26              <div>
27 27                {`Welcome, ${userDetails.username}`}
28 +               {userDetails.access === 'associate'
29 +                 ? <Link to="/orders">Orders</Link>
30 +                 : null}
28 31              <button type="button" onClick={logout}>
29 32                Logout
30 33              </button>
```

The new thing here is the ability to return `null` from one side of the ternary. React ignores the null and renders nothing.

You can also use and for short-circuiting like this:

```
{userDetails.access === 'associate' && <Link to="/orders">Orders</Link>}
```

React will not render the `false` that returns from that statement.

Display Orders

Now, you need to fetch the orders from the API and display them.

GET `/api/orders` will return an array of orders where each order contains `{ name, phone?, zipCode, items[] }`. Phone is optional and items is an array of items which is the same as the cart with the item id and quantity.

You will also display "No Orders" if the order array is empty.

Note: You will want to store the orders in state.

> **Practice**
>
> Feel free to try this on your own first.

src/App.js CHANGED

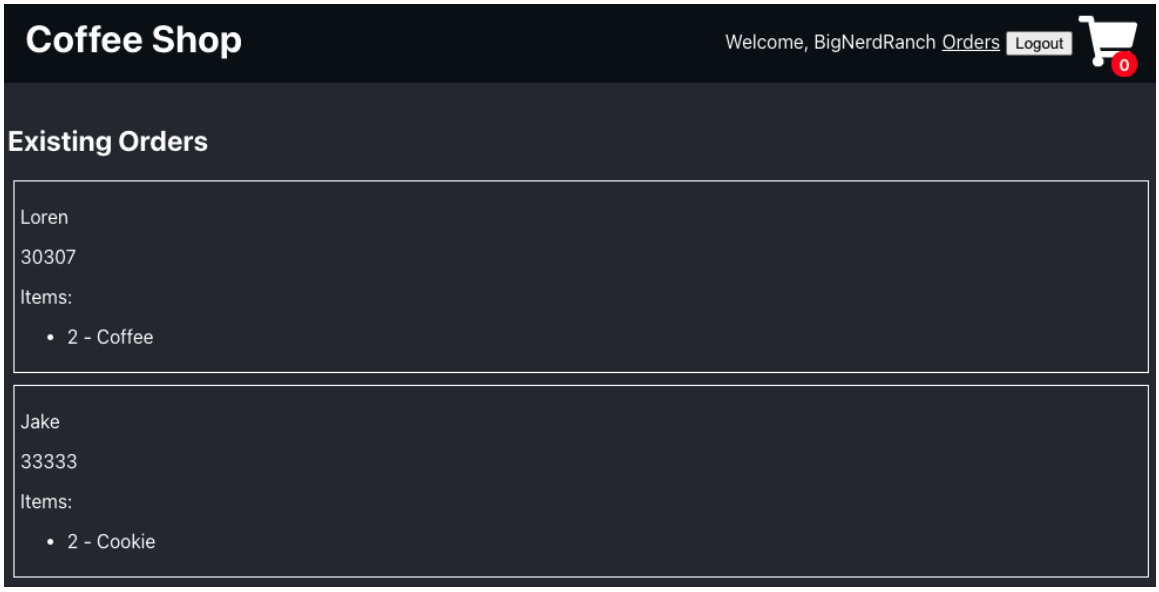
```
@@ -67,7 +67,7 @@ function App() {
                  <Login />
                </Route>
                <Route path="/orders">
-                 <Orders />
+                 <Orders items={items} />
                </Route>
                <Route exact path="/">
                  <Home items={items} />
```

src/components/Orders.js CHANGED

```
@@ -1,11 +1,53 @@
+ import axios from 'axios';
+ import { useEffect, useState } from 'react';
  import './Orders.css';
+ import PropTypes from 'prop-types';
+
+ function Orders({ items }) {
+   const [orders, setOrders] = useState([]);
+   useEffect(
+     () => {
+       axios.get('/api/orders')
+         .then((result) => setOrders(result.data))
+         .catch(console.error);
+     },
+     [],
+   );

- function Orders() {
    return (
      <div className="orders-component">
        <h2>Existing Orders</h2>
+       {orders.length === 0
+         ? <div>No Orders</div>
+         : orders.map((order) => (
+           <div className="order" key={order.id}>
+             <p>{order.name}</p>
+             {order.phone ? <p>{order.phone}</p> : null}
+             <p>{order.zipCode}</p>
+             <p>Items:</p>
+             <ul>
+               {order.items.map((item) => (
+                 <li key={item.id}>
+                   {item.quantity}
+                   {' '}
+                   -
+                   {' '}
+                   {items.find((i) => i.id === item.id).title}
+                 </li>
```

```
+                ))}
+              </ul>
+            </div>
+          ))}
        </div>
      );
    }

+ Orders.propTypes = {
+   items: PropTypes.arrayOf(PropTypes.shape({
+     id: PropTypes.string.isRequired,
+     title: PropTypes.string.isRequired,
+     price: PropTypes.number.isRequired,
+   })).isRequired,
+ };
+
  export default Orders;
```

Place a couple of orders. Then, visit the Orders page to view them.



Fulfilling Orders

You are going to provide a button to associates to delete orders when they have fulfilled them.

Clicking the button should make a `DELETE` call to `/api/orders/:id`.

Hint: `axios.delete` will make a call with the `DELETE` method. It accepts a string of the URL. You can use a template literal (backticks) to dynamically create a URL with the order ID at the end.

If the request is successful, the Order component should load the orders again. If not, log the error to the console.

> **Practice**
>
> Try this on your own.

src/components/Orders.js CHANGED

```
@@ -5,14 +5,18 @@ import PropTypes from 'prop-types';

 function Orders({ items }) {
   const [orders, setOrders] = useState([]);
-  useEffect(
-    () => {
-      axios.get('/api/orders')
-        .then((result) => setOrders(result.data))
-        .catch(console.error);
-    },
-    [],
-  );
+  const loadOrders = () => {
+    axios.get('/api/orders')
+      .then((result) => setOrders(result.data))
+      .catch(console.error);
+  };
+  useEffect(loadOrders, []);
+
+  const deleteOrder = (order) => {
+    axios.delete(`api/orders/${order.id}`)
+      .then(loadOrders)
+      .catch(console.error);
+  };

   return (
     <div className="orders-component">
@@ -36,6 +40,12 @@ function Orders({ items }) {
                   </li>
                 ))}
               </ul>
+              <button
+                type="button"
+                onClick={() => deleteOrder(order)}
+              >
+                Delete Order
+              </button>
             </div>
           ))}
       </div>
```

To avoid duplicate code, the code for loading orders now lives in a separate function.

Try deleting one of the orders you placed. If you delete them all, the page should say "No Orders".

Red Button

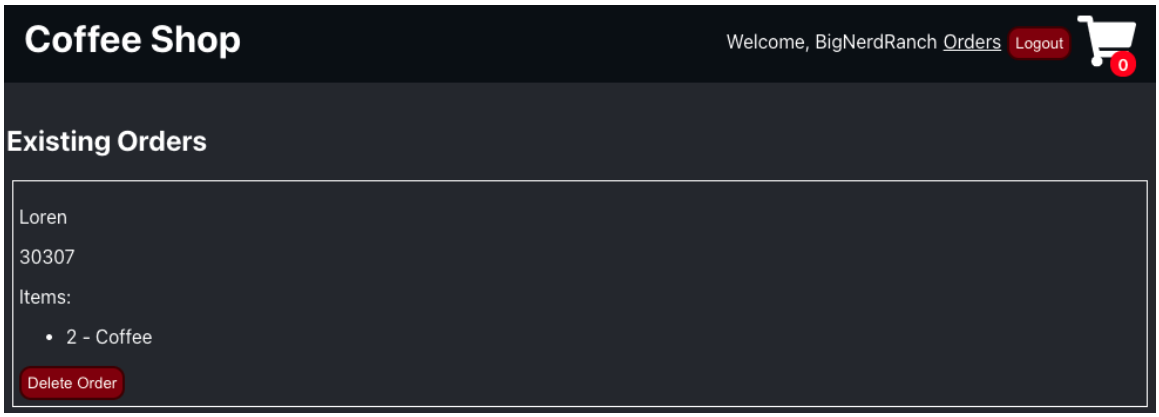
Deleting orders is not reversible. The designers want you to change the delete button to be red so users know it's dangerous to click. They give you the following mock to implement.

> **Practice**
>
> Try out your CSS skills.

src/components/Orders.css CHANGED

```
@@ -7,3 +7,11 @@ div.orders-component div.order {
    margin: 10px 5px;
    padding: 5px;
  }
+
+ button {
+   background-color: #880000;
+   color: #fff;
+   padding: 5px;
+   border-radius: 10px;
+   border: 2px solid #440000;
+ }
```

Nice! The button is styled.



Scoping Bug

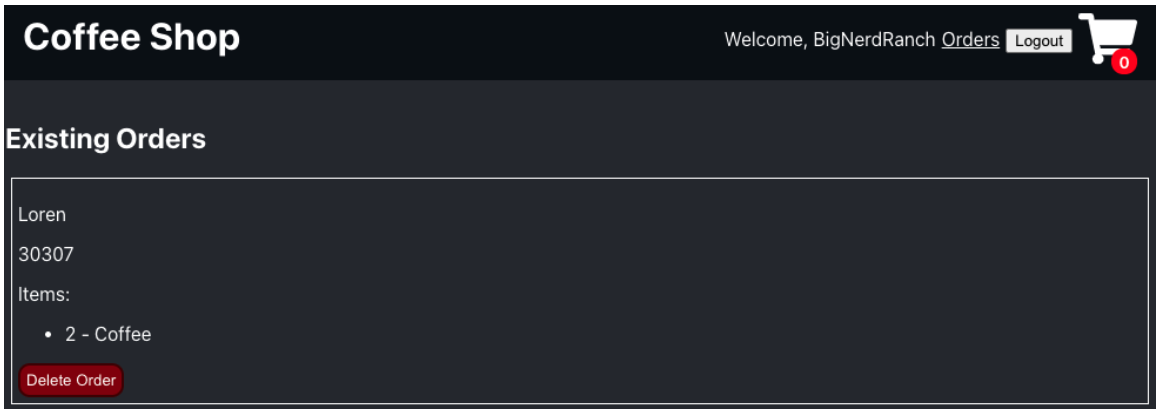
Your code may not have had an issue, but if you used the code above, the Login/Logout button is now also red. What happened?

Even though that button is part of a different component, all the CSS is merged together as part of the React built. Scope the changes to just the order component.

src/components/Orders.css CHANGED

