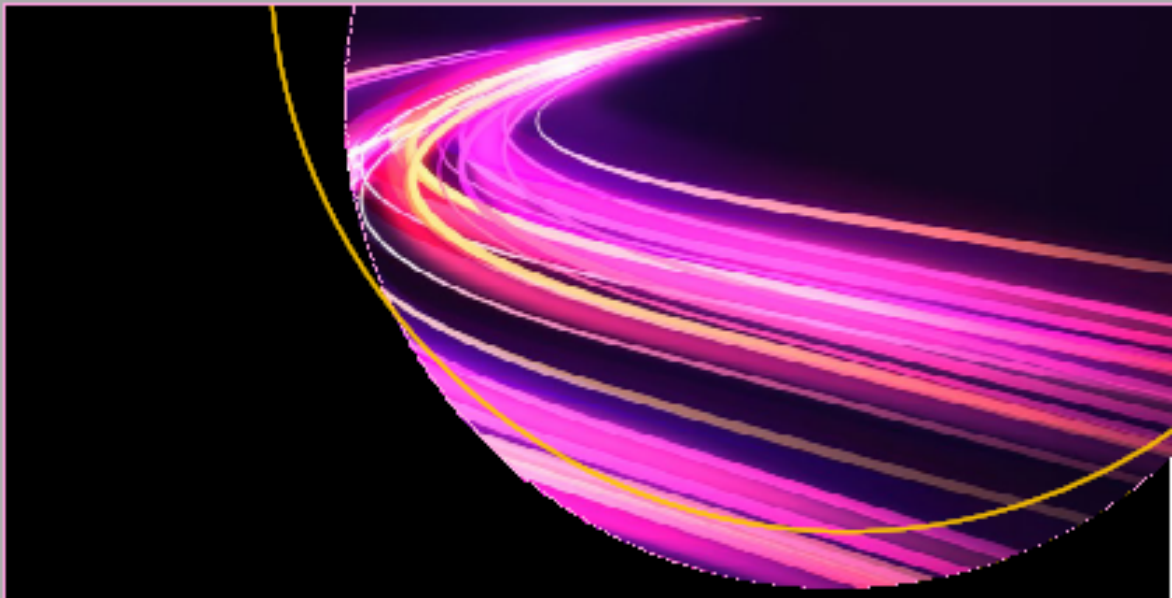

Modern Front-end Architecture

Optimize Your Front-end Development with Components, Storybook, and Mise en Place Philosophy

—

Ryan Lanciaux

Apress®

Modern Front-end Architecture

Optimize Your Front-end Development with Components, Storybook, and Mise en Place Philosophy

Ryan Lanciaux

Apress®

Modern Front-end Architecture

Ryan Lanciaux
Ann Arbor, MI, USA

ISBN-13 (pbk): 978-1-4842-6624-3 ISBN-13 (electronic): 978-1-4842-6625-0
https://doi.org/10.1007/978-1-4842-6625-0

Copyright © 2021 by Ryan Lanciaux

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484266243. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my family—to my wife, Rachel, and my amazing kids. They are incredible blessings from God!

Table of Contents

About the Author ix

About the Technical Reviewer xi

Acknowledgments xiii

Chapter 1: The Mise en Place Philosophy 1

Software Is Different, Right? 2

Good Software 2

What Does This Have to Do with Software? 4

Components 5

Storybook 9

Key Takeaways 10

Chapter 2: Configuring Our Workspace 13

System Requirements 13

Installing Node.js 14

Windows Installation Instructions 15

Confirming Node Version 16

Removing Existing Node.js Installations (Windows) 16

Install nvm-windows 17

Installing Node.js (Windows) 18

Mac/Linux Installation Instructions 20

Leveraging Node.js Built-In Utilities 21

Creating Our React Application 22
Running the Application 23
Adding Our Workspace 24
Storybook 24
Installing Storybook 25
Key Takeaways 26

Chapter 3: Our First Storybook Stories 29

Our First Storybook Stories 32
StoriesOf 33
Component Story Format 34
Storybook Add-ons 39
Story Variants 42
Key Takeaways 44

Chapter 4: Creating Reusable Components 47

What Makes a Good Component? 47
It All Comes Down to Purpose 47
Additional Components 49
When Should We Abstract Components? 51
Component States 51
How Should We Arrange Our Components? 54
Classifying Component Types 54
Key Takeaways 57

Chapter 5: Styling 59

CSS 60
Benefits of CSS 61
Drawbacks of CSS 61
Preprocessors 62

Benefits of Preprocessors 63
Drawbacks of Preprocessors 64
CSS-in-JS 64
Benefits of CSS-in-JS 67
Drawbacks of CSS-in-JS 67
Utility-First Styling Libraries 67
Benefits of Utility-First Libraries 68
Drawbacks of Utility-First Libraries 68
How to Choose a Styling Solution 68
Building a Theme 69
Key Takeaways 72

Chapter 6: Ensuring the Quality of Our Components 73

Unit Tests 73
Testing React Components 76
Testing Alongside Storybook 80
Key Takeaways 82

Chapter 7: Interacting with API Data 83

Some Main Considerations 84
Feature-Based Development 85
Loading Data 87
Container/Presentational Components 91
Mock Data 91
Wrapping Up 93

Chapter 8: Building Our Application 95

Navigating Between Pages 95

Routing 98

Updating Our Application to Use Routes 99

Navigation 104

Wrapping Up 106

Chapter 9: Automating Repetitive Tasks 107

Our Own CLI 108

A Brief Example 109

Building Some Generators for Our Project 111

Adding Additional Variables to Our Generator 114

Wrapping Up 116

Chapter 10: Communicating Our Components 117

Documenting Our Components 117

More Advanced Documentation 121

MDX 122

Sharing Our Workspace 124

Wrapping Up 124

Index 125

About the Author

Ryan Lanciaux is an independent software developer based out of Ann Arbor, Michigan. Concentrating on front-end development, Ryan helps organizations build scalable applications with a focus on efficiency and reusability. He regularly speaks at conferences and meet-ups and writes articles on the Web. You can find him on Twitter at @ryanlanciaux.

About the Technical Reviewer

Mwiza Kumwenda is a full-stack software engineer and content writer. Over the past decade, he has developed software for the following industries: banking, maritime, and electronic manufacturing. He is also interested in enterprise architecture, history, and politics. He likes to read in his free time.

Acknowledgments

I'd like to thank my parents for always encouraging me to explore my interests and my brothers, Nick and Joel, for being the best siblings you could ask for.

I'm super appreciative of the developer community in Ann Arbor, MI, and Toledo, OH. While we're not currently meeting in person due to a global pandemic, it's great to have a developer community to learn from and discuss ideas with.

Also, I'd like to thank Scott Sanzenbacher and Steven Cramer for helping me solidify my thinking on building applications with components. Through discussion, debate, and seeing things succeed or fail in real applications, their insight was instrumental to my current development processes.

Lastly, I'd like to thank the maintainers of React, Storybook, Cypress, Testing Library, and all of the other great open source tools that make the craft of developing software a lot easier and more fun.

CHAPTER 1

The Mise en Place Philosophy

There is nothing quite like the feeling of creating something that didn't exist before.

Whenever someone unfamiliar with coding asks me about why I am interested in software development, I like to talk about my love for creating things. While writing code, you can create your own worlds and build things that didn't exist previously.

The questioner may gloss over this response and see coding as an exercise in tedious algorithms, equations, and syntax. To those who write code and enjoy it, however, I would argue that this is a relatable mindset.

As opposed to software, many industries have years of standards, structure, and guidelines. Some may question if these principles help in the creation process. In most cases, they are both necessary and offer a head start. It would be tragic if someone in the construction industry threw out years of physics and structural engineering concepts to start with a "blank slate," so to speak. I wouldn't want to go to a dentist who performed creative root canals.

Think of the medical and aerospace industries. In both industries, the pace at which they are moving today is staggering. These fields are achieving great things by applying novel concepts to principles that have been around for decades. To start anything completely from scratch would be a terrible mistake.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_1

Software Is Different, Right?

Compared to many fields, software seems different. Software development, as a profession, is a very young field. Many developers are familiar with concepts like SOLID and design patterns from the Gang of Four. While helpful, these philosophies only go back several generations, not hundreds or thousands of years, as is the case in many other industries.

This relative "newness" of software development can be both freeing and problematic, especially in front-end development! Time and time again, a project that started off fun and exciting to work on turns into a nightmare after years of development. We often rewrite our applications only to find ourselves in the same situation a couple of years later. As we progress through this book, we'll discuss some strategies we can employ to keep our software both fun to work on and maintainable.

Good Software

Merely mentioning "good" software can cause fights among the developer population of the Internet. Many developers have some opinion of what good software is. While each person's opinion may be different, there is an ideal that we often don't live up to.

> *We build our computers [systems] the way we build our cities—over time, without a plan, on top of ruins.*
>
> —Ellen Ullman, programmer and author

At various points in my career, I've encountered software built like the proverbial city previously described. In these systems, you could see the reign of different CTOs in the company. You could tell when frameworks were popular and when they fell out of favor. It was clear where developers

gave up attempts to maintain any level of consistency and just tried to get something accomplished. These shortcomings ultimately led to applications being rewritten, only to start the process over again.

Have you ever

- Found code with the exact same functionality sprinkled throughout a codebase?
- Updated styles for one part of an application only to find out that you broke an entirely separate part of the application?
- Been on a team that found the challenges of maintaining a codebase so overwhelming that it warranted a rewrite? (Bonus question: Did the rewrite solve all the problems that were encountered?)

I don't know about you, but I can answer "yes" to all of the preceding questions. The sad thing about these responses is that things don't need to be this way. Few areas in the history of software prepare us for the constant change that besieges most front-end development teams. Thankfully, we can leverage ideas from other industries to write better software.

> *You don't have to know it all. Just take in the best big ideas from all these disciplines.*
>
> —Charlie Munger, investor

There are many industries that we, software developers, can obtain inspiration from, but one I particularly like is the culinary world—mise en place.

For generations, chefs tout the French term *mise en place* as a mindset that is critical to success in the kitchen. "What is mise en place?" you might ask. According to Wikipedia, mise en place means "everything in its place."

In many cases, a mise en place could refer to the setup that one takes before actually cooking a dish. This setup might look like organizing a workspace, chopping vegetables and other ingredients, and measuring seasonings, sauces, and other elements. When this workflow is in place, the final act of preparing a dish can consist of composing parts of the previous steps.

In some restaurants, the mise en place philosophy is so trusted that working with an entirely new kitchen staff is not seen as an impediment to success.

We'll see as we progress through this book that there are other parts to the mise en place philosophy we can use in software. For now, however, this focus on organization is where we'll start.

While constant staffing turbulence is to be avoided, it would be excellent to have this sort of resiliency in front-end applications.

What Does This Have to Do with Software?

You might be thinking, "Okay, this is neat, but what does this have to do with creating things outside the kitchen?" Adam Savage, the famed cohost of *MythBusters* and renowned crafter/maker, is a fan of the mise en place philosophy. He dedicates a whole chapter of his book to this concept. He describes the value of this philosophy outside the kitchen this way:

> *For all the alchemy that goes into building something, the magic of making is only possible because of the many repetitive processes we endure in preparation for final assembly.*
>
> —Adam Savage

In other words, a great outcome is only possible because of the little steps we take to build something.

In software development, there are many repetitive processes that we undertake. These processes may not always seem like they are helping us prepare for a great outcome, but with attention to detail and a focus on the individual aspects of an application, we can achieve great results. One such way we can focus on aspects of our application individually is by effectively leveraging components.

Components

We're in an exciting time for front-end development. Today's frameworks and libraries offer functionality that those developing front-end applications even five years ago could only dream of. Despite a lot of the new, fancy features that we get from today's libraries, the most significant impact is this convergence around components. Components are a central concept to Angular, Vue, React, Svelte, and others. Most browsers even support native web components!

Components let us break down our applications into what seems like a series of smaller applications. Frameworks like React provide an interface or contract that allows developers to compose components into other components. We can craft pages or screens out of a series of components. Let's take a look at a product search mockup shown in Figure 1-1.

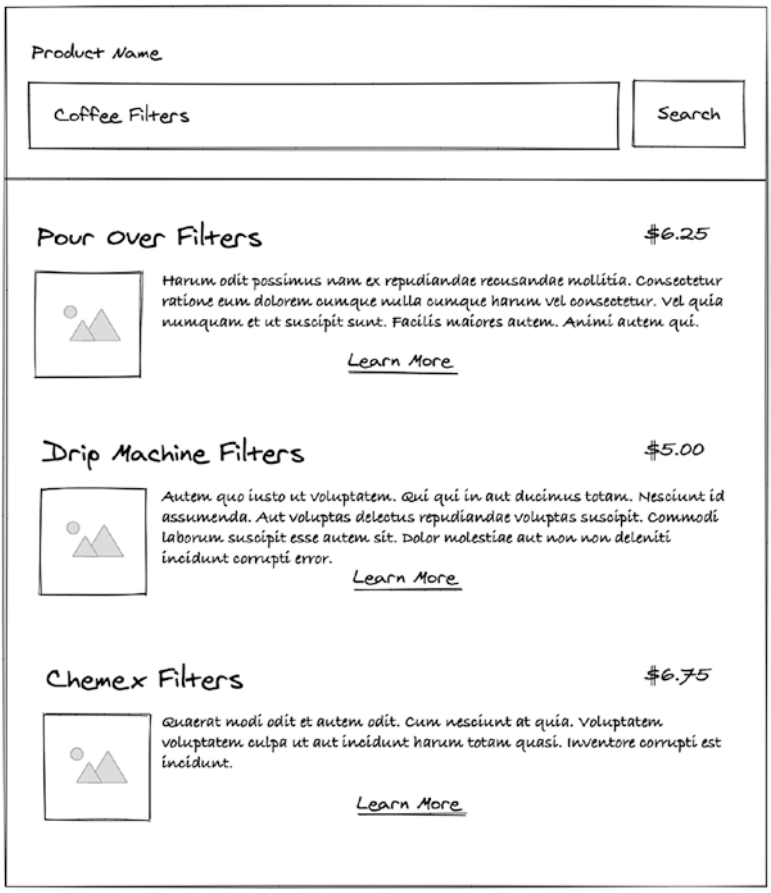

Figure 1-1. *A mockup of an online coffee retailer product search*

How do you imagine we would structure this product search? Do you visualize this mockup as a single page, or perhaps you see it as a series of components?

There's no wrong answer here, but in many cases, we could break this down into a product search input and search results. Continuing on this path, we could deconstruct these components to fields (the input and input label), buttons, photos, heading, text display, and links, as shown in Figure 1-2.

Figure 1-2. *Foundational components*

As ingredients in cooking can be used in other dishes, these components can be used as part of other screens.

This component concept that is foundational to many of today's front-end frameworks is great. Unfortunately, using components to build applications and using components effectively are not one and the same.

Meetings Don't Ensure Success

Earlier in my career, I was a front-end developer at an organization that had multiple teams working against a single codebase. Each team was responsible for its own part of the application. There should be no indication to someone using this application that they were switching into another team's domain.

A major challenge was making sure the teams were aligned and using similar components that looked the same. Every couple of weeks, some developers from these teams would meet to discuss the challenges and successes that their teams had. At almost every meeting someone would speak up, "We made a great component that solves this problem." Someone else would speak up, "Our team made one of these also." In some cases, it turned out that we had four instances of a component that served the same purpose!

This meeting was great to align on common, future goals, but it didn't help achieve this necessary consistency. There had to be a better way to not only make applications from a series of components but communicate what components exist.

The Workspace

In commercial kitchens, this mise en place philosophy is what promotes making consistent meals quickly. In many restaurants, there are different stations for a variety of elements of a meal. There could be stations for chopping vegetables, stations for cooking meat, stations for operating grilled items, and so forth. Pieces cooked in these various stations are composed together, transforming them into the meals served to customers.

If you're building web applications with components, you should consider adding a component workspace. This workspace highlights the components rather than standard application logic. A component workspace is a separate web application that you can run in development instead of your web application.

Building components in isolation has numerous benefits, but there are several key things to keep in mind:

- Using a component workspace can help us write truly reusable components. It's easy to have tunnel vision while working on a component as part of a page. Instead of thinking about the responsibilities of the page, we can focus on the responsibilities of the component.
- Focusing on one component at a time helps us ensure the quality of the component we are building. Like a chef in a kitchen taste testing individual parts of a meal to ensure a stellar outcome, high-quality components lead to better applications.
- A component workspace can communicate with other developers, designers, and stakeholders which elements are available in a system. This communication helps us avoid scenarios where components are duplicated simply because others were not aware that they existed.

Storybook

In this book, we will use Storybook as our component workspace. Storybook is an open source project to assist in building component libraries. While there are other ways to build component libraries and many great tools, Storybook has a mature developer ecosystem and is a solid choice.

In the next few chapters, we'll start coding with Storybook and React, but for now, let's take a quick look at what Storybook is in Figure 1-3.

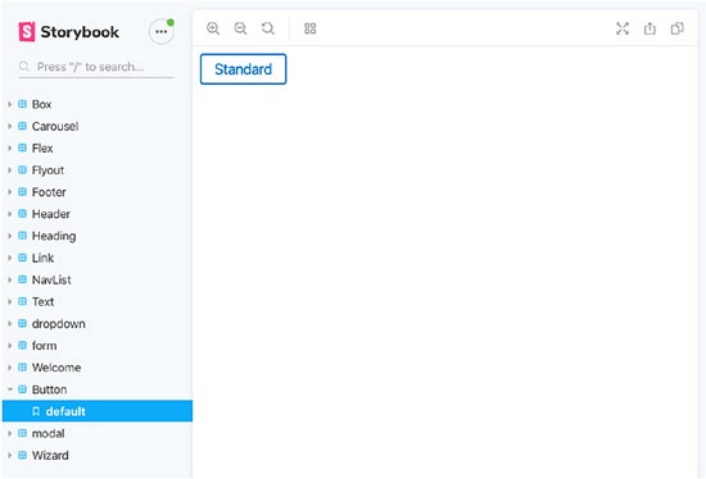

Figure 1-3. *Storybook inside a sample project*

Using Storybook, we interact with our workspace through stories. Stories are visual representations of a component in a specific state.

