

Pure React

DAVE CEDDIA

Contents

1	Introduction	6
	Why Just React?	9
	How This Book Works	10
	Environment Setup	13
	Debugging Crash Course	16
2	Hello World	18
3	JSX: What and Why	21
	What Is JSX?	21
4	Working With JSX	26
	Composing Components	26
	“If” in JSX	33
	Exercises	36
5	Example: Tweet Component	40
6	Props	54
	Passing Props	54
	Receiving Props	56
	Communicating With Parent Components	58

7 Example: Tweet With Props	60
ES6: Template Strings	62
What Exactly Should Be Passed as a Prop?	65
Guidelines for Naming Props	66
8 PropTypes	71
Documentation and Debugging In One	71
How Do I Validate Thee, Let Me Count the Ways	74
Example: Tweet with PropTypes	79
PropTypes as Documentation	82
Exercises	84
9 Children	86
Different Types of Children	88
Dealing with the Children	88
PropTypes for Children	89
Customizing Children Before Rendering	90
Exercises	92
10 Example: GitHub File List	94
The “key” Prop	99
Exercises	108

11 State in Classes	112
Example: A Counter	112
setState Is Asynchronous	115
Shallow vs Deep Merge	117
Cleaner Syntax for Class Components	118
Handling Events	120
Exercises	121
12 The Component Lifecycle	122
Phases	122
Mounting	126
Rendering	127
Unmounting	128
Error Handling	128
13 API Requests in React	129
Choose an HTTP Library	129
Fetch Data and Display It	131
Exercises	134
14 State in Functions	135
Introducing Hooks	135
The useState Hook	135

The “Magic” of Hooks	138
Rules of Hooks	141
Update State Based on Previous State	141
State as an Array	143
State as an Object	144
Exercises	146
15 Thinking About State	147
What to Put in State	147
Thinking Declaratively	149
Where to Keep State	152
“Kinds” of Components	156
16 Input Controls	157
Controlled Inputs	157
Uncontrolled Inputs	160
Exercises	162
17 The useReducer Hook	164
What’s a Reducer?	164
A More Complex Example	166
So... is Redux Dead?	170
Exercises	172

18 The useEffect Hook	173
Limit When an Effect Runs	174
Only Run on Mount and Unmount	177
Fetch Data With useEffect	179
Re-fetch When Data Changes	181
Making Visible DOM Changes	184
Exercises	185
19 The Context API	186
Before: A Prop Drilling Example	186
The “Slots” Pattern	190
Using the React Context API	192
The “Render Props” Pattern	197
Advanced Context Patterns	198
The useContext Hook	201
20 Example: Shopping Site	204
Create ItemPage	214
Create the Item Component	218
Create the CartPage Component	225
Exercises	236
21 Where To Go From Here	239

1 Introduction

There's a problem with frontend development today: it's *overwhelming*.

There are thousands of libraries out there, each doing one little thing. Groups of them evolve to become de facto standards, but with no *actual* standards in sight.

The React ecosystem is especially guilty of this: there's React, Redux, Webpack, Babel, React Router, and on and on.

Boilerplate projects exist to make this “easier” by bundling a bunch of choices together. “It'll be easier to learn,” the thinking goes, “if you don't have to make all the choices yourself.”

Instead, the opposite happens: rather than being overwhelmed by the choices, you're now overwhelmed by the sheer amount of code that came “for free” with the boilerplate, and you have *NO IDEA* what any of it actually does. That's a scary place to be.

Learning everything at once is massively overwhelming. So in this book, we will take a different approach. A more sane approach. We will learn Pure React.

Pure React: The core concepts of React, in isolation, without Redux, Webpack, and the rest.

When you achieve what's contained in this book – when you learn React *cold*, you'll be able to go on and learn all of its friends with ease: Redux, Router, and everything else you might need.

Not only will you be *able* to learn those other libraries, but you will be well-equipped. You'll have a solid foundation.

This book has been designed to get you from *zero* to *React* quickly, and with maximum understanding.

What you *won't* be doing here is what plays out in many tutorials, where you copy and paste each block of code until you have a working app at the end. “Voila!” they say. “Now you know React and Redux and GraphQL!”

Most tutorials try to teach too much, too quickly. You might learn one or two concepts with that approach, but it's a shaky foundation. Inevitably, when you sit down to write your own app, everything you supposedly “learned” instantly vanishes, leaving you staring at a blinking cursor wondering what that first line of code should be.

You can only get so far by copying and pasting.

And you know this already, otherwise you wouldn't be here.

So in this book we'll follow a different approach: I'll show you a concept, with some example code. *Then* (and this is the important part) you will *use* that concept in the exercises that follow, until it's second nature. Rinse and repeat until we've covered all the core pieces of pure React – and there aren't too many.

React Doesn't Do Much

React itself is small. It doesn't have a way to fetch data, it doesn't have anything built-in for animations or a special styling system, and it doesn't include a routing library. The popular React Router, despite the name, is a third-party library and not part of React.

The community around React has built all of these things, and most real-world apps will incorporate many third-party libraries like React Router, Redux, react-query, react-modal, and so on.

It's because React is small that it's easy to bolt on additional libraries as you need them. Installing all of them at the very beginning is a terrible way to learn the stack, though. It's much easier to learn them one at a time, starting with the thing they all revolve around: React itself.

What We'll Cover

We will start where most programming books start, with Hello World – just getting a few words on the screen. From there, we'll look at how to compose components together and how to work with JSX, React's HTML-like syntax for rendering elements to the page.

Once you have a grasp on how to create static components, you'll learn about “props” as a way to pass in the data they need, and “propTypes” for documenting and debugging the props that a component requires.

We'll cover React's special “children” prop, which is a powerful tool for building reusable components.

You'll learn about “state,” how it differs from props, and how to organize it in an application – both with the old standby, *classes*, and the fun new *Hooks*.

We'll touch on how to make API requests with React (because even though it's not a “React thing”, most apps need to do it at some point!), and how to work with form inputs.

You'll also learn how to use more advanced features of React: the `useReducer` hook, the `useEffect` hook, and the Context API.

Why Just React?

Without a solid understanding of React, learning libraries and tools like Redux and Webpack will only slow down your learning process. It's very tempting to dive in and learn it all at once, especially if you have a fun project in mind (or a deadline to meet).

However, learning everything at once will be slower in the long run.

Think of these libraries and tools as layers in a foundation.

If you were building a house, would you skip some steps to get it done faster? Say, start pouring the concrete before laying some rocks down? Start building the walls on bare earth?

Or how about making a wedding cake: the top part looks the most fun to decorate, so why not start there? Just figure out the bottom part later!

No?

Of course not. You know those things would lead to failure. They would, perhaps counterintuitively, *slow things down* rather than speed them up.

So does it make sense to learn React in tandem with ancillary tools like Webpack + Babel + Redux + React Router *all at once*? Doesn't that sound like a ton of overwhelming confusion?

Instead, the most efficient approach is to learn these one at a time. This book will teach you how to use React, and then you'll be ready to tackle the next piece of the puzzle.

How This Book Works

How Much Time Will This Take?

The basic concepts of React can be learned in a matter of days. This book covers those basics and also contains exercises after each major concept to reinforce your understanding.

Most of the exercises are short. There are a few that are more involved. The principle behind them is the same as the idea behind homework in school: to drill the ideas into your head by combining repetition and problem solving.

The theme behind the whole process is this: avoid getting overwhelmed. Quitting won't get you anywhere. Slow and steady, uh, learns the React.

Build Small Things and Throw Them Away

This is the awkward middle step that a lot of people skip. Moving on to Redux and other libraries without having a firm grasp of React's concepts will lead straight back to overwhelmsville.

But this step isn't very well-defined: what should you build? A prototype for work? Maybe a fancy Facebook clone, something substantial that uses the whole stack?

Well, no, not those things. They're either loaded with baggage or too large for a learning project. You want to build *small* things.

Don't Build a Prototype

"Prototypes" (for work) are usually terrible learning projects, because you *know in your heart* that a "prototype" will never die. It will live long beyond the prototype phase, morph into shipping software, and never be thrown away or rewritten. As soon as some manager sees that it works, features will be piled on. "We'll refactor it some day" will turn out to be a lie. The code will grow bloated and disorganized.

All of these, and more, are reasons why a prototype is a bad choice as a learning project.

When you know it won't be throwaway code, *the future* looms large. You start to worry... Shouldn't it have tests? Shouldn't I make sure the architecture will scale? Am I going to have to refactor this mess later? And shouldn't it have tests?

Worrying about architecture and scalability and “the future” is a bad strategy for learning the basics of a new technology.

On the flip side, if you build a prototype believing that it is throwaway code, it probably won’t be very good code. Then when your boss’ boss sees how awesome the prototype looks, they will absolutely not allow you to rewrite it with all the best practices you’ve learned. And that’s a recipe for an unmaintainable code base.

So What Should You Build?

This book exists to answer this question, and help you through it. The short answer is this:

Build small, throwaway apps.

The sweet spot is somewhere between “Hello World” and “entire clone of Twitter.”

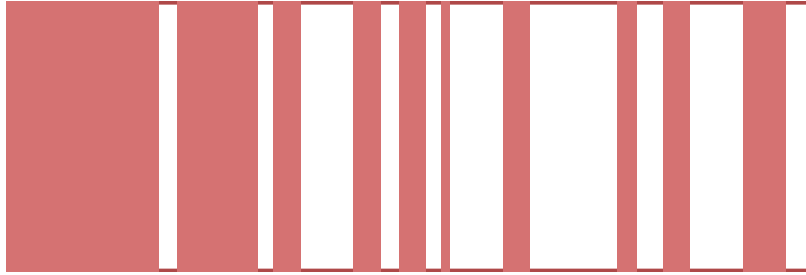
We’ll start off with Hello World, of course. No self-respecting programming book would be complete without that.

As your skills grow, low-fidelity copies of simple apps and sites like Reddit, Hacker News, and Slack make great projects. Designers call this “copywork,” and it’s great because it frees you from having to make *product decisions* like “what page the user should see after they log in” or “what color should the buttons be.” You can simply focus on learning React.

By the end of this book you’ll be building replicas of those popular apps and more. They’ll come together quickly once you can clearly “think in components,” a skill you’ll develop as you progress through the book.

Learning With Small Projects

I believe that you can get more learning value out of small projects than large or full-stack ones, at least in the beginning. Here’s an idea of what I mean. The colored bars are periods of maximum learning, and the gaps are where you’re doing things you already know how to do:



Learning with a Big Project



Learning with Small Projects

At some point, the larger projects have diminishing returns. The first few times you use a text input and have to wire it up to maintain its state, you're learning. By the tenth time, it's old hat.

That isn't to say that large projects aren't valuable, but I don't believe they're good *first* projects. Start small, build a few small things, then build a bigger thing or two.

This is the idea behind *deliberate practice* – the activity should be just beyond your current skill level. Not so hard that you get frustrated and quit, but not so easy that you can breeze right through, either. The exercises in this book are designed around that idea, to push you a *little* outside your comfort zone and make you think.

Environment Setup

Before we dive in, we'll need to set up an environment.

Don't worry, there's no boilerplate to clone from GitHub. No Webpack config, either.

Instead, we're using Create React App, a tool Facebook made. It provides a starter project and built-in build tools so you can skip to the fun part – creating your app!

Prerequisites

Tools

- Node.js (at least 8.10.0)
- NPM (ideally version ≥ 5.2)
- Google Chrome, Firefox, or some other modern browser
- React Developer Tools
- Your text editor or IDE of choice

You can use [nvm](#) to install Node and NPM on macOS & Linux, or [nvm-windows](#) on Windows. You can also download an installer from [nodejs.org](#), but the advantage of nvm is that it makes it very easy to upgrade your version of Node in the future.

Any modern browser should suffice. This book was developed against Chrome, but if you prefer another browser, it will probably work fine.

Create React App

Throughout this book, you'll be creating a lot of little projects with Create React App. Because the `create-react-app` command very rarely needs to be updated, I suggest installing the tool permanently, by running:

```
npm install -g create-react-app
```

If you don't want to install an extra tool, you can use the `npx` command to create your projects, as in `npm create-react-app my-project-name`. The upside of that is that you don't need to install a tool. The downside is that every command will take extra time because it needs to install Create React App from scratch every time.

Even without ever updating the global `create-react-app` command, it is designed to always pull down the latest version of React when it creates a new project. You don't need to worry about it becoming outdated.

Yarn

Yarn is an alternative package manager for JavaScript, released in June of 2016. It has all the same packages and it's often faster than NPM. Throughout the book I'll show the `npm` commands for installing packages, but feel free to use Yarn if you like.

React Developer Tools

The React Developer Tools can be installed from here:

<https://github.com/facebook/react-devtools>

Follow the instructions to install the tools for your browser. The React dev tools allow you to inspect the React component tree (as opposed to the regular DOM elements tree) and view the props and state assigned to each component. Being able to see how React is rendering your app is extremely useful for debugging.

Knowledge

You should already know JavaScript (at least ES5), HTML, and CSS. I'll explain the newer JavaScript features as they come up (you don't need to already know ES6 and beyond).

If you aren't very comfortable with CSS, don't worry too much – you can use the code provided in the book. A few exercises might be challenging without knowledge of CSS but you can feel free to skip those or modify the designs into something you can implement.

I don't recommend learning JavaScript and React at the same time. When everything looks new, it can be hard to tell where “JavaScript” ends and “React” begins. If you need to brush up on JS, here are some good (and free!) resources:

- Speaking JavaScript (book): <http://speakingjs.com/>
- Exercism (exercises): <http://exercism.io/languages/javascript>
- You Don't Know JS (book series): <https://github.com/getify/You-Dont-Know-JS>

A passing familiarity with the command line will be helpful as well. We'll mostly just be using it to install packages.

Project Directory

You'll be writing a lot of code throughout this book. To keep it organized, create a directory for the exercises. Name it `pure-react`, or whatever you like.

That's it! Let's get to coding.

Debugging Crash Course

When things go wrong, here are the steps to follow:

1. Don't panic.
2. Manually refresh the page. Sometimes the auto-refreshing mechanism breaks.
3. Check the browser console, manually refresh the page, and fix any errors you see. **Don't ignore errors and warnings!** Keep the console tidy. If you let problems pile up, non-urgent ones might end up obscuring the one that broke the app. Always scroll to the top and start with the first error; don't start with the most recent one.
4. Check the command line terminal where you ran `npm start`. Look for any errors, and fix those.

To open the browser console in Chrome on Mac, press Option+Command+I. In Chrome on Windows or Linux, press Ctrl+Shift+I. Then make sure you are on the “Console” tab.

If you don't know how to open your browser's Developer Tools, Google for “open browser console [your_browser]”.

Likewise, if you have no idea what an error means, copy and paste it into Google.

Still completely stuck? Stack Overflow has an abundance of React-related questions and answers, and the Reactiflux community at <https://www.reactiflux.com/> is full of friendly people who can help. There is also a #react IRC channel on Freenode, which you connect to with an IRC client or visit <http://irc.lc/freenode/reactjs>.

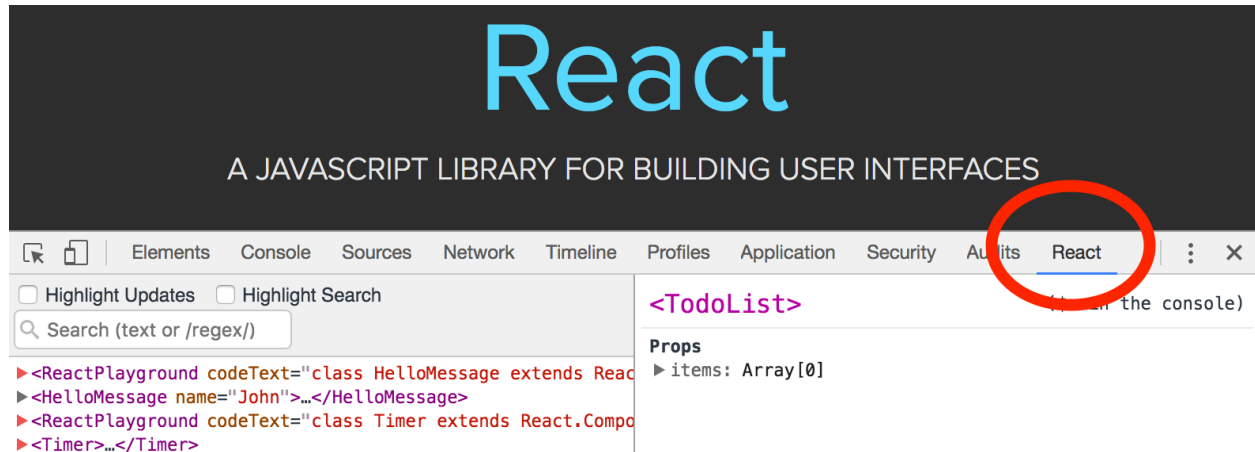
Keep the Console Open!

The browser console is an amazingly helpful tool for debugging, and it's a great idea to leave it open while you work through the examples and exercises. It'll help you catch typos and save you a lot of time hunting down problems.

Create React App does a good job of catching errors and displaying them full-screen, showing the snippet of code where things went wrong. Sometimes, though, an error will result in a completely blank screen. In those cases, always check the console first.

React Dev Tools

Once you have installed the React dev tools, you'll see a “React” tab in the browser devtools, like this:



Choose this tab, and look for your component in the tree. You can use the search box to find it without having to drill down. When you click on a component, its Props and State will appear in the side pane. Make sure the values agree with what you expect to see. You can even modify the Props and State right in the side pane, and see the component re-render with your changes. Try it out!

2 Hello World

At this point you have node and npm installed. All of the tools are ready. Let's write some code!

Step 1

Use `create-react-app` ("CRA") to generate a new project. CRA will create a directory and install all the necessary packages, and then we'll move into that new directory.

```
$ create-react-app react-hello
$ cd react-hello
```

The generated project comes with a prebuilt demo app. We're going to delete that and start fresh. Delete the files under the `src` directory, and create an empty new `index.js` file.

```
$ rm src/*
$ touch src/index.js
```

Step 2

Open up the brand new `src/index.js` file, and type this code in:

Type it out by hand? Like a savage? Typing it drills it into your brain *much* better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to understand React one day. Help 'em out.

```
import React from 'react';
import ReactDOM from 'react-dom';

function HelloWorld() {
  return (
    <div>Hello World!</div>
  );
}
```

```
    );  
  }  
  
  ReactDOM.render(  
    <HelloWorld/>,  
    document.querySelector('#root')  
  );
```

The `import` statements at the top are an ES6 feature. These lines will be at the top of every `index.js` file that we see in this book.

Unlike with ES5, we can't simply include a `<script>` tag and get React as a global object. So, the statement `import React from 'react'` creates a new variable called `React` with the contents of the `react` module.

The strings `'react'` and `'react-dom'` are important: they correspond to the names of modules installed by npm. If you're familiar with Node.js, `import React from 'react'` is equivalent to `const React = require('react')`.

Step 3

From inside the `react-hello` directory, start the app by running this command:

```
$ npm start
```

A browser will open up automatically and display "Hello World!"