```
@@ -8,7 +8,7 @@ div.orders-component div.order {
    padding: 5px;
  }

- button {
+ div.orders-component button {
    background-color: #880000;
    color: #fff;
    padding: 5px;
```

That's better only delete order is red now.



Access

The server rejects order deletions and modifications from users who are not associates.

Log out and place an order. Then, visit http://localhost:3000/orders .

Even though you are not logged in, you can still see the list of orders. This is because we wanted you to be able to see the list of orders by visiting http://localhost:3030/api/orders in a previous chapter.

Now try to delete an order. Nothing happens, but if you check the console or network tab, you will see that the request was rejected with the code `401` (Unauthorized). You will get the same error if you try to delete an order while logged in as a guest.

Getting New Orders

The order page has a problem. It does not update with new orders. Open the application in two tabs and view the order page in one. In the other tab, place an order. The tab with the order page will not show the new order until you refresh it.

You could solve this by polling the server at specified intervals, but websockets are a great way to subscribe to updates. That way the browser gets an update each time there is one and does not have to poll.

Let's switch to using a websocket.

Proxy the Websocket

Similar to the proxy for the API, we'll also proxy the websocket connection.

This is not built into the previous proxy configuration, so you will need to manually configure the proxy.

First, you will need to install a new package.

> **Install**
>
> In a separate terminal from your app, CD into your `coffee-shop` directory.
>
> Then, install http-proxy-middleware `npm install --save http-proxy-middleware@2.0.1`
>
> We've selected this version because it is the version already bundled in the dev server.

Now, you can create a custom proxy.

src/setupProxy.js ADDED

```
@@ -0,0 +1,8 @@
+ const { createProxyMiddleware } = require('http-proxy-middleware');
+
+ module.exports = (app) => {
+   app.use(createProxyMiddleware(
+     ['/api', '/ws'],
+     { target: 'http://localhost:3030', changeOrigin: true, ws: true },
+   ));
+ };
```

package.json CHANGED

```
@@ -23,7 +23,6 @@
      "lint": "eslint src --max-warnings=0",
      "eject": "react-scripts eject"
    },
-   "proxy": "http://localhost:3030",
    "husky": {
      "hooks": {
        "pre-commit": "npm run lint"
```

Create React App will automatically execute in `src/setupProxy.js` once you restart the app.

The dev server inside Create React App uses Express. `app.use` adds a piece of middleware. In this case, that middleware is a proxy from the `http-proxy-middleware` package. The first parameter of `createProxyMiddleware` is the path it should use. The API requests are at `/api` and the websocket will be `/ws`. This code matches both. The second parameter is the options: `target` is the proxy destination, `ws` lets it know to also proxy websocket requests, and `changeOrigin` changes the origin of the host header to the target URL. `changeOrigin` is necessary for name-based virtual hosted sites, and it's easiest to include it.

You can read more about the options in the documentation: https://github.com/chimurai/http-proxy-middleware

> **Restart React Coffee Shop**
>
> If you are currently running Coffee Shop, you will need to restart for this change to take effect. You can do that by killing the currently running process for `coffee-shop` with `Ctrl + C`, and starting it again with `npm start`.

> Assuming you are in the terminal where you had just run it, you can get to the most recently run command by pressing the up arrow. Then, press enter to run it again. Bash also provides a shortcut to the most recent command without needing the up arrow `!!`.

Using the Websocket

src/components/Orders.js CHANGED

```
@@ -5,16 +5,31 @@ import PropTypes from 'prop-types';
 
 function Orders({ items }) {
   const [orders, setOrders] = useState([]);
-  const loadOrders = () => {
-    axios.get('/api/orders')
-      .then((result) => setOrders(result.data))
-      .catch(console.error);
-  };
-  useEffect(loadOrders, []);
+  useEffect(() => {
+    const ws = new WebSocket(`${(
+      window.location.protocol === 'https:' ? 'wss://' : 'ws://'
+    )}${window.location.host}/ws`);
+    ws.onopen = () => {
+      console.log('connected');
+    };
+    ws.onerror = (event) => {
+      console.error(event);
+    };
+    ws.onmessage = (message) => {
+      const newOrders = JSON.parse(message.data);
+      setOrders(newOrders);
+    };
+    ws.onclose = () => {
+      console.log('disconnected');
+    };
+
+    return () => {
+      ws.close();
+    };
+  }, []);
 
   const deleteOrder = (order) => {
     axios.delete(`api/orders/${order.id}`)
-      .then(loadOrders)
       .catch(console.error);
   };
```

The code above opens and sets up a websocket (more details on that in a second). The websocket needs to be closed when the user navigates away from the order page. Otherwise, there would be a memory leak because the websocket would stay open and a new one would be opened if the user came back to the orders page later.

React accepts a cleanup function returned from `useEffect`. If a dependency changes, React calls the cleanup function before it runs the main effect again. If the component is about to be destroyed, it will also call the cleanup function. In this case, there are no dependencies, so the cleanup function is only called when the user navigates to another page and the `Order` component gets destroyed.

The URL to connect to the websocket looks a bit crazy. It is a template literal. The first expression in it determines the protocol for the web socket request. If the user is on `https://` it will was a secure

websocket `wss://`, otherwise, it uses `ws://`. The next expression `window.location.host` is the hostname and port (if non-empty). On your computer, this will be `localhost:3000`. Finally, `/ws` is the path for the websocket proxy set up above and the path for the websocket server on the server we provided in the last chapter.

You can read more about `window.location.host` at: https://developer.mozilla.org/en-US/docs/Web/API/Location/host

Try the experiment from "Getting New Orders" with two tabs again. The order page immediately updates with the new order!

🔎 Websocket API

The WebSocket API opens a WebSocket connection.

Once the web socket exists, it can be told what to do `onopen` (the initial connection), `onerror` (an error connecting), `onclose` (when it disconnects), and `onmessage` (when it gets a message). For most of them, the code logs the event to the console to help with debugging, but `onmessage` is important.

The server sends a message via the websocket with the list of orders right after the websocket connects and when orders are created, updated, or deleted. The message is sent as a string which has been JSON encoded, so the code needs to parse the JSON message before setting orders to the updated ones from the server.

🔎 Websocket API

Learn more at: https://developer.mozilla.org/en/docs/Web/API/WebSocket

Conclusion

Now associates can view and fill the orders! The application is nearly done.

We have provided several challenges below. Pick one or two to try.

Remember to make a backup of your code in case you run into issues and need to come back to the solution without your challenge code.

🥉 Smarter Redirect

Currently, all users are sent to the home page after they log in. Associates are usually checking the orders when they log in rather than trying to place an order. Edit the `Login` component so if a user is

an associate it redirects them to the orders page. The `Guest` user should still redirect to the home page.

🥈 Loader

Right now, the user sees "No Orders" while the orders are loading even if there are actually orders that will load.

Update the code to store a boolean state variable "loading" and display "Loading" while the orders are loading and only after they have loaded show "No Orders" if the server returned an empty array.

🥇 Check Access

Right now, you are relying on the server to reject requests for un-authenticated users, but you could check user details before making the server call.

Edit the `Orders` component to check `userDetails.access` and make sure the user has `associate` access. If they do not, display an error message that links them to the login page.

You can also check the access level before creating the effect with `useEffect` so the API to fetch orders is never called unless React thinks they have access.

With security, it is always important for the server to check access because users can edit the javascript, edit the JWT token, or use postman to make their own calls to the API. Checking in React gets non-malicious users faster feedback about what they may have missed. In this case, it is likely that an associate bookmarked the orders page, but their session timed out, so they end up on the orders page logged out.

🥇 Completing Orders

The API also has an `isComplete` field which is a boolean and lives at the same level as `name` and `zipCode`.

Change your code to make a PUT request to `/api/orders/:id` with the order and set `isComplete` to true.

For completed orders, render a short one line summary.

You can also strike through the summary with `<del>` or use other CSS formatting.

You can supply a conditional `className` with a ternary.