By default, we see our stories in a treelist view with all available component stories on the left and the workspace on the right. Selecting a different story from the treelist changes the component that's available in the workspace. Typically, we will configure our application to run with a script for launching Storybook by itself. This may look something like *npm run storybook* or *yarn storybook*.

In the next chapter, we'll begin adding Storybook to a project and continue to see some of the benefits that this tool unlocks in our codebase.

Key Takeaways

In this chapter, we talked about some of the challenges that we encounter in the front end today. Consistency, communication, and developer productivity can be hard to achieve in a large codebase.

Other industries have strategies for organization and systems in place to help achieve consistency and quality in an efficient way.

Mise en place is one such strategy that is used in the culinary world that we can use in software development. Mise en place refers to the organization and preparation that goes on prior to assembling a meal.

A workspace is an integral part of mise en place in commercial kitchens. We can think of "mise en place" as both the architecture and organization we apply to our codebase. Developers can benefit from a workspace to help focus on parts of the codebase in isolation.

Storybook is an excellent tool that serves as a workspace for our code.

CHAPTER 2

Configuring Our Workspace

Now that we understand a bit more about the mise en place philosophy we want to emulate, let's move on to the code. In this chapter, we're going to ensure our computer is set up and configured for our success.

System Requirements

Before we embark on our adventure, let's ensure that we have a great JavaScript environment at our fingertips. We'll walk through operating system–level details in the following, but from a high level, our requirements are

- **Node.js version 8 or higher** – In this book, we're going to assume you're using v10.
- **An editor you feel comfortable using for working with JavaScript files** – Visual Studio Code (`https://code.visualstudio.com`) is a solid choice, but any editor should work. This editor is a great choice because it works on many operating systems, has a great set of plugins, and has a large number of users.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_2

Note You should consider **only** using even-numbered major versions of Node.js for production applications. According to the Node.js documentation, even-numbered major releases have long-term support, or they will receive bug fixes for 30 months. Odd-numbered releases only have six months of support (https://nodejs.org/en/about/releases/).

Even though we are not working with Node.js directly, the tools and libraries that we'll be using, such as npm (a CLI tool for managing installed Node packages) and Create React App (CRA), depend on Node.js.

Installing Node.js

Today, Node.js comes preloaded on many computers and operating systems. Unfortunately, depending on a system-wide Node.js version can cause trouble as many codebases depend on a particular version of Node.js.

What happens if we are working on several applications where each expects a different version of Node.js? How do we know we update Node.js for one of our projects without changing all of our codebases? These questions may seem like hard problems to solve. Thankfully, version managers make this a nonissue.

Version Managers

A version manager is a software that allows us to install and manage many Node.js installations on our computer. This software will enable us to easily use one version of Node.js in one application and a completely different version in another. We are going to use nvm (https://github.com/nvm-sh/nvm) on Mac/Linux and nvm-windows (https://github.com/coreybutler/nvm-windows) on Windows to achieve version management excellence.

Note While the names of the projects Node Version Manager (NVM) and Node Version Manager for Windows are similar, they are entirely separate projects that have a similar purpose.

Throughout the next several sections, we will walk through how to configure our device to run Node.js. Feel free to jump to the operating system you use and skip all the rest.

Windows Installation Instructions

As noted previously, we're going to use nvm-windows to manage Node.js versions on our device. We'll start out by opening the Command Prompt. From the Start menu, type `cmd` and open the Command Prompt, as shown in Figure 2-1.

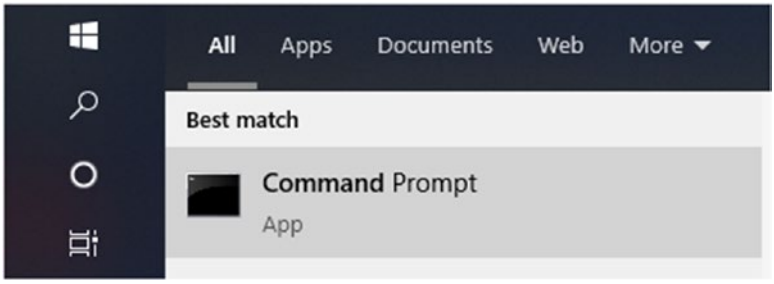

Figure 2-1. *Command Prompt application*

Confirming Node Version

We want to ensure that we don't have any existing versions of Node.js installed. We can check this from the Command Prompt by typing *node -v.* This command will show the version of Node.js that is installed on our device or confirm that Node.js is not present. For instance, the following is what it looks like if Node.js is currently installed:

```
> node -v
v10.20.1
```

whereas a device without Node.js appears as

```
> node
'node' is not recognized as an internal or external command,
operable program or batch file.
```

If we see a version listed, it means we currently have Node.js installed and should remove it before proceeding with a version manager.

Removing Existing Node.js Installations (Windows)

Caution Proceed with careful consideration before applying these changes. Thoughtfully evaluate the impact these changes could have on your device. While these instructions work for me and countless others, you are making changes to the software that other applications on your system may depend on. This operation may be unnecessary depending on the version of Node.js you have on your computer.

We'll start off by navigating to *Apps & features* in Windows Settings to remove the installed Node.js runtime. Click Node.js and select Remove.

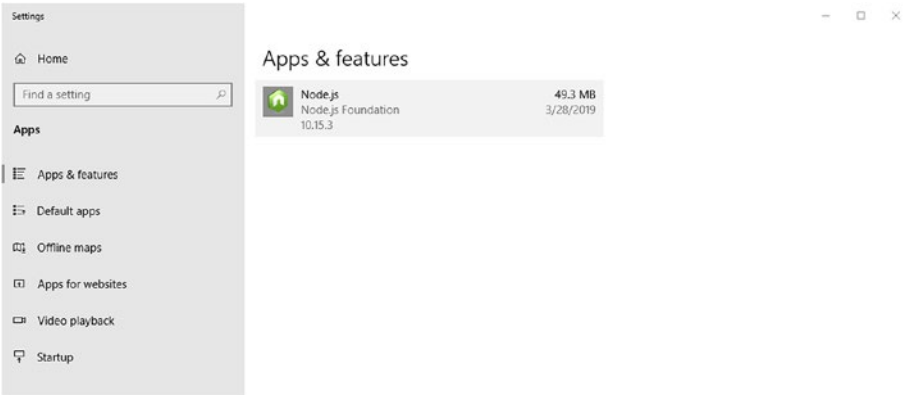

Figure 2-2. *The Apps & features window*

Next, we'll ensure that Node.js is removed by opening the Command Prompt again and typing *node -v*. Upon typing node, we should be presented with the message that node is not recognized as a command:

```
> node
'node' is not recognized as an internal or external command,
operable program or batch file.
```

We're now ready to install nvm-windows.

Install nvm-windows

NVM for Windows is a Node.js version manager software running on the Microsoft Windows operating system. Navigate to the Node Version Manager for Windows project on GitHub available at `https://github.com/coreybutler/nvm-windows` and download the latest nvm-windows installer (`https://github.com/coreybutler/nvm-windows/releases`). Once downloaded, we can run the installer using the default settings.

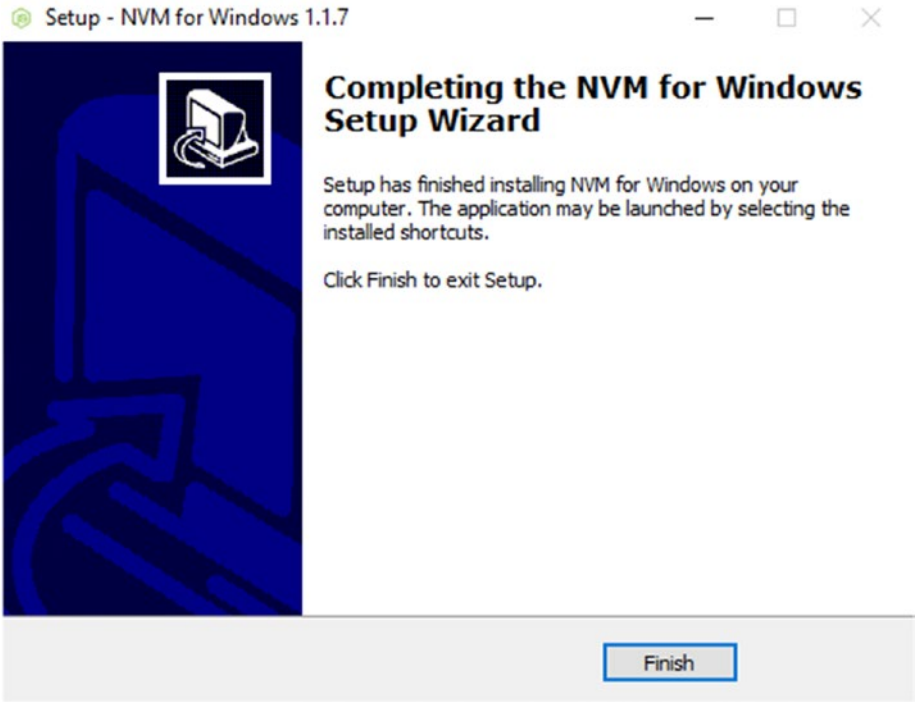

Figure 2-3. *The NVM for Windows installation wizard*

Once the installation is complete, we are ready to install Node.js.

Installing Node.js (Windows)

We're going to use the command line once again. From the Start menu, type cmd. Right-click the Command Prompt app (Figure 2-4) listing and select "Run as administrator." Node Version Manager requires running as an admin.

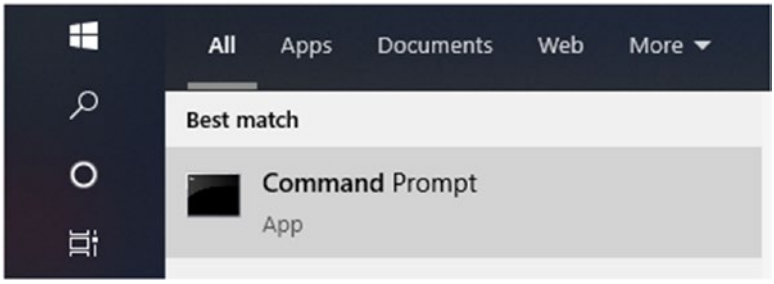

Figure 2-4. *Start the Command Prompt as an administrator*

Once we're in the Command Prompt, type "nvm." If everything is installed as expected, we should see the version of nvm followed by a list of usage instructions:

```
> nvm
Running version 1.1.7.
```

From here, we're ready to turn nvm on and install a version of Node.js:

```
> nvm install 10.20.1
Downloading node.js version 10.20.1 (64-bit)...
...
Installation complete. If you want to use this version, type
nvm use 10.20.1
```

Now, if we run `nvm on`, nvm will use the version of Node.js we just installed:

```
> nvm on
nvm enabled
Now using node v10.20.1 (64-bit)
```

```
> node -v
v10.20.1
```

Node.js is now configured and running on our Windows computer. Feel free to skip the following section on how to configure a Mac or Linux device.

Mac/Linux Installation Instructions

If you skimmed through the Windows installation instructions, you may have noticed that there were quite a few steps to install a Node.js version manager on Microsoft's platform. On Mac and most Linux devices, the installation instructions are much simpler. Following along with the nvm project's documentation, we can install nvm on our machine by downloading and running an install script. Launch the terminal you use and run the following command:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/
install.sh | bash
```

This command uses curl to obtain an install script and pipes it into our bash shell. In other words, it takes the output of the downloaded script and runs it in bash. At the time of writing, the latest version of nvm is v0.35.3. The referenced install command references this version. To obtain the most up-to-date version, please view the nvm project's documentation on GitHub: `https://github.com/nvm-sh/nvm`.

Verify Version Manager Installation (Mac/Linux)

Continuing with the project's documented steps, we should now confirm that our installation is a success. We should start off by either restarting our terminal or running *source ~/.bashrc* (or *~/.bash_profile, ~/.zshrc,* or the file containing the shell configuration). From there, run the following:

```
> command -v nvm
nvm
```

If the command output is `nvm`, it means that the installation was successful. If you encounter any problems, consult the nvm project documentation (https://github.com/nvm-sh/nvm) as it contains some excellent troubleshooting tips.

Install Node.js (Mac/Linux)

We can now install a version of Node.js and set it as the default version on our system:

```
> nvm install v10.20.1
Downloading and installing node v10.20.1...
...
Now using node v10.20.1 (npm v6.14.4)
```

Since nvm allows us to install many versions of Node.js on our computer, it can be a good idea to set a default version. Run the following command to set the recently installed v10.20.1 as the default Node version:

```
> nvm alias default v10.20.1
default -> v10.20.1
```

We're all set up and ready to start creating our application.

Leveraging Node.js Built-In Utilities

Now that our system is configured, we're ready to begin creating our application. Before we do this, however, let's briefly discuss two built-in tools that we will use while building our application, `npm` and `npx`. The npx tool allows us to run any command-line library stored in the npm

registry on our device. Before npx, it was necessary to install a CLI tool locally before using it. Let's take a look at how we can create a React app leveraging npx:

```
# with npx we can create a react app quickly
> npx create-react-app MyApp

# previously a command line tool would need to be installed
before it could be used
> npm install -g create-react-app
. . .
> create-react-app MyApp
```

Creating Our React Application

We're finally ready to start constructing the application that we'll be building upon throughout this book. We're going to build a web experience for a coffee and tea shop—we'll call it Rocket Coffee.

Most applications have humble beginnings, and ours is no different. Right now, we simply want to build something that is basically a "Hello World." While there are many ways to build a React application, we're going to stick with the "Create React App" tool released by the React team at Facebook (https://create-react-app.dev/).

We'll start by running the npx command to create our application. This will handle all of the setup steps that we need to build a React application:

```
> npx create-react-app rocket-coffee
. . .
Happy Hacking!
```

This command initialized a new React app. This command took the steps of creating a new Node.js package, installing the default dependencies, and providing a starting point for our application. We can now run our basic application and see it in action.

Running the Application

To run the application, we'll navigate to our newly created directory, "rocket-coffee," and run the script that starts our application. In the root of our app directory is a package.json file that contains the listing of dependencies, scripts, and other information about our application:

```
> cd rocket-coffee
> npm run start
```

Once the application is running, we should be able to navigate to `http://localhost:3000` to see the default React application as seen in Figure 2-5.

Figure 2-5. *The default output of the Create React App application running in a browser*

Fantastic! Our application environment is now fully operational. Before we get started on the app-level features, let's add our component workspace.

Adding Our Workspace

Now that we have our dependencies in place and a working "Hello World"–style application, we should add a component workspace. As we discussed in Chapter 1, a component workspace can be an incredible asset in creating excellent software as opposed to building components directly within the pages where they live.

This component workspace should be a little bit like the different stations in a restaurant where culinary artists build the elements of a dish. Similarly, our component workspace will be where we focus on individual elements of a page/screen.

Storybook

Storybook is an excellent tool for building component libraries. This tool is organized around the concept of stories. A story represents a unique state of a component. A quick look at an example Storybook is seen in Figure 2-6.

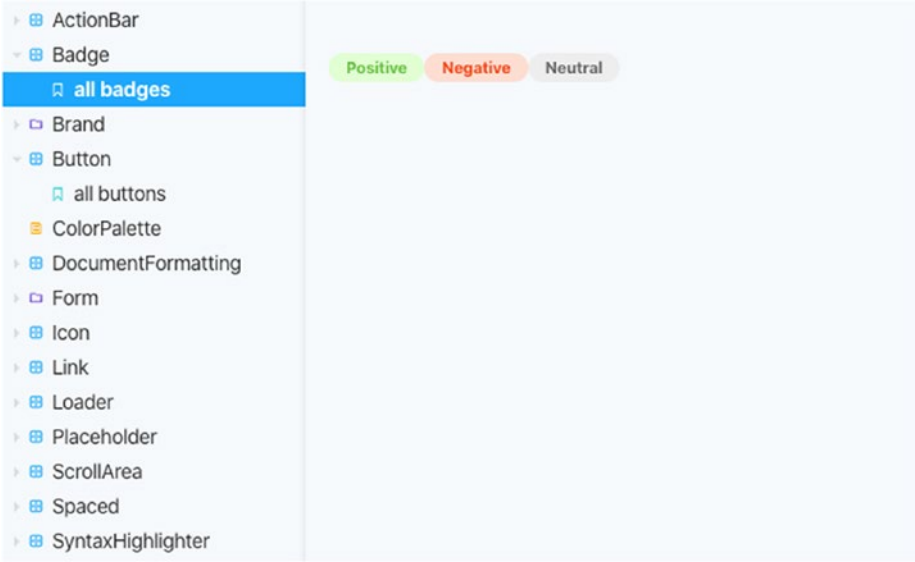

Figure 2-6. *Example Storybook implementation (from the Storybook Examples)*

Let's break this down a bit. On the right-hand side of Figure 2-6, we see the visual output of the story. Each component can have many stories, but in this example, we only have one.

On the left hand, we see a listing of all possible stories, organized in a tree view. We can change the active story by selecting a different item in this tree view.

Installing Storybook

From the terminal within our project directory, we'll run the following to install Storybook in our application:

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

This command will analyze our codebase, install Storybook, and add some scripts to our package.json file to make it easy to run the tool. Let's run Storybook from the command line now:

```
> npm run storybook
```

After this command completes, we should see the default example Storybook stories (Figure 2-7).

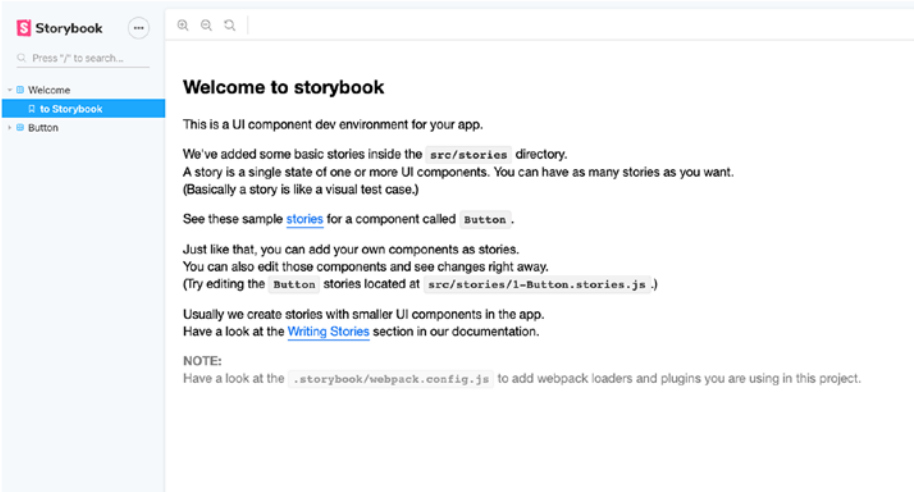

Figure 2-7. *Default example Storybook stories*

In the next several chapters, we'll begin to leverage Storybook to help us build well-structured, component-based applications. For now, however, take some time to explore the out-of-the-box Storybook experience.

Key Takeaways

This chapter was all about preparation. In many cases, using a system-wide version of Node.js can cause problems that we would like to avoid. We installed Node.js through a version manager that won't tie us to a particular version of the framework.

We created a React application through Create React App and initialized a default Storybook component workspace. Although preparing a codebase may not be as fun as writing the code that powers our applications, it is an integral step in obtaining a great outcome.

Before anything else, preparation is the key to success.

—Alexander Graham Bell

CHAPTER 3

Our First Storybook Stories

Now that the system configuration is complete and you've created an application and added Storybook, we'll dive into creating our real Storybook stories and components that serve as the basis of our application. In this chapter, we're going to dive into the code a bit more. Feel free to follow the steps we're taking directly, or navigate to `github.com/ryanlanciaux/example-rocket-coffee` and view the code on the branch labeled chapter3.

We are going to create an ecommerce store for a coffee/tea shop called Rocket Coffee. We'll pretend our designer gave us some designs to work with but not much direction outside of that. Let's take a look at the product listing in Figure 3-1.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_3

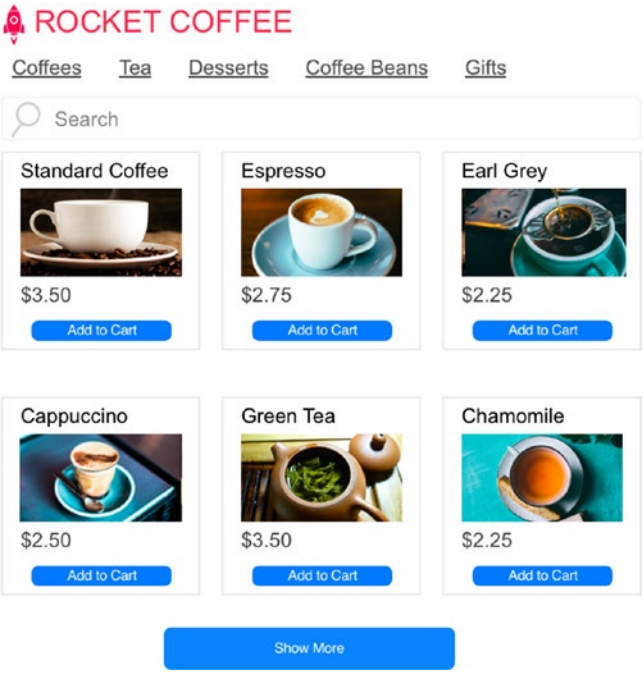

Figure 3-1. *Home page design*

We don't want to tackle this entire page at once. For now, we're going to focus on building a component for a single product item (Figure 3-2).

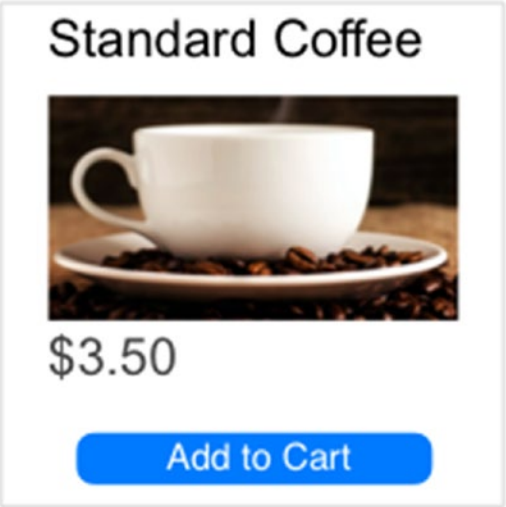

Figure 3-2. *Product item*

We'll continue with the React app we started in Chapter 2. Let's start off by creating a new folder under *src* called *components*. From there, we'll add a folder called *ProductListItem* with two JavaScript files, *index.js* and *ProductListItem.js*. We should end up with something like in Figure 3-3. You may notice that we're capitalizing the folders and filenames for components but using standard pascalCase for other types of files.

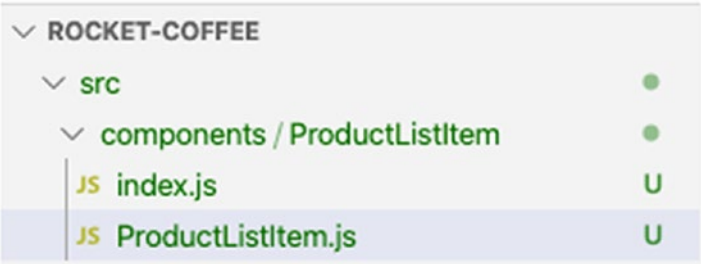

Figure 3-3. *Folder/file structure*

With our file and folder structure in place, we're ready to start adding some code. We'll begin by adding what amounts to the component equivalent of a "Hello World" application to ensure that we have everything wired up properly:

```
// ProductListItem.js
import React from 'react';

export default function ProductListItem() {
  return <p>Hello from Product List Item!</p>
}

// index.js
export { default } from './ProductListItem';
```

Note We could place all our code inside the index.js file and avoid creating what's basically a pass-through file. I've often found that storing my code with a very specific filename helps me find that code later. While a well-structured application will be broken down into concise folders, creating a separate file for your component can help you avoid getting lost in a sea of *index.js* files.

Our First Storybook Stories

We're now ready to see these new components in action. Instead of updating our application screens to include the components while we're actively developing them, we're going to add a story to serve as our workspace.

There are a couple of ways that we can create Storybook stories. Each has its own benefits and trade-offs. In the following examples, we'll end up with a Storybook workspace that looks like the image in Figure 3-4

after running *npm run storybook* (or *yarn storybook*) from the command line. We will create a story for the ProductListItem */src/components/ProductListItem/ProductListItem.stories.js*. You might have noticed that there is a global *stories* folder that's created when we install Storybook. Some teams prefer to place their stories in a folder outside the main src directory. We will place our stories next to the components they represent instead of this global stories folder. This is a personal/team preference item for discussion, but I often find that it's easier to navigate stories when they are near the components they describe.

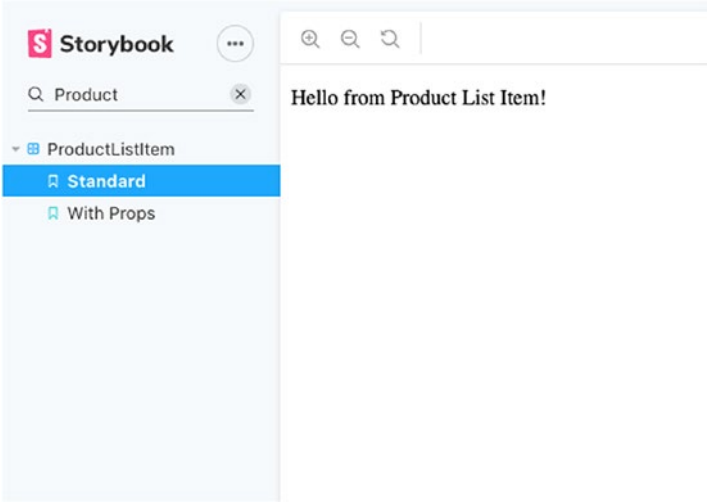

Figure 3-4. *First ProductListItem story*

StoriesOf

The StoriesOf API is the original way to create stories in Storybook. StoriesOf uses a custom function, provided by Storybook, to define our stories and render them in a structured way in our component workspace. For our example, usage of the StoriesOf methods may look like the following code example:

```
import React from 'react';
import { storiesOf } from '@storybook/react';
import ProductListItem from './ProductListItem';

storiesOf('Product List Item', module)
```

.add('standard', () => <ProductListItem />)

The StoriesOf API is a FluentInterface or a mechanism to use method chaining to define our stories. We supply the story group in the *storiesOf* method and the subsequent stories with the chained *add* method. We could continue to tack on add methods, but for right now, we're only representing one state of our component. We're supplying the story name as the first parameter followed by the story method as the second parameter. The story method can be a standard function, but I find the arrow functions a little more readable.

Component Story Format

In 2019, the Storybook team released another strategy for creating stories called the Component Story Format. The Component Story Format is a succinct way to create stories using, what is mostly, idiomatic JavaScript. While the StoriesOf API uses a custom, Storybook-specific interface to define stories, the Component Story Format uses JavaScript objects and functions:

```
import React from 'react';
import ProductListItem from './ProductListItem';

export default { title: 'ProductListItem' };

export const standard = () => <ProductListItem />
```

Using the Component Story Format, we create an object with the story properties (or metadata) as the default export. In this case, we've set the property of the story title to ProductListItem. Every named export function will be picked up by Storybook as a story to display.

According to the documentation, this API is the recommended way to write stories and is how we will write stories throughout the remainder of this book. There are many great reasons for this recommendation as we'll see in later chapters. From a high level, stories written with the Component Story Format are more portable and reusable in other areas of our codebase, such as tests, which we'll explore deeper in Chapter 6.

Component Code

Now that we've wired up our Storybook and ensured that the story and our component are in sync, we're ready to create the actual component code. We'll start out by thinking about what our component should do. This ProductListItem is responsible for displaying the product name, image, price, and a function that will be activated when a shopper clicks "Add to Cart." In code, we'll translate this to the React props, the inputs that the component receives, as *name, price, image,* and *onAddToCart*. Our example component could look like the following code output. If you're not currently running Storybook, now would be a good time to run *npm run storybook* (or *yarn storybook*).

We'll update ProductListItem.js to the following:

```
import React from 'react';
import './ProductListItem.css';

export default function ProductListItem({ name, price,
imageUrl, onAddToCart }) {
  return (
    <div className="card">
      <h2>{name}</h2>
```

```
      <img src={imageUrl} alt="" />
      <small>{price}</small>
      <button onClick={onAddToCart}>Add to Cart</button>
    </div>
  );
}
```

You may notice that we're referencing a CSS (Cascading Style Sheets) file that doesn't exist yet. The Storybook output should be displaying an error noting that there's "No such file or directory" for *ProductListItem.css*. Go ahead and create an empty *ProductListItem.css* file and check the Storybook output for the ProductListItem story.

Next, we'll update our story to include values for the necessary props:

```
import React from 'react';
import ProductListItem from './ProductListItem';

export default { title: 'ProductListItem' };
export const standard = () => (
  <ProductListItem
    name="Standard Coffee"
    price="2.50"
    onAddToCart={() => {
      console.log("CLICKED");
    }}
    imageUrl="https://source.unsplash.com/tNALoIZhqVM/200x100/"
  />
);
```

We're supplying some fake values to this component as well as an excellent coffee image that we're pulling in from the Unsplash placeholder image API.

Now, if we go back to Storybook, we should see something like Figure 3-5 in the content area of the Storybook window.

Figure 3-5. *First phase of the ProductListItem output in Storybook*

It looks like all the items are present, but the style doesn't quite match up. We'll make this a bit better with the following CSS in our CSS file:

```
.card {
  display: flex;
  flex-direction: column;

  border: 1px solid #EDEDED;
  padding: 8px;
  max-width: 240px;
}

.card button {
  background-color: #0A84FF;
  color: #fff;
  padding: 8px;
  border-radius: 4px;
}

.card small::before {
  content: '$'
}
```

While there are a lot of things we could improve here, this is "good enough" styling for now. We're creating a card style for our ProductListItem, as well as stating that we want to arrange the component as a column view. Finally, we're applying some basic colors and noting that a "$" should show up before the element that we're using to display prices. Our story output should look like Figure 3-6.

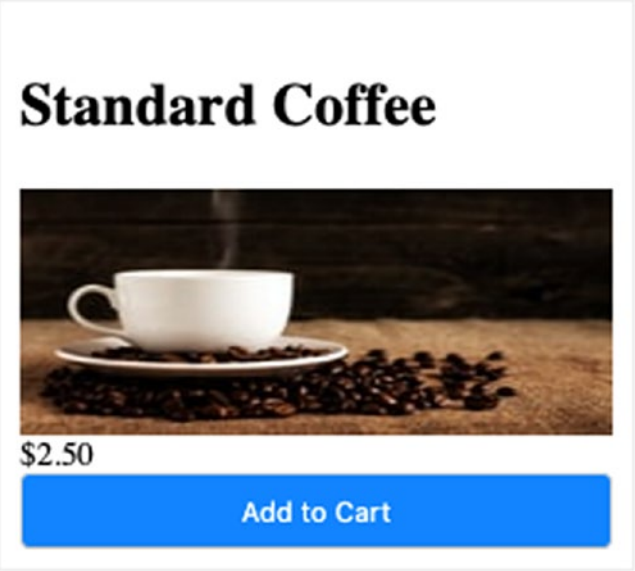

Figure 3-6. *Updated ProductListItem*

Clicking Add to Cart should now be adding log messages with the text "CLICKED" in the browser's JavaScript console. This can be accessed on Chrome by selecting "View ➤ Developer ➤ JavaScript Console" from the browser's file menu. While it's useful to see that our callbacks are working, there is a much better way to interact with our components.

Storybook Add-ons

Storybook has a series of add-ons that provide additional capabilities to the platform. There are a couple that come preinstalled when we installed Storybook. We're going to use Storybook's *actions* add-on to remove our log statement in the story while still seeing when our event runs.

We'll start by importing "@storybook/addon-actions" and referencing the "action" method instead of "console.log." We'll pass the label we want to apply to the callback as the parameter to this method, "Add to cart clicked." When we're done, our updated story for *ProductListItem.stories.js* should look like the following (only relevant updates displayed):

```
...
import { action } from '@storybook/addon-actions';

export const standard = () => (
  <ProductListItem
    ...
    onAddToCart={action("Add to cart clicked")}
    ...
  />
);
```

When we click the action, we should see a pane "Actions" at the bottom of our Storybook window that displays a message when the "Add to Cart" button is clicked. If additional parameters are sent with the callback, these values will show up here as well.

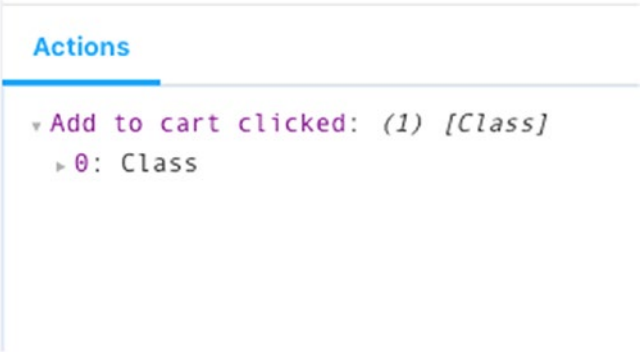

Figure 3-7. *Storybook Actions output*

Now, what if we want to see our component with different names, prices, and other parameters, without hardcoding a new story for each? This is a perfect use case for the *Knobs* Storybook add-on.

We'll add Knobs to the project *npm install —save-dev @storybook/addon-knobs* and import and register it in *.storybook/main.js*:

```
module.exports = {
  stories: ['../src/**/*.stories.js'],
  addons: [
    '@storybook/preset-create-react-app',
    '@storybook/addon-actions',
    '@storybook/addon-links',
    '@storybook/addon-knobs'
  ],
};
```

Next, we'll switch back to our story, import Knobs, and register it as a decorator in our story options. The story options object is our default export that previously contained the story's title. The decorators property is an array of add-ons we wish to register with this story:

```
import { text, withKnobs } from '@storybook/addon-knobs';
export default {
```

```
  title: 'ProductListItem',
  decorators: [withKnobs]
};
```

Following this, we'll want to switch out our hardcoded text props to take advantage of the newly imported Knobs. We'll supply a name to our knob as well as an initial value:

```
export const standard = () => (
  <ProductListItem
    name={text("Name", "Standard Coffee")}
    price={text("price", "2.50")}
    onAddToCart={action("Add to cart clicked")}
    imageUrl={text("imageUrl", "https://source.unsplash.com/
    tNALoIZhqVM/200x100/")}
  />
);
```

We are only using the text knob currently, but the add-on supports arrays, numbers, Boolean, dates, colors, and many others. Since we changed .storybook/main.js, we will need to restart our Storybook server. Once the server is restarted, at the bottom portion of our Storybook window, we should see the Knobs tab added to the add-ons pane.

Figure 3-8. *Storybook Knobs add-on form*

Typing in this pane will update our component in real time. Knobs are a fantastic way to interact with our components, but Storybook provides other ways that we can see our components in different states as well.

Story Variants

How should we represent different global states of our component, such as noting that an item is sold out? In the following example, let's assume that we add a prop called *isSoldOut* to our ProductListItem component. This prop will determine if the "Add to Cart" button is enabled, as well as what text should be displayed inside the button:

```
export default function ProductListItem({
  name,
  price,
  imageUrl,
  onAddToCart,
  isSoldOut
}) {
  return (
    <div className="card">
      <h2>{name}</h2>
      <img src={imageUrl} alt="" />
      <small>{price}</small>
      <button onClick={onAddToCart} disabled={isSoldOut}>
        {isSoldOut ? "Sold out" : "Add to Cart"}
      </button>
    </div>
  );
}
```

We could interact with this property as a Boolean knob, as we did with our text props. While this would work, it may be clearer to highlight this as a unique story:

```
export const soldOut = () => (
  <ProductListItem
    name={text("Name", "Standard Coffee")}
    price={text("price", "2.50")}
    onAddToCart={action("Add to cart clicked")}
    imageUrl={text("imageUrl", "https://source.unsplash.com/
    tNALoIZhqVM/200x100/")}
    isSoldOut
  />
);
```

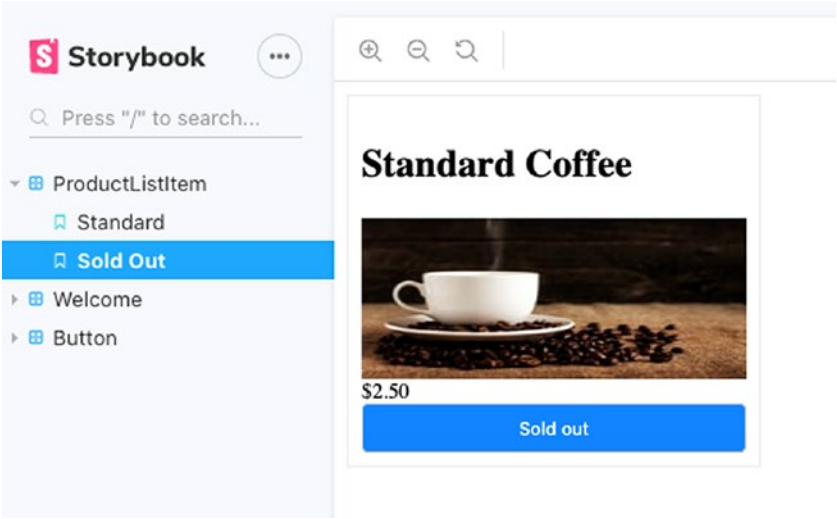

Figure 3-9. *Sold out story*

When Should We Create a Story vs. a Knob?

You may have noticed that we used different techniques to change how we're presenting our components. The distinction between what could be a knob vs. what warrants a new story can be a nuanced decision. While they are conceptually the same, in other words, changing which props are passed into a component, I generally like to use the following criteria:

Knob – For changing props that do not greatly change how a user is interacting with a component or change the purpose of the component.

Story – When the purpose of the story is impacted by the addition of a given prop or props. In our case, we slightly changed the purpose of the ProductListItem when we passed in the *isSoldOut* prop. Shoppers can no longer purchase an item by interacting with this component—it's purely an indicator noting that there is an item that's typically available but it's currently sold out.

One other mechanism that I use when determining when to use a knob vs. a new story is whether or not I would like to write unit tests for a component with the presence of given props. Any time I find it would be helpful to test a component with specific props, it generally is an indicator that a separate story would also be useful.

Key Takeaways

In this chapter, we created our first Storybook stories and discussed the front-end application that we'll be working on throughout a good portion of this book. We walked through a couple different ways that we can construct our stories and why we'll be using the Component Story Format to craft our stories. Finally, we talked about how to supply variants to our stories through Knobs and creating new stories and some reasons for choosing one strategy over another.

If you'd like to dive in a bit further, there are some ways that you can make this ProductListItem a little more ready for production.

EXERCISE: HIGHLIGHT A PRODUCT AS ON SALE

We want to call out a product as an item we want to sell. This should be a little bit like the isSoldOut prop, but it should cause the background color to change to #E8F6FF and should add the text (On Sale) after the product name. Create a new story to represent the On Sale state of the ProductListItem. When complete, Storybook should look like Figure 3-10.

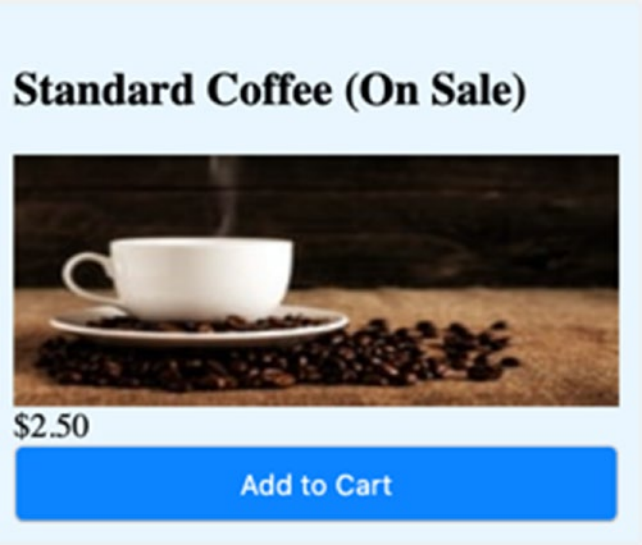

Figure 3-10. *Sale ProductListItem*

CHAPTER 4

Creating Reusable Components

So far, we've created a product list component for our Rocket Coffee example. While this component works, we want to start employing strategies that ensure reusability and resiliency to change.

What Makes a Good Component?

Within the front-end ecosystem, there are many opinions about how a component should be structured. If we polled the community about how we should structure our product list example, we might receive conflicting suggestions. Some may say we should simply make one component for all the product list item concerns. Others may suggest creating a new component for every HTML element.

Either of these strategies will work, but I find there's often nuance in what makes a good component.

It All Comes Down to Purpose

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.*
>
> —Doug McIlroy, programmer known for proposing Unix pipelines

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_4

I like to structure my components around purpose. Like the Unix philosophy, components should do one thing well. A series of smaller components working in concert can serve us better than HTML tag soup that can plague our front-end code.

Have you ever been involved in a project where adding a feature was challenging because it was hard to determine which div to place the new feature in? Building components by purpose can help us clearly see how a component is structured.

Let's go back to our product listing that we looked at in Chapter 3. As a refresher, our ProductListItem looks like this:

```
import React from 'react';
import './ProductListItem.css';
export default function ProductListItem({ name, price,
imageUrl, onAddToCart }) {
  return (
    <div className="card">
      <h2>{name}</h2>
      <img src={imageUrl} alt="" />
      <small>{price}</small>
      <button onClick={onAddToCart}>Add to Cart</button>
    </div>
  );
}
ProductListItem.js (from Chapter 3)
```

While this isn't too complex, we have plans to build a larger application. Many areas of our codebase will need ways to present content within a consistent card, display text and headings, handle click/press interactions, and so on. With that in mind, let's refactor this component into a series of reusable components.

Additional Components

We'll start off by moving the reusable elements to be wrapped in their own components. These will mostly be pass-through components for now—that is, we'll simply pass props to the JSX elements. As we progress through this book, we'll apply some powerful functionality to these components, taking advantage of this separation:

```
function Heading({ children }) {
  return <h2>{children}</h2>;
}
function Card({ children, highlight }) {
  const cardClassName = highlight ? "card sale" : "card";
  return <div className={cardClassName}>{children}</div>;
}
function Text({ children }) {
  return <span>{children}</span>;
}
function Button({ onClick, children }) {
  return <button onClick={onClick}>{children}</button>;
}
```

With this in place, we can update our existing ProductListItem:

```
export default function ProductListItem({
  name, price, imageUrl, onAddToCart,
  isSoldOut, isOnSale,
}) {
  return (
    <Card highlight={isOnSale}>
      <Heading>
        {name} {isOnSale && "(On Sale)"}
      </Heading>
```

```
      <img src={imageUrl} alt="" />
      <Text>{price}</Text>
      <Button onClick={onAddToCart} disabled={isSoldOut}>
        {isSoldOut ? "Sold out" : "Add to Cart"}
      </Button>
    </Card>
  );
}
```

You may notice that the structure is relatively close as when we were using HTML elements in this ProductListItem component. However, we're mostly using components that match the purpose they serve. Assuming we were looking at this code fresh, we'd have a pretty clear understanding of what it was doing just based on the component structure. Instead of needing to parse CSS classNames and make sense of the hierarchy, our component structure gives us some pretty strong clues.

If we type *npm run storybook* in our terminal, Storybook should load as it did before.

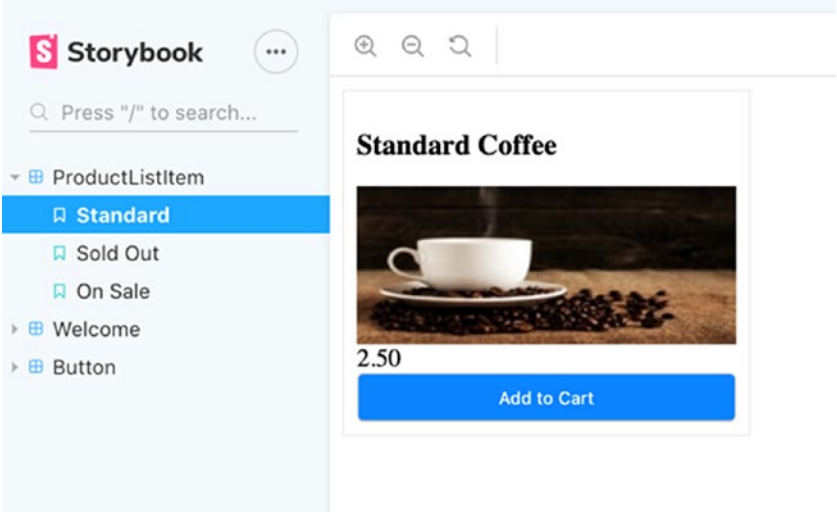

Figure 4-1. *Storybook output*

When Should We Abstract Components?

This process of removing some elements from one component into a new component can be referred to as abstracting a component. There's a problem with simply creating a new component out of any element that *could* be a new component. It's important to determine when we should create an abstraction.

> *Prefer duplication over the wrong abstraction.*
>
> —Sandi Metz, programmer and author

Sandi Metz offers one of my favorite quotes in all of software development. This warning reminds us that it's often better to duplicate code until we know we'll end up with an abstraction that will be valuable to our codebase.

The primary factor I use for determining if a new component would be useful is if there are other components in a codebase that have or will need a component that serves the same purpose. In our simple example, we can assume that other components built for Rocket Coffee will need a Card component. If we aren't sure that an element will be reused, it's a safer bet to wait before creating an abstraction.

Component States

Many components require the ability to represent different states. It seems reasonable that we may need to represent a state for when the product list is loading or when loading failed. Let's create a ProductList folder next to the ProductListItem folder in */src/components*. We want to have a state for when the component is loading and when an error has occurred. We could represent these states with Boolean flags as follows:

```
export default function ProductList ({
  isLoading, hasError, ...otherProps
}) {
  if(isLoading) {
    return <Loading />;
  }
  if(hasError) {
    return <Error />;
  }
  return // standard output when data present
}
```

This can work, but can lead to bugs. What happens if we are technically in the loading state, but an error occurred? We could end up with both *isLoading* and *isErrored* set to true. Our component should never be in a state where it's both loading and an error occurred. To anyone using our application, this would look like the data is consistently loading, when they should be presented with a note that there was an error loading the data. While there are many ways around this, such as error boundaries and handling errors higher up in the component hierarchy, we should strive to ensure that our components only end up in one, accurate state.

Taking a cue from state machines and state charts, we can change our props to take a *status* prop instead of several Boolean flags. According to Smashing Magazine, "the machine can have different states, but at a given time fulfills only one of them" (www.smashingmagazine.com/2018/01/rise-state-machines/). For our simple example, this means the state or status of our component is either *loading*, *loaded*, or *errored*. It wouldn't be possible to be in both loading and error states at the same time as it would with Boolean flags that control the component state.

Figure 4-2. *visualization of our simple state machine*

With that philosophy in place, we could update our simple example. We'll start by adding a *statusTypes* object to contain our component states:

```
export const statusTypes = {
  loading: "loading",
  errored: "errored",
  loaded: "loaded"
};
```

This object contains all the states that our component can be in. These values could be any other string, but using the same property name and value should work well enough for most instances:

```
export default function ProductList({
status, ...otherProps
}) {
  if (status === statusTypes.loading) {
    return <Loading />;
  }
  if (status === statusTypes.errored) {
    return <Error message="Failed to load data" />;
  }
   return ... // standard output when data present
}
```

Next, we can create some stories to see our various component states in action. We'll create ProductList.stories.js.

How Should We Arrange Our Components?

By default, React, the library, has very little opinion about where components should live in our codebase. While some libraries and frameworks are very opinionated about where things belong, React doesn't offer any strong recommendations. I've found when building a component library, there are a couple guidelines I like to follow.

Classifying Component Types

One guideline I like to follow is grouping like components together. There's a problem here—we don't necessarily have a great metric to ensure that we're classifying components the same way.

There's a fantastic book (`https://atomicdesign.bradfrost.com/`) and article (`https://bradfrost.com/blog/post/atomic-web-design/`) by Brad Frost called *Atomic Design*. In *Atomic Design*, Brad describes a mechanism for classifying UI elements based on the periodic table of elements. In this philosophy, he references building design systems out of components, where the lowest-level components are atoms, which can be composed into molecules, organisms, and so on. In other words, the foundational components, or atoms, get composed together into bigger components, which eventually make up our application screens or pages. I love this philosophy and find it's a really useful way to think about components that make up our applications.

That all said, I generally use a slightly different naming schema:

Atoms – Inspired by the naming in *Atomic Design*, I generally call my foundational components atoms. These are things like buttons, headings, text, and things that don't stand up so well on their own, but are necessary for the success of the other elements in our application.

Patterns – These components are reusable UI patterns that are composed from the atom-level components. I would classify our ProductListItem component as a pattern, since it is composed of the Card, Button, Heading, and Text components and can be valuable in many contexts.

Screens – These components represent our pages or screens in an application. A screen could be something like the product listing page that a user would see. This could consist of a layout, product list, and various other elements.

While this is the naming schema that I generally use right now, it's important to note that this is not a set of definitive rules. The more important thing is to use consistent naming/classification that works for you or your team.

With this in mind, we can go ahead and update our ProductList.js and ProductListItem.js to be in components/patterns.

While we're at it, we should create new folders that match the other components we moved out of the main ProductListItem component (but kept in the ProductListItem.js file). When we're done, we should end up with a file structure like this:

```
RocketCoffee/
├─ src/
  ├── components/
    ├── atoms/
      ├── Button/
      ├── Card/
      └── Heading/
    ├── patterns/
      ├── ProductList/
      └── ProductListItem/
```

Once we restart Storybook, everything should be working, using this structure. It should be noted that we're only looking at the folder listing for src right now. These folders should contain the index.js, component, and {componentName}.stories.js files as we've seen with the ProductListItem:

```
ProductListItem/
        ├── index.js/
        ├── ProductListItem.js/
        └── ProductListItem.stories.js
```

EXERCISE: HIGHLIGHT A PRODUCT AS ON SALE

Using the ProductListItem folder listing as a guide, create files for the atomic components in ProductListItem.js. These components should live in the folders we created in the preceding text under *src/components/atoms*. Feel free to reference the Chapter 4 example code if you get stuck.

Key Takeaways

In this chapter, we discussed how to write components that scale well. Taking inspiration from the Unix philosophy, we built components that focus on one thing and handle that one thing well. We talked about how a component's purpose should be clear and focusing on this purpose can help illuminate what items should exist as unique components. Finally, we talked about components as a series of states and some strategies we can use to classify our components.

CHAPTER 5

Styling

In this chapter, we're going to talk about styling front-end applications. Styling is an exciting topic. It's fun to see our code change how our application looks. Unfortunately, merely mentioning CSS can cause some developers to run away and look for any other development task to work on. Styling applications has unique challenges that other coding paradigms do not, but few areas of our code make quite as big of an impact as quickly as how we style our code.

How we style our applications can be a huge topic too! There are entire books and volumes of books dedicated to this topic, and many of these may fall short. To make things more challenging, the front-end landscape for styling has exploded with options lately regarding how we can style our apps. Like anything in programming, fans of a particular paradigm will often suggest that their preference is the best way, making it challenging to separate useful information from hype.

Navigating the styling landscape can be challenging, but it doesn't have to be. In this chapter, we're going to discuss some of the options we have and where we may want to choose one over another. We will mostly be focused on choosing how to style our applications and the structure we may want to put in place to ensure our styles scale with our applications. This chapter will not be a deep dive on the syntax of styling our applications.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_5

CSS

Any discussion around styling web applications should start with CSS (Cascading Style Sheets). CSS is the browser-supported way to style web applications. Using CSS, we apply attributes to HTML elements and define the styles that should be applied to these elements by referring to them through selectors. A selector is a way that we can refer to an HTML element. In some cases, this can be through applying a className prop or an id attribute to an element or finding the element through other means:

```
// index.html

<div id="login-box">
  <div>
    Login box here...
  </div>
</div>
<ul class="points">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
/* index.css */
#login-box {
  border: 1px solid #ededed;
  border-radius: 5px;
  ...
}
.points li:nth-child(2) {
  background-color: #ccc;
}
```

In the preceding example, we're using CSS to style some HTML elements. In the case of *login-box,* we're selecting this element with an id. Element IDs should be unique—we could use a class if we wanted to select multiple elements by an identifier. We're also applying a background color to the second element in list using the *nth-child* pseudo class. This allows us to select a single item by its order in a group. This example only scratches the surface of what we can do with CSS. For a more exhaustive exploration of CSS, you may want to look at *Architecting CSS: The Programmer's Guide to Effective Style Sheets* by Martine Dowden and Michael Dowden.

Benefits of CSS

CSS can be interpreted in just about every browser we will encounter when developing applications. Many of the other ways we can style an application, preprocessors and so on, require a translation or compilation step to work in a browser. By default, CSS needs no such translation. Additionally, while there is nuance to how different developers use CSS, it's a ubiquitous paradigm. Many web developers have some experience with CSS, where a more specialized styling library may be a foreign concept.

Drawbacks of CSS

This headline is intentionally misleading. When talking about the pros and cons of different styling technologies, one of the items that constantly comes up with CSS is cascading (the C in CSS). Cascading is the characteristic of CSS that allows properties to overwrite other properties. This can be confusing to some developers who aren't aware of this feature—especially in codebases that treat styling as an afterthought.

While the argument could definitely be made that cascading is one of the benefits of CSS, we're discussing it as a drawback as well, as it's something that those who may not be as comfortable with CSS may trip

over as they're styling applications. Have you ever tried to change the styling in one area of a codebase only to find that the styles you added were not getting applied? It's likely that something else in your application was overriding your styles. With CSS, the last property that's interpreted is the last one that is applied.

Due to how CSS selectors work, it can be easy to have naming conflicts or misapply styles to elements we're not intending to style. Careful naming and code structure should be applied to help avoid conflicts.

Note There is a technique called BEM (Block Element Modifier) that can help with element specificity and avoiding conflicts. You can learn more about BEM at getbem.com.

Preprocessors

A CSS preprocessor is, as the name implies, something that processes different styling syntax and converts it to CSS the browser understands. One common preprocessor is Sass. An example of Sass/SCSS syntax is as follows:

```
$heading-color: #00F;
$text-color: #2A2A2A;
.card {
  width: 300px;
  border-radius: 5px;
  border: 1px solid #EDEDED;
  padding: 12px;

  h3 {
    color: $heading-color;
  }
```

```
  p {
    color: $text-color;
  }
}
```

This styling corresponds to the following HTML:

```
<div class="card">
  <h3>This is the heading</h3>
  <p>This is some text</p>
</div>
```

At first glance, this code may seem just like CSS. Upon further inspection, however, it may be apparent that Sass/SCSS and standard CSS are quite a bit different. Sass gives additional functionality to styling like variables and nested element styles. This can be a solid option, but it should be noted that CSS supports custom properties (or variables). Additionally, as of the writing of this book, the CSS Working Group is discussing the addition of nesting to CSS.

Note Sass/SCSS and other CSS preprocessors can add a lot of useful functionality beyond the items discussed here. That said, the individual features are outside the scope of this book.

Benefits of Preprocessors

Preprocessors can add great functionality to CSS. Using a preprocessor allows us to write styling code in a way that can feel closer to how we write our application's logic by providing additional utilities to our styling code. Some such capabilities that preprocessors unlock are variables, conditional statements, loops, and property nesting.

Drawbacks of Preprocessors

Preprocessors can add extra build steps that may add time or even file size to our exported code. Additionally, the preprocessor syntax can add new code that developers may need to learn.

CSS-in-JS

Handling styles in JavaScript is potentially one of the most controversial topics in this book. Some developers love pushing more of the front-end responsibility into JavaScript, while others see it as a high crime.

CSS-in-JS techniques are a popular choice for many React codebases since they push the "Everything is a component" mentality that often comes with React. Where other frameworks promote separation of concerns, React components layer styling, markup, and functionality as one cohesive unit of code. Using CSS-in-JS pushes this concept a little further as it treats styles as a responsibility of the component.

One way we can leverage CSS-in-JS is using the React component *style* prop:

```
const Card = () => {
  return (
    <div
      style={{
        backgroundColor: "#EDEDED",
        borderRadius: 5,
        padding: 12
      }}
    >
      ...
    </div>
  )
}
```

In our preceding card example, we are supplying a *background-color*, *border-radius*, and *padding*. Notice that we're using camel case instead of the traditional CSS property names since dashes are not valid as JavaScript properties. Also, since this is just a JavaScript object, we could move this style, so it's not inline:

```
const cardStyle = {
  backgroundColor: "#EDEDED",
  ...
}
return (<div style={cardStyle} ...);
```

In addition to React's style prop, there are other libraries in the React ecosystem that provide mechanisms for styling such as emotion and styled-components. Using styled-components, we can style our application with tagged template literals and styled's API. We'll start off creating a Card component:

```
const Card = styled.div`
  width: 300px;
  border-radius: 5px;
  border: 1px solid #EDEDED;
  padding: 12px;
  h3 {
    color: ${(props) => props.headingColor};
  }
  p {
    color: ${(props) => props.textColor}
  }
`;
```

After this, we'll supply default props. Since *Card* is a React component, we will use the same API to add defaultProps as we would to any other React component:

```
Card.defaultProps = {
  headingColor: "#00F",
  textColor: "#2A2A2A"
}
```

Finally, we can render our *Card* component similar to how we would any other component:

```
export default function App() {
  return (
    <Card>
      <h3>This is a card heading</h3>
      <p>This is the text description</p>
    </Card>
  );
}
```

It's worth repeating that this *Card* we created with styled-components is a React component. The styled-components API allows us to nest styles and even receive props! This can be an extremely powerful way to style our React applications.

Note We're only scratching the surface of what we can achieve using styled-components or emotion. Both libraries have a mechanism for themes that provides a clear path for providing consistent styles to varying components. Additionally, there are entire ecosystems of plugins that these libraries can utilize such as styled-system, rebass, and theme-ui.

Benefits of CSS-in-JS

Using CSS-in-JS to style our components allows us to stick with a single paradigm to develop and style our components. It can be incredibly useful to style our components using the same properties that we're using to build our component functionality. Additionally, developers who are not comfortable with the cascading aspects of CSS may feel more at home using CSS-in-JS.

Drawbacks of CSS-in-JS

Like preprocessors, CSS-in-JS introduces new syntax that developers need to learn. Additionally, styling with JavaScript doesn't necessarily take advantage of performance benefits that we get by using raw CSS in the browser.

Utility-First Styling Libraries

In recent years, utility-first styling libraries have risen to the forefront of app styling options, popularized by libraries such as Tailwind and Tachyons. By using a utility-first approach, developers build many small CSS classes or styling functions that can be used repeatedly throughout a codebase. Instead of applying styles directly to a component, we reference a number of utility class names to define how it is styled. Using Tailwind, we could style a Card component like this:

A Card component using Tailwind

```
<div class="rounded p-5 m-5 shadow-lg max-w-md">
  <h3 class="text-blue-600 text-xl font-bold">...</h3>
  <p class="text-sm text-gray-600">...</p>
</div>
```

What do you think about this approach? My first reaction upon seeing Tailwind was negative, but having used Tailwind on some projects, I gradually found it to be a very quick and consistent way to style UI applications. Utility-first libraries embrace the "Do one thing well" philosophy that we discussed in Chapter 4 and provide a nice way to build UIs with composition.

Benefits of Utility-First Libraries

Leveraging utility-first libraries allows us to quickly build consistent UI applications. In many cases, the styles that we need to apply are available, and utilizing the style utilities can help us ensure that all of our components have the same look and feel.

Drawbacks of Utility-First Libraries

Oftentimes, utility-first libraries still depend on CSS preprocessors to achieve their results. This is useful because it can eliminate utilities that aren't used, but also adds an additional build step. What's more, some developers don't like having multiple class names to determine a component or element's styling.

How to Choose a Styling Solution

There are trade-offs we must consider when choosing any of these options. There are many factors that come into play when choosing a method for styling your application. Unfortunately, there is no way we can come up with a one-size-fits-all approach to choosing how you or your team should style your application.

What I often evaluate when recommending a styling solution is the level of comfort a team may have with CSS vs. JavaScript. Additionally, I

think about how a team may embrace composing class names to create components vs. providing unique styles. Finally, I explore the build steps that must occur to successfully use one of the techniques.

For this book, we're going to use CSS to style our components. All of the other options we discussed depend on CSS knowledge. The things we learn for CSS will almost always be valuable for other libraries, while the same may not be true of other techniques.

Building a Theme

When constructing a component library, we want our components to use consistent colors, spacing, fonts, and so on. Have you ever used an app where you could tell a different team was responsible for a different area of the codebase based solely on how it looked? We want to avoid this scenario, and we'll achieve this through the use of themes.

We'll start by creating a CSS file that contains the custom properties we'll use in our app to define our global template. We'll use `:root` to define variables that apply to any element in our codebase:

```
:root {
  --background: #fff;
  --text: #222;
  ...
  --border-default: 1px solid;
  --font-sm: 12px;
  --spacing-sm: 4px;
  ...
  --spacing-xlarge: 48px;
  --shadow-default: 10px 9px 33px -17px rgba(0, 0, 0, 0.75);
  --radius-default: 8px;
}
```

Next, we'll define our card styles:

```
.card {
  background-color: var(--background);
  color: var(--primary);
  padding: var(--spacing-medium);
  margin: var(--spacing-large);
  border: var(--border-default);
  border-color: var(--light);
  border-radius: var(--radius-default);
  box-shadow: var(--shadow-default);
}
```

Finally, we're ready to use this style for our Card component:

```
import React from "react";
import "../../../theme.css";
import "../../../Card.css";
export default function Card() {
  return (
    <div className="card">
         ...
    </div>
  );
}
```

Now that we have a pretty good handle on how we're going to style components in our application, let's jump back to our Rocket Coffee example and apply these findings there. We'll start off by copying our theme.css file to /src/theme.css. Next, we'll create a new file under `/src/components/atoms/Card/` called `Card.css`:

```
.card {
  background-color: var(--background);
  color: var(--primary);
  padding: var(--spacing-medium);
  margin: var(--spacing-large);
  border: var(--border-default);
  border-color: var(--light);
  border-radius: var(--radius-default);
  box-shadow: var(--shadow-default);
}
```

If we run Storybook or the app directly, we'll notice that our styles are not showing up. We need to import our theme.css file in both our main app's entry point and Storybook's entry point.

For our main app, navigate to *src/index.js* and import the following line after the rest of the imports:

```
import './theme.css'
```

For Storybook, navigate to the *.storybook* directory and add a new file, *config.js.* Add the following import statement to the newly created file:

```
import "../src/theme.css";
```

Now, if we run `npm run storybook`, the Card with the styles based on the theme should be present.

Key Takeaways

In this chapter, we discussed the various strategies we can use to style our applications and some of the pros and cons of the various techniques. We also discussed why using CSS is a solid choice for our exploration. Finally, we talked about CSS custom properties and how we can use them to establish themes for our application.

EXERCISE

So far, we've only added the styling for our Card component in Rocket Coffee. Spend some time and add styles for the rest of the items. Check to ensure that it's working with Storybook by running *npm run storybook.*

CHAPTER 6

Ensuring the Quality of Our Components

So far, we've built our components in isolation and have tested them for quality manually through Storybook. We've verified that our code is working by running our components directly through Storybook, but there are other techniques we may want to apply to ensure that we're providing a quality codebase. Can you imagine checking every component in a web app before every release?

Unit Tests

Storybook is an excellent workspace for our components. So far, it's also served as the primary method of ensuring our components are working as expected. This works great for our smaller examples, but we are in the business of creating professional applications. In these professional applications, there are generally a lot of components in play. Testing each thing manually will not scale as well as we would like. Thankfully, we can automate this process through unit testing.

Unit tests are blocks of code that we can use to test a specific unit of application code. In other words, we write code to ensure that the various elements of our codebase are functioning as intended. This may seem odd to write code to verify that our code functions, but creating tests can result in code that scales better over time, with lesser bugs and higher resiliency

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_6

to change. Instead of manually going through all of our components and features, we can run a test script to verify that our application is working as intended.

JavaScript has a built-in way that we can perform unit tests:

```
> console.assert(1 + 3 === 7);
Assertion failed
```

While this assert works, it suffers from a lack of tooling. We would have to come up with our own way of running these tests. Thankfully, there are many open source options that we can leverage to achieve a great testing experience for our applications.

In our time together, we'll use Jest to write tests. Jest is an open source unit testing library created by Facebook. Jest serves as the tool that we will run our tests with and provides some guidance on how we should structure our unit test code. It balances ease of use with flexibility and has a pretty large ecosystem of developers. This large ecosystem generally results in more people who have encountered and fixed issues and more plugins.

Let's take a look at simple calculator and related Jest tests. By default, Create React App (or CRA), the starting point we are using for our application, comes with Jest preconfigured. We'll proceed with the assumption that we are using Jest under this CRA umbrella.

Note If you need Jest outside of Create React App, the configuration is relatively straightforward. Check out the Jest documentation for more on that at `https://jestjs.io/`. The information on this site also provides information on how to use Jest outside of React!

We'll also assume that we have a *test* script in our package.json file that runs *jest* for us when we type `npm run test` from the command line:

```
export function add(a, b) {
  return a + b;
}
```

We're going to write a simple test and ensure that things are working as expected. But before we check that that things are working as expected, it can be a good practice to make sure that a scenario that should result in a failed test actually causes a test failure. I've seen tests that appeared to be working, but failed to catch a number of bugs. Examining the tests while fixing the bugs, it turned out that the tests always passed. The tests gave a false sense of security. To help avoid this situation, we'll start out by making sure that our test fails before we make it pass:

```
import { add } from './calculate';

it('adds two numbers correctly', () => {
  expect(add(5, 6)).toEqual(17);
})
```

We're using Jest to state that the test adds two numbers correctly. This is described in the `it` method. The first parameter to the method is the test label, or what we'll see in the output to reference this test. The *expect* method is where we will add our assertions. Jest and the testing ecosystem provide many other extension methods, like `toEqual`, we can use to verify our assertions. Additionally, we have an arrow function that performs the test. Clearly, adding 5 to 6 does not result in 17. First, we want to ensure that our test fails with incorrect values supplied. We make sure the test

fails initially because want to avoid any scenario where our tests provide us with false confidence. Running `npm run test` will result in the following message:

```
adds two numbers correctly
  expect(received).toEqual(expected) // deep equality
  Expected: 17
  Received: 11
```

Excellent. As expected, the test failed as we hoped. We see the title of the test "adds two numbers correctly" followed by the result. Now, let's go ahead and fix this test so it runs:

```
it('adds two numbers correctly', () => {
  expect(add(5, 6)).toEqual(11);
})
```

Running `npm run test` again results in the following message from the Jest test runner:

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

This is a great way to test standard JavaScript methods and classes, but how would you approach testing a React component?

Testing React Components

It's true that React components are standard JavaScript methods or classes, but there is more nuance to these components since they have the concept of a rendering surface. On the Web, React components render to the DOM (document object model). Tests need some way to handle rendering as well. There are many things we could do to obtain a rendering surface in our React component tests. We're going to focus on React Testing Library.

In my opinion, React Testing Library offers an incredible testing experience. In this paradigm, we'll use React Testing Library to render our components and then interact with our components using the query methods that React Testing Library provides.

Let's add some tests to our product listing for Rocket Coffee. We'll start off adding Testing Library to our project via the CLI:

```
npm install –save-dev @testing-library/react
```

From there, we'll move on to our code. We're going to start by adding tests to the `ProductListItem.js` component, located in `src/components/patterns/ProductListItem/`. For our exploration, this component has a good balance between number of features within the component and succinctness. We can cover the items that we wish to without being overwhelmed.

We'll start by thinking about the things that can test. Personally, I usually think about the branching code paths (things that show up in one context and not another) and the items that have interaction points. We could add more testing for ensuring things render the items we're expecting, but we'll leave that as an exercise for you.

In this case, we want to test the following:

- Show (On Sale) when the component is on sale.
- Call `onAddToCard` when the button is enabled and clicked.
- Disable the button when sold out.

We'll create a new file *ProductListItem.test.js* next to the *ProductListItem.js.* We'll open the test file and create our first test. Keep in mind, we won't need to import anything for Jest, but we will for React, React Testing Library, and our component.

One thing that would be good to quickly discuss before we embark on testing these components: React Testing Library has quite a few helper

methods and functions that we will not be using in these examples. It would be good to familiarize yourself a bit with the documentation at some point, as these helper functions may save you a lot of time and debugging heartache if you're aware of them.

With that out of the way, let's continue adding our tests!

```
it('shows on sale label when isOnSale', () => {
  const { getByText } = render(<ProductListItem
    name="Mocha"
    price={3.50}
    imageUrl="..."
    isOnSale
  />)
    expect(getByText(`(On Sale)`))
      .toBeInTheDocument();
})
```

Here, we are rendering our component with React Testing Library with an *isOnSale* prop. We're expecting the text "(On Sale)" to exist in the text rendered here. If we open up our terminal and type in *npm run test*, we should be presented with a note that our test passed:

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

Note Keep in mind it can be a good practice to ensure that the test fails so we don't think that the test works when it does not. Tests that provide false security are worse than no tests.

With this test out of the way, let's continue on. We'll start with the test for checking that we disable the button when the *isSoldOut* prop is present:

```
it('disables the button when disabled', () => {
  const { getByText } = render(<ProductListItem
    name="Mocha"
    price={3.50}
    isSoldOut
  />)
  expect(getByText("Sold out"))
    .toHaveAttribute('disabled');
});
```

The last test we want to add is for ensuring that clicking addToCart actually fires an event. We're going to use a Jest mock function. A mock function is something that we can use while testing in place of a real function. We'll interact with the mock function as *jest.fn*. In the following example, we're setting a value `onAddToCart` to this mock function and referencing it in our component.

In addition to a mock function, we'll want to click the button in our test to ensure that the `onAddToCart` method is being called. React Testing Library has a *fireEvent* helper that we will use to achieve our goals:

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

it('calls callback when button Add to Cart pressed', () => {
  const onAddToCart = jest.fn();
  const { getByText } = render(<ProductListItem
    name="Mocha"
    price={3.50}
    onAddToCart={onAddToCart}
  />)

  fireEvent.click(getByText("Add to Cart"))
  expect(onAddToCart).toHaveBeenCalled()
})
```

Testing Alongside Storybook

These tests may feel a little familiar. That's because they closely resemble our Storybook stories. In both the Storybook stories and tests, we're examining our component in different states. For instance, we have a standard state, an "On Sale" state, and a "Sold out" state.