How the Code Works

Let's start at the bottom, with the call to `ReactDOM.render`. That's what actually makes this work. This bit of code is regular JavaScript, despite the HTML-looking `<HelloWorld/>` thing there. Try commenting out that line and watch how Hello World disappears.

React uses the concept of a *virtual DOM*. It creates a representation of your component hierarchy and then *renders* those components by creating real DOM elements and inserting them where you tell it. In this case, that's inside the element with an `id` of `root`.

`ReactDOM.render` takes 2 arguments: what you want to render (your component, or any other React Element) and where you want to render it into (a real DOM element that already exists).

```
ReactDOM.render([React Element], [DOM element]);
```

Above that, we have a *component* named `HelloWorld`. The primary way of writing React components is as plain functions like this. Most people call them “function components” but you might also see them called “functional components” or “stateless function components” (SFC for short).

There are 2 other ways to create components: ES6 classes, and the now-deprecated `React.createClass`. You may still see the `createClass` style in old projects or Stack Overflow answers, but it’s not in common use anymore. Primarily we’ll be writing components as functions.

The HTML-like syntax inside the render function is called *JSX*, and we’ll cover that next.

3 JSX: What and Why

One of the first things you probably noticed about React code is that it looks like the component function is returning HTML. This HTML-like syntax is actually called JSX.

What Is JSX?

JSX is a syntax invented for React that looks very similar to (X)HTML. It allows you to create elements by writing in a familiar-looking syntax, instead of writing out function calls by hand. The HTML-like syntax actually compiles down to real JavaScript.

Did you notice how there are no quotes around the “HTML”? That’s because **it’s not a string**. The lack of quotes is not just a trick, either. React is *not* parsing the tags and converting them into HTML.

I know, I know, it looks like HTML. In reality though, JSX is just a nice syntax for function calls that create DOM elements.

So what is React actually doing here? How does this work?

JSX Is Compiled to JavaScript

The JSX elements you write are actually compiled down to JavaScript by a tool called Babel. Babel is a compiler that transforms code into valid ES5 JavaScript that all browsers can understand, and it’s bundled in with projects created by Create React App.

After you run `npm start`, a tool called Webpack is watching for files to change. When they do, it feeds those files into Babel, which turns JSX into JS, and sends it to your browser via the development server running on port 3000.

Each JSX element becomes a function call, where its arguments are its attributes (“props”) and its contents (“children”).

Here’s an example of a simple React component that returns some JSX:

```
function Hello() {  
  return <span>Hello!</span>;  
}
```

And here is the JavaScript generated by the Babel compiler:

```
function Hello() {  
  return React.createElement(  
    'span',  
    {},  
    'Hello!'  
  );  
}
```

The `React.createElement` function signature looks like this:

```
React.createElement(  
  string|element,  
  [propsObject],  
  [children...]  
)
```

The `string|element` can be a string describing an HTML or SVG tag (like `'div'` or `'span'`), or it can be a component a.k.a. function (like `HelloWorld`, with no quotes).

The `propsObject` and `children` are optional, and you can also supply more than one child by passing additional arguments:

```
function HelloWorld() {  
  return React.createElement(  
    'div',  
    {},  
    'Hello',  
    'World'  
  );  
}
```

You can also nest the calls:

```
function ManyChildren() {
  return React.createElement('div', {},
    React.createElement('div', {}, 'Child1'),
    React.createElement('div', {}, 'Child2',
      React.createElement('div', {}, 'Child2_child')
    )
  );
}
```

Try it yourself! Rewrite the `HelloWorld` component to call `React.createElement` instead of returning JSX.

```
function HelloWorld() {
  return React.createElement(
    /* fill this in */
  );
}
```

Here is a slightly more complicated bit of JSX, and a preview of what's to come. You can see that it references a function parameter named `props`. We haven't talked about props yet, but this is the way you pass arguments to React components.

```
function SongName(props) {
  return (
    <span className='song-name'>
      {props.song.name}
    </span>
  );
}
```

And here is what it compiles to:

```
function SongName(props) {  
  return (  
    React.createElement('span',  
      { className: 'song-name' },  
      props.song.name  
    )  
  );  
}
```

See how JSX is essentially a nice shorthand for writing function calls? You don't even have to use JSX if you don't want to – you can write out these function calls manually.

Your first instinct might be to avoid writing JSX because you don't like the look of “HTML in JS.” Maybe you'd rather write real JavaScript function calls, because it feels more “pure” somehow. I suggest giving JSX an honest try before you give up on it.

Writing out the `React.createElement` calls is not a common approach in the React community. Essentially all React developers use JSX, which means code that you see in the wild (on GitHub, Stack Overflow, etc.) is likely to be written with it.

But... Separation of Concerns!

If you've internalized the idea that mixing JS with HTML is just *wrong*, and that each language should be kept in its own file, you're not alone! On the other hand, if you're new to web development and never had any of these hangups, you can treat the next few paragraphs as a history lesson.

Before I got into using React, I had a lot of apprehension about using JSX. It took some time (and writing a few small apps) before I began to understand why it was good. If this is you, too, don't worry – the exercises in this book will give you enough practice to come to grips with it.

I believe the disdain for mixing HTML with JS has a bit of cargo-cult “tradition” behind it. It is (let's say *was*) a piece of lore passed down through the generations about the Right Way to build web apps.

If you have flashbacks to PHP or JSPs where SQL statements were mixed in with HTML and thousands of lines of JavaScript, and you never want to go back to that world, I don't blame you. React's pattern of building with components helps prevent this.

Unseparated Concerns

When you step back and think about it, there are some good reasons to combine the logic and the view together.

If you've ever used something like Angular 1.x, or even a backend MVC framework like Ruby on Rails, you've probably written some logic in one file and the HTML in a separate template file.

Tell me, how often have you opened up the template to tweak something without having to look at (or change!) the associated logic? How often have you changed the JS without having to touch the template?

In most code I've worked with, it's rare that I could add new functionality without changing both the template and its associated JS code. If they were truly separate concerns, this would not be necessary.

We like to think that splitting the JS and the HTML into separate files magically transforms them into "separated concerns." Reusability here we come!

Except it rarely works that way. The JS code and its related template are usually pretty tightly coupled, and naturally so – they're two sides of the same coin.

Splitting code into separate files does not automatically lead to separation of concerns.

The story is similar with good old jQuery. The HTML is blissfully unaware that JavaScript even exists, and yet the JS and HTML are tightly coupled by the fact that the jQuery selectors must know something about the page structure. If the structure changes, the code must change.

If you haven't noticed, I'm trying to make the case that the template and the view logic could actually coexist in the same file and it *might actually make more sense to do it that way*.

You don't have to believe me right now. Just keep the idea in the back of your mind as you work through the examples and exercises. You may find (as I did) that merging the logic and view makes your code easier to navigate, easier to write, and easier to debug. You'll spend less time hopping between files when all the related functionality is in one place.

4 Working With JSX

Composing Components

JSX, like HTML, allows you to nest elements inside of one another. This is probably not a big surprise.

Let's refactor the `HelloWorld` component from earlier to demonstrate how composition works. Here's the original `HelloWorld`:

```
function HelloWorld() {  
  return (  
    <div>Hello World!</div>  
  );  
}
```

Leaving the `HelloWorld` component intact for now, create two *new* components: one named `Hello` and one named `World`. `Hello` should render `Hello` and `World` should render `World`. You can basically copy-and-paste the `HelloWorld` component and just change the text and the function name.

Go ahead and try making that change yourself. I'll wait.

...

...

Got it?

It's important to actually *type this stuff out* and try it yourself! Don't just read while nodding along, because you won't actually learn it that way. It's very easy to look at code and think, "Yep, that all makes sense." You'll never know if you truly understand until you *try it*.

Your two new components should look like this:

```
function Hello() {  
  return <span>Hello</span>;  
}
```

```
function World() {  
  return <span>World</span>;  
}
```

Now, change the HelloWorld component to use the two new components you just created. It should look something like this:

```
function HelloWorld() {  
  return (  
    <div>  
      <Hello/> <World/>!  
    </div>  
  );  
}
```

Assuming the app is still running, the page should automatically refresh. If not, make sure the app is running (run `npm start` if it's not).

You should see the same “Hello World!” as before. Congrats!

I know this seems painfully simple, but there are some lessons here, I promise.

The next few examples depend upon the `Hello` and `World` components that you should have created above, so make sure those exist before you continue.

Wrap JSX with Parentheses

A quick note on formatting: you might notice I wrapped the returned JSX inside parentheses, `()`. This isn't strictly necessary, but if you leave off the parens, the opening tag must be on the same line as the return, which looks a bit awkward:

```
function HelloWorld() {  
  return <div>  
    <Hello/> <World/>!  
  </div>;  
}
```


Just for kicks, try moving the `<div>` onto its own line, without the surrounding parens:

```
function HelloWorld() {  
  return  
    <div>  
      <Hello/> <World/>!  
    </div>;  
}
```

This will fail with an error.

If you look in the browser console, you'll also likely see a warning about "Nothing was returned from render."

This is because JavaScript assumes you wanted a semicolon after that return (because of the newline), effectively turning it into this, which returns undefined:

```
function HelloWorld() {  
  return;  
  <div>  
    <Hello/> <World/>!  
  </div>;  
}
```

So: feel free to format your JSX however you like, but if it's on multiple lines, I recommend wrapping it in parentheses.

Return a Single Element

Notice how the two components are wrapped in a `<div>` in the `HelloWorld` example:

```
function HelloWorld() {  
  return (  
    <div>  
      <Hello/> <World/>!  
    </div>  
  );  
}
```

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

Here's a little exercise: try removing the `<div>` wrapper and see what happens. You should get this error:

Adjacent JSX elements must be wrapped in an enclosing tag.

If this seems surprising, remember that JSX is compiled to JS before it runs:

```
// This JSX:
function HelloWorld() {
  return (<Hello/> <World/>);
}

// Becomes this JS:
function HelloWorld() {
  return (
    React.createElement(Hello, null) React.createElement(World, null)
  );
}
```

Returning two things at once is pretty obviously not gonna work. So that leads to this very important rule:

A component function must return a single element.

But wait! Could you return an array? It's just JavaScript after all...

```
// This JSX:
function HelloWorld() {
  return [<Hello/>, <World/>];
}
```

```
}  
  
// Would turn into this JS  
// (notice the brackets).  
function HelloWorld() {  
  return [  
    React.createElement(Hello, null),  
    React.createElement(World, null)  
  ];  
}
```

Try it out! It renders correctly.

But if you open up the browser console, you'll see a warning:

Each child in an array or iterator should have a unique "key" prop.

As the warning suggests, React requires a unique key prop for each JSX element in an array. We'll learn more about the key prop later on, but in the meantime, there are two ways to solve this problem: either wrap the elements in a single enclosing tag, or wrap them in a fragment.

Wrap With a Tag

The most obvious way to return multiple elements is to wrap them in an enclosing tag, like a `<div>` or ``. However, it has the side effect of influencing the DOM structure.

For example, this React component...

```
function HelloWorld() {  
  return (  
    <div>  
      <Hello/> <World/>!  
    </div>  
  );  
}
```

...will render a DOM structure like this:

```
<div>
  <span>Hello</span>
  <span>World</span>
</div>
```

A lot of the time, this is perfectly fine. But sometimes, you won't want to have a wrapper element, like if you have a component that returns two table cells:

```
function NameCells() {
  return (
    <td>First Name</td>
    <td>Last Name</td>
  );
}
```

You can't wrap these elements in a `<div>`, because the `<td>` table cells need to be direct descendants of a `<tr>` table row. How can you combine them?

Fragments

React's answer is the *fragment*. This component was added in React 16.2, and can be used like this:

```
function NameCells() {
  return (
    <React.Fragment>
      <td>First Name</td>
      <td>Last Name</td>
    </React.Fragment>
  );
}
```

After rendering, the `React.Fragment` component will “disappear”, leaving only the children inside it, so that the DOM structure will have no wrapper components.

Fragments make it easier to produce valid HTML (such as keeping `<td>` elements directly inside `<tr>`s), and they keep the DOM structure flatter which makes it easier to write semantic HTML (which is also usually more accessible HTML).

Fragment Syntax

If you think `React.Fragment` looks clunky, I don't blame you. JSX supports a special syntax that looks like an "empty tag" and is much nicer to write:

```
function NameCells() {
  return (
    <>
      <td>First Name</td>
      <td>Last Name</td>
    </>
  );
}
```

This `<></>` syntax is the preferred way to write fragments, and this feature will be available as long as you're working in a new-enough project (Babel 7+, Create React App 2+)

JavaScript in JSX

You can insert real JavaScript expressions inside JSX code, and in fact, you'll do this quite often. Surround JavaScript with single braces like this:

```
function SubmitButton() {
  const buttonLabel = "Submit";
  return (
    <button>{buttonLabel}</button>
  );
}
```

Remember that this will be compiled to JavaScript, which means that the JS inside the braces must be an *expression*. An expression produces a value. These are expressions:

```
1 + 2
buttonLabel
aFunctionCall()
aFunctionName
```

Each of these results in a single value. In contrast, *statements* do not produce values and can't be used inside JSX. Here are some examples of statements:

```
const a = 5
if(true) { 17; }
while(i < 7) { i++ }
```

None of these things produces a value. `const a = 5` declares a variable with the value 5, but it does not *return* that value.

Another way to think of statement vs expression is that *expressions* can be on the right hand of an assignment, but statements cannot.

```
// These aren't valid JS:
a = let b = 5;
a = if(true) { 17; }
```

You can also ask yourself, “Could I return this value from a function?” If the answer is yes, that’s an expression and you can write it inside JSX within `{single braces}`.

“If” in JSX

The next question you might wonder is, “How do I write a conditional if I can’t use ‘if’?” There are a couple of options.

The first is the ternary operator (the question mark, `?`). Use it like this:

```
function ValidIndicator() {  
  const isValid = true;  
  return (  
    <span>{isValid ? 'valid' : 'not valid'}</span>  
  );  
}
```

You can also use boolean operators such as `&&` like this:

```
function ValidIndicator() {  
  const isValid = true;  
  return (  
    <span>  
      {isValid && 'valid'}  
      {!isValid && 'not valid'}  
    </span>  
  );  
}
```

Comments in JSX

If you need to put a comment into a block of JSX, the syntax is a little awkward. Remember that any JavaScript code needs to be inside single braces, and think of the comments as JavaScript. To comment chunks of JSX, put the comments inside a JavaScript block like this:

```
function ValidIndicator() {  
  const isValid = true;  
  return (  
    <span>  
      {/* here is a comment */}  
      {isValid && 'valid'}  
      {!isValid && 'not valid'}  
      {  
        // Double-slash comments are
```

```

    // OK in multi-line blocks.
  }
  { /*
    <span>thing one</span>
    <span>thing two</span>
    */ }
</span>
);
}

```

Capitalize Component Names

The components you write must begin with an uppercase letter. This means using names like `UserList` and `Menu` and `SubmitButton`, and not names like `userList`, `menu`, and `submit_button`.

In JSX, a component that starts with a lowercase letter is assumed to be a built-in HTML or SVG element (`div`, `ul`, `rect`, etc.).

A bit of history: early versions of React kept a “whitelist” of all the built-in element names so it could tell them apart from custom ones, but maintaining that whitelist was time-consuming and error-prone. If a new SVG element made its way into the spec, you couldn’t use it until React updated that list! So they killed off the list, and added this rule.

Close Every Element

JSX requires that every element be closed, similar to XML or XHTML. This includes the ones you might be used to leaving open in HTML5, like `
` or `<input>` or maybe even ``.

```

// DO THIS:
return <br />;
return <input type='password' ... />;
return <li>text</li>;

// NOT THIS:
return <br>;
return <input type='password' ...>;
return <li>text;

```


Exercises

Create a new app for these exercises by running:

```
$ create-react-app jsx-exercises
```

Open the `src/index.js` file and replace the contents, similar to Hello World. Fill in the rest.

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyThing() {
  // ...
}

ReactDOM.render(
  <MyThing/>,
  document.querySelector('#root')
);
```

You can delete or ignore the `src/App*` files and the logo. If you don't explicitly import them, they won't get bundled into your app.

1. Create a component that renders this JSX:

```
<div className='book'>
  <div className='title'>
    The Title
  </div>
  <div className='author'>
    The Author
  </div>
  <ul className='stats'>
    <li className='rating'>
      5 stars
    </li>
```

```
<li className='isbn'>
  12-345678-910
</li>
</ul>
</div>
```

2. See how JSX interprets whitespace. Try rendering these arrangements and take note of the output (hint: leading and trailing spaces are removed, and so are newlines):

a. Single lines

```
<div>
Newline
Test
</div>
```

b. Empty newlines

```
<div>
Empty

Newlines

Here
</div>
```

c. Spaces with newlines

```
<div>
&nbsp;Non-breaking
&nbsp;Spaces&nbsp;
</div>
```

d. Inserting spaces when content spans multiple lines

```

<div>
Line1
{' '}
Line2
</div>

```

3. Make a copy of the component from Exercise 1, and replace the JSX with calls to `React.createElement`. The output should be identical.
4. Return the appropriate JSX from this component so that when `username` is undefined or null, it renders “Not logged in”. When `username` is a string, render “Hello, username”.