```
className={`order ${order.isComplete ? 'completed' : ''}`}
```

🥇 Confirmation

Add a modal when the user clicks to delete an order that confirms they are sure they want to delete the order.

The user should be able to say "Ok" to delete the order or cancel the action.

Testing Overview

You will finish Coffee Shop by adding tests to make sure things are working properly. Normally, we recommend developing tests at the same time as the feature. In this case, we saved it until the end because we wanted to allow you to focus on learning one thing at a time. Now that you have learned how to write React code, use hooks, and integrate with an API. You are ready to learn to test them.

This chapter introduces the types of testing, and the trade-offs between them. Then, you will implement unit tests for your application in the next chapter. After that, you will implement end-to-end tests for it with Cypress.

Questions

To decide what type of testing to use, it is helpful to first ask yourself a few questions.

Why are you testing?

The first question looks into your initial motivations for testing. Sometimes the answer is that your boss or team lead said to write tests, so you are doing it. Other times, people write tests mostly as an insurance policy, so they do not have to take them blame for production bugs. In either of these motivations, it's easy to write tests that create high code coverage from automated code inspections but would not actually catch bugs.

One reason to introduce tests to a previously un-tested project might be that the project is about to undergo refactoring, and you want to be sure it still works the same after the refactors as it did before. In this case, you could guide how you write the tests to ensure you are testing endpoints that you expect to behave the same before and after the refactor. For example, you probably should not write a unit test for a custom checkbox component if a part of the refactor is switching to a new checkbox component from a library. You would instead want to write a higher level test.

Another reason to test might be to create dependable code from the beginning of a project.

What do you want to know?

What you want to know from your tests affects how many of them you write and at what level of the application you choose to write them. Some possibilities here might be:

- The basic happy paths work
- Every happy and unhappy (error) path works
- The parts work in isolation

- The parts work together
- The design looks right
- The validators/computed values work

An important word of warning is that trying to test every possible path through an application is rarely possible due to path explosion. Imagine there are 2 ways to add an item to the cart, a user could edit or not edit the cart, checkout with Visa or PayPal, and the transaction could be approved or declined. That's `2 * 2 * 2 * 2` or `2^4` for a total of 16 paths. 16 paths is potentially testable, but the paths did not include the user searching for the items or needing to go back in the checkout flow, so it quickly becomes impossible to test everything as the complete flow.

What does the future look like?

Are you planning to start a refactor? Do you need to be ready to pivot the app in a new direction because you are current in a discovery phase? Alternatively, the application might be just adding incremental functionality or a legacy application that is purely receiving bug fixes. As mentioned above in "Why are you testing?", testing granularly can lead to many testing refactors if you refactor the item you were testing. Granular testing does, however, come with the advantage that it gives a very specific point of failure when the test fails.

Types of Testing

Now, let's look at the different types of testing in more detail.

Static Testing

Static tests come in as the first line of defense. In the coffee shop, you already have some static testing. `ESLint` helps serve this purpose. Using `TypeScript` with React adds stronger typechecking.

Advantages

- Fast
- Gives you an exact line of failure
- Setup once and it's done for ESLint
- Built in to some languages such as TypeScript

Disadvantages

- ESLint mixes code style and code quality checks, so it can be hard to tell which is which

Unit Testing

There are quite a few tests that can be run by the `Jest` testing system. They all fall broadly under the Unit Testing category. Each is discussed individually below.

Unit Testing - Functions

Tests can be written which test a single function in the code. This function would take in some number of parameters and return some value but would not be returning JSX/HTML. An example might be a tax calculation function which accepts an array of items in the cart and the user's location and returns the tax amount.

Advantages

- Fast and Isolated to a single function
- Easy to find the source of failure

Disadvantages

- If you decide to refactor later, you potentially have to refactor the tests as well.

Now image refactoring such that there is a function to compute the total cost of the items and then the tax function accepts the total and the location. The resulting tax amount the user sees is the same, but you would need to refactor the tests with new inputs to get the expected output values.

Unit Testing - Shallow

Shallow mounting allows you to render only one component and not render any of its child components. It would, however, run all functions contained inside the component. You can do this in React with Jest and Enzyme.

Advantages

- Fast, isolated to one component
- Easy to find the source of failure

Disadvantages

- Refactoring requires test changes (ie if you decide to extract part of a component into a child component)
- Fake DOM means the page is not drawn so partially or fully obscured buttons are still clickable

Integration Testing - Full Render

This is the standard mode of testing provided by React Testing Library. This renders the component and all it's children into HTML nodes.

Advantages

- Fast and somewhat isolated depending on how many child components exist
- Extracting code to a sub-component is unlikely to require test changes

Disadvantages

- Cannot test the data passed between child components
- Finding source of failure requires some knowledge of the component tree
- Fake DOM means the page is not drawn so partially or fully obscured buttons are still clickable
- May need to mock APIs

Do not be afraid of the longer list of disadvantages of this type of test. The advantage that refactoring is less likely to force you to re-write tests is a big win!

Integration Testing - Full App

This is still React Testing Library, but rather than rendering an individual component. It can render the entire application including the router which allows jest to "navigate" around the application. jsdom cannot actually change pages such as `page1.html` -> `page2.html`, but React uses the History API to show different URLs in the address bar. This means tests can "navigate" between components with React Router in jsdom just like users can in the browser.

Advantages

- Fast and extracting code to a sub-component is unlikely to require test changes
- Can navigate with React Router

Disadvantages

- Cannot test the data passed between child components
- Finding source of failure requires knowledge of the component tree
- Still a fake DOM
- Need to mock APIs

This test can do almost everything an end-to-end test can, but it does it in a fake DOM. With the full render, you might be able to avoid mocking some APIs by providing data via props or context. When you mock the whole application, you will have to also mock out the API which makes test setup more complicated, but these tests will be much faster than end-to-end tests which have to start a browser.

Snapshot Testing

Snapshot tests avoid writing the conditions for truth and instead store the rendered HTML as the source of truth. This means less code to write to verify that the DOM looks correct, but it also means that you are assuming the DOM was correct when you took the first snapshot.

For the author, I do not find these tests very useful. I frequently find bugs while writing the expectations for my test cases. With a snapshot test, I do not have to write expectations, and it's

easy for me to skim over a mistake when I'm verifying the snapshot.

Snapshots can be taken with unit or integration tests.

Advantages: easier to write, snapshot changes get reviewed as part of the PR

Disadvantages: Any HTML change requires a change in the snapshot, snapshot changes can be large make it hard to tell intentional changes from unintentional ones

It is also important to reiterate that a snapshot is not a picture. It's the HTML. This means that a global CSS change could significantly alter how the page looks, but as long as the HTML is the same, the snapshot would not be changed.

End-to-End Testing

End-to-End tests work on a fully rendered page in a real browser. Some popular tools include: Cypress, Selenium WebDriver, Puppeteer, and CodeceptJS.

End-to-end tests in general come with the following advantages and disadvantages.

Advantages

- Refactoring front-end code should not require any test changes
- Can test navigation between pages
- Can test full workflows
- Uses a real browser
- Catches issues with obscured buttons

Disadvantages

- Launching real browsers is slow
- Requires running a web server for the browser to load the page
- Significantly slower than unit tests
- Easy to write flaky tests

The chapter on end-to-end testing includes tips on avoiding flaky tests. A flaky test is a test which passes sometimes and fails other times even though there have not been any code changes to the test or project.

End-to-End Testing - Mocked Back-end

When writing your end-to-end tests some tools, such as Cypress, allow you to mock out requests as a part of your tests. Other tools would require you to write a separate fake back-end to accomplish this goal.

Advantages

- Mocked back-ends are usually fast

Disadvantages

- Have to maintain the mock back-end
- Mock back-end and the live back-end might not have the same API contract leading to failures in production even though the test passed

End-to-End Testing - Live Back-end

The other option for end-to-end tests is to allow them to hit the real back-end code.

Advantages

- No extra back-end to maintain
- Ensures the contract between the front-end and back-end works

Disadvantages

- Live back-end may be slow which will lead to slow end-to-end tests
- Must set up test data and isolate the tests in the live backend
- It can be difficult to know where the error is coming from since the back end is live and the front end is fully rendered as well

Visual/Screenshot Testing

Visual tests take an image based screenshot of the page to compare for style changes. These tests can be written by adding a line to match a screenshot to end-to-end tests. (The exact code varies by testing library.) As a result, they come with all the same advantages and disadvantages of end-to-end tests and can be done with a mocked or live back-end. They do include a few advantages and disadvantages of their own.

Advantages

- Prevents design regressions
- Catches side effects from shared CSS
- Catches slightly obscured elements that might still pass end-to-end tests

Disadvantages

- Anti-aliasing (which helps text and other objects on screen look smooth around the edges) can be rendered differently on different runs and using windows vs Mac or updating Chrome to a newer version can cause objects to shift by a pixel or two on screen. Because of this, a direct pixel by pixel comparison can easily fail a test which should pass.
- To combat pixel shifts, you can allow a tolerance level for a percentage of difference that is okay, but with too much tolerance a border color change might not cause a desired test failure.

- A number of companies have worked on using machine learning to better compare screenshots. Most of this work is only available through paid software-as-a-service offerings which can be expensive or prohibited by larger companies where everything needs to be on site.

Component Testing

You are ready to add tests to your project. The first round of tests will use React Testing Library which was installed for you by create react app. React Testing Library enforces some patterns such as only using fully mounted components and encourages (as much as possible) only interacting with the site like a user by reading text and clicking on things rather than mocking functions, looking for element IDs, or looking for class names.

Testing the Thumbnail

The first test will be the most like a unit test. The only component it renders from your code is `Thumbnail`, but it will also render the `Link` component from React Router. You will test the `Thumbnail` component to makes sure it displays the item's title and an image with the alt of the item's title.

You will put the test in a new file `Thumbnail.test.js` which will live alongside `Thumbnail.js`. Most developers either put the tests alongside the file or in a folder named `__tests__` (ex. `src/components/__tests__/Thumbnail.test.js`). I prefer a flatter directory structure, which matches the example test Create React App produced in `src/App.test.js`, but you are welcome to use another structure.

For this test, you will not test navigation. You will test that in an integration test later in this chapter. That test will also use React Testing Library.

You will need to delete `src/App.test.js` for now since the tests in it will fail. (You'll re-create it later.)

src/components/Thumbnail.test.js ADDED

```
@@ -0,0 +1,16 @@
+ import { render, screen } from '@testing-library/react';
+ import { MemoryRouter as Router } from 'react-router-dom';
+ import Thumbnail from './Thumbnail';
+ import { itemImages } from '../items';
+
+ describe('Thumbnail', () => {
+   it('Displays Thumbnail', () => {
+     render(
+       <Router>
+         <Thumbnail id="apple" title="Apple" image={itemImages.apple} />
+       </Router>,
+     );
+     screen.getByText('Apple');
+     screen.getByAltText('Apple');
+   });
+ });
```

src/App.test.js DELETED

```
@@ -1,8 +0,0 @@
- import { render, screen } from '@testing-library/react';
- import App from './App';
-
- test('renders learn react link', () => {
-   render(<App />);
-   const linkElement = screen.getByText(/learn react/i);
-   expect(linkElement).toBeInTheDocument();
- });
```

Why does the code import `MemoryRouter`? The `Thumbnail` component uses a `Link` provided by React Router. `Link` will throw an error if it is used outside, of one of the routers provided by React Router. `MemoryRouter` provides a fake browser History API since one is not provided by JSDOM.

`screen` is the fake DOM which was populated by the `render` function.

Inside screen `getBy` commands tell React Testing Library that exactly one item should match. If no items match or multiple items match, an exception will be thrown. An exception will fail the test, so you do not need add an `expect` clause to those statements.

Kent C. Dodds, however, suggests including the `expect` clause so the expectation is explicit. In that case, the end of the test would look like this:

```js
expect(screen.getByText('Apple')).toBeInTheDocument();
expect(screen.getByAltText('Apple')).toBeInTheDocument();
```

You can use whichever you prefer.

You can read more about the queries at: https://testing-library.com/docs/dom-testing-library/api-queries

For help deciding which query to use see: https://testing-library.com/docs/guide-which-query/

Now, you are ready to run your test.

You can do that with `npm test`. This enters watch usage by default. After the test run, you can press `w` to see all the commands.

After pressing `w`, you should see something like:

```
 PASS  src/components/Thumbnail.test.js
  Thumbnail
    ✓ Displays Thumbnail (27 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.178 s
Ran all test suites.

Watch Usage
```

```
› Press f to run only failed tests.
› Press o to only run tests related to changed files.
› Press q to quit watch mode.
› Press p to filter by a filename regex pattern.
› Press t to filter by a test name regex pattern.
› Press Enter to trigger a test run.
```

The test passed! (Not listed above, pressing `a` will run all the tests.)

You can also run `CI=true npm test` to avoid watch mode.

Home Test

Now, let's make a test of the home page. This test moves toward integration testing because it will render `Home` and `Thumbnail` and each `Link` in the thumbnails. This test will look at the same data in `Thumbnail` just to make sure it is rendering, but the main goal is to test that `Home` displays all the items it is given as thumbnails.

Again, you will not test the navigation because this test only tests the `Home` component.

There is one difficulty with this test, there is not a way to grab all the thumbnails without resorting to an element selector. For this purpose, React Testing Library, provides a way to get elements by test ID. A test ID is a data attribute `data-testid`. Data attributes attach arbitrary data to a DOM node.

Let's add the test attribute now:

src/components/Thumbnail.js CHANGED

```
@@ -6,6 +6,7 @@ function Thumbnail({ id, image, title }) {
    return (
      <Link
        className="thumbnail-component"
+       data-testid="thumbnail-component"
        to={`/details/${id}`}
      >
        <img src={image} alt={title} />
```

Now that you have that in place, you can write a test to render a home page and count how many items were rendered.

src/components/Home.test.js ADDED

```
@@ -0,0 +1,12 @@
+ import { render, screen } from '@testing-library/react';
+ import { MemoryRouter as Router } from 'react-router-dom';
+ import Home from './Home';
+ import { items } from '../items';
+
+ describe('Home', () => {
+   it('Displays Items', () => {
+     render(<Router><Home items={items} /></Router>);
+     const thumbnails = screen.queryAllByTestId('thumbnail-component');
+     expect(thumbnails).toHaveLength(items.length);
+   });
+ });
```

`Home` renders `Thumbnail` which renders `Link` so you need to wrap this in a router as well.

This time you used `queryAll` which will allow all matching elements to be returned or none. If no elements match the selector, it will return an empty array. Then, the `expect` clause to checks the length of the element. `expect` comes from the Jest library. You could have written this statement many ways:

```js
expect(thumbnails.length === items.length).toBeTruthy();
expect(thumbnails.length).toBe(items.length);
expect(thumbnails).toHaveLength(items.length);
```

By using the more specific matcher `toHaveLength`, Jest will print more helpful messages when the expectation fails. `toBeTruthy` prints "expected false to be true", but `toHaveLength` prints:

```
Expected length: 9
Received length: 10
Received array:  [...] // actual array contents omitted for brevity
```

Having the actual array contents is helpful for debugging and happens automatically with matchers such as `toHaveLength` and `toContain`

You can read about more of the Jest matchers at https://jestjs.io/docs/en/expect .

Run `npm test` again, and you should see your new passing test!

Mocking the Server

The next step will be adding integration tests to the full application. The first test will make sure the username is displayed for a logged-in user. To test that, you will need to mock the API request for `/api/auth/current-user`.

There are a number of options for mocking API calls. So far the tests have used components which receive data from `App` so the test can supply in the result of the calls (such as passing `items` to `Home`). For this test, you will use the whole `App` component. It's possible to use Jest mock functions to mock `axios.get`. There are also npm packages specifically to help mock axios. Another option is a package called `msw` (Mock Service Workers). This package mocks out all the API calls with a mock server. These same mocks can be used in the browser if you are developing the client app and the server is not ready, but we will not cover that in this course.

> **Install**
>
> Install msw with: `npm install --save msw`

Time to set up the mock server:

src/mocks/handlers.js ADDED

```
@@ -0,0 +1,11 @@
+ import { rest } from 'msw';
+ import { items } from '../items';
+
+ const handlers = [
+   rest.get('/api/items', (req, res, ctx) => res(ctx.json(items))),
+   rest.get('/api/auth/current-user', (req, res, ctx) => (
+     res(ctx.json({ access: 'associate', username: 'Tester' }))
+   )),
+ ];
+
+ export default handlers;
```

src/mocks/server.js ADDED

```
@@ -0,0 +1,5 @@
+ import { setupServer } from 'msw/node';
+ import handlers from './handlers';
+
+ // Setup requests interception using the given handlers.
+ export default setupServer(...handlers);
```

src/setupTests.js CHANGED

```
@@ -3,3 +3,8 @@
   // expect(element).toHaveTextContent(/react/i)
   // learn more: https://github.com/testing-library/jest-dom
   import '@testing-library/jest-dom';
+ import server from './mocks/server';
+
+ beforeAll(() => server.listen({ onUnhandledRequest: 'error' }));
+ afterAll(() => server.close());
+ afterEach(() => server.resetHandlers());
```

The first file is handlers. This file is separate from the server, so you could pull it into a file to mock the calls in the browser if you wanted to.

Mock service worker provides a `rest` to allow mocking `get` / `post` / `put` / `delete` calls. Inside a handler, the function returns a response. MSW also provides context which contains a variety of helper functions for things like responding with json.

This code mocks out two calls. The call for items and the call for the current user because the main `App` file will request both.

The second file, sets up the server using `setupServer` from the node folder of msw. It needs to be passed the handlers created above.

Last, Jest has to run the mock server before it runs tests, to stop it at the end of the tests, and to reset it between each test. This is accomplished by calling `beforeAll`, `afterAll`, and `afterEach` respectively in `setupTests.js`. (This file was originally created by create react app.)

The docs for Mock Service Work are located at: https://mswjs.io/docs/ .

Testing the Logged In User

Now that the server is mocked out. You are ready to test the logged in user.

> **WARNING**
>
> This test will fail as written below. You will fix it shortly.

src/App.test.js ADDED

```
@@ -0,0 +1,9 @@
+ import { screen, render } from '@testing-library/react';
+ import App from './App';
+
+ describe('App', () => {
+   it('Displays the Logged In User\'s Username', async () => {
+     render(<App />);
+     await screen.findByText(/Welcome, Tester/);
+   });
+ });
```

The new test reads nicely "App: It displays the logged-in user's username". (Test names should always read like a sentence with the `describe` and `it` blocks.)

`<App />` already contains a router so it can be rendered directly. This test uses `findBy` instead of `getBy` because the user's username loads in asynchronously, so it will not be present in the first render. The test needs to give it a second to show up. `findBy` returns a promise so the test must `await` its resolution. This means the function itself must be `async`. Jest notices when an async function is provided in an `it` block and automatically waits for the promise to complete and fails if the promise fails.

`/Welcome, Tester/` is a regular expression. It matches elements that contain other spaces or text around the specified text.

Let's run the test. As promised, it fails:

```
Test suite failed to run

react-modal: No elements were found for selector #root.
```

Why? The `Cart` and `Login` components set the React modal root outside the main function. This worked fine for the web browser because the javascript loaded after the main HTML page, but in the test suite the JS loads before anything is rendered. Let's delay setting the React modal root by moving it into a function that will run one time when the component is mounted (`useEffect`).

src/components/Cart.js CHANGED

```
@@ -1,6 +1,11 @@
  import axios from 'axios';
  import PropTypes from 'prop-types';
- import { useMemo, useState, useRef } from 'react';
+ import {
+   useEffect,
+   useMemo,
+   useState,
+   useRef,
+ } from 'react';
  import Modal from 'react-modal';
```

```
 5  10  import { useHistory } from 'react-router-dom';
 6  11  import { CartTypes } from '../reducers/cartReducer';
     @@ -18,8 +23,6 @@ const customStyles = {
18  23    },
19  24  };
20  25
21     - Modal.setAppElement('#root');
22     -
23  26  function Cart({ cart, dispatch, items }) {
24  27    const [name, setName] = useState('');
25  28    const [phone, setPhone] = useState('');
     @@ -29,6 +32,10 @@ function Cart({ cart, dispatch, items }) {
29  32    const zipRef = useRef();
30  33    const history = useHistory();
31  34
     +   useEffect(() => {
     +     Modal.setAppElement('#root');
     +   }, []);
     +
32  39    const subTotal = cart.reduce((acc, item) => {
33  40      const details = items.find((i) => i.id === item.id);
34  41      return item.quantity * details.price + acc;
```

src/components/Login.js CHANGED

```
     @@ -1,4 +1,4 @@
1       - import { useContext, useState } from 'react';
     1  + import { useContext, useEffect, useState } from 'react';
2    2  import axios from 'axios';
3    3  import Modal from 'react-modal';
4    4  import { useHistory } from 'react-router-dom';
     @@ -17,8 +17,6 @@ const customStyles = {
17  17    },
18  18  };
19  19
20     - Modal.setAppElement('#root');
21     -
22  20  function Login() {
23  21    const { setUserDetails } = useContext(UserContext);
24  22    const [username, setUsername] = useState('');
     @@ -26,6 +24,10 @@ function Login() {
26  24    const [apiError, setApiError] = useState('');
27  25    const history = useHistory();
28  26
     +   useEffect(() => {
     +     Modal.setAppElement('#root');
     +   }, []);
     +
29  31    const login = (event) => {
30  32      event.preventDefault();
31  33      axios.post('/api/auth/login', {
```

Run your tests again. They should all pass now!

Testing Navigation

Over the next few tests, you will test the full flow of the App. This test is an integration test, but it runs with Jest, the same framework as the unit tests you wrote above. You will start with the home page and navigating to the detail page for an item.

src/App.test.js CHANGED

```
     @@ -1,9 +1,20 @@
1       - import { screen, render } from '@testing-library/react';
     1  + import { screen, render, waitFor } from '@testing-library/react';
     2  + import userEvent from '@testing-library/user-event';
```

```
  import App from './App';
+ import { items } from './items';

  describe('App', () => {
    it('Displays the Logged In User\'s Username', async () => {
      render(<App />);
      await screen.findByText(/Welcome, Tester/);
    });
+   it('Allows the user to build a cart and place an order', async () => {
+     render(<App />);
+     await waitFor(() => {
+       const thumbnails = screen.queryAllByTestId('thumbnail-component');
+       expect(thumbnails).toHaveLength(items.length);
+     });
+     userEvent.click(screen.getByRole('link', { name: /Apple/i }));
+     await screen.findByText(/Price:/);
+   });
  });
```

The test renders the app and waits to make sure the thumbnails loaded on the home page. Like the username, these load by an asynchronous API, but this test uses `waitFor` to wait for their to be exactly `items.length` thumbnails. `findAllBy` would have waited for their to be at least one, but using `expect` inside `waitFor`, makes sure there are 10.

Then, the `user-event` library allows clicking on the link containing the text "Apple". (Just like a user clicking to order an Apple.) Many roles are set by default such as `link` and `button`. You can read more about querying by role here: https://testing-library.com/docs/dom-testing-library/api-queries/#byrole

Then, the test waits for routing to occur and makes sure next page includes the items price (the detail page).

Run your tests, they should all pass.

Testing Add to Cart

After the user clicks "Add to Cart", you will want to make sure the cart quantity updates. To do that, you will need a data-testid, so you can grab the cart quantity item.

src/components/Header.js CHANGED

```
@@ -15,7 +15,7 @@ function Header({ cart }) {
          to="/cart"
        >
          <img src={CartIcon} alt="Cart" />
-         <div className="badge">{cartQuantity}</div>
+         <div className="badge" data-testid="cart-quantity">{cartQuantity}</div>
        </Link>
      </header>
    );
```

Now that the testid is in place. You can build a test to click the Add to Cart button and make sure an item is added to cart. Add two apples to the cart.

src/App.test.js CHANGED

```
@@ -16,5 +16,13 @@ describe('App', () => {
    });
```

```
17  17        userEvent.click(screen.getByRole('link', { name: /Apple/i }));
18  18        await screen.findByText(/Price:/);
    19  +     userEvent.click(screen.getByRole('button', { name: 'Add to Cart' }));
    20  +     await waitFor(() => (
    21  +       expect(screen.getByTestId('cart-quantity')).toHaveTextContent('1')
    22  +     ));
    23  +     userEvent.click(screen.getByRole('button', { name: 'Add to Cart' }));
    24  +     await waitFor(() => (
    25  +       expect(screen.getByTestId('cart-quantity')).toHaveTextContent('2')
    26  +     ));
19  27      });
20  28    });
```

(We had to break the waitFor onto multiple lines, so it would fit in the browser, but you do not need to split it in your code.)

The `toHaveTextContent` matcher comes from `jest-dom` which was installed by Create React App. You can read about those matchers at https://github.com/testing-library/jest-dom#custom-matchers .

Testing Checkout

Before test can checkout, you need to mock the post request in the MSW handlers.

src/mocks/handlers.js CHANGED

```
        @@ -6,6 +6,7 @@ const handlers = [
6   6       rest.get('/api/auth/current-user', (req, res, ctx) => (
7   7         res(ctx.json({ access: 'associate', username: 'Tester' }))
8   8       )),
    9   +   rest.post('/api/orders', (req, res, ctx) => res(ctx.status(201))),
9   10    ];
10  11
11  12    export default handlers;
```

This new request does not do any validation, it just responds with 201 (created).

Now you are ready to fill in the checkout form and submit an order.

> **WARNING**
>
> This test will fail.

src/App.test.js CHANGED

```
        @@ -24,5 +24,15 @@ describe('App', () => {
24  24        await waitFor(() => (
25  25          expect(screen.getByTestId('cart-quantity')).toHaveTextContent('2')
26  26        ));
    27  +     userEvent.click(screen.getByRole('link', { name: /Cart/i }));
    28  +     await screen.findByLabelText(/Name/);
    29  +     expect(screen.getByRole('button', { name: 'Place Order' })).toBeDisabled();
    30  +     userEvent.type(screen.getByLabelText(/Name/), 'Big Nerd Ranch');
    31  +     userEvent.type(screen.getByLabelText(/Zip Code/), '30307');
    32  +     expect(screen.getByRole('button', { name: 'Place Order' })).toBeEnabled();
    33  +     userEvent.click(screen.getByRole('button', { name: 'Place Order' }));
    34  +     await screen.findByText(/Thanks for your order/i);
    35  +     userEvent.click(screen.getByRole('button', { name: 'Return Home' }));
    36  +     await screen.findAllByTestId('thumbnail-component');
27  37    });
```

```
28  38    });
```

The test gets the cart image by its alt text and clicks it (just like a user would click the cart). Then, to be sure it's on the checkout page, it looks for the name input. Next, it checks that the button to place an order is initially disabled. Using the labels, it fills out the checkout form. After making sure the place order button is enabled, it places the order and expects the modal to show. Finally, clicking "Return Home" to take the test back to the home page where at least one item thumbnail will be showing. `getAllBy` will throw an error if no items show up, but it does not care how many items are present. (No need to double check the exact thumbnail count here.)

Run the test, it fails. There is a lot of output, but you will eventually find this specific error.