What if we could eliminate the areas where we're defining the JSX elements to render and leverage what we already set up for Storybook instead? We can, and there are a couple of immediate advantages to this:

- **We can visually "debug" our tests.** If one of our tests fails, instead of logging out the rendered HTML (a common debugging technique), we could fire up Storybook and manually step through the same steps. Since the test and the story are using the same code to determine how an element should be rendered, we can have confidence that we're not out of sync between manually walking through our code and the automated test.
- **We keep our stories up to date.** I've seen many projects where Storybook eventually became out of sync with the current state of a product. What started as a useful tool lost its relevance as developers became less vigilant about creating and maintaining stories. This is a tragedy as Storybook can provide a lot of value, but treating Storybook as part of the testing process helps us ensure that Storybook stays up to date.
- **It's less code/less duplication.** While duplication is a better option when things are not the same, we want to run our components in different states in Storybook. Similarly, we want to test our components in various states in our unit tests. Take our example of the ProductListItem. We have one state of the component when the product is in stock and another state of the component when the product is sold out.

We'll make some adjustments to our ProductListItem stories. Instead of applying the Storybook Knobs and actions directly to our elements, we'll bring in the values as default parameters to the story. This way, our tests can override properties as necessary while still providing the default add-on values when desired:

```
export const Standard = ({
  name=text("Name", "Standard Coffee"),
  price=text("price", "2.50"),
  onAddToCart=action("Add to cart clicked"),
  imageUrl=text(
      "imageUrl",
      "https://source.unsplash.com/tNALoIZhqVM/200x100/"
    )
}) => (
  <ProductListItem
    name={name}
    price={price}
    onAddToCart={onAddToCart}
    imageUrl={imageUrl}
  />
);
```

Now, we can import these stories and use them in our test, instead of duplicating the creation of these elements:

```
import { OnSale, SoldOut, Standard } from './ProductListItem.
stories';
it('shows on sale label when isOnSale', () => {
  const { getByText } = render(<OnSale />)
  expect(getByText(`(On Sale)`)).toBeInTheDocument();
})
```

```
it('disables the button when disabled', () => {
  const { getByText } = render(<SoldOut />)
  expect(getByText("Sold out")).toHaveAttribute('disabled');
});

it('calls onAddToCart when button pressed', () => {
  const onAddToCart = jest.fn();
  const { getByText } = render(<Standard
    onAddToCart={onAddToCart}
  />)
  fireEvent.click(getByText("Add to Cart"))
  expect(onAddToCart).toHaveBeenCalled()
})
```

While less code isn't the goal, especially when testing, this is a great way to share our stories with other areas of our codebase to achieve better software.

Key Takeaways

In this chapter, we discussed ways in which we can test our components automatically and why we may wish to do so. We discussed Jest and React Testing Library from a high level. Finally, we talked about how we can leverage our existing Storybook stories to power our unit tests.

EXERCISE: CREATE ADDITIONAL TESTS

So far, we've added tests to the ProductListItem only. Go through */src/components/atoms* and *src/components/patterns/ProductList* and add tests to the components in those folders.

Be sure to check out the included examples if you get stuck!

CHAPTER 7

Interacting with API Data

So far, we've worked on very specific components for a small portion of our application. In this chapter, we're going to discuss ways that we can promote consistency beyond simply defining where our components live.

In Chapter 4, we talked about classifying our component types and how component classification should have some impact on where those files live in our codebase. In addition, we discussed using atoms, patterns, and screen or page components. Atoms are lowest-level building block components in our system, whereas patterns are slightly more complex reusable components. Lastly, we discussed screens; these would be the page-level features.

Unfortunately, in very large systems, there can be a lot of space between what we call patterns and screens. That is, so far, we've mostly focused on creating components in a reusable way and not focused on how we should glue everything together into a cohesive application. With anything in software development, there are quite a few options at our disposal. We'll discuss a couple of these options and some of the trade-offs we should consider when employing the various techniques.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_7

Some Main Considerations

Several of the major considerations we need to take into account are how we're handling data loading, application state management, and routing between pages.

Data loading – This part of the code refers to how we interact with systems outside of our front-end application. This could be something like API servers or third-party platforms.

State management – One of the trickier things we need to consider when building applications is how we manage the application state. In many scenarios, we'll need to keep track of variables that are shared between pages and components. Additionally, we need to determine how to pass application state between various components. There could be entire books on this topic alone, but in this book, we'll use React's built-in state management.

Additionally, we'll structure our application in such a way that our folder hierarchy can help us determine where to place our state variables. We'll see this in action more as we progress through this chapter.

Routing – We link our various pages and provide entry points to our application via the browser's URL with routing. Our front-end applications, much like standard static HTML documents, have paths that they respond to from the browser bar.

We're going to organize our system around the concept of routing. What this means is our src/components/pages folder will be arranged to match the routes that the patrons of our coffee shop will visit to purchase coffee.

This means that for every main route, we'll have a corresponding folder in our pages directory and every subroute will map to a specific subcomponent within that directory. Let's assume that we will have a Checkout screen in our application and the checkout process consists of a screen where a customer will input their payment information and a

screen where they confirm their order and payment information is correct. On this second screen is where the purchase will actually occur. Our folder structure in our pages may look a little bit like this:

```
RocketCoffee
|_ src
|__ pages
|____ Checkout
|______ Checkout.js
|______ Confirm.js
|______ index.js
|______ PaymentInfo.js
```

Our screens would then map to /checkout, /checkout/paymentInfo, and checkout/confirm. A little bit later, we'll set this up, but for now, we're building our mental model of how the various pieces we're building connect.

Now, our routing structure is established, but how do we handle application state and data loading?

Feature-Based Development

We're going to take a cue from a technique called feature-based development to determine where our data loading will occur. In feature-based development, we arrange our code around features. Features represent a unit of value in our application and encapsulate everything that the feature may need.

Applications not embracing this paradigm will often organize files around technology. This is especially pronounced in applications using Redux. While we are not using Redux in our application, this example can illuminate some of the overhead we can incur when arranging our codebase around the technologies that are in play.

Let's assume we are building our shopping cart functionality with Redux. In a traditional application, there are folders for components, pages, and the various technologies that are used with Redux, actions, reducers, selectors, and so on. We would end up with something like this:

```
src
|_ actions
|___ cartActionCreators.js
|_ components
|___ cart
|_____ Cart.js
|_ reducers
|___ cartReducers.js
|_ selectors
|___ cartSelectors.js
```

Product lists, user accounts, and everything else would theoretically have files dispersed between actions, reducers, selectors, and so on. This works and provides some consistency, but it can be trouble when you want to update, move, or delete things since files are dispersed throughout the codebase.

In a feature-based codebase, we would end up with a base folder for our cart, and all of the unique items of the cart would live within that folder instead of spread out by technology:

```
src
|_ features/
|___ cart/
|_____ components/
|_____ actionCreators.js
|_____ index.js
|_____ reducer.js
|_____ selectors.js
```

While we are not going to use feature folders directly, at some level, we will treat our pages folder a little bit like a feature. That is, everything distinct to the page will live within the pages folder. What that means is we'll keep our data loading and state management mechanisms near the page they represent.

With this mindset established, we're going to update our product list page with data loading that we can use throughout that section of our application. In Chapter 8, we'll look at routing and how we can view our page screens outside of Storybook.

Loading Data

For the data loading, we're going to start by performing an API request directly in a Products component. We'll create a component named *Products* in *src/screens/Products/Product.js*, along with sibling files for our Storybook story *Products.stories.js* and an *index.js* file.

Inside our Products component, we'll use our ProductList that we created earlier and render data fetched from an API. Our initial component could look a little like this:

```
import React, { useEffect, useState } from 'react';
import ProductList, { statusTypes }  from '../../components/
patterns/ProductList';

export default function Products() {
  const [ productState, setProductState ] = useState({
    data: [],
    status: statusTypes.loading
  });
```

```
  return <ProductList
    data={productState.data}
    status={productState.status}
  />
}
```

In this component, we're building state to eventually hold our product data that we'll pass to our ProductList component. In our Products.stories.js file, we will reference the newly created component as follows:

```
import React from 'react';
import Products from './Products';

export default { title: 'screens/products' }

export const standard = () => {
  return <Products />
}
```

Now, when we run Storybook and navigate to screens ➤ products from the tree view, we see our component in a Loading state.

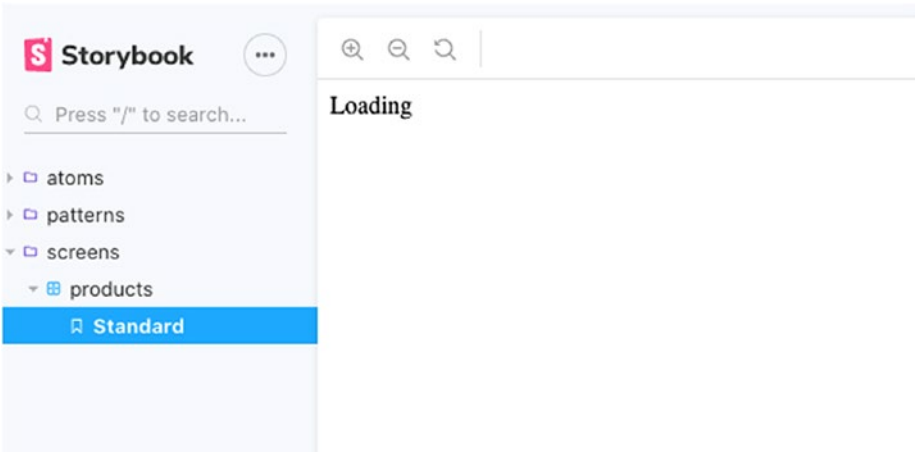

Figure 7-1. *Storybook output*

Now, let's fetch some data in our component. We're going to assume that in the production version of our application, we're running an API on the same server as our front-end application on the path */api*. We'll add a *useEffect* statement in our component directly before the return statement for interacting with this API:

```
import React, { useEffect, useState } from 'react';
import ProductList, { statusTypes }  from '../../components/
patterns/ProductList';
export default function Products() {
  const [ productState, setProductState ] = useState({
    data: [],
    status: statusTypes.loading
  });
  useEffect(() => {
    const getData = async () => {
      try {
        const productFetch = await fetch('/api/products');
        const productResponse = await productFetch.json();
        setProductState({ data: productResponse.data, status:
        statusTypes.loaded })
      } catch (ex) {
        console.error(ex);
        setProductState({ data: [], status: statusTypes.errored })
      }
    }

    getData();
  }, []);
  return <ProductList
    data={productState.data}
    status={productState.status}
  />
}
```

For those who aren't familiar, *useEffect* is a little bit like *componentDidMount* in class-based React components. We're defining an async method to interact with our API and setting our state when we get a response back. We don't need to change anything in the return statement, because we are already basing the results off of the data property on our `productState` object.

Note We normally don't need to use another method inside a useEffect statement. When we want to use async/await, however, we create another method since useEffect cannot be async directly.

Running Storybook may illuminate a problem, however. We our attempting to interact with an API. In our stories and our tests, we do not want to use a real API to obtain our data.

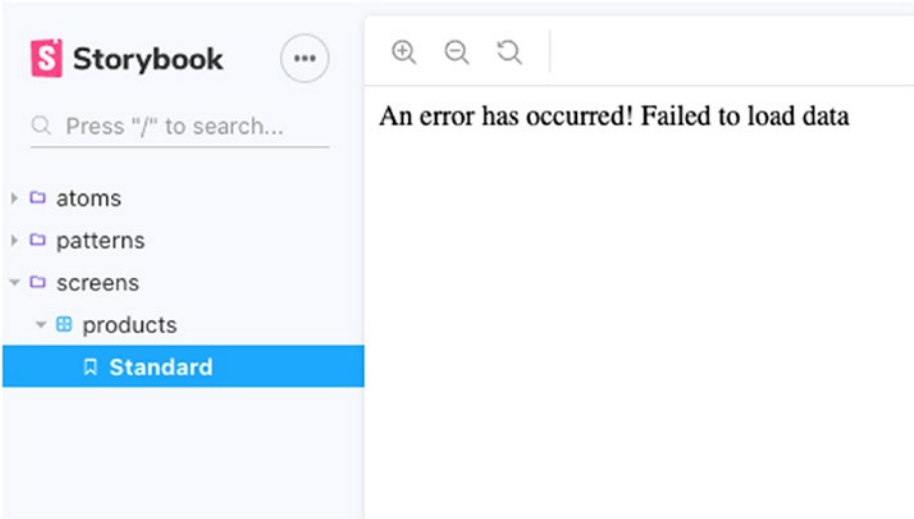

Figure 7-2. *Storybook output after attempting to obtain data from an API*

This is great that we can see the component in an error state; however, we're trying to see the list of products as the standard view of this screen.

There are a couple of strategies we could employ to ensure that we could still derive value from Storybook for our screen/page-level components.

Container/Presentational Components

The first thing we could attempt is split up our component into a Container and View component. The Container component would be responsible for obtaining data, and the View component would only be responsible for displaying data. With this strategy, we would have distinct files Products.container.js and Products.js where Products.js is the View component.

This is a strategy that I've used successfully on many projects, but there's a downside. You're creating more files than you may need. More code means there is more surface area for bugs and things you need to maintain.

Mock Data

The technique we're going to use in this book is mocking the API endpoint for Storybook (and since we're using Storybook stories to power our tests, our tests too). This functionality is provided through the open source tool Mirage JS (miragejs.com). Mirage will allow us to interact with the endpoint in our components while replacing any calls to the API with the data when viewing components in Storybook. We'll install miragejs with npm:

```
npm install –save-dev miragejs
```

Once npm has finished, we'll update our test to build a mock API endpoint that's used in Storybook (Listing 7-1).

Listing 7-1. Products.stories.js

```
import { Server } from 'miragejs';

let server = new Server();
server.get('/api/products/', { data:
  [
    {
      id: 1,
      name: "Mocha",
      price: 3.5,
      imageUrl: "https://source.unsplash.com/tNALoIZhqVM/
      200x100/",
    },
    ...
  ]
});
```

We import the server from Mirage and set up a mock that says "when I request /api/products/, respond with this data." Now if we run Storybook, we can see our mock data being returned instead of the error we saw previously.

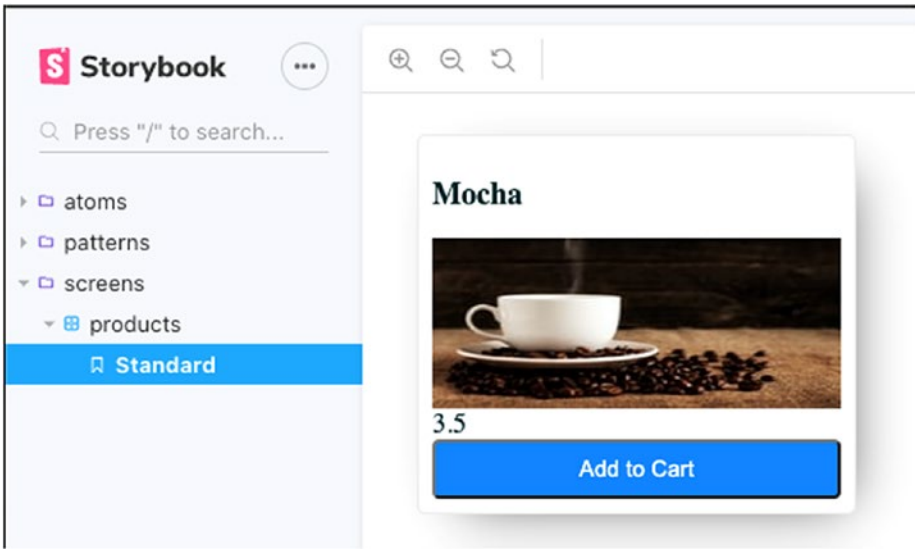

Figure 7-3. *Products page*

Wrapping Up

In this chapter, we discussed how we can load data in our Screen/Page components. We talked about some of the various techniques we can use to load data and how we can quickly mock data for Storybook stories and unit tests.

EXERCISE

So far, we've used our mock data to power our Storybook story. The goal is that we will have a nice test coverage as well. A great exercise would be creating a series of unit tests that covers this functionality. Using some of the concepts from this chapter and Chapter 6, create a suite of unit tests that ensure that we load and display the product list.

CHAPTER 8

Building Our Application

We've been working in the safety of our component workspace. Unfortunately, unless our product is a component library, our users won't receive much value from our components unless we place them in our real application. In this chapter, we're going to take everything we've made so far using Storybook and apply it to a real application. Our approach with Storybook is a bit different from how many applications are built, but as we'll see in this chapter, when we're ready to build our actual application, everything falls into place pretty quickly.

Navigating Between Pages

One thing that we'll need in our application is a way for our coffee shoppers to visit different pages. We could simply show and hide our components based on which part of our application is active. This could work for small applications, but would be unmaintainable at scale. There are some other problems with this strategy that may be more apparent with a simple example.

Let's assume we have some basic page components that represent a Home and About page:

```
const Home = () => <h1>Hi from the home page!</h1>;
const About = () => <h1>Hi from about!</h1>;
```

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_8

Next, we'll add a basic Navigation component that receives a callback that's called when one of the navigation items is clicked. In this case, we're going to use a button since we're currently technically providing button-like functionality instead of typical link functionality (i.e., we're not changing the browser route and instead switching out which content is present—there's some nuance here, but we'll fix this up later):

```
export const Navigation = ({ onChangeNavigation }) => {
  return (
    <nav>
      <ul>
        <li>
          <button onClick={() => onChangeNavigation("home")}>
          Home</button>
        </li>
        <li>
          <button onClick={() => onChangeNavigation("about")}>
          About</button>
        </li>
      </ul>
    </nav>
  );
};
```

Next, we're going to add our Application component as follows. This Application component will use our pages and our navigation:

```
function App() {
  const [currentPage, setCurrentPage] = useState("home");
  let content = undefined;
  if (currentPage === "home") {
    content = <Home />;
  }
```

```
  if (currentPage === "about") {
    content = <About />;
  }

  return (
    <div className="App">
      <Navigation onChangeNavigation={setCurrentPage} />
      {content}
    </div>
  );
}
export default App;
```

In the previous App component, we're defining a variable, *content*, that will hold our page-level content and switching the content based on state. What are some problems you see with this approach?