```

function Greeting() {
  // Try all of these variations:
  //let username = "root";
  //let username = undefined;
  //let username = null;
  //let username = false;

  // Fill in the rest:

  // return (...)
});

```

5. One good way to learn a new syntax is to try breaking it – discover its boundaries. Try some of the things this chapter warned about and see what kind of errors you get. At the very least, it’ll familiarize you with what the errors mean if you make one of these mistakes later on.
 - a. Name a component starting with a lowercase letter, like “testComponent”.
 - b. Try returning 2 elements at once
 - c. Try returning an array with 2 elements inside
 - d. Can you put 2 expressions inside single braces, like `{x && 5; x && 7}`?
 - e. What happens if you use `return` inside a JS expression?
 - f. What about a function call like `alert('hi')`? Does it halt rendering?
 - g. Try putting a quoted string inside JSX. Does it strip out the quotes?

6. The HTML spec says that tables must be structured with a `table` element surrounding a `tbody`, which surrounds multiple `tr`s (the rows), which each surround multiple `td`s (the columns). Create a component called `Table` that renders a table with 1 row and 3 columns and any data you like. Open the browser console and make sure there are no warnings. Then, create a component called `Data` that renders the 3 columns, and replace the 3 `<td>`s with the `<Data/>` component.

5 Example: Tweet Component

To learn to “think in components” you’ll need to build a few, and we’re going to start with a nice simple one. We’ll follow a 4-step process:

1. Make a sketch of the end result
2. Carve up the sketch into components
3. Give the components names
4. Write the code!

1. Sketch

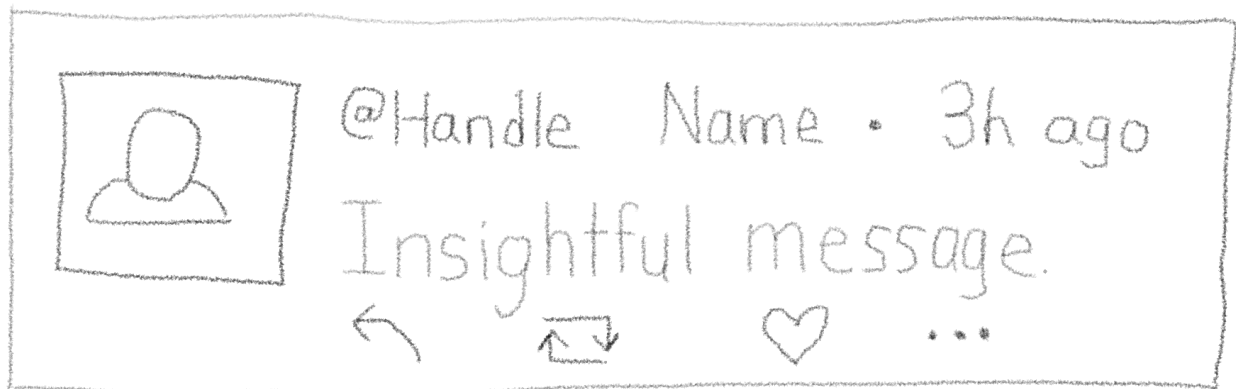
Spending a few seconds putting pen (or pencil) to paper can save you time later. Even if you can’t draw (it doesn’t need to be pretty).

We’ll start with a humble sketch because it’s *concrete* and gives us something to aim for. In a larger project, this might come from a designer (as wireframes or mockups) but here, I recommend sketching it out yourself before writing any code.

Even when you can already visualize the end result, spend the 30 seconds and sketch it out on real dead-tree paper. It will help tremendously, especially for the complicated components.

There’s something satisfying about building a component from a drawing: you can tell when you’re “done.” Without a concrete picture of the end result, you’ll compare the on-screen component to the grand vision in your head, and it will never be good enough. It’s easy to waste a lot of time when you don’t know what “done” looks like. A sketch gives you a target to aim for.

Here is the sketch we’ll be building from:



It’s rough on purpose: you don’t need a beautiful mockup. A simple pen-and-paper sketch works fine.

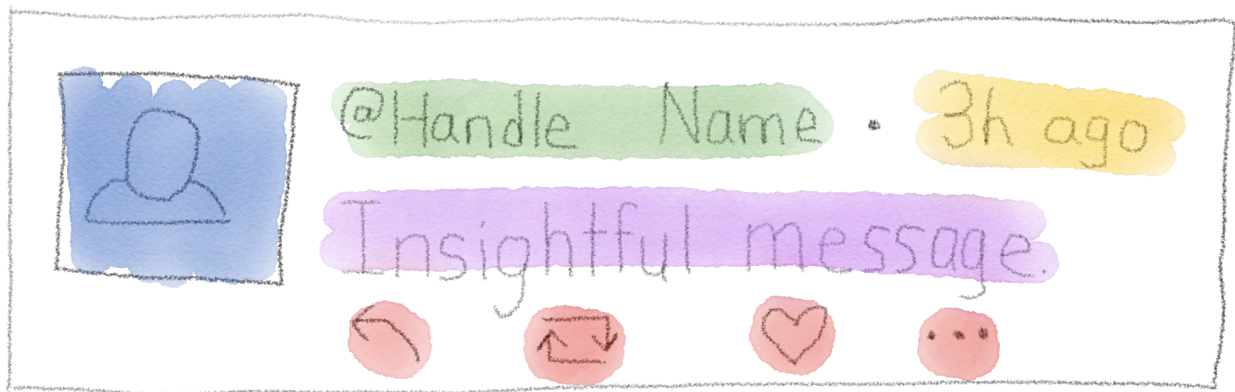
2. Carve into Components

The next step is to break this sketch into components. To do this, draw boxes around the “parts,” and think about reusability.

Imagine if you wanted to display 3 tweets, each with a different message and user. What would change, and what would stay the same? The parts that change would make good components.

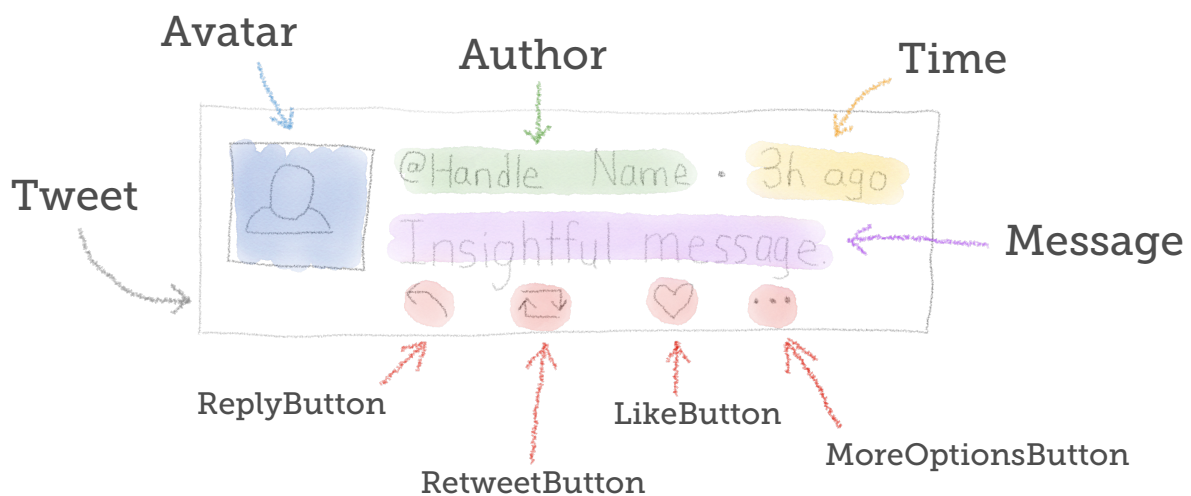
Another strategy is to make every “thing” a component. Things like buttons, chunks of related text, images, and so on.

Try it yourself, then compare with this:



3. Name the Components

Now that we’ve broken the sketch into pieces, we can give them names.



Each of these named items will become a component, with `Tweet` being the “parent” that groups them all together. The hierarchy looks like this:

- `Tweet`
 - `Avatar`
 - `Author`
 - `Time`
 - `Message`
 - `ReplyButton`
 - `LikeButton`
 - `RetweetButton`
 - `MoreOptionsButton`

4. Build

Now that we know what the component tree looks like, let’s get to building it! There are two ways to approach this.

Top-Down, or Bottom-Up?

Option 1: Start at the top. Build the `Tweet` component first, then build its children. Build `Avatar`, then `Author`, and so on.

Option 2: Start at the bottom (the “leaves” of the tree). Build `Avatar`, then `Author`, then the rest of the child components. Verify that they work in isolation. Once they’re all done, assemble them into the `Tweet` component.

So what’s the best way? Well, it depends (doesn’t it *always*?).

For a simple hierarchy like this one, it doesn’t matter much. It’s easiest to start at the top, so that’s what we’ll do here.

For a more complex hierarchy, start at the bottom. Build small pieces, test that they work in isolation, and combine them as you go. This way you can be confident that the small pieces work, and, by induction, the combination of them should also work (in theory, anyway).

Writing small components in isolation makes them easier to unit test, too. Though we won’t cover unit testing in this book (learning how to use React is hard enough by itself!), when you’re ready, look into Jest and Enzyme for testing your components.

You'll likely combine the top-down and bottom-up approaches as you build larger apps.

For example, if we were building the whole Twitter site, we might build the Tweet component top-down, then incorporate it into a list of tweets, then embed that list in a page, then embed that page into the larger application. The Tweet could be built top-down while the larger application is built bottom-up.

Rewriting an Existing App

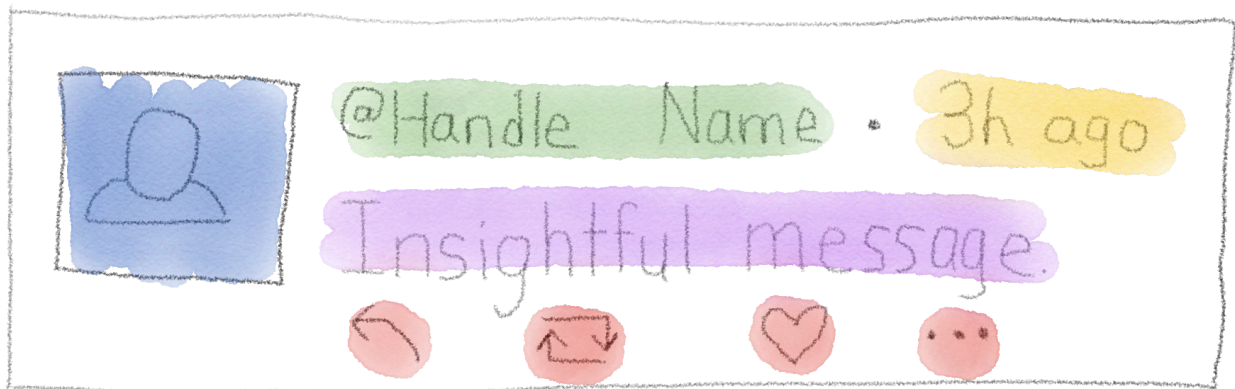
Another situation where building from the bottom is preferable is when you are converting an app to React. If you have an existing app written in another framework like AngularJS or Backbone and you want to rewrite in it React, starting at the top makes little sense, because it'll have a ripple effect across your entire code base. You'd have to finish *all* of it before anything would work.

Starting at the bottom is manageable and controlled. You can build the “leaf nodes” of your app – the small, isolated pieces. Get those working, then build the next level up, and so on, until you reach the top. At that point you have the option to replace your current framework with React if you choose to.

The other advantage of bottom-up development in a rewrite is that it fits nicely with React's one-way data flow paradigm. Since the React components occupy the bottom of the tree, and you're guaranteed that React components only contain other React components, it's easier to reason about how to pass your data to the components that need it.

Build the Tweet Component

Here's our blueprint again:



We'll be building a plain static tweet in this section, starting with the top-level component, Tweet.

Create a new project with Create React App by running this command:

```
$ create-react-app static-tweet && cd static-tweet
```

Similar to before, we'll delete some of the generated files and create our own empty `index.js`. Since we'll need some style too, we'll also create an empty `index.css`.

```
$ rm src/*  
$ touch src/index.js src/index.css
```

The newly-generated project comes with an `index.html` file in the `public` directory. Add Font Awesome by putting this line inside the `<head>` tag (put it all on one line):

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/  
font-awesome/4.6.3/css/font-awesome.min.css">
```

Then open up our blank `src/index.css` file and replace its contents with this:

```
.tweet {  
  border: 1px solid #ccc;  
  width: 564px;  
  min-height: 68px;  
  padding: 10px;  
  display: flex;  
  font-family: "Helvetica", arial, sans-serif;  
  font-size: 14px;  
  line-height: 18px;  
}
```

The `index.js` file will be very similar to the one from Hello World. It's basically the same thing, with "Tweet" instead of "Hello World". We'll make it better soon, I promise. Replace the contents of `src/index.js` with this (don't forget to type it out! Repetition is your friend, here):

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

function Tweet() {
  return (
    <div className="tweet">
      Tweet
    </div>
  );
}

ReactDOM.render(<Tweet/>,
  document.querySelector('#root'));
```

That should do it. Start up the server, same as before, by opening up a command line terminal and running:

```
$ npm start
```

And the page should render something like this:

A screenshot of a web browser window. Inside the window, the word "Tweet" is displayed in a simple, black, sans-serif font. The text is centered within a light gray rectangular border that serves as a container for the component.

This is nothing you haven't seen before. It's a simple component, with the addition of a special `className` attribute (which React calls a "prop", short for property).

We'll learn more about props in the next section, but for now, just think of them like HTML attributes. Most of them are named identically to the attributes you already know, but `className` is special in that its value becomes the `class` attribute on the DOM node.

One other new thing you might've noticed is the `import './index.css'` which is importing... a CSS file into a JavaScript file? Weird?

What's happening is that behind the scenes, when Webpack builds our app, it sees this CSS import and learns that `index.js` depends on `index.css`. Webpack reads the CSS file and includes it in the bundled JavaScript (as a string) to be sent to the browser.

You can actually see this, too – open the browser console, look at the Elements tab, and notice under `<head>` there's a `<style>` tag that we didn't put there. It contains the contents of `index.css`.

Back to our outline. Let's build the Avatar component.

Later on we'll look at extracting components into files and importing them via `import`, but to keep things simple for now, we'll just put all the components in `index.js`. Add the Avatar component to `index.js`:

```
function Avatar() {  
  return (  
      
    );  
}
```

If you want to use your own Gravatar, go to davedceddia.com/gravatar to figure out its URL.

Next we need to include Avatar in the Tweet component:

```
function Tweet() {  
  return (  
    <div className="tweet">  
      <Avatar/>  
      Tweet  
    </div>  
  );  
}
```

Now just give Avatar some style, in `index.css`:

```
.avatar {  
  width: 48px;  
  height: 48px;  
  border-radius: 5px;  
  margin-right: 10px;  
}
```

It's getting better:



Next we'll create two more components, the Message and Author:

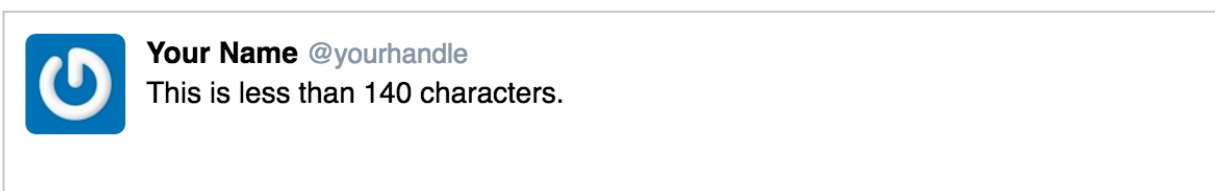
```
function Message() {  
  return (  
    <div className="message">  
      This is less than 140 characters.  
    </div>  
  );  
}  
  
function Author() {  
  return (  
    <span className="author">  
      <span className="name">Your Name</span>  
      <span className="handle">@yourhandle</span>  
    </span>  
  );  
}
```

If you refresh after adding these, nothing will have changed because we still need to update Tweet to use these new components, so do that next:

```
function Tweet() {  
  return (  
    <div className="tweet">  
      <Avatar/>  
      <div className="content">  
        <Author/>  
        <Message/>  
      </div>  
    </div>  
  );  
}
```

It's rendering now, but it's ugly. Spruce it up with some CSS for the name and handle:

```
.name {  
  font-weight: bold;  
  margin-bottom: 0.5em;  
  margin-right: 0.3em;  
}  
  
.handle {  
  color: #8899a6;  
  font-size: 13px;  
}
```



It's looking more like a real tweet now!

Next up, add the Time and the buttons (we'll talk about the new syntax in a second – just type these in as shown):

```
const Time = () => (  
  <span className="time">3h ago</span>  
)  
;  
  
const ReplyButton = () => (  
  <i className="fa fa-reply reply-button"/>  
)  
;  
  
const RetweetButton = () => (  
  <i className="fa fa-retweet retweet-button"/>  
)  
;  
  
const LikeButton = () => (  
  <i className="fa fa-heart like-button"/>  
)  
;  
  
const MoreOptionsButton = () => (  
  <i className="fa fa-ellipsis-h more-options-button"/>  
)  
;
```

These components don't look like the functions you've been writing up to this point, but they are in fact still functions. They're *arrow functions*. Here's a progression from a regular function to an arrow function so you can see what's happening:

```
// Here's a normal function component:  
function LikeButton() {  
  return (  
    <i className="fa fa-heart like-button"/>  
  );  
}  
  
// It can be rewritten as an  
// anonymous function and stored in a  
// (constant) variable:
```

```

const LikeButton = function() {
  return (
    <i className="fa fa-heart like-button"/>
  );
}

// The anonymous function can be
// turned into an arrow function:

const LikeButton = () => {
  return (
    <i className="fa fa-heart like-button"/>
  );
}

// It can be simplified by removing
// the braces and the `return`:

const LikeButton = () => (
  <i className="fa fa-heart like-button"/>
);

// And if it's really simple,
// you can even write it on one line:

const Hi = () => <span>Hi</span>;

```

Arrow functions are a nice concise way of writing components.

Notice how there's no return in the last couple versions: when an arrow function only contains one expression, it can be written without braces. And *when it's written without braces*, the single expression is implicitly returned.

We'll continue to use arrow functions throughout the book, so you'll get lots of practice. Don't worry if they look foreign now. Your eyes will adapt after writing them a few times. You'll get used to them, I promise.

Also: arrow functions don't "replace" regular functions, and aren't strictly "better than" functions. They're just different. For function components, they're effectively interchangeable. Personally, I tend to write function when the component is a bit larger, and use a `const () => (...)` when it's only a couple lines. Some people prefer to write arrow functions everywhere. Use what you like.

The `let` and `const` Keywords

If you're familiar with languages like C or Java, you're familiar with block scoping. As you may know, JavaScript's `var` is actually *function-scoped*, not block-scoped. This has long been an annoyance (especially in `for` loops). But no more!

ES6 added `let` and `const` as two new ways of declaring block-scoped variables.

The `let` keyword defines a mutable (changeable) variable. You can use it instead of `var` almost everywhere.

The `const` defines a constant. It will throw an error if you try to reassign the variable, but it's worth noting that it does not prevent you from modifying the data *within* that variable. Here's an example:

```
const answer = 42;
answer = 43;    // error!

const numbers = [1, 2, 3];
numbers[0] = 'this is fine'; // no error
```

Using `const` is more of a signal of intent than a bulletproof protection scheme, but it is still worthwhile.

Add the Buttons and the Time

Now that we've got all those new components, update `Tweet` again to incorporate them:

```
function Tweet() {
  return (
    <div className="tweet">
      <Avatar/>
```



```
    <div className="content">
      <Author/><Time/>
      <Message/>
      <div className="buttons">
        <ReplyButton/>
        <RetweetButton/>
        <LikeButton/>
        <MoreOptionsButton/>
      </div>
    </div>
  </div>
);
}
```

Finally, add a few more styles to cover the time and buttons:

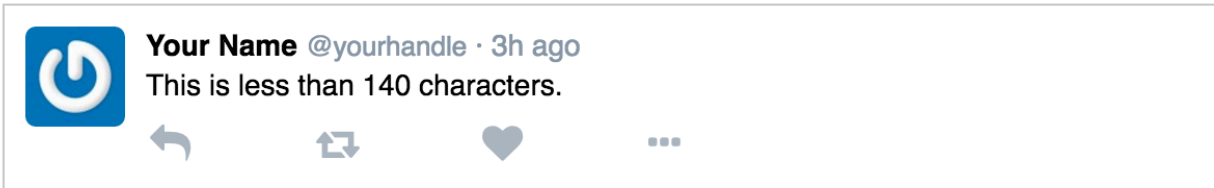
```
.time {
  padding-left: 0.3em;
  color: #8899a6;
}

.time::before {
  content: "\00b7";
  padding-right: 0.3em;
}

.buttons {
  margin-top: 10px;
  margin-left: 2px;
  font-size: 1.4em;
  color: #aab8c2;
}

.buttons i {
  width: 80px;
}
```

And here we have a fairly respectable-looking tweet!



You can customize it to your heart's content: change the name, the handle, the message, even the Gravatar icon. Pretty sweet, right?

“But... wait,” I hear you saying. “I thought we were going to build *reusable* components!” This tweet is pretty and all, but it's only good for showing *one* message from *one* person...

Well don't worry, because that's what we're going to do next: learn to parameterize components with props.

6 Props

Where HTML elements have “attributes,” React components have “props” (short for “properties”). It’s a different name for essentially the same thing.

Think about how you would customize a plain function. This might seem a bit basic but bear with me. Let’s say you have this function:

```
function greet() {  
  return "Hi Dave";  
}
```

It works great, but it’s pretty limited since it will always return “Hi Dave”. What if you want to greet someone else? You’d pass in their name as an argument:

```
function greet(name) {  
  return "Hi " + name;  
}
```

Where functions have arguments, components have *props*. Props let you pass data to your components.

Passing Props

Here is a bit of JSX passing a prop called name with a string value of "Dave" into the Person component:

```
<Person name='Dave' />
```

Here’s another example, passing a className prop with the value "person":

```
<div className='person' />
```

JSX uses `className` instead of `class` to specify CSS classes. You will forget this over and over. You will type “class” out of habit, and React will warn you about it. That’s fine. Just change it to `className`.

Notice how the `div` is self-closing? This ability isn’t just for `<input/>` anymore: in JSX, *every* component can be self-closing. In fact, if the component has no children (no contents), the convention is to write it like this, instead of using a closing tag like `<div></div>`.

In this next component, we’re passing a string for `className`, a number for the `age` prop, and an actual JavaScript expression as the `name`:

```
function Dave() {
  const firstName = "Dave";
  const lastName = "Ceddia";

  return (
    <Person
      className='person'
      age={33}
      name={firstName + ' ' + lastName} />
  );
}
```

Remember that in JSX, singles braces must surround JavaScript expressions. The code in the braces is real JavaScript, and it follows all the same scoping rules as normal JavaScript.

It’s important to understand that the JS inside the braces must be an *expression*, not a statement. Here are a few things you can do inside JSX expressions:

- Math, concatenation: `{7 + 5}` or `{'Your' + 'Name'}`
- Function calls: `{this.getFullName(person)}`
- Ternary (?) operator: `{name === 'Dave' ? 'me' : 'not me'}`
- Boolean expressions: `{isEnabled && 'enabled'}`

Here are some things you cannot do:

- Define new variables with `let`, `const`, and `var`
- Use `if`, `for`, `while`, etc.
- Define functions with `function`

Remember that JSX evaluates to JavaScript, which means the props become keys and values in an object. Here's that example from above, transformed into JavaScript:

```
function Dave() {
  const firstName = "Dave";
  const lastName = "Ceddia";

  return React.createElement(Person, {
    age: 33,
    name: firstName + ' ' + lastName,
    className: 'person'
  }, null);
}
```

All of the rules that apply to function arguments apply to JSX expressions. Could you call a function like this?