```
App › Allows the user to build a cart and place an order

react-modal: No elements were found for selector #root.
```

Ugh, more react-modal errors. The previous code was able to delay fixing the issue because no test had rendered the `Cart` or `Thumbnail` component. The div with the ID `root` exists in `public/index.html` not in the React code so tests that modals need to create the div as part of the render. Wrap `App` in a div with the id root in the test.

src/App.test.js CHANGED

```
@@ -9,7 +9,7 @@ describe('App', () => {
9   9       await screen.findByText(/Welcome, Tester/);
10  10    });
11  11    it('Allows the user to build a cart and place an order', async () => {
12      -     render(<App />);
        12  +     render(<div id="root"><App /></div>);
13  13        await waitFor(() => {
14  14          const thumbnails = screen.queryAllByTestId('thumbnail-component');
15  15          expect(thumbnails).toHaveLength(items.length);
```

Run the tests, they pass again!

Verify Order Submission

The last thing to do is make sure the order made it to the server. The best case would be to visit the Orders page and make sure it shows up there. Unfortunately, mocking WebSockets requires other testing libraries and is uncommon enough that we will not cover it in this chapter. Feel free to work on it as a challenge later. The orders page will be tested with end-to-end tests in the next chapter.

What you can do is add a data layer to the mock API. (This data layer would have worked with the orders page prior to adding websockets.)

Let's set up the data layer now.

src/mocks/data.js ADDED

```
@@ -0,0 +1,11 @@
1   + let orders = [];
```

```
+
+ export const reset = () => {
+   orders = [];
+ };
+
+ export const getOrders = () => orders;
+
+ export const addOrder = (newOrder) => {
+   orders.push({ ...newOrder });
+ };
```

src/mocks/handlers.js CHANGED

```
@@ -1,12 +1,17 @@
  import { rest } from 'msw';
  import { items } from '../items';
+ import { addOrder, getOrders } from './data';

  const handlers = [
    rest.get('/api/items', (req, res, ctx) => res(ctx.json(items))),
    rest.get('/api/auth/current-user', (req, res, ctx) => (
      res(ctx.json({ access: 'associate', username: 'Tester' }))
    )),
-   rest.post('/api/orders', (req, res, ctx) => res(ctx.status(201))),
+   rest.get('/api/orders', (req, res, ctx) => res(ctx.json(getOrders()))),
+   rest.post('/api/orders', (req, res, ctx) => {
+     addOrder(req.body);
+     return res(ctx.status(201));
+   }),
  ];

  export default handlers;
```

src/setupTests.js CHANGED

```
@@ -3,8 +3,12 @@
  // expect(element).toHaveTextContent(/react/i)
  // learn more: https://github.com/testing-library/jest-dom
  import '@testing-library/jest-dom';
+ import * as data from './mocks/data';
  import server from './mocks/server';

  beforeAll(() => server.listen({ onUnhandledRequest: 'error' }));
  afterAll(() => server.close());
- afterEach(() => server.resetHandlers());
+ afterEach(() => {
+   data.reset();
+   server.resetHandlers();
+ });
```

The mock data layer uses an array to store and return the orders and provides a function to reset the orders between each test.

Then, in the handlers, there is a network call to get the orders. (This code will not use this, but if you decided to remove the websockets, you could.) The post request now adds the order to the data layer.

Last, the `afterEach` block resets the data layer in addition to the server handlers after each test.

Add one final line to the checkout test to make sure the server has an order.

src/App.test.js CHANGED

```
@@ -2,6 +2,7 @@ import { screen, render, waitFor } from '@testing-library/react';
  import userEvent from '@testing-library/user-event';
  import App from './App';
```

```
 4   4   import { items } from './items';
     5 + import { getOrders } from './mocks/data';
 5   6
 6   7   describe('App', () => {
 7   8     it('Displays the Logged In User\'s Username', async () => {
           @@ -32,6 +33,7 @@ describe('App', () => {
32  33       expect(screen.getByRole('button', { name: 'Place Order' })).toBeEnabled();
33  34       userEvent.click(screen.getByRole('button', { name: 'Place Order' }));
34  35       await screen.findByText(/Thanks for your order/i);
     36 +     expect(getOrders()).toHaveLength(1);
35  37       userEvent.click(screen.getByRole('button', { name: 'Return Home' }));
36  38       await screen.findAllByTestId('thumbnail-component');
37  39     });
```

Run your tests. Everything passes! You have completed a full happy path test for the user.

Checkout Error

The next test makes sure the error modal shows up if something goes wrong during the checkout flow, but there is no need to spend time testing the entire add to cart flow again. This is where the ability to write tests that read like unit tests really shines. In this case, you will render `Cart`. You will still use the mock server and allow the modals to render so there is still integration, but the test focuses on one component.

src/components/Cart.test.js ADDED

```
@@ -0,0 +1,38 @@
 1 + import { screen, render } from '@testing-library/react';
 2 + import { MemoryRouter as Router } from 'react-router-dom';
 3 + import userEvent from '@testing-library/user-event';
 4 + import { rest } from 'msw';
 5 + import Cart from './Cart';
 6 + import { items } from '../items';
 7 + import server from '../mocks/server';
 8 +
 9 + describe('Cart Errors', () => {
10 +   it('Shows Checkout Failure Error', async () => {
11 +     const testErrorMessage = 'Coffee Shop is Closed';
12 +     server.use(
13 +       rest.post('/api/orders', async (req, res, ctx) => (
14 +         res(ctx.status(500), ctx.json({ error: testErrorMessage }))
15 +       )),
16 +     );
17 +     const cart = [{ id: items[0].id, quantity: 1 }];
18 +     const dispatch = jest.fn(() => {});
19 +     render(
20 +       <div id="root">
21 +         <Router>
22 +           <Cart cart={cart} dispatch={dispatch} items={items} />
23 +         </Router>
24 +       </div>,
25 +     );
26 +     expect(screen.getByRole('button', { name: 'Place Order' })).toBeDisabled();
27 +     userEvent.type(screen.getByLabelText(/Name/), 'Big Nerd Ranch');
28 +     userEvent.type(screen.getByLabelText(/Zip Code/), '30307');
29 +     expect(screen.getByRole('button', { name: 'Place Order' })).toBeEnabled();
30 +     userEvent.click(screen.getByRole('button', { name: 'Place Order' }));
31 +     await screen.findByText(/There was an error/);
32 +     screen.getByText(new RegExp(`${testErrorMessage}`, 'i'));
33 +     userEvent.click(screen.getByRole('button', { name: 'Ok' }));
34 +     expect(screen.queryByText(/There was an error/)).not.toBeInTheDocument();
35 +     expect(screen.getByRole('button', { name: 'Place Order' })).toBeEnabled();
36 +     expect(dispatch).not.toHaveBeenCalled();
37 +   });
```

```
+ });
```

Mock Service Worker allows temporarily overriding a server endpoint, so the order endpoint can fail with an error message without having to make the happy path worker run any validations.

This test needs a mock item in the cart. It mocks `dispatch` with a Jest function wrapper, so it can check if `dispatch` was called. The code could have provided a real implementation for the function, but since the function should not be called, the code provides a blank function.

Then, the test fills out the form as before, but when the form is submitted, it expects the error modal to pop up. The test closes the modal and expects the place order button to still be enabled and the dispatch function not to have been called since an error should not clear the cart.

Run the test, it passes!

Mocking `console.error`

Right now when the test runs, it dumps a giant error log to the console which shows up during the test run. This is helpful if the error is unexpected, but since the error is expected, it clutters up the console.

Let's clear up the clutter while also ensuring that real errors are logged to the console by mocking console in the test suite.

src/components/Cart.test.js CHANGED

```
@@ -7,6 +7,14 @@ import { items } from '../items';
 import server from '../mocks/server';