While this may seem pretty straightforward, there are some definite challenges that we may run into using this strategy. First and foremost, this will not scale well. For two pages, it's pretty clear what's going on, but how would we handle this if we had 20 or more pages? Beyond the developer experience, this technique would not provide a great experience to our coffee shoppers either.

What would happen if a patron wanted to bookmark the About page? Every time they tried to visit the bookmark, they would come back to the main page since this is controlled by the App state unless we created some mechanism for rehydrating the active page based on a URL parameter. At that point, we may be asking ourselves, "Are we just reinventing the concept of a router?" The answer to that question would be "Yes." Thankfully, there is an excellent router that we can use.

Routing

We're going to utilize the awesome React Router library. We can now change our example application to the following:

```
export default function App() {
  return (
    <BrowserRouter>
      <Navigation />
      <Switch>
        <Route path="/about" component={About} />
        <Route path="/" component={Home} />
      </Switch>
    </BrowserRouter>
  );
}
```

From there, we'll update our Navigation component to use the React Router–provided *Link* component, instead of the button elements we were previously using:

```
// still wrapped in existing nav / ul
<li>
  <Link to="/">Home</Link>
</li>
<li>
  <Link to="/about">About</Link>
</li>
```

With this strategy, we define our browser router; this is the top-level provider for our router and creates our routes contained within a Switch component. The Switch component basically allows us to say that only one route is active at a time. Conceptually, we can think of this like a

switch statement where only one code path will be executed based on the provided values. The Route components receive a path and the component that should be active based on the given path.

For our Navigation component, we've updated the buttons to the Link component from React Router. The link renders as an anchor tag and will, as the name implies, link to another route. We provide the route path to our Link components via the *to* prop (i.e., *<Link to="/about">)*.

Using a router has numerous benefits over manually handling content visibility based on state and so on. First, using React Router scales quite a bit better than manually handling content visibility based on state. We could continue to add routes to our switch statement or even provide nested routes (see the React Router documentation for more on nested routes).

React Router provides URL route updates. You may notice if you run the preceding router example, clicking a link updates the URL. This means that users can bookmark the pages that they're on. Additionally, the *Link* component is more sensible, since it's an actual anchor tag rather than buttons that manipulate state. Buttons worked well as the element for calling callbacks, but since we're actually changing the URL with the router, an anchor tag is better.

Updating Our Application to Use Routes

Let's update our Rocket Coffee application to use routes. Since Chapter 7, I've added some pages for Cart and "My Account." They're not hooked up to a real database or anything, but should suffice for dealing with routing.

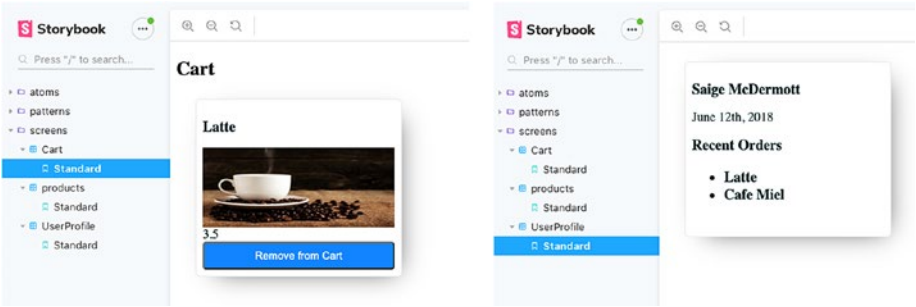

Figure 8-1. *Basic Cart and User Profile components*

We want to organize our system around the concept of routing. What this means is our src/components/pages folder will be moved to match the routes that the patrons of our coffee shop will visit to purchase coffee.

This means that for every main route, we'll have a corresponding folder in our pages directory and every subroute will map to a specific subcomponent within that directory. Our file structure could look like the following:

```
RocketCoffee
|_ src/
|__ components/
|____ atoms/
|____ patterns/
|__ screens/
|____ Cart/
|____ Products/
|____ UserProfile/
```

Each of our screen folders contains a component, an index file, and a story (and optionally contains a test and style information). We can view our pages through Storybook, but we want a real functioning application. Let's go ahead and add React Router and use our real app outside of Storybook.

We'll start by adding the *react-router-dom package from npm*. We're building a web application, so the DOM version of React Router is what we want. It has some areas that interact directly with the browser:

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

From there, we're going to add a new file called *Routes.js* at the top level of our src directory. We're going to include React, *BrowserRouter, Switch*, and *Route* from react-router-dom and our page-level components. Finally, we'll return the routes to our pages in the Switch component:

```
import React from "react";
import { BrowserRouter, Switch, Route } from "react-router-dom";

import Cart from "./screens/Cart";
import Products from "./screens/Products";
import UserProfile from "./screens/UserProfile";

export default function Routes() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/cart" component={Cart} />
        <Route path="/userProfile" component={UserProfile} />
        <Route path="/" component={Products} />
      </Switch>
    </BrowserRouter>
  );
}
```

Our application screens depend on an API for the data we present. Unfortunately, this API doesn't exist. We'll use the mocking technique that we applied in Chapter 7 to provide a mock API to our application. We'll start by creating a file, Mock.server.js, in our *src* directory. This file will be responsible for

```
import { Server } from "miragejs";

const productData = [
  {
    id: 1,
    name: "Mocha",
    price: 3.5,
    imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",
  },
  {
    id: 2,
    name: "Latte",
    price: 3.5,
    imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",
  },
  {
    id: 3,
    name: "Vanilla Latte",
    price: 3.5,
    imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",
  },
];

const server = new Server({
  routes() {
    this.namespace = "/api";
    this.get("/cart", () => ({ data: [productData[1]] }));
    this.get("/products", () => ({ data: productData }));
    this.get("/profile", () => ({
        name: "Saige McDermott",
        memberSince: "June 12th, 2018",
        recentOrders: [
```

```
          { orderId: 12, name: "Latte" },
          { orderId: 27, name: "Cafe Miel" },
        ],
    }));

    this.get("/users", () => [
      { id: "1", name: "Luke" },
      { id: "2", name: "Leia" },
      { id: "3", name: "Anakin" },
    ]);
  },
});

export default server;
```

Next, we need to import this mock server in our app. When we have our real API, we'll want to remove the code that imports our mock, but for now, it works out well:

```
import React from "react";
import ./Mock.server'
import Routes from ./Routes';

function App() {
  return (
    <Routes />
 );
}

export default App;
```

Now if we run our application via *npm start* (or yarn), we should be able to visit our pages by typing in localhost:3000/, localhost:3000/userProfile, and localhost:3000/cart, respectively.

Navigation

Let's add a basic navigation using the Link component from React Router as well. For the sake of example, we'll do this by creating a Navigation component under patterns along with a corresponding index.js barrel file. We'll go ahead and create patterns/Navigation, an index.js file that only exports our soon-to-be-created Navigation component, and our component Navigation.js:

```
// patterns/Navigation/Navigation.js
import React from "react";
import { Link } from "react-router-dom";
export default function Navigation() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Products</Link>
        </li>
        <li>
          <Link to="/cart">Cart</Link>
        </li>
        <li>
          <Link to="/userProfile">Profile</Link>
        </li>
      </ul>
    </nav>
  );
}
```

We'll jump back over to our Routes.js file and add our navigation. We can use our Products screen from Chapter 7 and other screens from the example code. From a high level, these screens all obtain data from our API and use the existing components to present that data to our customers. We'll import these screens in our *Routes.js* file and place them in a React Router browser router as follows:

```
import React from "react";
...
import Navigation from "./components/patterns/Navigation/
Navigation";

export default function Routes() {
  return (
    <BrowserRouter>
      <Navigation />
      <Switch>
        <Route path="/cart" component={Cart} />
        <Route path="/userProfile" component={UserProfile} />
        <Route path="/" component={Products} />
      </Switch>
    </BrowserRouter>
  );
}
```

We now have a rudimentary navigation along with routes that can be bookmarked in our application.

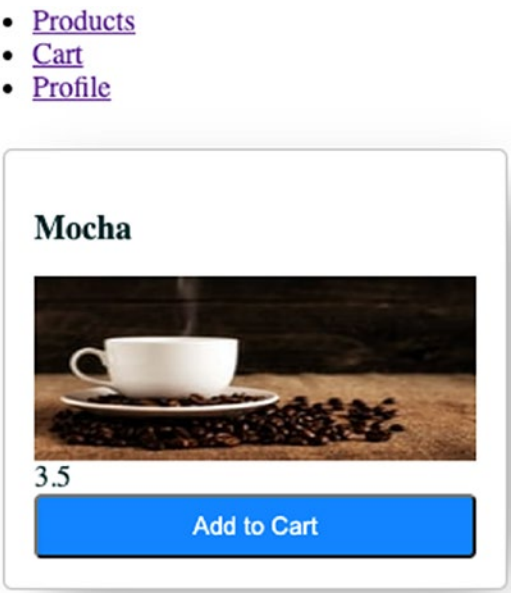

Figure 8-2. *Navigation + route output*

Wrapping Up

In this chapter, we discussed how we can start migrating out of Storybook into our real application. We were able to quickly make pages out of our existing components and provide navigation through React Router.

EXERCISE

Our navigation is currently pretty rudimentary. We want something a bit more sophisticated for our real application. Please create a Header/Footer component that will show up on each page. The header should link to "/" and the footer should contain the same links as our Navigation component. Finally, this Header/Footer component should live in a Layout component that will contain the header, footer, navigation, and page-level content.

CHAPTER 9

Automating Repetitive Tasks

In this chapter, we're going to look into how we can speed up our development. We've discussed constructing our application out of a series of smaller parts. For every page component, pattern component, and atom component, we end up creating a component, an index file, a test file, a story, and in many cases a style file. I don't know about you, but that's a lot of effort that I'd rather not do. It's weird to say, because it's not a complicated task, but it is tedious.

> *Do not use humans for jobs computers can do better—this is a waste of human energy and creativity, the only real resource on this planet, and demeans the human spirit.*
>
> —J. Paul Morrison, programmer and inventor

I really appreciate this quote. To me, it brings up thoughts of Tony Stark (Iron Man's alter ego) being assisted by computers to do his job way better so he can focus on more important things like saving the world. I don't know about you, but I wish I had a robot assistant. I would love to hand off tedious work to a computer so I can keep my attention on the more important things in a project.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_9

While we're not going to be using as sophisticated computing powers as science fiction stories, we can leverage computers to make our jobs easier. Constructing our directory structure is not hard. Let's use some software to make that job more straightforward.

Our Own CLI

We're going to create a CLI specific to our project. A CLI is a command-line interface, that is, an application that runs in the command line. If you've ever used Create React App, npm, yarn, or git, you have interacted with a CLI. We're going to construct one that makes the process of creating our files and directories in the structure we want a process that requires almost no thought at all.

There are a number of tools that we could use to achieve this goal, such as Gluegun, Plop, and others. In this book, however, we're going to use Hygen (`www.hygen.io/`).

Hygen is deemed "The scalable code generator that saves you time" by its documentation. I've used several other generators/CLI tools, and I appreciate the clear-cut results-driven architecture of Hygen. Let's take a look at a simple example where we'll create a file with a hello world function in a specific location. Don't worry. Hello world isn't exciting, but we'll soon be applying these concepts to our Rocket Coffee codebase. Ideally, we will be able to run Listing 9-1 from the command line to generate a component in our app.

Listing 9-1. Theoretical CLI code generator

```
> yarn generate component UserForm –location atom
 added: src/components/atoms/Button/index.js
 added: src/components/atoms/Button/Button.js
 added: src/components/atoms/Button/Button.stories.js
 added: src/components/atoms/Button/Button.test.js
 added: src/components/atoms/Button/Button.css
```

A Brief Example

We'll start by adding Hygen to our computer. There are a couple of ways we could handle this, but adding it as a global module via npm works well:

```
npm install -g hygen
```

Note If we don't want to add the Hygen tool to our computer, we could take advantage of npx to achieve a similar result. The npx tool is a way that we can run a CLI tool that's hosted on the npm registry. If you prefer to use npx, anywhere where we reference the globally installed version of the tool in this guide, use npx instead.

For our example, we're going to use the Rocket Coffee project that we've been building throughout this book. That said, these same steps should work in any project.

Hygen uses template files and the surrounding folders to generate CLI commands. For our "hello world" example, we simply want to run *hygen function new HelloWorld*. There are some things beyond installing Hygen that we need to do to initialize our generator. We'll start off by running the following:

```
hygen init self
added: _templates/generator/help/index.ejs.t
added: _templates/generator/with-prompt/hello.ejs.t
added: _templates/generator/with-prompt/prompt.ejs.t
added: _templates/generator/new/hello.ejs.t
```

This statement initializes Hygen with a generator that we'll use to create our other generators. We're ready to create our function generator:

```
hygen generator new function
```

Hygen will create a new folder in our project. We should navigate to our *_templates/function/new* folder and edit "hello.ejs.t." Hygen uses the .t filetype to prevent editors from trying to apply additional IDE features to this file. It's likely the IDE won't know about these templates. The generator command we will soon run corresponds directly to the folder structure. For instance, we will run *hygen function new {Name}* where Name is the input we'll provide to our generator. You may notice we have a folder for *function*, the first statement in our command, and a function for *new* as well.

Note For this example, keeping the template name as hello should be fine. The template name is more important when we plan on keeping these files around; the name helps our teammates and "future us" recognize the intent of the template.

We'll update the code in _templates/function/new/*hello.ejs*.t to the following:

```
---
to: src/components/<%=name%>.js
---
console.log("hello world!")
```

Hygen templates are based on ejs, a popular templating engine for Node applications. Don't worry if you're not familiar with this, as knowing the ins and outs of ejs is not necessary for building a generator. For our purposes, we want to understand that code in between <%= %> will be output to the file. For example, we are using the variable *name* to populate the filename for our document.

In addition to ejs, it's important to realize that this template consists of two parts. The top part of the template, surrounded in "—-", is the front matter section of the template. Template settings belong in this section of the document. In the front matter for our "hello world" example, we're supplying where the generated output of this generator belongs as the

to property. For this generator, we're stating that we want a file with a given name placed in *src/components*. Now what happens if we run our *generator*? If you run into any errors, please ensure that you've run the previous steps, *hygen init self* and *hygiene generator new function*. Hygen will not function as we're anticipating without these steps:

```
> hygen function new Test
Loaded templates: _templates
       added: src/components/Test.js
```

Building Some Generators for Our Project

With this knowledge in hand, let's build the generators we want to make working in our codebase easier. We'll start by renaming the function folder used in the preceding text from *_templates/function* to *_templates/component*. You could remove the folder and start the generator process over again, but I generally prefer this route.

From there, we'll navigate to _templates/component/new/, rename the hello.ejs.t file to component.ejs.t, and copy this file to create a basic index.ejs.t, component.stories.ejs.t, and component.test.ejs.t file. We'll give each one of these templates a purpose that relates to its name. We'll start by editing the component file. We want it to contain the code in Listing 9-2.

Listing 9-2. component.ejs.t

```
---
to: src/components/atoms/<%=name%>/<%=name%>.js
---
import React from 'react'

export default function <%= name %> () {
  return <h1>Hello from <%= name %></h1>;
}
```

Now, when we run our generator, we see that we create an *atom* component in our src folder:

```
> hygen component new First
Loaded templates: _templates
       added: src/components/atoms/First/First.js
```

Let's go ahead and add our basic stories, tests, and index files. These are definitely not profound, but the time this saves scales as we need more files in our codebase.

We'll start with the basic index file (_templates/component/new/index.ejs.t). For now, we only want this to export the default output from our component file:

```
---
to: src/components/atoms/<%=name%>/index.js
---
export { default } from './<%=name%>';
```

Next, we'll move on to the Storybook stories file (_templates/component/new/component.stories.ejs.t):

```
---
to: src/components/atoms/<%=name%>/<%=name%>.stories.js
---
import React from 'react'
import <%=name%> from './<%= name %>'

export default { title: 'atoms/<%= name %>'}
export const Standard = () => <<%= name %> />
```

Lastly, we'll create our test file (_templates/component/new/component.test.ejs.t):

```
---
to: src/components/atoms/<%=name%>/<%=name%>.test.js
---
import React from 'react'
import { screen, render } from '@testing-library/react';

import <%= name %> from './<%= name %>';
describe('<%= name %>', () => {
  it('renders as expected', () => {
    const { container } = render(<<%= name %> />);
    expect(container).toMatchSnapshot();
  })
})
```

Now, we can run our generator and see that it created a new component and sibling story/test files:

```
> hygen component new TextInput
Loaded templates: _templates
  added: src/components/atoms/TextInput/TextInput.js
  added: src/components/atoms/TextInput/TextInput.stories.js
  added: src/components/atoms/TextInput/TextInput.test.js
  added: src/components/atoms/TextInput/index.js
```

We can run our tests or Storybook and everything works as expected, and our new tests pass. This is great and can save a lot of time; however, we have a problem. Remember we are classifying our components as atoms, patterns, and pages? Unfortunately, we're currently hardcoding our generated components to be atoms. Let's fix that.

Adding Additional Variables to Our Generator

We need to make our generator accept an additional parameter, `type`. While there are a couple ways we could handle this (such as logic in our template files), we're going to build a reusable helper. All of our templates need to know what type of component is being created, so this reusable technique will help us avoid duplicating code.