```
myFunc(const x = true; x && 'is true');
```

Of course not! That looks completely wrong. If you tried to pass that argument to a JSX expression, this is what you'd get:

```
<Broken value={const x = true; x && 'is true'}/>
// gets compiled to:
React.createElement(Broken, {
  value: const x = true; x && 'is true'
}, null);
```

So when you're trying to decide what to put in a JSX expression, ask yourself, "Could I pass this as a function argument?"

Receiving Props

Props are passed as an object, and are the first argument to a component function, like this:

```
function Hello(props) {
  return (
    <span>Hello, {props.name}</span>
  );
}

// Used like:
<Hello name="Dave"/>
```

It works the same way for arrow functions:

```
const Hello = (props) => (
  <span>Hello, {props.name}</span>
);
```

ES6 has a new syntax called *destructuring* which makes props easier to work with. It looks like this:

```
const Hello = ({ name }) => (
  <span>Hello, {name}</span>
);
```

You can read `{ name }` as “extract the ‘name’ key from the object passed as the first argument, and put it in a variable called name”. You can extract multiple keys at the same time, too:

```
const Hello = ({ firstName, lastName }) => (
  <span>Hello, {firstName} {lastName}</span>
);
```

In practice, props are very often written using this destructuring syntax. Just so you know, it works outside of function arguments as well. You can destructure props this way, for instance:

```
const Hello = (props) => {  
  const { name } = props;  
  return (  
    <span>Hello, {name}</span>  
  );  
}
```

Remember, arrow functions need a `return` if the body is surrounded by braces, and it needs braces if the body contains multiple lines.

Modifying Props

One important thing to know is that props are *read-only*. Components that receive props must not change them.

If you come from a framework that has two-way binding this is a change.

In React, data flows *one way*. Props are read-only, and can only be passed *down* to children.

Communicating With Parent Components

If you can't change props, but you need to communicate something up to a parent component, how does that work?

When a child needs to send data to its parent, the parent can send down a *function* as a prop, like this:

```
function handleAction(event) {  
  console.log('Child did:', event);  
}  
  
function Parent() {  
  return (  
    <Child onAction={handleAction}/>  
  );  
}
```

The Child component receives the `onAction` prop, which it can call whenever it needs to send up data or notify the parent that something happened.

One place where it's common to pass functions as props is for handling events. For instance, the standard button element accepts an `onClick` prop, which it'll call when the button is clicked.

```
function Child({ onAction }) {  
  return (  
    <button onClick={onAction}/>  
  );  
}
```

We'll learn more about event handling later on.

7 Example: Tweet With Props

Now that you have a basic understanding of props, let's see how they work in practice.

We'll take the static Tweet example from Chapter 5 and rework it to display dynamic data by using props.

For this example, make a copy of the static-tweet project folder so that we can work without fear of breaking the old code. We'll also start up a development server (stop the old one if it's still running).

```
$ cp -a static-tweet props-tweet && cd props-tweet
$ npm start
```

Note: Don't use `cp -r` since it does not preserve symlinks, and can break `npm start`.

If, after copying the project, the `npm start` command fails, delete the `node_modules` folder and run `npm install`. Then try `npm start` again.

Open up `src/index.js`. To begin with, update the Tweet component to accept a `tweet` prop as shown below. Then add the `testTweet` object, which will serve as our fake data, and update the call to `ReactDOM.render` to pass the `testTweet` object as the `tweet` prop.

Refresh the page after making these changes and make sure everything looks the same as before (nothing should be different yet).

```
// add the { tweet } prop, destructured
function Tweet({ tweet }) {
  return (
    <div className="tweet">
      <Avatar/>
      <div className="content">
        <Author/><Time/>
        <Message/>
        <div className="buttons">
          <ReplyButton/>
          <RetweetButton/>
          <LikeButton/>
        </div>
      </div>
    </div>
  )
}
```

```

        <MoreOptionsButton/>
      </div>
    </div>
  </div>
);
}

// ...

const testTweet = {
  message: "Something about cats.",
  gravatar: "xyz",
  author: {
    handle: "catperson",
    name: "IAMA Cat Person"
  },
  likes: 2,
  retweets: 0,
  timestamp: "2016-07-30 21:24:37"
};

ReactDOM.render(<Tweet tweet={testTweet}/>,
  document.querySelector('#root'));

```

Avatar

Let's start converting the static components to accept props, starting with Avatar. In the render method of Tweet, replace this line:

```
<Avatar/>
```

With this line:

```
<Avatar hash={tweet.gravatar}/>
```

This passes the tweet's gravatar property into the hash prop. Now update Avatar to use this new prop:

```
function Avatar({ hash }) {  
  const url = `https://www.gravatar.com/avatar/${hash}`;  
  return (  
    <img  
      src={url}  
      className="avatar"  
      alt="avatar" />  
  );  
}
```

The Gravatar hash, passed in as hash using ES6 destructuring, is incorporated into the URL and passed to the image tag as before.

ES6: Template Strings

Here's a little more ES6 for you: the backticks around the URL string are a new syntax for *template strings*. The `${hash}` part will be replaced with the hash itself. This new syntax is a bit cleaner than concatenating strings like `"https://www.gravatar.com/avatar/" + hash`. It's especially nice when you need to concatenate multiple strings together. Consider:

```
// Using a template string:  
person = `${firstName} ${lastName}`;  
// versus concatenation:  
person = firstName + " " + lastName;  
  
// Or, building a url:  
url = `foo.com/users/${userId}/items/${itemId}`;  
// versus:  
url = "foo.com/users/" + userId + "/items/" + itemId;
```

The avatar should render the same as before.

You can replace the Gravatar hash with your own if you like. Visit <https://daveceddia.com/gravatar>, type in your email, and copy the hash. (If you don't have a Gravatar account, you can create one at <https://gravatar.com>).

Message

Now we'll do Message. Replace this line in the render method of Tweet:

```
<Message/>
```

With this line:

```
<Message text={tweet.message}/>
```

We're extracting the message from the tweet and passing it along to the Message component as text. Then update the Message component to use the new prop:

```
function Message({ text }) {  
  return (  
    <div className="message">  
      {text}  
    </div>  
  );  
}
```

Instead of static text in the div, we're rendering the text prop that was passed in. Refresh now, and you'll see the message is now "Something about cats." Success!

Author and Time

This time we'll convert two components at once: Author and Time. Update Tweet to pass the relevant data by replacing this line in render:

```
<Author/><Time/>
```

With these lines:

```
<Author author={tweet.author}/>
<Time time={tweet.timestamp}/>
```

We're also going to introduce a library called Moment.js to work with dates and times. We will use it to calculate the relative time string ("3 days ago"). Run this command in the terminal to install Moment:

```
npm install moment
```

Then we need to import Moment at the top of our `index.js` file, so add this line at the top:

```
import moment from 'moment';
```

Now update the Author and Time components. You'll notice that Time uses the new moment library.

```
function Author({ author }) {
  const { name, handle } = author;
  return (
    <span className="author">
      <span className="name">{name}</span>
      <span className="handle">@{handle}</span>
    </span>
  );
}
```

```
const Time = ({ time }) => {
  const timeString = moment(time).fromNow();
  return (
    <span className="time">
      {timeString}
    </span>
  );
};
```

In the static tweet example, `Time` was written as an arrow function, so I've left it that way. However, because it now has 2 statements, it needs the surrounding braces, which then causes it to need a `return`. You could of course simplify this by moving the `moment(time).fromNow()` directly inside the ``, and then you could remove the `return` and the braces. I'll leave it up to your own sense of style.

What Exactly Should Be Passed as a Prop?

Props can accept all sorts of things: Numbers, Booleans, Strings, Objects, even Functions. Early on you might start to wonder, should you pass an object and let the component extract what it needs? Or should you pass the specific pieces of data that the component requires?

In the examples here, we're passing in the specific pieces of data.

We could instead just pass the tweet object itself. For instance, instead of passing a timestamp directly to the `Time` component, we could pass in `tweet` and let it extract the timestamp from the tweet. Why not do it that way? Here are a few reasons:

- **It hurts reusability** - If `Time` expects an object with a property called "timestamp", then it's locked in to that structure. What if you have a user with an timestamp property called "updated_at" and want to render it with a `Time` component? Well, you can't, without hacking together a temporary object that "looks like" what `Time` expects, or changing the implementation of `Time` (potentially breaking other uses of it).
- **It's harder to change** - The `Time` component would have knowledge of the inner structure of a tweet object. This might not seem like a big deal, until you have 10 components like this, and the backend developer decides that "timeStamp" with a capital "S" looks better than "timestamp" and now you have to update all those components. It's a good idea to keep the knowledge of data structures contained to as few places as possible to reduce the cost of change.

Guidelines for Naming Props

As the saying goes, “There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.”

The names you choose for props influence how the component will be used, as well as how components are coupled. Imagine if we had written `Time` to take a prop named `tweetTime`. Even if we passed in the timestamp directly (instead of the entire tweet object), this is still not a great choice for a prop name. Why?

Consider what happens when the next developer comes along (maybe that’s you, 3 weeks from now) and wants to reuse the `Time` component in a new part of the app that has nothing to do with tweets. The rendered result will come out fine, but the code will be awkward. Writing `<Time tweetTime={timestamp}/>` expresses the wrong intent if the timestamp doesn’t actually belong to a tweet.

More likely, the developer will look at `Time`, see that it expects a `tweetTime`, and then decide not to reuse that component – either because they assume it’s not suitable for their purpose, or they worry that its underlying implementation might change and break their code.

The Remaining Components

Let’s get back to converting `Tweet` and its children to use props. There are only two components left: `RetweetButton` and `LikeButton`, which need to display counts of retweets and likes. Currently, they don’t display any numbers at all, just an icon.

Change these two lines:

```
<RetweetButton/>
<LikeButton/>
```

To these lines (passing in the counts):

```
<RetweetButton count={tweet.retweets}/>
<LikeButton count={tweet.likes}/>
```

Then, update the RetweetButton and LikeButton to accept the new count prop and render their respective numbers:

```
function getRetweetCount(count) {
  if(count > 0) {
    return (
      <span className="retweet-count">
        {count}
      </span>
    );
  } else {
    return null;
  }
}

const RetweetButton = ({ count }) => (
  <span className="retweet-button">
    <i className="fa fa-retweet"/>
    {getRetweetCount(count)}
  </span>
);
```

```
const LikeButton = ({ count }) => (
  <span className="like-button">
    <i className="fa fa-heart"/>
    {count > 0 &&
      <span className="like-count">
        {count}
      </span>
    }
  </span>
);
```

Finally, the styling needs an update too. Take these lines out:

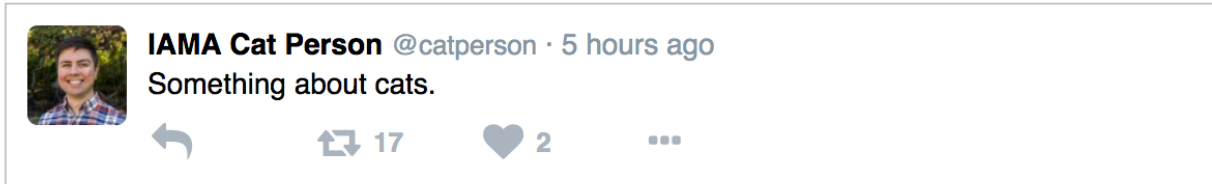
```
.buttons i {
  width: 80px;
}
```


Then replace them with this:

```
.reply-button, .retweet-button,
.like-button, .more-options-button {
  width: 80px;
  display: inline-block;
}

.like-count, .retweet-count {
  position: relative;
  bottom: 2px;
  font-size: 13px;
  margin-left: 6px;
  font-weight: bold;
}
```

There we go!



Let's go over the code. Despite looking different, these components have the same logic for determining how to show the count.

RetweetButton

In `RetweetButton`, the computation is extracted into a separate function called `getRetweetCount`. It either returns a `` element or `null`.

Notice that we added a wrapper span element around the icon and the count. Remember that React components can only return a single element (try removing the span and see what happens). We could've used a `Fragment` here but it's useful to have a wrapper for styling purposes.

Also note that when an expression evaluates to `null` or `false` inside single braces within JSX, nothing is rendered at all. To prove it, open up the Dev Tools Element Inspector in Chrome and inspect the Retweet button when it has 0 retweets: there should be no “like-count” span in sight.

It’s worth noting that the `getRetweetCount` function could be written as a component instead – it almost is one already. Here’s what that might look like:

```
function Count({ count }) {
  if(count > 0) {
    return (
      <span className="retweet-count">
        {count}
      </span>
    );
  } else {
    return null;
  }
}

const RetweetButton = ({ count }) => (
  <span className="retweet-button">
    <i className="fa fa-retweet"/>
    <Count count={count} />
  </span>
);
```

The function had to be renamed to start with a capital letter (remember that component names must start with a capital letter), and its argument needed to be replaced with destructuring to extract `count`. Otherwise it looks the same. Then, using it in `RetweetButton` is just a matter of inserting some JSX.

LikeButton

For `LikeButton`, the logic is done slightly differently. Instead of extracting the logic to a function, it’s done inline with a boolean operator (the `&&`). But again, it renders a span, or null.

Another Option

Here's a third alternative, just for fun, which differs slightly in that it will always output a `` with class "like-count", but where the contents might be empty:

```
const LikeButton = ({ count }) => (  
  <span className="like-button">  
    <i className="fa fa-heart"/>  
    <span className="like-count">  
      {count ? count : null}  
    </span>  
  </span>  
>);
```

As you can see, there's more than one way to accomplish the same thing with JSX.

8 PropTypes

Documentation and Debugging In One

We've seen what "props" are, and how they're passed into React components – but what happens if you forget to pass one of the props?

Well, it ends up being undefined, just as if you'd forgotten to pass an argument to a plain old function. This can be totally fine, or a code-breaking disaster (just as if you'd forgotten to pass an argument to a plain old function).

If you want to avoid this, there are two main options: write your app in TypeScript, or stick with JS and be diligent about using PropTypes.

TypeScript is outside the scope of this book (one thing at a time, remember?), but in the last chapter I'll suggest a few resources to help you explore it if you're interested.

In this chapter, we'll look at how to write PropTypes so that you can catch missing or incorrect props quickly.

How to Write PropTypes

When you create a component, you can declare that certain props are optional or required, *and* you can declare what type of value that prop expects. Here's an example:

```
import PropTypes from 'prop-types';

function Comment({ author, message, likes }) {
  return (
    <div>
      <div className='author'>{author}</div>
      <div className='message'>{message}</div>
      <div className='likes'>
        {likes > 0 ? likes : 'No'} likes
      </div>
    </div>
  );
}
```

```
Comment.propTypes = {  
  message: PropTypes.string.isRequired,  
  author: PropTypes.string.isRequired,  
  likes: PropTypes.number  
}
```

First, notice that `PropTypes` must be explicitly imported from the `'prop-types'` package.

Prior to React 15.5, `prop-types` was bundled together with the `react` package, but today you'll probably need to install this package separately (and restart the Create React App dev server after you do):

```
$ npm install prop-types
```

Next, notice that `propTypes` are set as a property on the component itself. This pattern is the same whether the component is a regular function, an arrow function, or even an ES6 class (which we'll see later).

Finally, notice that the `propTypes` attribute starts with a lowercase "p" while the imported `PropTypes` starts with a capital "P".

With this set of `propTypes`, `message` and `author` are required, and must be strings. The `likes` prop is optional, but must be a number if it's provided. Try rendering it a few different ways, and check the browser console each time:

```
<Comment author='somebody' message='a likable message' likes={1}/>  
<Comment author='mr_unpopular' message='unlikable message'/>  
<Comment author='mr_unpopular' message='another message' likes={0}/>  
<Comment author='error_missing_message'/>  
<Comment message='mystery author'/>
```

React will warn you in the browser console if you forget a required prop:

```
<Comment author='an_error' />
```

Warning: Failed prop type: The prop message is marked as required in Comment, but its value is undefined.

Likewise, you'll get a warning if you pass the wrong type:

```
<Comment author={42} />
```

Warning: Failed prop type: Invalid prop author of type number supplied to Comment, expected string.

These warning messages are invaluable for debugging. It tells you the mistake you made, *and* gives you a hint where to look! This is a fair bit better than some other frameworks that silently fail when you forget a required attribute.

The Catch

So: what's the catch? All this nice error handling must have a catch, right?

Well, sort of. The only catch with prop types is that *you must remember to declare them*. Providing propTypes is optional, and React won't give you any warnings if you don't specify propTypes on one of your components.

You can either vow to be super-diligent about writing those propTypes, or you can have a *linter* tool check for you. I recommend the second one.

ESLint is a popular choice, and there is a React plugin for ESLint that will check for things like missing propTypes and that props are passed correctly.

How Do I Validate Thee, Let Me Count the Ways

PropTypes comes with a lot of validators built in.

First, there are validators for the standard JavaScript types:

- `PropTypes.array`
- `PropTypes.bool`
- `PropTypes.func`
- `PropTypes.number`
- `PropTypes.object`
- `PropTypes.string`

You saw the `string` and `number` validators in the `Comment` component earlier. The others work the same way – validating that an array is passed in, or a boolean, or a function, or whatever.

There are validators for `node` and `element`. A node is anything that can be rendered, meaning numbers, strings, elements, or an array of those. An element is a React element created with JSX or by calling `React.createElement`:

- `PropTypes.node`
- `PropTypes.element`

There's an `instanceOf` validator for checking that the prop is an instance of a specific class. It takes an argument:

- `PropTypes.instanceOf(SpecificClass)`

You can limit to specific values with `oneOf`:

- `PropTypes.oneOf(['person', 'place', 1234])`

You can validate that the prop is one of a few types:

```
PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.boolean
])
```

You can validate that it's an array of a certain type, or an object whose properties are values of a certain type:

- `PropTypes.arrayOf(PropTypes.string)`
 - Would match: `['a', 'b', 'c']`
 - Would not match: `['a', 'b', 42]`
- `PropTypes.objectOf(PropTypes.number)`
 - Would match: `{age: 27, birthMonth: 9}`
 - Would not match: `{age: 27, name: 'Joe'}`

You can validate that an object has a certain *shape*, meaning that it has particular properties. The object passed to this prop is allowed to have *extra* properties too, but it must at least have the ones in the shape description.