 describe('Cart Errors', () => {
+  beforeEach(() => {
+    jest.spyOn(console, 'error').mockImplementation(() => {});
+  });
+
+  afterEach(() => {
+    console.error.mockRestore();
+  });
+
   it('Shows Checkout Failure Error', async () => {
     const testErrorMessage = 'Coffee Shop is Closed';
     server.use(
@@ -34,5 +42,6 @@ describe('Cart Errors', () => {
     expect(screen.queryByText(/There was an error/)).not.toBeInTheDocument();
     expect(screen.getByRole('button', { name: 'Place Order' })).toBeEnabled();
     expect(dispatch).not.toHaveBeenCalled();
+    expect(console.error).toHaveBeenCalledTimes(1);
   });
 });
```

`spyOn` and `mockRestore` could have been added at top and bottom of the `it` block, but if `mockRestore` was at the bottom of the test and the test failed, it would not get restored. Using try/catch/finally in the `it` block still restore, but it is easier to let Jest handle that by using a local `beforeEach` and `afterEach`. (There is also `beforeAll` and `afterAll`.) Since the `describe` block is for "Cart Errors", it makes sense to mock console errors on all the tests in this describe block.

Then, at the bottom, the test expects there to be exactly 1 console error as a result of this test.

Run the test. It passes with just the normal Jest output.

Conclusion

You now have unit tests and some integration tests that run with Jest.

These tests are quite fast, running in 1.95 seconds on my machine and allow you to be more confident that future code changes have not broken the coffee shop.

🥇 Login Tests

Can you add a happy path login test and a failure path test?

You will also want to test that the "Orders" link only shows up for associates.

You will need to create base handlers for login as well as override them for the failure.

🥈 Phone Number

Test that typing in the phone number field causes the dashes to be added as expected.

Ex. Typing "5555555555" should result in a value of "555-555-5555", and typing "555-5555555" should also result in a value of "555-555-5555".

🥈 Cart Details

Test that the tax details adjust properly.

Also, check the adding multiple different items and multiple of the same item works as expected.

🥈 Remove Item

Test that the remove item button works as expected.

🥈 More tests

Or implement some other test that you come up with.

End-to-End Testing

With component testing done, you are ready to move on to end-to-end testing. This chapter uses Cypress to write end-to-end tests.

Install Cypress

The first step is installing Cypress, an eslint plugin, and the testing library extension for Cypress.

> **Install**
>
> Run `npm install --save cypress@9.4.1 eslint-plugin-cypress@2.12.1 @testing-library/cypress@8.0.2`
>
> Note: Cypress takes a few minutes to install so don't worry if the install command looks stuck.

Configure Scripts

Next, you need to configure some new package scripts to run cypress.

package.json CHANGED

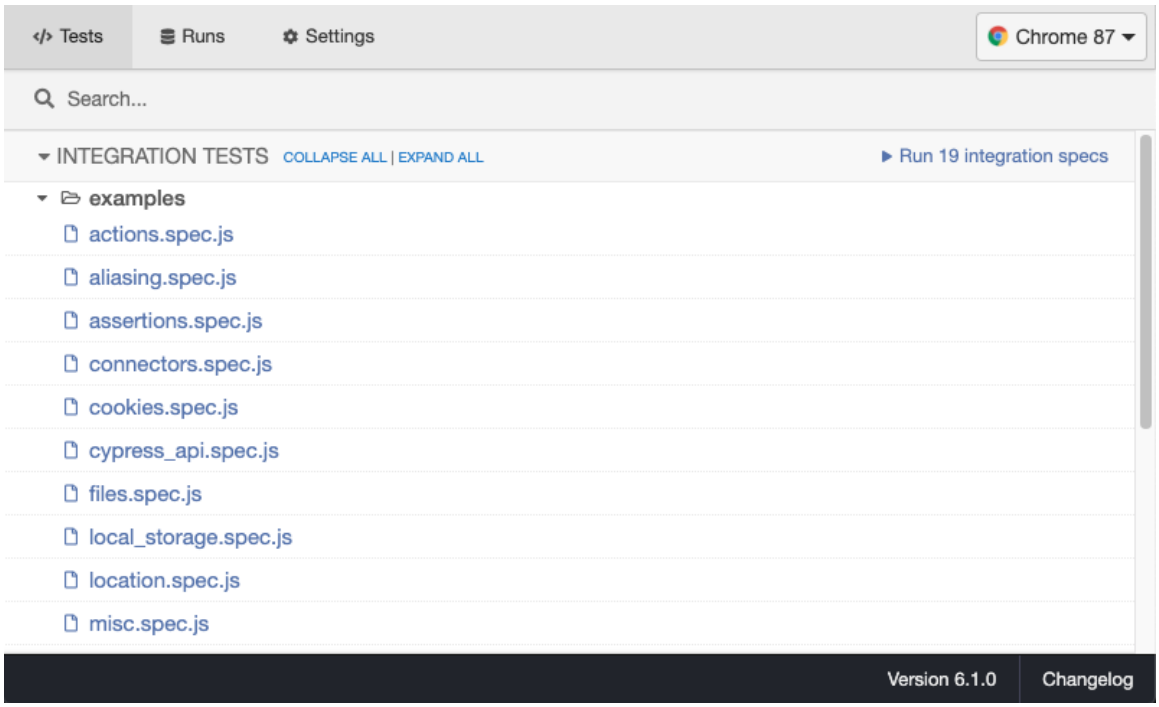
```
@@ -27,6 +27,8 @@
27 27       "web-vitals": "^2.1.3"
28 28     },
29 29     "scripts": {
30 +       "cypress:open": "cypress open",
31 +       "cypress:run": "cypress run",
30 32       "start": "react-scripts start",
31 33       "build": "react-scripts build",
32 34       "test": "react-scripts test",
```

`package.json` has two new scripts. The first `cypress:open` will open cypress and allow you to run tests one at a time. The second `cypress:run` will run cypress in headless mode and run all the tests. (This mode is mostly meant for CI.)

Try out your new script.

```sh
npm run cypress:open
```

Cypress takes a few seconds to open, but you should see the cypress window.



Cypress has also created a number of configuration files and example tests.

Close the Cypress Window.

ESLint Setup

The lint command previously only linted the `src` directory. Since Cypress is outside that, you need to change the configuration to lint the cypress directory as well.

You also need to add the cypress plugin and globals to the cypress directory.

Note: If you do not already have a directory named `cypress`, you need to go back up a step and run `npm run cypress:open` so Cypress will create it.

cypress/.eslintrc.js ADDED

```
@@ -0,0 +1,13 @@
+ module.exports = {
+   plugins: ['eslint-plugin-cypress'],
+   extends: [
+     'plugin:cypress/recommended',
+   ],
+   env: {
+     'cypress/globals': true,
+   },
+   rules: {
+     'testing-library/await-async-query': 'off',
+     'testing-library/prefer-screen-queries': 'off',
+   },
+ };
```

package.json CHANGED

```
@@ -32,7 +32,7 @@
32  32      "start": "react-scripts start",
33  33      "build": "react-scripts build",
34  34      "test": "react-scripts test",
35      -   "lint": "eslint src --max-warnings=0",
    35  +   "lint": "eslint cypress src --max-warnings=0",
36  36      "eject": "react-scripts eject"
37  37    },
38  38    "eslintConfig": {
```

ESLint allows overriding and extending the configuration by directory. In this case, `cypress/.eslintrc.js` adds cypress specific rules from the cypress eslint plugin and the cypress globals, so eslint will not flag them as undefined. We are also disabling two lint rules that are useful when using testing-library with jest but are not useful when using Cypress.

Fixing ESLint

If you run `npm run lint` now, you will see a lot of lint errors.

As a first step to cleaning these up, delete all the examples.

> **Delete**
>
> Delete the folder `cypress/integration/examples`.

You are down to just a few errors now. Some will autofix if you run `npm run lint -- --fix`. Ignore the last one.

cypress/plugins/index.js CHANGED

```
@@ -15,7 +15,8 @@
15  15  /**
16  16   * @type {Cypress.PluginConfig}
17  17   */
    18  + // eslint-disable-next-line no-unused-vars
18  19  module.exports = (on, config) => {
19  20    // `on` is used to hook into various events Cypress emits
20  21    // `config` is the resolved Cypress config
21      - }
    22  + };
```

cypress/support/index.js CHANGED

```
@@ -14,7 +14,7 @@
14  14  // ***********************************************************
15  15
16  16  // Import commands.js using ES2015 syntax:
17      - import './commands'
    17  + import './commands';
18  18
19  19  // Alternatively you can use CommonJS syntax:
20  20  // require('./commands')
```

Now, you should be able to commit your work without any complaints from the linter.

.gitignore

When run in headless mode, Cypress will save screenshots and videos. These should not be committed by git. Update `.gitignore` to exclude them.

.gitignore CHANGED

```
@@ -1,5 +1,9 @@
  # See https://help.github.com/articles/ignoring-files/ for more about ignoring files
+ # cypress
+ /cypress/screenshots
+ /cypress/videos
+
  # dependencies
  /node_modules
  /.pnp
```

The `.gitignore` file lists files that should not be committed to git version control. The file was initially created by Create React App. You can list directories or specific files or patterns to ignore. You can also include `.gitignore` files in subdirectories to extend the rules.

For more reading on `.gitignore` see: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#_ignoring

Import Commands

Testing Library has a Cypress plugin which will gives Cypress similar commands to the ones used in the last chapter. You already installed it above, but you need to import the commands into Cypress.

cypress/support/commands.js CHANGED

```
@@ -23,3 +23,4 @@
  //
  // -- This will overwrite an existing command --
  // Cypress.Commands.overwrite("visit", (originalFn, url, options) => { ... })
+ import '@testing-library/cypress/add-commands';
```

Cypress Configuration

There is one last piece of configuration to set before writing your first test.

cypress.json CHANGED

```
@@ -1 +1,3 @@
- {}
+ {
+   "baseUrl": "http://localhost:3000"
+ }
```

`baseUrl` makes it easy to change the expected URL of the application, and you do not have to type as much. Rather `cy.visit('http://localhost:3000/...')`, with a base url the code can be `cy.visit('/...')`.

Testing Login

You are finally ready to write your first test! For this test, you will test that login works and the logged in user's username shows in the header.

cypress/integration/login.spec.js ADDED

```
@@ -0,0 +1,10 @@
+ describe('Login', () => {
+   it('Show Logged In User\'s Username', () => {
+     cy.visit('/');
+     cy.findByRole('link', { name: 'Login' }).click();
+     cy.findByLabelText(/Username/).type('Tester');
+     cy.findByLabelText(/Password/).type('pass');
+     cy.findByRole('button', { name: 'Login' }).click();
+     cy.findByText(/Welcome, Tester/);
+   });
+ });
```

Cypress uses the same `describe` and `it` block structure for test files as jest.

This test starts by visiting the home page. Then, it finds and clicks the Login link.

All the queries for elements in Cypress use `findBy` rather than `getBy` or `queryBy`. This is specific to Cypress testing library. Cypress requires everything to be retryable so `getBy` is unsupported because it will not retry. `findBy` normally errors if no items are found, but with Cypress, it is modified to work with the chainer such that `cy.findByText(...).should('not.exist')` works. Therefore, there is no reason to use `queryBy` commands. In the end, Cypress only needs `findBy` and `findAllBy`.

Interacting with the page is also a bit different. Cypress provides a chainer `.click()` which can be added after selecting the item to click.

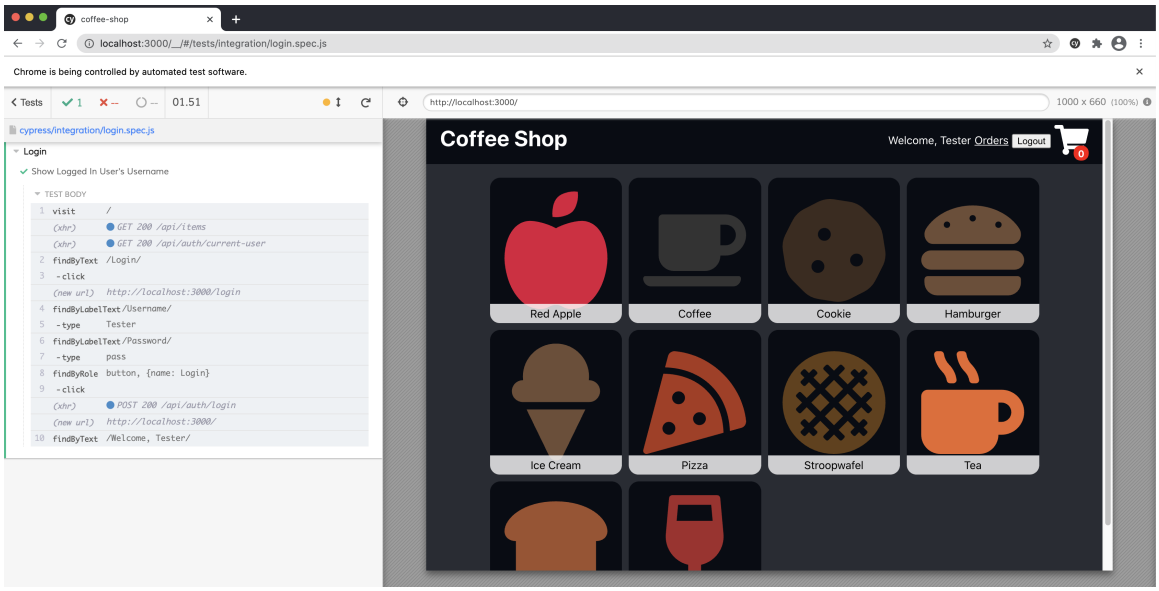
Similar to clicking, Cypress also provides a chainer for typing `.type('Tester')`.

The last line makes sure the welcome text exists.

Let's run the tests. First, you need to be sure your coffee shop front end and backend is already running. You should be able to visit http://localhost:3000 and see your coffee shop.

In another terminal window, run `npm run cypress:open`. You should see the list of test files.

Click `login.spec.js`. This will open another window where you can watch the test run. It should pass.



Testing Checkout

For the next test, rewrite the checkout test from the `App` test in the previous chapter in Cypress.

cypress/integration/orders.spec.js ADDED