Hygen has a top-level h object that has the default Hygen helpers. We can add on to this object by creating a file in the root of our project named `.hygen.js` and creating a helpers object with the functions we wish to leverage in our templates (Listing 9-3). We're going to make two helper functions that receive the type supplied from the command line: one to obtain the folder path and one to obtain the name of the type. We'll create an object to hold the values for each possible type. That way, we can add on to this later or change things without needing to update every helper function where a value is present.

Listing 9-3. .hygen.js in the root of our project

```
const types = {
  atom: { name: 'atoms', path: 'components/atoms/'},
  pattern: { name: 'patterns', path: 'components/patterns/'},
  screen: { name: 'screens', path: 'screens/'}
}
module.exports = {
  helpers: {
    getTypePath: (type = 'atom') => {
      return types[type].path;
    },
    getType: (type = 'atom') => {
      return types[type].name
    }

  }
}
```

Now when we use this helper method, any type that we pass in as an argument to our generator will be used to determine where the files live. Now, let's update the story template to use our helper methods. One thing to note is we generally could reference our type variable simply as type; however, that would imply that type is required. We have our generator set up so that when type is not specified, it will default to an atom. We will use Hygen's *locals* object to obtain our type rather than the type variable directly (Listing 9-4).

Listing 9-4. _templates/component/new/component.stories.ejs.t

```
---
to: src/<%= h.getTypePath(locals.type) %><%=name%>/<%=name%>
.stories.js
---
import React from 'react'
import <%=name%> from './<%= name %>'

export default { title: '<%= h.getType(locals.type) %>/<%=
name %>'}

export const Standard = () => <<%= name %> />
```

We can go ahead and update the rest of our templates to use this new helper method. After that's complete, running our generator works for atoms, patterns, and screen components. The following command will generate a new component under our screens directory with supporting test, story, and index files:

```
> hygen component new UserProfile --type screen

Loaded templates: _templates
  added: src/screens/UserProfile/UserProfile.js
  added: src/screens/UserProfile/UserProfile.stories.js
  added: src/screens/UserProfile/UserProfile.test.js
  added: src/screens/UserProfile/index.js
```

Wrapping Up

We now have the ability to generate our files that we'll use to power our front-end application. While some may argue that creating files is not challenging, this now frees us to focus on more important things. We've placed the responsibility of building our app structure on a repeatable process. Instead of needing to constantly check pull requests for correct folder structure, we can rely on the "robot doing it just effectively."

Additionally, the generator can help facilitate communication. When a new developer is onboarded to our team, we can talk about running the generator instead of tedious discussions around where different files belong. Additionally, if we ever want to update where a file belongs, we can update the template and avoid scenarios where we create files in their previous schema out of habit.

EXERCISE

Our primary focus has been on the component, test, and story files. We didn't address the creation of a style file. Create a template that will generate a local CSS module as part of the *hygen component new* command.

CHAPTER 10

Communicating Our Components

So far, we've mostly been focused on strategies to make developing our applications more straightforward. Success in software development hinges on much more than merely writing good code, however. Projects can be a win or a loss, depending on how well people communicate. Thankfully, Storybook helps us here too.

As you may have guessed, Storybook helps us communicate with other developers which components exist in a codebase. How many projects have you been on where the same component or snippet of code lived in many places, simply because a developer had no idea the functionality existed elsewhere? Using a component workspace that all the developers can reference and see a working version of the component helps with this communication. The strategies we've employed so far are useful to developers, but we can do a bit more.

Documenting Our Components

We want to help other developers get up to speed with components as quickly as possible. Only having stories helps in that endeavor, but we could make the experience a bit nicer than merely directing developers to look at the code that powers the stories. We'll start by adding additional documentation properties to our stories. Using our Rocket Coffee codebase, let's start making these updates.

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0_10

Storybook can help us with this documentation also. Remember that our stories' default export is an object? We have some additional properties we can use to provide some documentation with our stories.

To add some detailed documentation to our stories, we'll need to include the *docs* add-on:

```
> yarn add @storybook/addon-docs
```

From there, we'll need to update our Storybook's main.js file to include this new add-on in Listing 10-1.

Listing 10-1. .storybook/main.js

```
module.exports = {
  stories: ["../src/**/*.stories.js"],
  addons: [
    "@storybook/preset-create-react-app",
    "@storybook/addon-actions",
    "@storybook/addon-links",
    "@storybook/addon-knobs",
    "@storybook/addon-docs"
  ],
};
```

Now, if we fire up Storybook, we'll see a new tab called *Docs* next to the main Storybook output. You'll notice it shows our component, a way to see the markup for the component, and our args table.

Now, if we fire up Storybook we'll see a new tab called *docs* next to the main Storybook output.

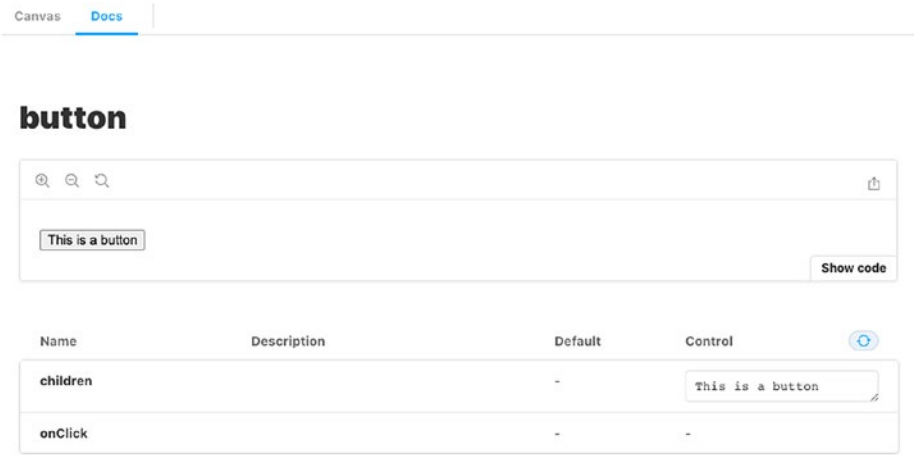

Figure 10-1. *Storybook's Docs tab*

This Docs tab is pretty useful out of the box. We have another view of our stories and have a way to quickly view the code. Let's add a description to the args table at the bottom of the listing.

In addition to the standard, title, and component properties that we've seen before on our story's default export, we're going to leverage the `argTypes` property (Listing 10-2). This property allows us to provide some additional metadata to the args table.

Listing 10-2. Button.stories.js

```
export default {
  title: "atoms/button",
  component: Button,
  argTypes: {
    children: {
      description: "The element(s) that should be rendered
      within the button",
      default: "undefined",
    },
```

```
    onClick: {
      description: "The action that is fired when the button is
      pressed",
    },
  },
};
```

If we load our Storybook, we will see the descriptions we provided in the args table.

Name	Description	Default	Control
children	The element(s) that should be rendered within the button	-	This is a button
onClick	The action that is fired when the button is pressed	-	-

Figure 10-2. *Updated args table*

These descriptions that we've provided as part of the argTypes property support Markdown to provide some light styling to our text. If you're not familiar with Markdown, I recommend visiting Mastering Markdown from GitHub (https://guides.github.com/features/mastering-markdown/).

Note Markdown is a nice addition, but it's not necessary to use with Storybook docs. Feel free to only dive into learning more about Markdown if it's something you want to use.

Taking a similar approach, we can now add an overall description to the story as well:

```
export default {
  title: "atoms/button",
  component: Button,
```

```
  parameters: {
    docs: {
      description: {
        component:
          "The **button** is the component that should fire an
          action based on a click event.",
      },
    },
  },
  ...
};
```

We'll add a property on our default export *parameters.docs.description.component* that contains the description we want to display with our story. This field also supports Markdown.

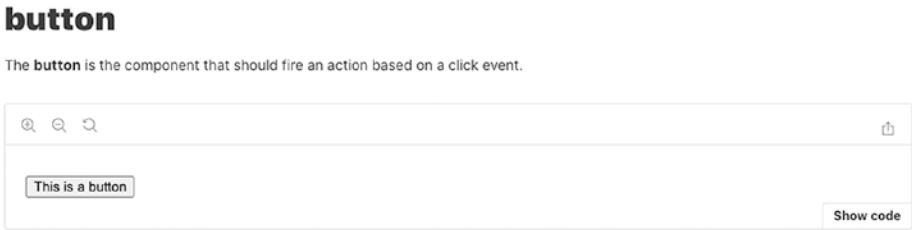

Figure 10-3. *Our story docs with a description*

More Advanced Documentation

This works pretty well, but what if we want more control over our documentation? There's quite a lot we can do with the Storybook default object, but it can be tedious to update our docs entirely through a JavaScript object. Thankfully, we can provide documentation using a different strategy.

MDX

We also can use Markdown, or more specifically MDX, to write our stories. According to the documentation site (mdxjs.com), "MDX is an authorable format that lets you seamlessly write JSX in your Markdown documents." This is an optional way to document Storybook stories, but may be preferred if the goal is to provide stories alongside documentation.

We'll start off by updating our *.storybook/main.js* to add support for mdx files:

```
module.exports = {
  stories: ["../src/**/*.stories.js", "../src/**/*.stories.mdx"],
  addons: [...],
};
```

Next, we'll create a new file to contain our MDX documentation at *src/components/atoms/Button/Button.stories.mdx* and populate it with the following MDX content. Remember, MDX is a combination of JSX and Markdown. We'll still import our references as we would in any standard JavaScript file:

```
import { Meta, Story, Canvas } from "@storybook/addon-docs/blocks";
import Button from "./Button";
<Meta title="atoms/Button" component={Button} />
# Button

This is the **Button** component. The intent of the **Button** is ...

<Button onClick={console.log}>Click here</Button>

_This is a button outside of the Storybook Canvas_

## More documentation
<Canvas>
  <Story name="standard">
    <Button
```

```
      onClick={...}
    >
      Click me
    </Button>
  </Story>
</Canvas>
Other markdown like [Links](http://www.google.com) work also!
```

In the previous example, we're rendering some standard Markdown, along with our Button component inline with the Markdown and our Button component contained within a Storybook canvas.

Now, visiting the Docs tab for the button story displays Figure 10-4.

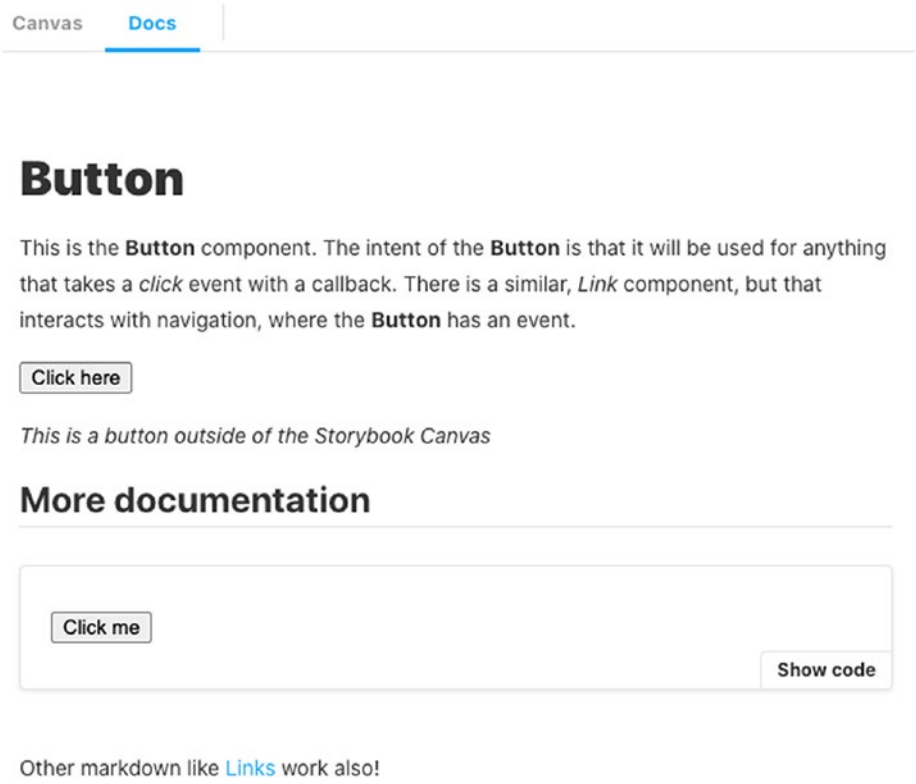

Figure 10-4. *Storybook docs output*

This provides great documenting functionality to our component workspace. The ease and flexibility of Markdown coupled with the power of JSX give us documentation superpowers.

Sharing Our Workspace

We mentioned that one of the goals we have in using Storybook is to use it as a communication tool, as well as a tool for developers. Unfortunately, right now, anyone that wants to view the component workspace needs to pull down our repository from git and run *npm run storybook* (or *yarn storybook*). This works really well for developers, but isn't so great for those who don't want to interact with the code (or git). Thankfully, it's pretty straightforward to build our Storybook in a way that we can deploy it to a server:

```
npm run build-storybook
```

We should be notified that Storybook has created a compiled version of our workspace in *storybook-static*. We can deploy the code from this folder to a server or have this happen automatically as part of a CI process. This is a bit outside the scope of this study, but it is a scenario that is supported by Storybook.

Wrapping Up

In this book, we've looked at why we should use a component workspace, how it helps us build better software, and how we can use it to help us facilitate better communication. It's interesting that these are things that we need to write good software. The tooling isn't the only way that we can achieve good software, but it certainly helps.

Index

A, B

API data
- considerations, 84, 85
- data loading
  - container/presentational components, 91
  - mock data, 91–93
  - ProductList component, 88
  - products component, 87
  - Storybook output, 88, 90
  - useEffect statement, 89

argTypes property, 119

Automating repetitive tasks
- build generators
  - adding variables, 114, 115
  - component file, 111
  - index file, 112
  - navigation, 111
  - run, 112, 113
  - Storybook stories file, 112
  - test file, 112, 113
- CLI
  - generator, 108
  - Hygen, 108
  - tools, 108

C

Cascading Style Sheets (CSS), 60
- benefits, 61
- building theme, 69, 70
- drawbacks, 61, 62
- element IDs, 61
- HTML elements, 60
- JavaScript
  - benefits, 67
  - Card component, 65
  - defaultProps, 66
  - drawbacks, 67
  - emotion, 66
  - React components, 64
  - rendering, 66
  - style, 65
  - styled-components, 66
- login box, 61
- preprocessors, 62, 63
  - benefits, 63
  - drawbacks, 64
- selector, 60
- styling solution, 68
- utility-first styling libraries, 67, 68

© Ryan Lanciaux 2021
R. Lanciaux, *Modern Front-end Architecture*, https://doi.org/10.1007/978-1-4842-6625-0

Cascading Style Sheets (CSS) (*cont.*)
benefits, 68
drawbacks, 68
Command-line interface (CLI), 108
Component, documenting, 117–121
Create React App (CRA), 14, 74

D, E

Data loading, 84
Documentation
MDX, 122, 123
workspace, 124
Document object model (DOM), 76

F, G

Feature-based development, 85, 86

H, I, J, K, L

Hygen
code, 110
create folder, 110
generator, 109
installation, 109
templates, 109, 110

M

Mac/Linux installation instructions
command, 20
Node js, 21
version, 20, 21
Mise en place philosophy, 3, 4

N, O, P, Q

Navigating between pages
Application component, 96, 97
bookmark, 97
challenges, 97
change navigation, 96
Home/About page, 95
navigation components, 96
Navigation
index.js file, 104
Routes.js file, 105
rudimentary, 105
Node Version Manager (NVM), 15

R

React components, 76
add tests, 77, 78
branching code paths, 77
fireEvent, 79
isSoldOut prop, 78, 79
mock function, 79
onAddToCart, 79

ProductListItem.js, 77
React Testing Library, 77
rendering, 78
React Router library, 98, 99
React Testing Library, 77
Reusable components
abstraction
creation, 51
primary factor, 51
additional components, 49, 50
arranging, 54
components types, 55, 56
component states
Boolean flags, 51
bugs, 52
loading/error, 52
name/value, 53
ProductList folder, 51
state charts, 52
state machines, 52, 53
statusTypes object, 53
ProductListItem, 48
purpose, 48
Routes
basic carts, 99, 100
codebase, 100
import, 103
Mock.server.js, 101–103
react-router-dom
package, 101
Routes.js, 101
screen folders, 100
user profile
components, 99, 100
Routing, 84, 85
benefits, 99
link component, 98
navigation component, 99
switch component, 98
URL, 99

S, T

Software
components, 5
foundational, 6, 7
meeting, 8
workspace, 8, 9
good software, 2–4
newness, 2
processes, 5
product search mockup, 5, 6
State management, 84
Storybook, 9, 10, 24, 73
add-ons
actions, 39, 40
importing, 39
knobs, 40–42
name/value, 41
ProductListItem.stories.js, 39
story options, 40
advantages, 80
components, 31
Component Story Format, 34

Storybook (*cont.*)
component code, 36
create object, 35
CSS file, 37
documentation, 35
JavaScript console, 38
JavaScript objects/functions, 34
values, 36
creation, 32
folder/file structure, 31
home page design, 29, 30
implementation, 24, 25
import, 81, 82
index.js file, 32
installation, 25, 26
less code/duplication, 80
personal/team preference item, 33
product item, 30, 31
ProductListItem, 33
first phase, 36, 37
updation, 35, 38
ProductListItem stories, 81
StoriesOf API, 33, 34
variants
Add to Cart, 42
Boolean knob, 43
isSoldOut, 42
sold out story, 43
story *vs*. knob, 44

U

Unit tests, 73
assert works, 74
expect method, 75
JavaScript, 74
Jest, 74, 75
npm run test, 76
test script, 75
write test, 75

V

Version manager, 14, 15

W, X, Y, Z

Windows installation instructions
Command Prompt application, 15
Node js, 18–20
Node version, 16
nvm, 17, 18
removing Node js, 16, 17
Workspae
adding, 24
built-in utilities, 21, 22
React application, 22–24
system requirements, 13
Node.js, 14
version manager, 14, 15