```
@@ -0,0 +1,22 @@
+ describe('Orders', () => {
+   it('User can build a cart and place an order', () => {
+     cy.visit('/');
+     cy.findAllByTestId('thumbnail-component').should('have.length', 10);
+     cy.findByRole('link', { name: /Apple/ }).click();
+     cy.findByText(/Price:/);
+     cy.findByRole('button', { name: 'Add to Cart' }).click();
+     cy.findByTestId('cart-quantity').should('contain', '1');
+     cy.findByRole('button', { name: 'Add to Cart' }).click();
+     cy.findByTestId('cart-quantity').should('contain', '2');
+     cy.findByRole('link', { name: /Cart/i }).click();
+     cy.findByLabelText(/Name/);
+     cy.findByRole('button', { name: 'Place Order' }).should('be.disabled');
+     cy.findByLabelText(/Name/).type('Big Nerd Ranch');
+     cy.findByLabelText(/Zip Code/).type('30307');
+     cy.findByRole('button', { name: 'Place Order' }).should('be.enabled');
+     cy.findByRole('button', { name: 'Place Order' }).click();
+     cy.findByText(/Thanks for your order/);
+     cy.findByRole('button', { name: 'Return Home' }).click();
+     cy.findAllByTestId('thumbnail-component');
+   });
+ });
```

Most of this code looks just like the previous test except for replacing `getBy` with `findBy` and moving to chainers for user interaction (`.click()` and `.type()`).

The assertions have also changed to use `.should()` chainers. One example is `.should('have.length', 10)` which asserts that there are 10 thumbnails on the page.

Checking the cart quantity uses `.should('contain', '1')` . There as also `.should('be.disabled')` .

There is also an assertion for `.should('exist')`, but `findBy` errors by default if the item does not exist.

These two lines are equivalent:

```js
cy.findByText(/Price:/);
cy.findByText(/Price:/).should('exist');
```

Feel free to try out both. You'll notice a slight difference in output because the second line shows an `assert`. If you change the text to cause the command to fail, you will get the same error message either way though the second line will again show the assertion. If it seems more readable to you, feel free to include `.should('exist')` on the find commands to make the assertion explicit.

Cypress supports many assertions from Chai, Sinon, and jQuery, you can read about them here: https://docs.cypress.io/guides/references/assertions.html

Go back to the Cypress test window, and click `orders.spec.js`. Your test should run and pass!

Avoiding Flaky Tests

The test currently asserts that the coffee shop has 10 items. The previous test got the number 10 from the mock data, so it knew to expect exactly 10 items. In this test, the items are loaded from the real server rather than mock data. The server does currently return 10 items, but more items might be added to the coffee shop in the future. It's probably better not to assert a specific number of items in this test. Otherwise, the test would have to be changed anytime items are adding or removed from the coffee shop.

cypress/integration/orders.spec.js CHANGED

```
@@ -1,7 +1,6 @@
  describe('Orders', () => {
    it('User can build a cart and place an order', () => {
      cy.visit('/');
-     cy.findAllByTestId('thumbnail-component').should('have.length', 10);
      cy.findByRole('link', { name: /Apple/ }).click();
      cy.findByText(/Price:/);
      cy.findByRole('button', { name: 'Add to Cart' }).click();
```

Note: Cypress does support mocking the server in which case it would be perfectly safe to assert a specific number. The end-to-end tests in this chapter will use the real server to test integration between the server and the client.

Test Viewing/Deleting Orders

Because you are not mocking anything, you can easily add a test to check the orders page and complete/delete the order.

> **Warning**
>
> This test will likely fail if you run it.

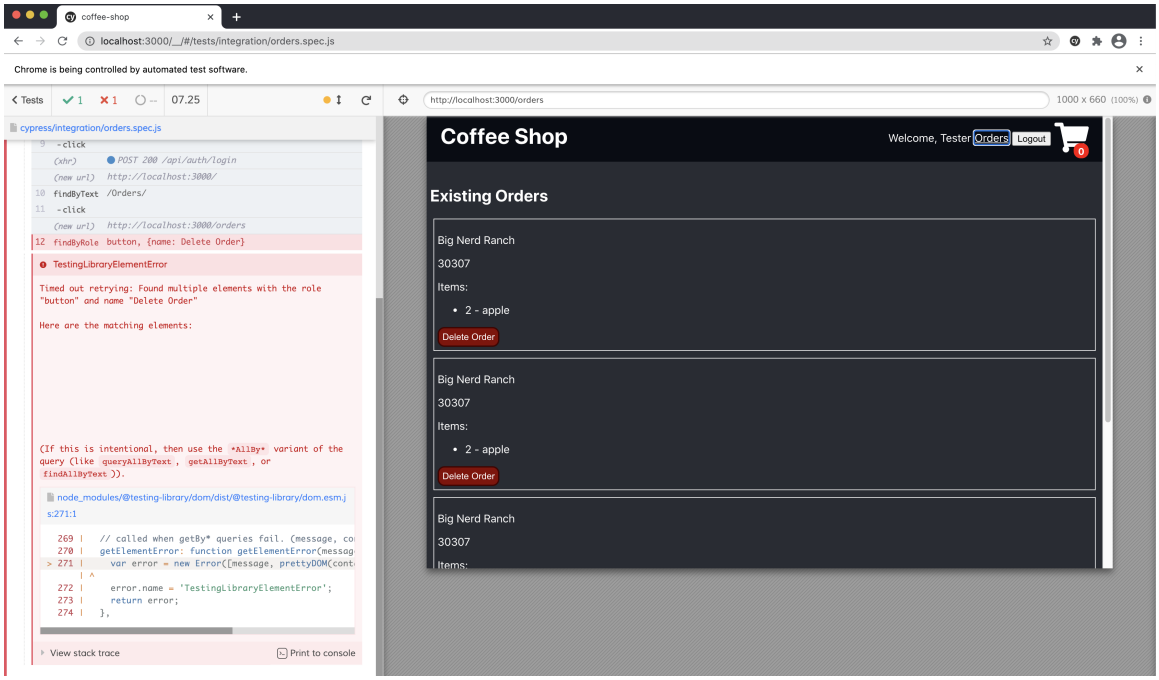
cypress/integration/orders.spec.js CHANGED

```
@@ -18,4 +18,14 @@ describe('Orders', () => {
      cy.findByRole('button', { name: 'Return Home' }).click();
      cy.findAllByTestId('thumbnail-component');
    });
+   it('Can View and Delete Orders', () => {
+     cy.visit('/');
+     cy.findByRole('link', { name: 'Login' }).click();
+     cy.findByLabelText(/Username/).type('Tester');
+     cy.findByLabelText(/Password/).type('pass');
+     cy.findByRole('button', { name: 'Login' }).click();
+     cy.findByRole('link', { name: 'Orders' }).click();
+     cy.findByRole('button', { name: 'Delete Order' }).click();
+     cy.findByText(/No Orders/);
+   });
  });
```

This test has to login and has duplicated the login flow. You will factor that out later.

Then, the test clicks to go to the orders page and clicks the button to delete the order. Since the test only placed one order, the page should now show "No Orders".

Rerun your `orders.spec.js` file. Most likely the test failed.



Why did it fail? The error says it failed because `findBy` expected to find only one button, but it found multiple buttons. Looking at the browser window on the right, there are several existing orders rather than just one.

Why are there several orders? The server was not cleared between test runs, so it sees the orders from the previous test runs. The server clears each time it restarts, so you could restart it before each test run, but there might be several Cypress tests that place orders. If you plan to run your tests in parallel, you would need to separate them so they did not interfere with each other. By default, Cypress runs the tests in sequence so as long as Cypress clears the orders before the test starts it will work.

Clearing the Orders

Let's clear the orders, so the previous test will work. Cypress can send network requests, so it can use the delete all endpoint on the server to erase all the orders.

cypress/integration/orders.spec.js CHANGED

```
@@ -1,4 +1,7 @@
  describe('Orders', () => {
+   before(() => {
+     cy.request('DELETE', '/api/orders');
+   });
    it('User can build a cart and place an order', () => {
      cy.visit('/');
      cy.findByRole('link', { name: /Apple/ }).click();
```

In Cypress, the hook that runs once before all the tests is called `before` rather than `beforeAll` as it is called in Jest. (Cypress does also have `beforeEach` which behaves like `beforeEach` in Jest.)

`cy.request` sends requests to the server. The delete request to `/api/orders` will delete all the orders.

Run your `orders.spec.js` test again. It should pass again!

🔎 Cypress Network Requests

Cypress can do many things with network requests. It can send them to setup test data like above.

It can also stub them out. Cypress can also wait for them to complete, so you can make sure the network request is done before looking at the browser for updates. This is particularly helpful for slower requests that might otherwise time out.

Cypress can make assertions with them, so the test which placed an order could have called get `/api/orders` to make sure it made it to the server.

To learn more about using Network Requests in Cypress tests see:
https://docs.cypress.io/guides/guides/network-requests.html

Testing Checkout Errors

Next, create a test for checkout errors. Unfortunately, Cypress cannot start with a mock cart, so the test will have to build the cart out again.

cypress/integration/checkoutErrors.spec.js ADDED

```
@@ -0,0 +1,16 @@
+ describe('Checkout Errors', () => {
+   it('Shows an error when the order fails', () => {
+     cy.visit('/');
+     cy.findByRole('link', { name: /Apple/ }).click();
+     cy.findByRole('button', { name: 'Add to Cart' }).click();
+     cy.findByRole('link', { name: 'Cart 1' }).click();
+     cy.findByLabelText(/Name/).type('Big Nerd Ranch');
+     cy.findByLabelText(/Zip Code/).type('99999');
+     cy.findByRole('button', { name: 'Place Order' }).click();
+     cy.findByText(/There was an error/);
+     cy.findByText(/We don't ship to 99999/i);
+     cy.findByRole('button', { name: 'Ok' }).click();
+     cy.findByText(/There was an error/).should('not.exist');
+     cy.findByRole('button', { name: 'Place Order' }).should('be.enabled');
+   });
+ });
```

This test used simpler code to put an item in the cart since the happy path test already asserts the cart quantities. This test also uses the `99999` zipcode which will produce an error from the server. Otherwise, this looks similar to your component test.

Run your new `checkoutErrors.spec.js` file in Cypress. It should pass.

Testing Console

Cypress does not show the console error by default, but it would still be nice to assert that it was logged. Cypress can spy on the console.

cypress/integration/checkoutErrors.spec.js CHANGED

```
@@ -1,6 +1,10 @@
  describe('Checkout Errors', () => {
    it('Shows an error when the order fails', () => {
-     cy.visit('/');
+     cy.visit('/', {
+       onBeforeLoad(win) {
+         cy.stub(win.console, 'error').as('consoleError');
+       },
+     });
      cy.findByRole('link', { name: /Apple/ }).click();
      cy.findByRole('button', { name: 'Add to Cart' }).click();
      cy.findByRole('link', { name: 'Cart 1' }).click();
@@ -12,5 +16,6 @@ describe('Checkout Errors', () => {
      cy.findByRole('button', { name: 'Ok' }).click();
      cy.findByText(/There was an error/).should('not.exist');
      cy.findByRole('button', { name: 'Place Order' }).should('be.enabled');
+     cy.get('@consoleError').should('be.calledOnce');
    });
  });
```

Each `cy.visit` call creates a window. Just before Cypress loads the content, this code stubs `console.error` and saves a reference to it as `consoleError`. `cy.get('@consoleError')` returns that reference, and the test expects it to have been called once.

Run your test again. It should pass.

For details see the Cypress FAQ: https://docs.cypress.io/faq/questions/using-cypress-faq.html#How-do-I-spy-on-console-log

You can also check out their sample code: https://github.com/cypress-io/cypress-example-recipes/tree/master/examples/stubbing-spying__console

Factoring out Custom Commands

To avoid repeating common bits of code, Cypress allows creating custom commands.

Let's factor out the login command.

cypress/integration/login.spec.js CHANGED

```
@@ -1,10 +1,7 @@
  describe('Login', () => {
    it('Show Logged In User\'s Username', () => {
      cy.visit('/');
-     cy.findByRole('link', { name: 'Login' }).click();
-     cy.findByLabelText(/Username/).type('Tester');
-     cy.findByLabelText(/Password/).type('pass');
-     cy.findByRole('button', { name: 'Login' }).click();
+     cy.login();
      cy.findByText(/Welcome, Tester/);
    });
  });
```

cypress/integration/orders.spec.js CHANGED

```
@@ -23,10 +23,7 @@ describe('Orders', () => {
    });
    it('Can View and Delete Orders', () => {
      cy.visit('/');
-     cy.findByRole('link', { name: 'Login' }).click();
-     cy.findByLabelText(/Username/).type('Tester');
-     cy.findByLabelText(/Password/).type('pass');
-     cy.findByRole('button', { name: 'Login' }).click();
+     cy.login();
      cy.findByRole('link', { name: 'Orders' }).click();
      cy.findByRole('button', { name: 'Delete Order' }).click();
      cy.findByText(/No Orders/);
```

cypress/support/commands.js CHANGED

```
@@ -24,3 +24,10 @@
  // -- This will overwrite an existing command --
  // Cypress.Commands.overwrite("visit", (originalFn, url, options) => { ... })
  import '@testing-library/cypress/add-commands';
+
+ Cypress.Commands.add('login', (username = 'Tester', password = 'pass') => {
+   cy.findByRole('link', { name: 'Login' }).click();
+   cy.findByLabelText(/Username/).type(username);
+   cy.findByLabelText(/Password/).type(password);
+   cy.findByRole('button', { name: 'Login' }).click();
+ });
```

The custom login command allows the user to specify the username and password, but also provides default values if they do not.

If you run your tests, they will still pass, and you've removed some duplicate code!

Invisible Elements

One of the main benefits of running end-to-end tests in a browser is the elements are actually drawn on the page. This means CSS applies.

Let's make a couple edits to prove this. First, hide the checkout inputs with CSS:

src/components/Cart.css CHANGED

```
@@ -13,4 +13,5 @@
 .cart-component form input {
   margin: 5px;
   padding: 2px;
+  display: none;
 }
```

Run your component tests (`npm run test`). You might be surprised that they still pass.

Now, run `orders.spec.js`

Cypress displays a helpful error message about why it could not type in the input field. It will allow you to override the message with `{force: true}`, but this helped catch a bug.

Covered Elements

Let's try one more thing and assume the code accidentally opened the thank you modal early.

```
15        -     cy.findByRole('button', { name: 'Place Order' }).should('be.disabled');
16  15          cy.findByLabelText(/Name/).type('Big Nerd Ranch');
17  16          cy.findByLabelText(/Zip Code/).type('30307');
18  17          cy.findByRole('button', { name: 'Place Order' }).should('be.enabled');
```

src/components/Cart.css CHANGED

```
            @@ -13,5 +13,4 @@
13  13        .cart-component form input {
14  14          margin: 5px;
15  15          padding: 2px;
16        -     display: none;
17  16        }
```

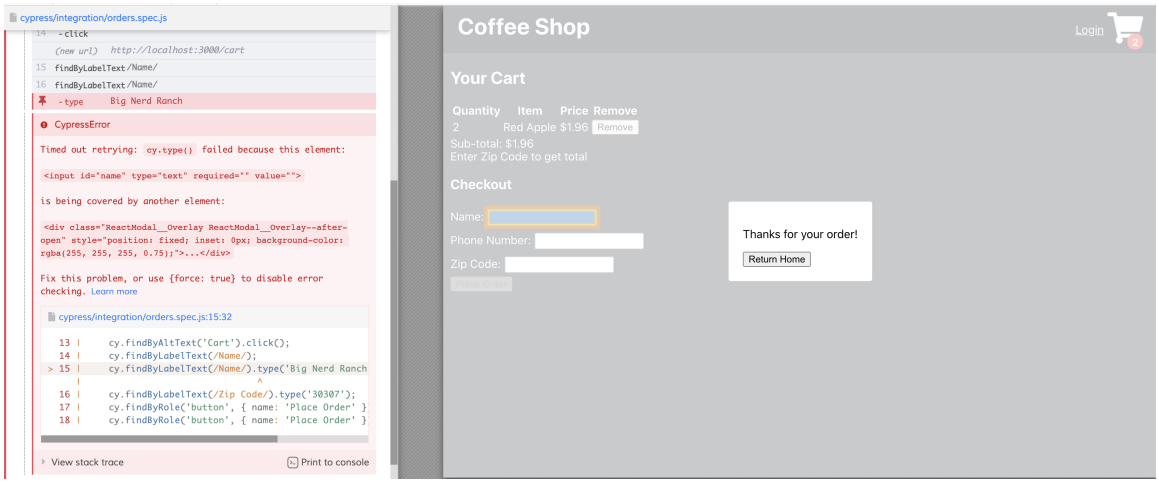
src/components/Cart.js CHANGED

```
            @@ -34,6 +34,7 @@ function Cart({ cart, dispatch, items }) {
34  34
35  35          useEffect(() => {
36  36            Modal.setAppElement('#root');
        37  +       setThankYouOpen(true);
37  38          }, []);
38  39
39  40          const subTotal = cart.reduce((acc, item) => {
```

This removed one line from the Cypress test because you get a more helpful error message with the `type` command than the command checking for the disabled element. It would normally fail on the test for the disabled button because the button is not accessible since it is covered.

Run the unit tests. They spit out a warning, but they still pass. The warning is created because after submitting the order the test very quickly navigates back home from the thank you modal so when the mock API response happens the Cart component has already been unmounted.

Run the Cypress `orders.spec.js` again. Cypress again properly fails because the elements are covered.



Shown in the screenshot above, if you hover over an element, Cypress will highlight the element it is choosing. If you hover over a click event, it will also show a red dot where it clicked.

Before continuing, restore all the files to how they were before.

cypress/integration/orders.spec.js CHANGED

```
@@ -12,6 +12,7 @@ describe('Orders', () => {
      cy.findByTestId('cart-quantity').should('contain', '2');
      cy.findByRole('link', { name: /Cart/i }).click();
      cy.findByLabelText(/Name/);
+     cy.findByRole('button', { name: 'Place Order' }).should('be.disabled');
      cy.findByLabelText(/Name/).type('Big Nerd Ranch');
      cy.findByLabelText(/Zip Code/).type('30307');
      cy.findByRole('button', { name: 'Place Order' }).should('be.enabled');
```

src/components/Cart.js CHANGED

```
@@ -34,7 +34,6 @@ function Cart({ cart, dispatch, items }) {

    useEffect(() => {
      Modal.setAppElement('#root');
-     setThankYouOpen(true);
    }, []);

    const subTotal = cart.reduce((acc, item) => {
```

Conclusion

Run all your tests. (`npm run cypress:run`) On my machine, they run in 6 seconds. That's 3x slower than the component tests from the last chapter. End-to-end tests also had to redo most of the checkout flow to test checkout errors which you did not have to do in the component tests since you could pass in a cart.

So, why use end-to-end tests at all? Because these tests are uses the real server, they help test the integration between the server and React. Even if the tests mock the server, they help ensure elements are actually visible and not covered in the DOM which is only possible if the page is drawn in a browser. Cypress can also test in multiple browsers in case different browsers draw the page differently.

In many cases, you can write most tests with the faster component tests and just have a few end-to-end tests to catch covered elements and test integration with the server.

🥈 Phone Number

Test that typing in the phone number field causes the dashes to be added as expected.

Ex Typing "5555555555" should result in a value of "555-555-5555", and typing "555-5555555" should also result in a value of "555-555-5555".

🥈 Cart Details

Test that the tax details adjust properly.

Also, check the adding multiple different items and multiple of the same item works as expected.

🥈 Remove Item

Test that the remove item button works as expected.

🥈 More tests

Or implement some other test that you come up with.

Afterword

Congratulations! You are at the end of this guide. Not everyone has the discipline to do what you have done and learn what you have learned. Take a quick moment to give yourself a pat on the back.

Your hard work has paid off: You are now a React developer.

The Final Challenge

We have one last challenge for you: Become a *good* React developer. Good developers are good in their own ways, so you must find your own path from here on out.

Where might you start? Here are some ideas:

Write code. Now. You will quickly forget what you have learned here if you do not apply your knowledge. Contribute to a project or write a simple application of your own. Whatever you do, waste no time: Write code.

Learn. You have learned a little bit about a lot of things in this book. Did any of them spark your imagination? Write some code to play around with your favorite thing. Find and read more documentation about it – or an entire book, if there is one. Also, check out the React Podcast https://reactpodcast.com/ .

Meet people. Local meetups are a good place to meet like-minded developers. Lots of top-notch React developers are active on Twitter such as Sophie Alpert `@sophiebits`, Dan Abramov `@dan_abramov`. And you can attend conferences to meet other developers (maybe even us!).

Explore the open source community. Front-end development is exploding on www.github.com . When you find a cool library, check out other projects from its contributors. Share your own code, too – you never know who will find it useful or interesting. The React newsletter can clue you in to some cool things http://reactjsnewsletter.com/ .

You can also checkout https://overreacted.io/ .

Shameless Plugs

You can find us on Twitter `@bignerdranch`.

If you enjoyed this book, check out the other Big Nerd Ranch Guides at www.bignerdranch.com/books . We also have a broad selection of week-long courses for developers, where we make it easy to learn a book's worth of stuff in only a week. And of course, if

you just need someone to write great code, we do contract programming, too. For more info, go to www.bignerdranch.com .

Thank You

Without students like you, our work would not exist. Thank you for attending our class!