```
PropTypes.shape({  
  name: PropTypes.string,  
  age: PropTypes.number  
})
```

This will match an object of the exact shape, like this:

```
person = {  
  name: 'Joe',  
  age: 27  
}
```

It will also match an object with additional properties, like this:

```
person = {  
  name: 'Joe',  
  age: 27,  
  address: '123 Fake St',  
  validPerson: false  
}
```


Since this PropTypes shape is requiring an object that has name and age keys, if we leave one of them off, it won't pass. This would generate a warning:

```
person = {  
  age: 27  
}
```

Similarly, if we pass the wrong type for one of those keys, we'll get a warning:

```
person = {  
  name: false, // boolean instead of string  
  age: 27  
}
```

Required Props

By default, a propTypes validation is *optional*. Optional props will not warn if they're missing, but will warn if the wrong type is passed in.

Any PropTypes validation can be made required by adding `.isRequired` to the end of it. Here are a few examples:

```
PropTypes.bool.isRequired
```

```
PropTypes.oneOf(['person', 'place', 1234]).isRequired
```

```
PropTypes.shape({  
  name: PropTypes.string,  
  age: PropTypes.number  
}).isRequired
```

Remember, all PropTypes are *optional* by default, which means you won't get console warnings if you forget to pass a value, or make a typo in a prop name. In most cases, you'll want to append `.isRequired` to your PropTypes validations.

Custom Validators

If the built-in PropType validators aren't expressive enough, you can write your own! The function should take the props, propName, and componentName, and return an Error if validation fails. Note that's *return* an Error, not *throw* an Error.

A custom validator to check that the passed prop is exactly length 3 (either a string or an array) would look like this:

```
function customValidator(props, propName, componentName) {
  // here, propName === "myCustomProp"
  if (props[propName].length !== 3) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + `'. Length is not 3.'
    );
  }
}

const CustomTest = ({ myCustomProp }) => (
  <span>{myCustomProp}</span>
);
CustomTest.propTypes = {
  myCustomProp: customValidator
}

// This will produce a warning:
ReactDOM.render(
  <CustomTest myCustomProp='not_three_letters' />,
  document.querySelector('#root')
);

// This will pass:
ReactDOM.render(
  <CustomTest myCustomProp={[1, 2, 3]} />,
  document.querySelector('#root')
);

// This will also pass:
ReactDOM.render(
```

```
<CustomTest myCustomProp="abc"/>,  
document.querySelector('#root')  
);
```

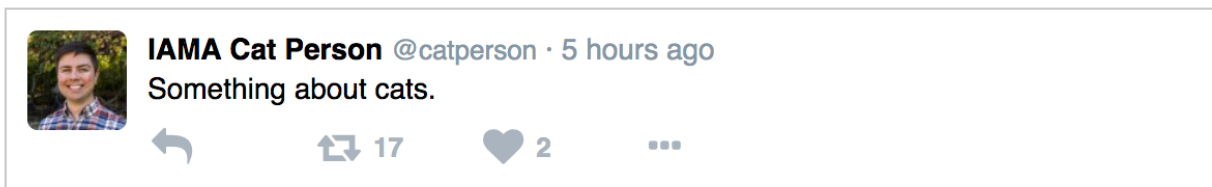
Example: Tweet with PropTypes

One last time, let's go back to the Tweet example to see how to apply PropTypes to a real set of components.

Recall that the structure of the app looks like this:

- Tweet
 - Avatar
 - Author
 - Time
 - Message
 - ReplyButton
 - RetweetButton
 - LikeButton
 - MoreOptionsButton

And here's what it should look like:



Starting from the bottom up, let's add PropTypes to these components. Make a copy of the props-tweet code from earlier so we can modify it without fear:

```
$ cp -a props-tweet proptypes-tweet && cd proptypes-tweet
$ npm start
```

The buttons are a good place to start. MoreOptionsButton doesn't take props at all, so we don't need to change it.

LikeButton can optionally take a count prop, so we'll add a PropTypes for that:

```
const LikeButton = ({ count }) => (
  <span className="like-button">
```

```

    <i className="fa fa-heart"/>
    {count > 0 &&
      <span className="like-count">
        {count}
      </span>}
    </span>
  );

LikeButton.propTypes = {
  count: PropTypes.number
};

```

To verify that it works, go to the Tweet component, and try passing a string instead of a number to the LikeButton:

```

// Replace this:
<LikeButton count={tweet.likes}/>

// With this:
<LikeButton count='foo'/>

```

If you refresh now (or wait for the browser to refresh automatically), you should see a warning in the browser console like this:

Warning: Failed prop type: Invalid prop 'count' of type 'string' supplied to 'LikeButton', expected 'number'. Check the render method of 'Tweet'.

Good! Everything is working correctly.

Change that line back to what it was before, and now add a count PropType to the RetweetButton component in the same way. I'll let you do this one on your own. Go ahead and do that now.

Got it?

I bet you can figure out the PropTypes for Message and Time too (they both take optional strings). Go ahead and do those yourself too.

Author is a little more interesting: it takes an author object that should have name and handle properties. Any idea how you'd implement that using one of the validators that comes with React?

If you said `PropTypes.shape`, you would be correct! I'll also accept `PropTypes.object`, even though that's a bit less explicit. Let's see how the shape validator would look in the component:

```
function Author({ author }) {
  const { name, handle } = author;

  return (
    <span className="author">
      <span className="name">{name}</span>
      <span className="handle">@{handle}</span>
    </span>
  );
}

Author.propTypes = {
  author: PropTypes.shape({
    name: PropTypes.string.isRequired,
    handle: PropTypes.string.isRequired
  }).isRequired
};
```

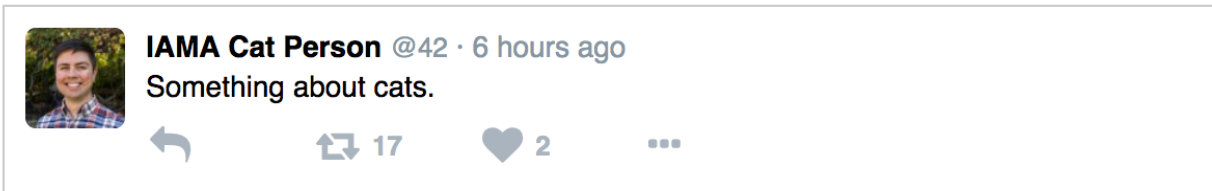
Here, the author prop is required, and it must have a name and handle as strings.

Let's try an experiment: set the tweet's `author.name` to a number like 42 instead of a string. Then check the console.

React complains, as expected:

Warning: Failed prop type: Invalid prop 'author.handle' of type 'number' supplied to 'Author', expected 'string'. Check the render method of 'Tweet'.

Take a look at the component though:



React rendered the number even though it didn't pass props validation!

Here is an important thing to keep in mind: `PropTypes` are a great debugging tool, but a failed validation won't stop your code from running. Keep an eye on that console window.

Two components left: `Avatar` and `Tweet`. They're all yours!

`Avatar` is easy, it just takes a string.

Since `Tweet` takes an object, you could validate it with `PropTypes.object`, or use a more complex shape. It's up to you! (I suggest trying to figure out the shape though!).

How Explicit Should You Be?

When the object passed to a component is simple, it's easy to justify writing out a shape `PropType`. When the object is more complex, though, you might start to wonder if it's worth the effort. The answer will depend on your particular case. If you go overboard with shape it could be a pain to maintain later.

It's also good to follow the DRY (Don't Repeat Yourself) principle. If you have an explicit object shape required in one place, for instance in `Author`, there's little value in duplicating the shape in the parent `Tweet` component. If the shape of `author` changes some day, there will be *two* places to update code. Having that second check doesn't buy you anything, and instead, could actually cost you time in the future.

`PropTypes` as Documentation

Here's one last benefit of `PropTypes`: in addition to helping out with debugging, they serve as nice documentation. When you come back to a component a few days, a week, or a month later, the `PropTypes` will serve as a "README" of sorts.

Rather than having to scan through the code, you'll be able to instantly figure out which props are required, which are optional, and what their types need to be.

Exercises

Who says you have to model components after web interfaces? Real-world objects provide good practice too. You'll build some of a few of those below.

The aim of these exercises is to give you some practice creating components that use props. Getting the CSS to look right is a bonus, so don't struggle with it – if styling is not your strong suit, just get the React component working and move on. No shame.

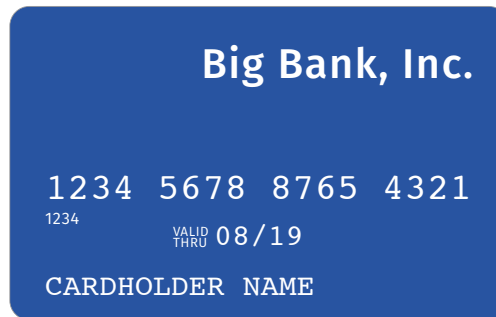
1. Create an `AddressLabel` component that takes a `person` object as a prop and renders their name and address like so:

```
Full Name  
123 Fake St.  
Boston, MA 02118
```

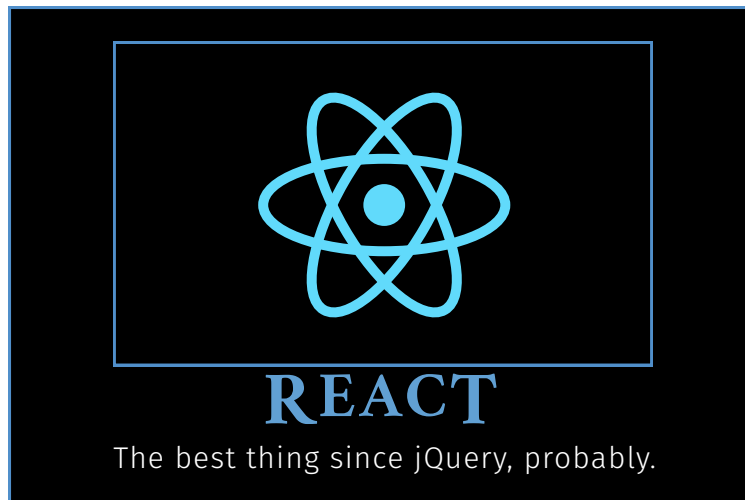
2. Create an `Envelope` component that takes `toPerson` and `fromPerson` as props and uses your `AddressLabel` from Exercise 1 to display the return address and the recipient address. Make sure to include a `Stamp` too!



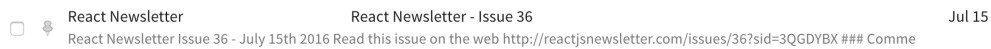
3. Create a `CreditCard` component based on this design. Style it up with CSS or inline styles. Accept a `cardInfo` prop that contains the person's name, expiration date, credit card number, and bank name.



4. Create a Poster component that takes image, title, and text as props. Render something like the image below. Google “demotivational posters” for inspiration.



5. Create a single-line email, as would appear in an inbox. Reference the screenshot below. It should accept an email prop, which contains the sender, subject, date, and message.



9 Children

We've seen how JSX supports nesting components, just like HTML. And we've seen that custom components can accept *props* as arguments, and use those props to render content or pass along to child components.

There's a special prop we haven't talked about yet: it's called `children`.

Let's say you wanted to make a reusable `IconButton` component that looked something like this:



You might imagine using it like this:

```
<IconButton>Do The Thing</IconButton>
```

What happens to that inner text, "Do The Thing"? Well, that's where the `children` prop comes in. When React renders `IconButton`, it will pass all of its sub-elements (in this case, the text "Do The Thing") into `IconButton` as a prop called `children`.

The `children` are automatically passed in, but they're not automatically displayed anywhere. You have to explicitly place the `children` somewhere in your component. If you don't, they'll be ignored. For instance, if the `IconButton` component looked like this:

```
function IconButton() {  
  return (  
    <button>  
      <i className="target-icon"/>  
    </button>  
  );  
}
```

The rendered component would be a button with an icon, and *no text*. It would look like this:



With a small tweak – accepting the `children` prop and then inserting it after the icon – the passed-in children will be rendered where we want them:

```
function IconButton({ children }) {
  return (
    <button>
      <i className="target-icon"/>
      {children}
    </button>
  );
}
```

Notice how `children` is available as a prop in `IconButton` even though we didn't explicitly pass a prop named `children`. That's because React automatically passes in the content of a component as the `children` prop. You could achieve the same result by using `IconButton` by explicitly passing the `children` prop, instead:

```
<IconButton children={"Do The Thing"}></IconButton>
```

It looks a bit unnatural, but it works.

The convention is to put children *inside* the tag, and not pass them explicitly as a prop, but I wanted to show you that children can be used like any other prop.

By the way, you can also pass JSX into a prop. If we wanted to make the button text italic, we could write it either of these ways:

```
<IconButton children={<em>Just Do It</em>} />
<IconButton>
  <em>Just Do It</em>
</IconButton>
```

You can also pass JSX into *any* prop, not just the one named “children”.

As an example, you might write a `Confirmation` component that displays a dialog with OK and Cancel buttons. If you wanted to be able to customize the title and message of the dialog, you could write it to accept title and message as separate props and then use it like this:

```
<Confirmation
  title={<h1>Are you sure?</h1>}
  message={<strong>Really really sure?</strong>}
/>
```

Different Types of Children

The `children` prop is always pluralized as `children` no matter whether there’s a single child or multiple children. This means that `children` can be a single element *or* an array, depending on what was passed in.

When there are multiple children, `children` will be an *array* of `ReactElements`.

When there is only one child, it is a *ReactElement object*.

This might seem a little weird: wouldn’t it be easier to work with if `children` was always an array?

Well, yes, it would. But `children` is used so often and the single-child use case is so common that the React team decided to optimize it by not allocating an array when there’s only one child.

Dealing with the Children

React provides utility functions for dealing with this opaque data structure.

- `React.Children.map(children, function)`
- `React.Children.forEach(children, function)`
- `React.Children.count(children)`
- `React.Children.only(children)`
- `React.Children.toArray(children)`

The first two, `map` and `forEach`, work the same as the methods on JavaScript's built-in `Array`. They accept `children`, whether it's a single element or an array, and a function that'll be called for each element. `forEach` iterates over the `children` and returns nothing, whereas `map` returns an array made up of the values you return from the function you provide.

`count` is pretty self-explanatory: it returns the number of items in `children`.

`toArray` is similarly intuitive: it converts `children` into a flat array, whether it was an array or not.

`only` returns the single child, or throws an error if there is more than one child.

You have access to every child element individually, so you can reorder them, remove some, insert new ones, pass the `children` down to further children, and so on.

PropTypes for Children

If you want your component to accept zero, one, or more children, use the `.node` validator:

```
propTypes: {  
  children: PropTypes.node  
}
```

If you want it to accept only a single child, use the `element` validator:

```
propTypes: {  
  children: PropTypes.element  
}
```

Beware that this expects a single *React Element* as a child. This means it has to be a custom component, or a tag like `<div>`. `PropTypes.element` will warn if you pass a string or number.

If you need to allow an element *or* a string, you can use the node validator (which will accept elements, strings, and more) or be more explicit with a `oneOfType` validator like this:

```
propTypes: {  
  children: PropTypes.oneOfType([  
    PropTypes.element,  
    PropTypes.string  
  ])  
}
```

As with any other `propTypes`, they are optional unless you append `.isRequired`.

Customizing Children Before Rendering

In the example above, we only inserted some text. What if we wanted to do something more expressive, like creating our own custom component hierarchy?

Imagine that we constructed our own “API” of sorts for expressing a navigation header:

```
<Nav>  
  <NavItem url='/'>Home</NavItem>  
  <NavItem url='/about'>About</NavItem>  
  <NavItem url='/contact'>Contact</NavItem>  
</Nav>
```

Using the children prop, the Nav component can do things like insert a separator between each item:

```
function Nav({ children }) {
  let items = React.Children.toArray(children);
  for(let i = items.length - 1; i >= 1; i--) {
    items.splice(i, 0,
      <span key={i} className='separator'>|</span>
    );
  }

  return (
    <div>{items}</div>
  );
}
```

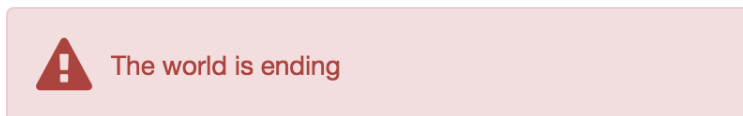
The code converts children into an array, then walks backward from the end as it inserts a new element between every existing element.

You will implement `NavItem` in the exercises coming up. It could render a simple link, or an icon next to a link, or anything you'd like.

Exercises

Now that you’ve seen how the `children` prop works, here are some exercises to improve your understanding.

1. Make a component to display an “error box” that looks like this:

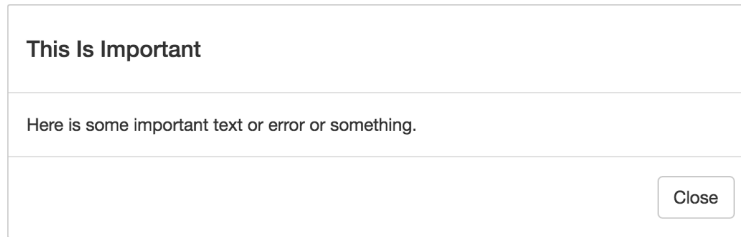


Invoking the component should look like this:

```
<ErrorBox>
  Something has gone wrong
</ErrorBox>
```

Use the `children` prop to place the text properly. The image above uses [Bootstrap](#) for styling and [Font Awesome](#) for the icon. You can add these libraries to your `public/index.html` file for the styling icon if you like, or just make a plain-looking box. It’s more important to get practice with the `children` prop than to get the style perfect.

2. Practice using the `React.Children.toArray` function with these next few exercises. You can put them all in a single app.
 - a) Write a component called `FirstChildOnly` that accepts any number of children but only renders the first.
 - b) Write a component called `LastChildOnly` that only renders its last child.
 - c) Create a component named `Head` that takes a `number` prop, and renders the first `[number]` children. e.g. If you pass `number=3`, and 7 child elements, it will render the first 3.
 - d) Create a component named `Tail` that takes a `number` and renders the last `N` children.
3. Create a `Dialog` component which accepts as children `Title`, `Body`, and `Footer` components, all optional. `Dialog` should verify that all of its children are one of these types, and should output something that looks like this:



Feel free to use Bootstrap's modal styles, or create your own.

10 Example: GitHub File List

You have a few components under your belt now. You've seen props and propTypes, and written some JSX.

Before we move into state and interactivity, I want to go through an example that incorporates everything we've seen so far. There'll be a few new techniques covered here too.

So with that in mind, let's create a new mini-app that replicates GitHub's file list.

Just as we did with the Tweet example, we'll follow a 4-step process:

1. Start with a sketch/mockup/screenshot
2. Break it into components
3. Name the components
4. Build it!

Here's a screenshot of the GitHub UI which we'll be modeling our app after:

📁 build	Close #1687, Replace es3ify with Babel ES3 transforms (#1688)	2 months ago
📁 docs	Mention that we're Observable in the API.	9 hours ago
📁 examples	Update doc to use test with Enzyme (#1692)	18 days ago
📁 logo	Use Redux logo as favicon on GitBook docs (#1761)	2 months ago
📁 src	Only warn for unexpected key once per key (#1818)	22 days ago
📁 test	Only warn for unexpected key once per key (#1818)	22 days ago
📄 .babelrc	Close #1687, Replace es3ify with Babel ES3 transforms (#1688)	2 months ago
📄 .editorconfig	editorconfig: do not trim trailing whitespaces in Markdown files	5 months ago
📄 .eslintignore	Really ignore all node_modules and dist in eslint.	4 months ago
📄 .eslintrc	Bump eslint version	9 months ago
📄 .flowconfig	Add Flow type annotations	a year ago

Break It Into Components

The first step is to outline or highlight all the components in this screenshot. Basically what we're doing here is drawing boxes around the elements in the design, be they divs or some other HTML building block.

build	Close #1687, Replace es3ify with Babel ES3 transforms (#1688)	2 months ago
docs	Mention that we're Observable in the API.	9 hours ago
examples	Update doc to use test with Enzyme (#1692)	18 days ago
logo	Use Redux logo as favicon on GitBook docs (#1761)	2 months ago
src	Only warn for unexpected key once per key (#1818)	22 days ago
test	Only warn for unexpected key once per key (#1818)	22 days ago
.babelrc	Close #1687, Replace es3ify with Babel ES3 transforms (#1688)	2 months ago
.editorconfig	editorconfig: do not trim trailing whitespaces in Markdown files	5 months ago
.eslintignore	Really ignore all node_modules and dist in eslint.	4 months ago
.eslintrc	Bump eslint version	9 months ago
flowconfig	Add Flow type annotations	a year ago

Name the Components

Once we've decided which parts will be components, we can assign names to them. For the highlighted areas above, I came up with these names (indented to show the parent/child relationships):

- **FileList**
 - **FileListItem**
 - * **FileName**
 - **FileIcon**
 - * **CommitMessage**
 - * **Time**

Can You Reuse Anything?

Look through the list of components and see if you can save yourself work anywhere.

Hey, look! That **Time** component looks very similar to the one we wrote for **Tweet**. Luckily we wrote that **Time** component generically, to just accept a time instead of a tweet. We can reuse it here.

What Data Does Each Component Need?

Next, let's figure out what data each component needs to render. This will give us the props and propTypes for each one.

See if you can figure out the propTypes definition for each of these without looking ahead at the code.

The `FileList` should take one prop, `files`, which is an array of file objects.

The `FileListItem` will just take a single file object as the `file` prop. That object should have a name, type, a commit with a message, and a last-modified time.

`FileName` will take a file object and expect it to have a name property.

`FileIcon` will take a file object and use its type property to decide which kind of icon to show.

`CommitMessage` will take a commit object and expect it to have a message property.

Finally, `Time` will take an absolute time string. We're going to reuse the `Time` component we made for `Tweet`, propTypes and all.

Top-Down or Bottom-Up?

We'll start from the top and work down for this one.

Keep it Working

Have you ever had the experience of coding, head down, for a long chunk of time without ever running the code? Inevitably, there's something wrong when you run it the first time.

It's disappointing, and it takes the wind out of your sails. You end up tracking down a bunch of syntax errors, logic errors, and whatever else before you can see your hard work come to life.

So as you write, try to make small changes, and refresh often. Make sure the code always works.

FileList

We've got enough direction to start working. Create a new project the same way we've done a few times now:

```
$ create-react-app github-file-list
$ cd github-file-list
$ rm src/*
$ touch src/index.js src/index.css
```

Copy the `index.html` file from the tweet project, from the “public” directory. This will set us up with the Font Awesome icons already loaded in. Change the `<title>` if you'd like, but aside from that, `index.html` is fine as it is.

Open up `src/index.js`.

Do you remember how to start off the file with the imports, and how to set up the initial render call with `ReactDOM.render`? Try to do it from memory. Look back at the Tweet example if you have trouble.

Create the `FileList` component. In the interest of doing the simplest thing that can possibly work, we'll render a plain unordered list of file names. Once it works we'll extract the list items into a `FileListItem` component.

```
// put the imports here

const FileList = ({ files }) => (
  <table className="file-list">
    <tbody>
      {files.map(file => (
        <tr className="file-list-item" key={file.id}>
          <td className="file-name">{file.name}</td>
        </tr>
      ))}
    </tbody>
  </table>
);
FileList.propTypes = {
```

```
    files: PropTypes.array
  };

const testFiles = [
  {
    id: 1,
    name: 'src',
    type: 'folder',
    updated_at: "2016-07-11 21:24:00",
    latestCommit: {
      message: 'Initial commit'
    }
  },
  {
    id: 2,
    name: 'tests',
    type: 'folder',
    updated_at: "2016-07-11 21:24:00",
    latestCommit: {
      message: 'Initial commit'
    }
  },
  {
    id: 3,
    name: 'README',
    type: 'file',
    updated_at: "2016-07-18 21:24:00",
    latestCommit: {
      message: 'Added a readme'
    }
  },
];

// put the ReactDOM.render call here
// pass testFiles as FileList's file prop
```

It should look like this, nice and ugly:

```
src
tests
README
```

Mapping over an array like this is how you render lists of things in React.

If you haven't seen it before, Array's map function returns a new array which is the same size as your existing array, but where each item is replaced by a new value. The function you provide to map decides, by returning a value, how to transform each item into a new value.

As the name "map" implies, it creates a "mapping" from your existing array to a new array. In our case, it returns a new array where each file has been turned into a table row with one cell showing the file name.

The "key" Prop

The key prop on the `<tr>` is a special one, and it's required any time you render an array of elements. React uses it to tell components apart when reconciling differences during a re-render. If you'd like more details about why keys are important, read the official docs on the [reconciliation algorithm](#).

Any time you use map to render an array, you'll also need key on the topmost element. React consumes the key prop before rendering, so the component you pass key to will not actually receive that prop.

You as the developer need to decide what to pass in as the key. The important thing to keep in mind is that keys should be *stable*, *permanent*, and *unique* for each element in the array.

- **Stable:** An element should always have the same key, regardless of its position in the array. This means `key={index}` is a bad choice.
- **Permanent:** An element's key must not change between renders. This means `key={Math.random()}` is a bad choice.
- **Unique:** No two elements should have the same key.

The item's array index is not a good choice because if the index changes, for instance when an element is added to the front of the array, React's mapping of indexes will become outdated. The item that was previously index "0" will now be "1", but React doesn't know that, and it can cause tough-to-diagnose rendering bugs. Duplicate items can appear, or items can appear out of order. So it's important to choose keys wisely.

If an item has a unique ID attached to it, that's a great choice for the key. If the items don't have IDs, try combining several fields to create a unique identifier.

FileListItem

It's generally a good idea to create a standalone component to render the individual items in a list, so we'll do that now. Even though this example is small, and it could be left alone, we'll pull it out to demonstrate the process.

Notice that we still need to pass the key prop. The key needs to be decided at the time of the "map" and passed in right there. You can't push that detail into the FileListItem component.

```
const FileList = ({ files }) => (  
  <table className="file-list">  
    <tbody>  
      {files.map(file =>  
        /* now we use FileListItem here */  
        <FileListItem key={file.id} file={file}/>  
      )}  
    </tbody>  
  </table>  
);  
FileList.propTypes = {  
  files: PropTypes.array  
};  
  
const FileListItem = ({ file }) => (  
  /* this code has been extracted from FileList */  
  <tr className="file-list-item">  
    <td className="file-name">{file.name}</td>  
  </tr>  
);  
FileListItem.propTypes = {  
  file: PropTypes.object.isRequired  
};
```

We'll also add some CSS to make it more presentable. Open up `src/index.css` and replace its contents with this code:

```
.file-list {
  font-family: Helvetica, sans-serif;
  width: 100%;
  max-width: 980px;
  color: #333;
  margin: 0 auto;
  border: 1px solid #ccc;
  border-collapse: collapse;
}

.file-list td {
  border-top: 1px solid #ccc;
}

.file-name {
  padding: 4px;
  max-width: 180px;
}
```

Don't forget to add the line to import the CSS file if you haven't already (`import './index.css'`).

src
tests
README

Now that we've got some basic structure in place, we can add the remaining components to the row component: `FileIcon`, `FileName`, `CommitMessage`, and `Time`.

Let's take care of `FileIcon` and `FileName` first, since they're nested together. According to the mockup, `FileName` is the parent of `FileIcon`. And since we're putting them into a table, it would be nice to keep them in separate cells. Try to write the code yourself before looking ahead. There's more than one right way to do it, so don't worry if your code doesn't look exactly like mine!

Here's `FileIcon`. This one is straightforward – a stateless component written as a plain function.

```
function FileIcon({ file }) {
  let icon = 'fa-file-text-o';
  if(file.type === 'folder') {
    icon = 'fa-folder';
  }

  return (
    <td className="file-icon">
      <i className={`fa ${icon}`} />
    </td>
  );
}
FileIcon.propTypes = {
  file: PropTypes.object.isRequired
};
```

Here is `FileName`, which uses the `<>` fragment syntax to wrap the two elements it returns. Remember that table cells (`<td>`) need to be direct children of table rows (`<tr>`) without any wrapper elements in between, which is why we need a fragment instead of a `div` here. (This rule comes from HTML, not React)

```
function FileName({ file }) {
  return (
    <>
      <FileIcon file={file} />
      <td className="file-name">{file.name}</td>
    </>
  );
}
FileName.propTypes = {
  file: PropTypes.object.isRequired
};
```

Then here's the updated `FileListItem` that uses the `FileName` component:

```
const FileListItem = ({ file }) => (
  <tr className="file-list-item">
    <FileName file={file}/>
  </tr>
);
FileListItem.propTypes = {
  file: PropTypes.object.isRequired
};
```

Now let's add some CSS and see how it looks.

```
.file-icon {
  width: 17px;
  padding-left: 4px;
}

.file-icon .fa-folder {
  color: #508FCA;
}
```

src
tests
README

CommitMessage

Let's create the `CommitMessage` component next, and then we can render a `CommitMessage` inside `FileListItem`.

```
const FileListItem = ({ file }) => (
  <tr className="file-list-item">
    <FileName file={file} />
    <CommitMessage
```

```

        commit={file.latestCommit} />
      </tr>
    );
    FileListItem.propTypes = {
      file: PropTypes.object.isRequired
    };

    const CommitMessage = ({ commit }) => (
      <td className="commit-message">
        {commit.message}
      </td>
    );
    CommitMessage.propTypes = {
      commit: PropTypes.object.isRequired
    };

```

And spruce it up with some style:

```

.commit-message {
  max-width: 442px;
  padding-left: 10px;
  overflow: hidden;
}

```

 src	Initial commit
 tests	Initial commit
 README	Added a readme

Notice that we're passing in the commit itself instead of the whole file object. `CommitMessage` doesn't need to know anything about files, and the fewer components that have knowledge of data structures, the better.

Time

We'll add the time now. Remember that we can reuse the `Time` component from the `Tweet` exercise. Rather than just copy-and-paste the `Time` component into this file, we'll extract it into its own file so other components can use it too.

We're going to need Moment.js again, so install that now:

```
$ npm install moment
```

Then create the file `src/Time.js` and paste in the `Time` component from earlier. We also need to add imports at the top, and an export at the bottom.

```
import React from 'react';
import PropTypes from 'prop-types';
import moment from 'moment';

const Time = ({ time }) => {
  const timeString = moment(time).fromNow();
  return (
    <span className="time">
      {timeString}
    </span>
  );
};

Time.propTypes = {
  time: PropTypes.string.isRequired
};

export default Time;
```

You might not recognize the `export default Time` syntax at the bottom. This is the ES6 way of making a component available so it can be imported into other files. The “default” means that this is the component we’ll get when we use `import Time from './Time'`.

The alternative is to make this a *named export*, which would look like `export { Time }`, with the braces. Then the corresponding import would look like `import { Time } from './Time'`.

Imports are all about the braces. No braces? You’re importing the default. With braces? You’re importing a named export. You can even mix them:

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

Think of it like destructuring, where the module is the “object” and you’re extracting named items from it.

Now let’s use Time inside FileListItem. Import it first by adding this line to the top of index.js:

```
import Time from './Time';
```

Since this is our own file instead of something from node_modules, the path needs to be relative (./Time), instead of merely the module name (Time). It’s common to name files with components in PascalCase (with the leading capital letter) but you can name them however you prefer.

Then, we can update FileListItem, and Time doesn’t render a <td> so we have to wrap it in one:

```
const FileListItem = ({ file }) => (  
  <tr className="file-list-item">  
    <FileName file={file} />  
    <CommitMessage commit={file.latestCommit} />  
    <td className="age">  
      <Time time={file.updated_at}/>  
    </td>  
  </tr>  
>);  
FileListItem.propTypes = {  
  file: PropTypes.object.isRequired  
};
```

Add a little bit of styling...

```
.age {  
  width: 125px;  
  text-align: right;  
  padding-right: 4px;  
}
```

And it works!

src	Initial commit	9 days ago
tests	Initial commit	9 days ago
README	Added a readme	2 days ago

I must say though, the code does not look very nice. We’ve got a mix where some `<td>`’s are inside components and some aren’t. It just doesn’t look consistent, and moreover, the components that contain table cells can only be used inside table rows – not very reusable at all.

A better way to organize it would be to leave the table “stuff” inside the table, and let the components worry about *just* their own data. Something like this:

```
const FileListItem = ({ file }) => (
  <tr className="file-list-item">
    <td><FileIcon file={file} /></td>
    <td><FileName file={file} /></td>
    <td><CommitMessage commit={file.latestCommit} /></td>
    <td><Time time={file.updated_at} /></td>
  </tr>
);
FileListItem.propTypes = {
  file: PropTypes.object.isRequired
};
```

At this point I could’ve gone back and edited the examples so that the code came out better. But I left it this way as an example: sometimes, maybe even often, the code won’t come out quite as clean as you imagined it. You might not fully realize the impact of a decision until you’ve lived with it for a while.

Reality often gets in the way. Sometimes you adhere too closely to the mockup, and take yourself down a path that results in awkward code. Or you try three different approaches in the same component and it becomes inconsistent and confusing.

But nothing is permanent! Now would be a perfect time to refactor this code (and in fact, you’ll do that in the exercises).

As you’re plugging away, writing your code... when that thought pops into your head that says, “Wait! These components won’t be reusable at all because every one of them contains a table cell!”... well, listen to that voice. Refactor early and often.

Exercises

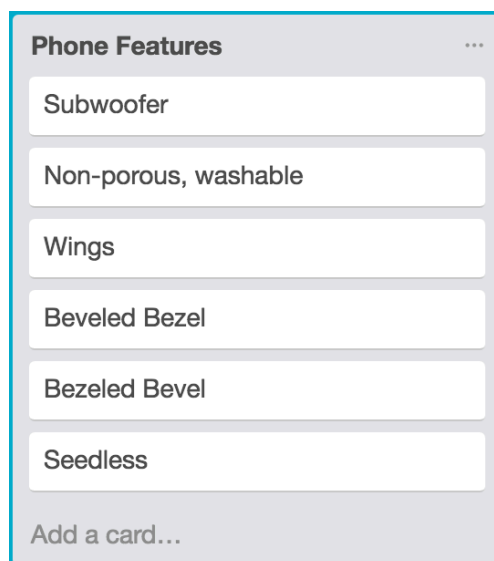
1. Refactor the GitHub file listing example so that none of the components return a table cell (`<td>`). Every component should return a `` or `<div>` instead. This makes them more reusable, and should also improve the code inside `FileList`. Change the CSS if necessary.
2. Sometimes a file will contain a few related components when those components are always used together, and when they're small (as in the `Nav/NavItem` example from earlier). But most of the time, in real applications, you'll want to have only one component per file. Refactor the code from Exercise 1 to pull out components into separate files, using `import` and `export`.
3. Now you know how to create lists, using `Array`'s `.map` function. Reuse the `Tweet` component from earlier and create a list of `Tweets`.

Lists are all over the place. It's been said that most web applications are basically just a bunch of lists. Implement these interfaces from sites around the web. Follow the same process we've done a few times now – highlight which pieces of the screen will be components, give them names, then build them.

4. Trello

Work on rendering a single list of cards. For more practice, render multiple lists of cards side-by-side.

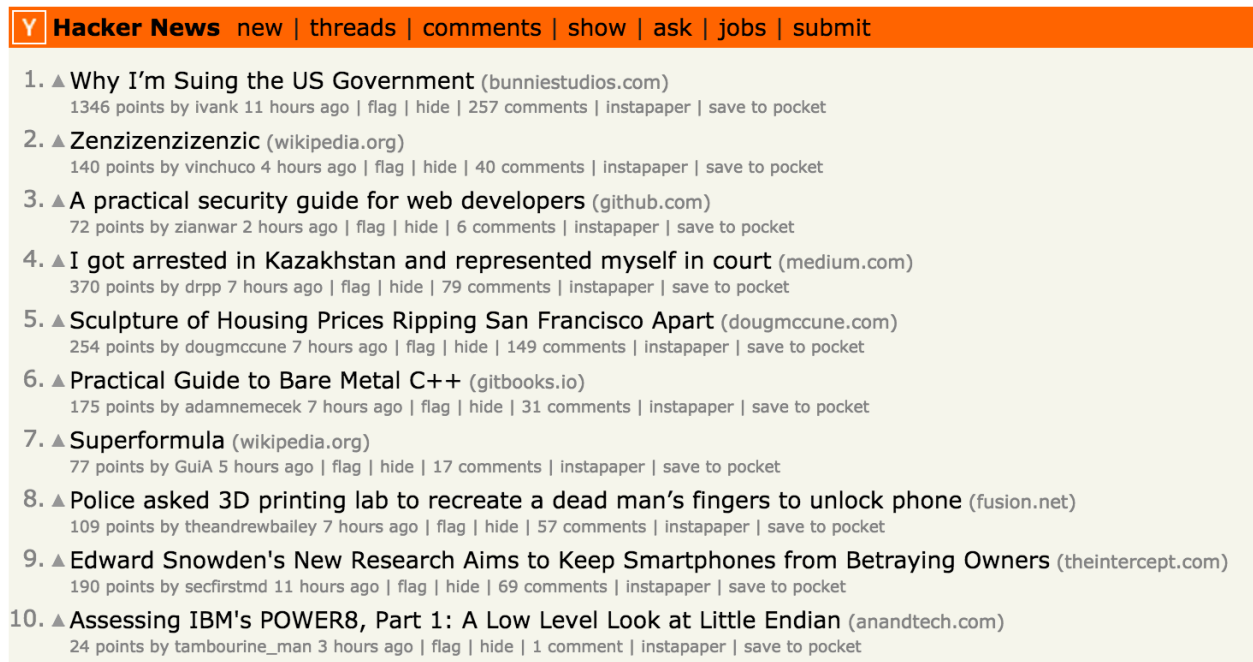
(screenshot from <https://trello.com>)



5. Hacker News

Implement the list of stories. For more practice, implement the header too.

(screenshot from <https://news.ycombinator.com/news>)



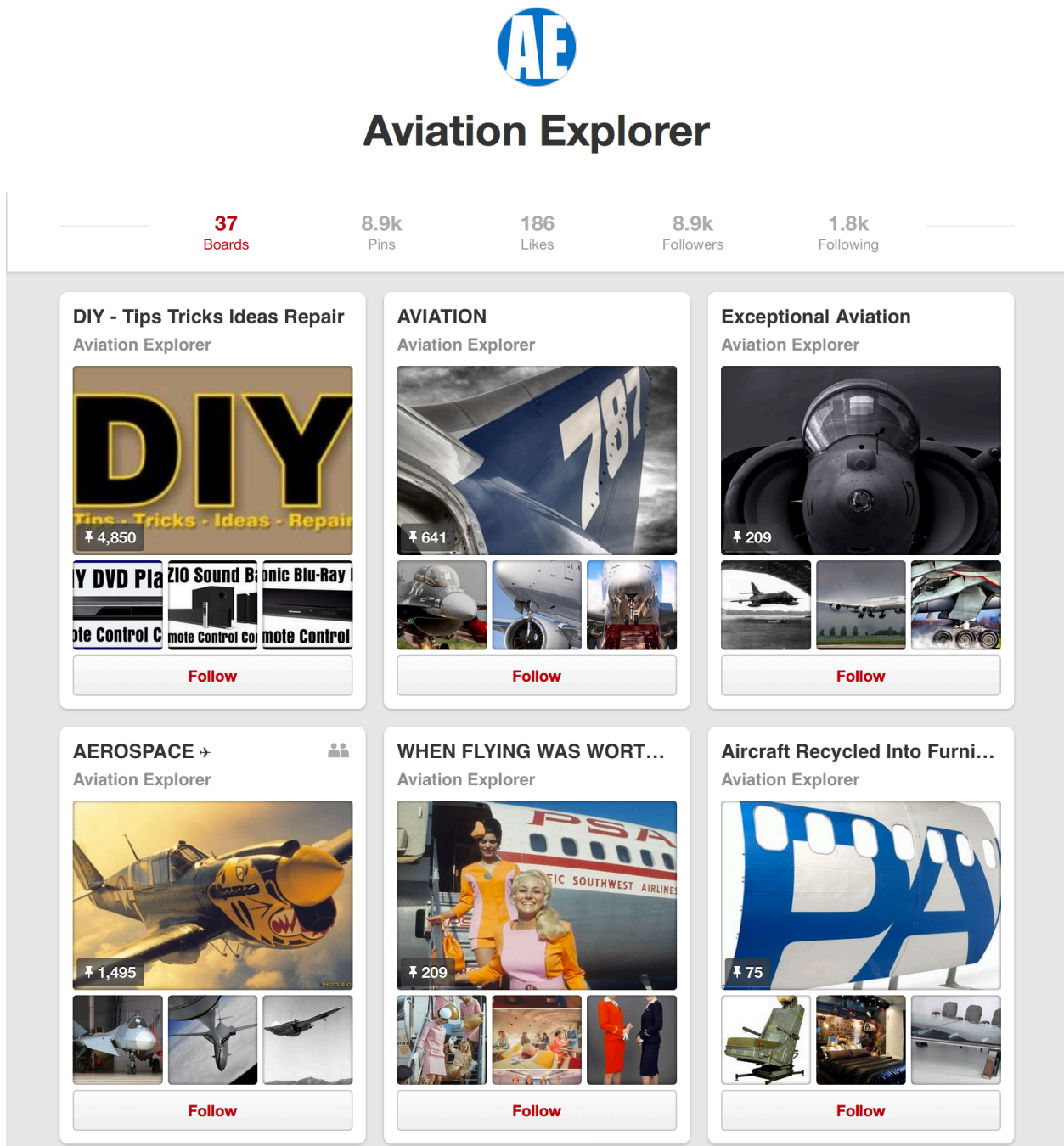
Y Hacker News new | threads | comments | show | ask | jobs | submit

- ▲ **Why I'm Suing the US Government** (bunniestudios.com)
1346 points by ivank 11 hours ago | flag | hide | 257 comments | instapaper | save to pocket
- ▲ **Zenzizenzizencic** (wikipedia.org)
140 points by vinchuco 4 hours ago | flag | hide | 40 comments | instapaper | save to pocket
- ▲ **A practical security guide for web developers** (github.com)
72 points by zianwar 2 hours ago | flag | hide | 6 comments | instapaper | save to pocket
- ▲ **I got arrested in Kazakhstan and represented myself in court** (medium.com)
370 points by drpp 7 hours ago | flag | hide | 79 comments | instapaper | save to pocket
- ▲ **Sculpture of Housing Prices Ripping San Francisco Apart** (dougmccone.com)
254 points by dougmccone 7 hours ago | flag | hide | 149 comments | instapaper | save to pocket
- ▲ **Practical Guide to Bare Metal C++** (gitbooks.io)
175 points by adamnemecek 7 hours ago | flag | hide | 31 comments | instapaper | save to pocket
- ▲ **Superformula** (wikipedia.org)
77 points by GuiA 5 hours ago | flag | hide | 17 comments | instapaper | save to pocket
- ▲ **Police asked 3D printing lab to recreate a dead man's fingers to unlock phone** (fusion.net)
109 points by theandrewbailey 7 hours ago | flag | hide | 57 comments | instapaper | save to pocket
- ▲ **Edward Snowden's New Research Aims to Keep Smartphones from Betraying Owners** (theintercept.com)
190 points by secfirstmd 11 hours ago | flag | hide | 69 comments | instapaper | save to pocket
- ▲ **Assessing IBM's POWER8, Part 1: A Low Level Look at Little Endian** (anandtech.com)
24 points by tambourine_man 3 hours ago | flag | hide | 1 comment | instapaper | save to pocket

6. Pinterest

Implement the list of image cards. For more practice, implement the header too (consider reusing the Nav and NavItem components from earlier).

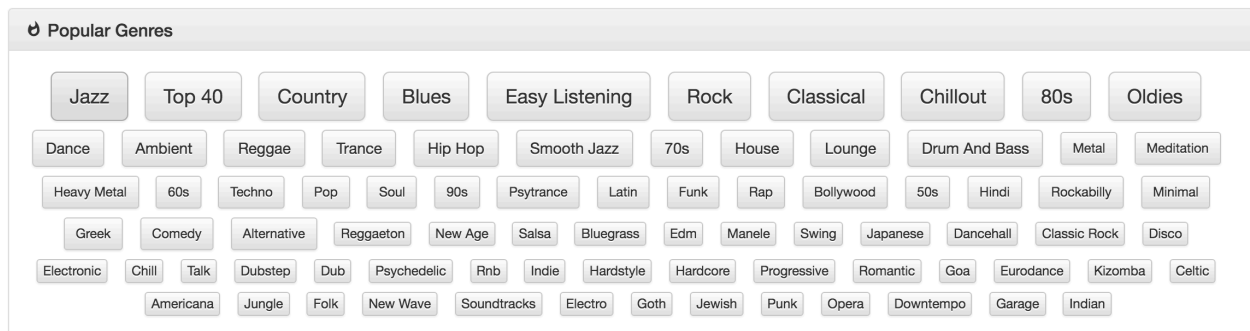
(screenshot from <https://www.pinterest.com/aviationhd/>)



7. InternetRadio genre cloud

Can you come up with a nice way of sizing the buttons so they get progressively larger?

(screenshot from <https://www.internet-radio.com/>)



11 State in Classes

Up until this point we've used *props* to pass data to components. But props are *read-only*. What if you need to keep track of data that can change? What if you want to make your app interactive? This is where *state* comes in.

To see how state is useful, let's look at the tiniest possible example: keeping track of button clicks.

Example: A Counter

Here is a Parent component that contains a Child component. Parent passes down a function which Child calls whenever a button is clicked:

```
function handleAction(event) {
  console.log('Child did:', event);
}

function Parent() {
  return (
    <Child onAction={handleAction}/>
  );
}

function Child({ onAction }) {
  return (
    <button onClick={onAction}>
      Click Me!
    </button>
  );
}
```

What if we wanted Parent to keep track of how many times the button was clicked? In other words, Parent should track how many times the `handleAction` function is called.

To do this, Parent needs to remember the count using *state*, and persist that state between renders. We'll want the *initial state* of the counter to be 0, a way to increment the counter, and a way to display the current count. Plus, whenever the count changes, we'll need to re-render the app to show the latest count.

React has two ways of adding state to components. The first is to rewrite the component as a *class*. Class components have been able to maintain state since the very first versions of React, and every version of React supports classes.

The second, more modern way, is to use *hooks* to add state directly to a function component. Hooks were added to React in version 16.8.

Here you'll learn how to use both approaches: first Classes, then Hooks. Even though hooks are the "latest and greatest," classes aren't deprecated. Classes are still well supported, and it's good to understand both styles, especially if you'll be working with code that's been around a while – like the kind you're likely to see at a company with a pre-existing React app.

Here's the new version of Parent, renamed to CountingParent and converted into a class – make sure to update the call to ReactDOM.render!

```
class CountingParent extends React.Component {
  // The constructor is called when a
  // component is created
  constructor(props) {
    super(props);

    // Set the state here. Use "props" if needed.
    this.state = {
      actionCount: 0
    };

    // Bind the event handler function, so that its
    // `this` binding isn't lost when it gets passed
    // to the button
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick(action) {
    console.log('Child says', action);
    // Replace actionCount with an incremented value
    this.setState({
      actionCount: this.state.actionCount + 1
    });
  }
}
```

```

render() {
  return (
    <div>
      <Child onAction={this.handleAction}/>
      <p>Clicked {this.state.actionCount} times</p>
    </div>
  );
}
}

```

The Child component doesn't have to change at all. Try it out! Click the button. Watch it increment. Ooooh. Aaaah. Such counter.

Ok so maybe incrementing a number is not a groundbreaking achievement, but let's see what's going on here.

When the component is first created, its initial state is set in the constructor. (Later, we'll see how to write a class without needing the constructor)

Next, you click the button, as commanded. The button's `onClick` handler is called. In this case that's the `onAction` prop, which ultimately calls the `handleAction` function in `CountingParent`. The `handleAction` function logs a message and then calls `this.setState` with an object that describes the new state.

The `setState` function will update the state and then re-render the component and all of its children. So that's what happens next: `CountingParent`'s render function is called, which looks at `this.state.actionCount`, which has now been incremented to 1. Or 2, or some other number if you've been click-happy with that button.

Note that every instance of a component has its own state. If you have more than one `CountingParent` component on the page, each will have its own counter that starts at 0 and increments independently of the others. You can prove this to yourself by creating a component that contains a few `CountingParents`:

```

const Page = () => (
  <div>
    <CountingParent/>
    <CountingParent/>
  </div>
)

```

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

Try this out! Make sure to update the `ReactDOM.render` call to render a `Page` instead of a `CountingParent`. Now click those buttons and watch the counters change independently.

Exercise: Reset Button

Here's a quick exercise: add a 'Reset' button to `CountingParent` that resets the counter to 0 when clicked. Just put the button directly inside `CountingParent`.

Once that's working, refactor your code to move the 'Reset' button down into `Child`. You'll need to pass down the click handler for the Reset button, and make sure to bind the handler in the constructor.

setState Is Asynchronous

I kinda lied to you up there. I'm sorry. I implied that the `setState` function would immediately update the state and call `render`. That's not really what happens. The `setState` function is actually *asynchronous*.

If you call `setState` and immediately `console.log(this.state)` right afterwards, it will very likely print the old state instead of the one you just set.

```
// Assume state is { count: 3 }  
  
// Then call setState:  
this.setState({ count: 4 });  
  
// Then try to print the "new" state:  
console.log(this.state);  
  
// It'll likely print { count: 3 }  
// instead of { count: 4 }
```


If you need to set the state and immediately act on that change, you can pass in a callback function as the second argument to `setState`, like this:

```
this.setState({name: 'Joe'}, function() {  
  // called after state has been updated  
  // and the component has been re-rendered  
  // this.state now contains { name: 'Joe' }  
});
```

Functional setState

Another way to make it so that sequential state updates run in sequence is to use the *functional* form of `setState`, like this:

```
this.setState((state, props) => {  
  return {  
    value: state.value + 1  
  }  
});
```

In this form, you pass a *function* to `setState` instead of an object. The function receives the current state and props as arguments, and it is expected to return an object, which will be merged with the old state. If you were to run a few of these sequentially...

```
this.setState((state, props) => {  
  return {  
    value: state.value + 1  
  }  
});  
this.setState((state, props) => {  
  return {  
    value: state.value + 1  
  }  
});  
this.setState((state, props) => {
```

```
return {  
  value: state.value + 1  
}  
});
```

This would work as expected, eventually incrementing value by 3.

This works because each call to `setState` “queues” an update in the order they’re called, and when they’re executed, they receive the latest state as an argument instead of using a potentially-stale `this.state`.

A side benefit to the functional style of `setState` is that the state update functions can be extracted from the class and reused because they are “pure” functions – that is, they only operate on their arguments, they don’t modify the arguments, and they return a new value. A “pure” function has no side effects, which means that calling it multiple times with the same arguments will always return the same result.

Functional `setState` is the preferred way to call `setState` because it’s guaranteed to work correctly, every time. Try to use it whenever you can.

Passing an object to `setState` works fine most of the time, but it’s a little like playing with fire. It’s fun until you get burned and then spend 30 minutes trying to figure out why your state isn’t updating.

Shallow vs Deep Merge

When you call `this.setState`, whether you call it with an object or in the functional form, the result is that it will *shallow merge* the properties in your object with the current state. Here’s how that works.

Let’s say you start with a state like this:

```
{  
  score: 7,  
  user: {  
    name: "somebody",  
    age: 26  
  },  
}
```

```
products: [ /*...*/ ]
}
```

After you run `this.setState({ score: 42 })`, the new state will be:

```
{
  score: 42,           // new!
  user: {              // unchanged
    name: "somebody", // unchanged
    age: 26            // unchanged
  },
  products: [ /* unchanged */ ]
}
```

That is, it *merges* the object you pass to `setState` (or return from the functional version) with the existing state. It doesn't erase the existing state, and it doesn't replace the whole top-level state with your object.

If instead you run `this.setState({ user: { age: 4 } })` then it would replace the entire `user` object with the new one:

```
{
  score: 7,    // unchanged
  user: {      // new!
    age: 4     // no more 'name'
  },
  products: [ ... ], // unchanged
}
```

A “deep” merge would peek into the `user` object and only update its `age` property while leaving the rest alone. A “shallow” merge overwrites the whole `user` object with the new one. It won't replace the top-level state, but it will only update one level deep.

Cleaner Syntax for Class Components

Earlier I mentioned that classes can be written without a constructor. Let's see how that works.

Here's the same example from earlier, with the `CountingParent` component rewritten without a constructor:

```
class CountingParent extends React.Component {
  // initialize state with a property initializer
  // you can access this.props if needed
  state = {
    actionCount: 0
  };

  // writing the handler as an arrow function
  // means it will retain the proper value of
  // `this`, so we can avoid having to bind it
  handleAction = (action) => {
    console.log('Child says', action);
    // Replace actionCount with an incremented value
    this.setState({
      actionCount: this.state.actionCount + 1
    });
  }

  render() {
    return (
      <div>
        <Child onAction={this.handleAction}/>
        <p>Clicked {this.state.actionCount} times</p>
      </div>
    );
  }
}
```

The constructor is gone, and in its place we're using *property initializers* to initialize the state and create the `handleAction` function.

This saves us a few lines of boilerplate code and makes the class easier to read.

As of this writing the property initializer syntax isn't yet finalized, but it's already widely used in the React community and has been supported by Babel & Create React App for quite a while. I'd consider it safe to use.

Handling Events

We've seen a few components that take an `onClick` prop. This is just one of many events that React components can handle. In fact, React components can respond to every event that plain old HTML elements can, for the most part.

The convention is that React's events are named with camelCase like `onClick`, `onSubmit`, `onKeyDown`... whereas the HTML events are all lowercase (`onclick`, `onsubmit`, `onkeydown`). React will actually warn you if you use the wrong capitalization:

Warning: Unknown event handler property `onclick`. Did you mean `'onClick'`?

At least one event differs by more than just capitalization: `ondblclick` is renamed to `onDoubleClick`. A complete list of events can be found in the [official React docs](#).

Your event handler function will receive the event object, which looks a lot like a native browser event. It has the standard `stopPropagation` and `preventDefault` functions if you need to prevent bubbling or cancel a form submission, for example. It's not actually a native event object though – it is a `SyntheticEvent`.

The event object passed to a handler function is only valid right at that moment. The `SyntheticEvent` object is *pooled* for performance. Instead of creating a new one for every event, React replaces the contents of the one single instance.

If you print it out with `console.log(event)`, the instance logged to the console will be cleared out by the time you go to look at it. Yes, even if you look immediately. You also can't access it asynchronously (say, after a timeout, or after a state update).

If you need to access an event asynchronously, call `event.persist()` and React will keep it around for you.

Exercises

1. Create a component called `House` to model a home with 4 rooms, each with its own light and lightswitch. Use state to keep track of whether each light is on or off. Add 4 buttons to represent the lightswitches, and flip the respective light on or off when the buttons are clicked. Use this initial state:

```
state = {  
  kitchen: true,  
  bathroom: false,  
  livingRoom: true,  
  bedroom: false  
}
```

2. Change the `House` component initial state to look like this, with state nested under the `rooms` key. Update the rest of the component accordingly.

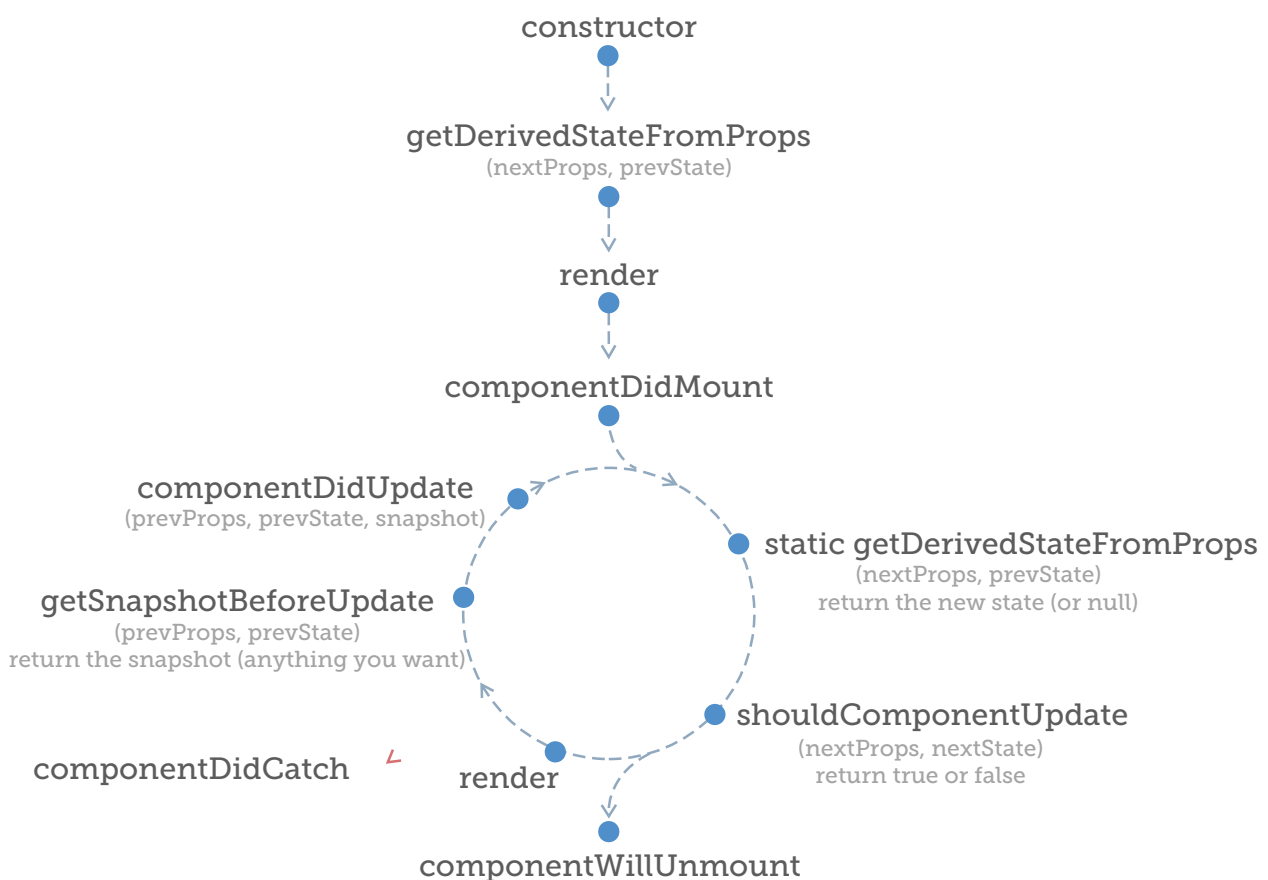
```
state = {  
  rooms: {  
    kitchen: true,  
    bathroom: false,  
    livingRoom: true,  
    bedroom: false  
  }  
}
```

12 The Component Lifecycle

Every React component goes through a *lifecycle* as it's rendered on screen. There are a sequence of methods called in a specific order. Some of them only run once, and some run multiple times.

For most of the components you build, tapping into the lifecycle methods won't be necessary. But when you need them, you need them. Whether you need to make AJAX calls to fetch data, insert your own nodes into the DOM, or set up timers, the lifecycle methods are the place to do it.

Here's the entire lifecycle of a component:



Tapping into a lifecycle hook is as simple as adding the appropriate function to your component class. Lifecycle methods are only available to *class* components, not functional ones.

Phases

Within the lifecycle of a component there are a few phases that occur.

1. **Mount** occurs when the component is first added to the DOM. Initialization and setup are done here.
2. **Render** occurs when the component renders for the first time, and whenever it re-renders due to a change in props or state. Despite its name, the *render* phase doesn't change what you see on the page.
3. **Commit** takes the output from render and updates the DOM to match.
4. **Unmount** happens when the component is being removed from the DOM.

Here's an example that uses every lifecycle method. Run it, open up the browser console, click the button, and watch the console to see the order in which they're called. There are two components here, `ErrorCatcher` and `LifecycleDemo`, because error boundaries (components that implement `componentDidCatch`) can only catch errors in their *children*, not in themselves.

```
import React from 'react';
import ReactDOM from 'react-dom';

class ErrorCatcher extends React.Component {
  state = { error: null }

  componentDidCatch(error, info) {
    console.log('[componentDidCatch]', error);
    this.setState({ error: info.componentStack });
  }

  render() {
    if(this.state.error) {
      return (
        <div>
          An error occurred: {this.state.error}
        </div>
      )
    }

    return this.props.children;
  }
}

class LifecycleDemo extends React.Component {
  // Initialize state first
  // (happens before constructor)
```



```
state = {counter: 0};

// The first method called after initializing state
constructor(props) {
  super(props);
  console.log('[constructor]');
  console.log('  State already set:', this.state);
}

// Called after initial render is done.
//
// This is a good place to kick off network
// requests to fetch data.
componentDidMount() {
  console.log('[componentDidMount]', 'Mounted.');
```

// ** Don't forget to make this `static`! **
// Called before initial render, and any time new props
// are received.
// Not commonly used.

```
static getDerivedStateFromProps(nextProps, prevState) {
  console.log('[getDerivedStateFromProps]');
  console.log('  Next props:', nextProps);
  console.log('  Prev state:', prevState);
  return null;
}

// Called before each render. Return false to prevent rendering.
//
// This (and PureComponent) are the primary ways to optimize
// class components. If you notice performance is slow,
// measure with the profiler, then try implementing this method
// to prevent needless renders. React is fast out of the box,
// and a few extra renders won't hurt. I wouldn't recommend
// implementing this method unless you know you need it.
shouldComponentUpdate(nextProps, nextState) {
  console.log('[shouldComponentUpdate]', 'Deciding to update');
  return true;
}
```

```

}

// Called after render() but before updating the DOM
// A good time to make calculations based on old DOM nodes.
// The value returned here is passed into componentDidUpdate
getSnapshotBeforeUpdate(nextProps, nextState) {
  console.log('[getSnapshotBeforeUpdate]', 'About to update...');
  return `Time is ${Date.now()}`;
}

// Called after render() and after updating the DOM. The whole
// render/commit/update cycle is done.
//
// This is a good time to check if a prop has changed,
// by checking prevProps.whatever === this.props.whatever.
// Useful for re-fetching data when a record ID changes.
componentDidUpdate(prevProps, prevState, snapshot) {
  console.log('[componentDidUpdate]', 'Updated.');
  console.log('  snapshot:', snapshot);
}

// Called right before the component is unmounted
// Time to clean up! Remove any event listeners, cancel
// timers, etc.
componentWillUnmount() {
  console.log('[componentWillUnmount]', 'Goodbye cruel world.');
```

```

}

handleClick = () => {
  this.setState({
    counter: this.state.counter + 1
  });
};

causeErrorNextRender = () => {
  // Set a flag to cause an error on the next render
  // This will cause componentDidCatch to run in the parent
  this.setState({
    causeError: true
  });
};

```

```

    });
  };

  render() {
    console.log( '[render]' );
    if( this.state.causeError ) {
      throw new Error( 'oh no!!' );
    }

    return (
      <div>
        <span>Counter: {this.state.counter}</span>
        <button onClick={this.handleClick}>
          Click to increment
        </button>
        <button onClick={this.causeErrorNextRender}>
          Throw an error
        </button>
      </div>
    );
  }
}

ReactDOM.render(
  <ErrorCatcher>
    <LifecycleDemo/>
  </ErrorCatcher>,
  document.querySelector( '#root' )
);

```

Mounting

These methods are called only once, when the component first mounts.

constructor: This is the first method called when your component is created. If state is initialized with a property initializer then it will already be set by the time the constructor executes.

componentDidMount: Called immediately *after* the first render. The component's children are already rendered at this point, too. This is a good place to make AJAX requests to fetch any data you need.

componentWillMount *[deprecated]*: Up until React 16.3, this method served a similar purpose to the constructor. It executes before the first render. You'll still probably come across code that uses it, but it will likely be removed in React 17. If you need to fetch data or do anything else "before" a component renders, just do it in `componentDidMount` instead. Rendering happens quickly, so you don't need to worry about performance, and you'll probably want to display a "loading" indicator or something while work is in progress anyway.

Rendering

These are called, in order, before and after each render. Only `getDerivedStateFromProps` is called during the initial render.

componentWillReceiveProps(nextProps) *[deprecated]*: This method is deprecated as of React 16.3, though you'll probably run into code that uses it. Use `getDerivedStateFromProps` instead.

static getDerivedStateFromProps(nextProps, prevState): This is an opportunity to change the state based on the value of props, which can be useful for initialization. It's not used very frequently. Don't call `setState` here, but instead, return an object that represents the new state. This method must not have side effects. Also, don't forget the `static` keyword before this method or it won't work. Since this method is `static`, you can't access the `this` object.

shouldComponentUpdate(nextProps, nextState): This is an opportunity to prevent rendering if you know that props and state have not changed. The default implementation always returns `true`. If you return `false`, the render will not occur (and children will not render either), and the remaining lifecycle methods will be skipped.

componentWillUpdate(nextProps, nextState) *[deprecated]*: This method is deprecated as of React 16.3, though you'll probably run into code that uses it. Use `getSnapshotBeforeUpdate`

render: You know this one well. It fits in the lifecycle right between `componentWillUpdate` and `componentDidUpdate`.

getSnapshotBeforeUpdate(prevProps, prevState): This is called after render, but before the changes are committed to the DOM. If you need to do any calculations based on the old DOM (tracking changes to scroll position, etc.) this is the time to do it. Return anything you want from this function, and that value will get passed as the third argument (snapshot) to `componentDidUpdate`. Use it to pass along anything you need to keep track of between DOM updates.

componentDidUpdate(prevProps, prevState, snapshot): Render is done. DOM changes have been committed. You can use this opportunity to operate on the DOM if you need to. Prior to this, DOM nodes could still be in flux. The snapshot argument comes from `getSnapshotBeforeUpdate`, if you returned something from there.

Unmounting

componentWillUnmount: The component is about to be unmounted. Maybe its item was removed from a list, maybe the user navigated to another tab... whatever the case, this component's time is numbered. You should invalidate any timers you created (using `clearInterval()` or `clearTimeout()`), disable event handlers (with `removeEventListener()`), and perform any other necessary cleanup. Note that this method will only get called if `componentDidMount` was called. If the component was never fully mounted (such as if it threw an error during the first render), then `componentWillMount` won't be called.

Error Handling

componentDidCatch: This method is called when a child component or one of its children throws an error inside the constructor, during rendering, or in a lifecycle method. (This doesn't include errors that happen inside event handlers, for instance). This method lets you implement *error boundaries* within your app, and in practice, you'd usually only have one or a handful of components that implement this, and use them at or near the root of your app. It's important to note that this is *not* called when the component that implements it throws an error from within itself, it's only triggered by errors in children.

13 API Requests in React

Now that you know about state and lifecycles, we can talk about the fun stuff: fetching data from a server and displaying it!

First, though, you should understand that React itself doesn't have any allegiance to any particular way of fetching data. In fact, as far as React is concerned, it doesn't even know there's a "server" in the picture at all.

React simply renders components, using data from only two places: **props** and **state**. You've already learned the essentials of React at this point. There are no built-in functions for making HTTP calls; no hidden secrets I've been keeping from you.

So, in order to display some data from the server, you need to get that data into your components' props or state.

Choose an HTTP Library

To fetch that data from the server, you'll need an HTTP library. There are a ton of them out there. Ultimately they all do the same thing, but they have different features.

Axios

My personal favorite is [axios](#). It uses Promises, automatically parses JSON for you, and treats non-200 response codes as errors. It also results in less verbose code for non-GET operations like POST/PUT/etc. The only downside is that it's an extra library to install.

Here's the axios code to fetch some data from reddit:

```
import axios from 'axios';

axios.get(`http://www.reddit.com/r/reactjs.json`)
  .then(response => {
    const posts = response.data.data.children.map(
      obj => obj.data
    );
    console.log(posts);
  });
```

```
  })  
  .catch(error => {  
    console.error(error);  
  });
```

Notice there's no React-specific code here. This same code could fetch data in any app! Once you receive data, you can put it in state, and render it from there. That's what I mean when I say React doesn't know anything about fetching data!

Fetch

Another good option is the `fetch()` function built in to modern browsers. The Fetch API is part of the JavaScript standard. It requires an extra step to parse JSON responses, and it treats basically every response as success, including 404s and 500s, so handling errors involves a little more code. It's also much more verbose if you need to do a POST/PUT/DELETE or another non-GET operation.

Here's that same call to Reddit, but using `fetch` this time:

```
// hey! we don't need to import anything!  
  
fetch(`http://www.reddit.com/r/reactjs.json`)  
  .then(response => {  
    if(response.ok) {  
      return response.json();  
    }  
    throw new Error('Request failed');  
  })  
  .then(json => {  
    const posts = res.data.data.children.map(  
      obj => obj.data  
    );  
    console.log(posts);  
  })  
  .catch(error => {  
    console.error(error);  
  });
```

Some browsers don't have the `fetch` function available. For those, Create React App bundles the `fetch` polyfill, so you can rest assured that `fetch` will be available when your code runs.

Fetch Data and Display It

Here's a simple example component that fetches posts from Reddit. Create a new project in the same way we've done before, and type it out. Then we'll go over what's happening.

```
import React from 'react';
import ReactDOM from 'react-dom';
import axios from 'axios';

class Reddit extends React.Component {
  state = {
    posts: []
  };

  componentDidMount() {
    axios
      .get(
        `https://www.reddit.com/r/${
          this.props.subreddit
        }.json`
      )
      .then(res => {
        const posts = res.data.data.children.map(
          obj => obj.data
        );
        this.setState({ posts });
      });
  }

  render() {
    const { posts } = this.state;

    return (
      <div>
        <h1>`/r/${this.props.subreddit}`</h1>
      </div>
    );
  }
}
```



```

        <ul>
          {posts.map(post => (
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      </div>
    );
  }
}

ReactDOM.render(
  <Reddit subreddit="reactjs" />,
  document.querySelector('#root')
);

```

First, we initialize state at the top. This is important because the component is going to render at least once before the call returns from Reddit, and if we left the `posts` array uninitialized, the call to `this.state.posts.map(...)` would fail during the first render.

Always remember to initialize your state!

The `componentDidMount` lifecycle is where the magic happens. This method will be executed after the component mounts for the first time. This method is only executed *once* during the component's life.

It uses the `axios.get` function to fetch the data from the subreddit, based on the `subreddit` prop passed in during render at the bottom.

Since `axios` returns a Promise, we chain the call with `.then` to handle the response. The `posts` are extracted after a little bit of transformation, and then the important bit:

The component's state is updated by calling `this.setState` with the new array of `posts`. This triggers a re-render, and then the posts are visible.

Reddit Trivia

A couple things to note, specific to Reddit:

- You can tack on `.json` to the end of any subreddit URL and get a JSON representation of the posts there.
- If you forget the `www` the request will probably fail with a CORS error.
- The data it returns is complex with a lot of nested fields, which is why we needed this bit of code to pull out the posts: `res.data.data.children.map(obj => obj.data);`

Exercises

1. Modify the Reddit example code to display an error message if the request fails. To cause an error, try passing in the name of a subreddit that doesn't exist. Use the axios code example as a guide for how to intercept the error. Hint: you'll need to save the error in state to be able to display it!
2. Modify the Reddit example code to display a "Loading..." message while the request is in progress.

14 State in Functions

You know how React class components can hold state, and function components can't?

And how class components can have lifecycles, and function components can't?

Introducing Hooks

Well – hooks change all that.

Officially released as part of React 16.8, hooks make it possible to take a React function component and add state to it, or *hook into* lifecycle methods like `componentDidMount` and `componentDidUpdate` (see what I did there).

From here on out, if you write a function component, and later decide that it needs a bit of state, you don't have to refactor the whole thing into a class. Those functions are no longer relegated to being “stateless function components”.

But I just learned classes!

Hooks don't replace classes. They're just a new tool that you can use, if you want to.

The React team has said that they have **no plans to deprecate classes** in React, so if you want to keep using them, please do!

The `useState` Hook

Let's look at a new example. We'll start off with a class and then change it into a function that uses the `useState` hook to retain state.

Here's a “button” component that can only be clicked once. Once you click it, the `clicked` state becomes true, and the button becomes disabled.

Create a new project called “usestate” and start off with a blank `index.js`:

```
$ create-react-app usestate
$ cd usestate
$ rm src/*
$ touch src/index.js
```

Open `src/index.js`, start the project with `npm start`, and type this in:

```
import React from 'react';
import ReactDOM from 'react-dom';

class OneTimeButton extends React.Component {
  state = {
    clicked: false
  }

  handleClick = () => {
    // The handler won't be called if the button
    // is disabled, so if we got here, it's safe
    // to trigger the click.
    this.props.onClick();

    // Ok, no more clicking.
    this.setState({ clicked: true });
  }

  render() {
    return (
      <button
        onClick={this.handleClick}
        disabled={this.state.clicked}
      >
        You Can Only Click Me Once
      </button>
    );
  }
}

ReactDOM.render(
```

```

<OneTimeButton onClick={() => alert("hi")} />,
document.querySelector('#root')
);

```

Got it working? The button should disable itself as soon as you click it, and you should only see the alert “hi” once.

Now, let’s convert this into a function component:

```

function OneTimeButton({ onClick }) {
  const [clicked, setClicked] = React.useState(false);

  const handleClick = () => {
    onClick();

    // Ok, no more clicking.
    setClicked(true);
  };

  return (
    <button onClick={handleClick} disabled={clicked}>
      You Can Only Click Me Once
    </button>
  );
}

```

We’ve effectively moved all of the stuff into the render method of the class. The state and the click handler are both inside the function now. We updated a couple references too: because this is no longer a concern, and the `onClick` prop is now a proper variable, we can call it directly from the click handler. The `handleClick` function is also right there in scope, and we can pass it straight into the button without a `this.` qualifier.

The big change is the introduction of `React.useState`, and the `clicked/setClicked` variables.

`useState` is a *hook*. You can tell because its name starts with “use” (that’s one of the Rules of Hooks – their names must start with “use”).

The `useState` hook takes the initial state as an argument (we passed `false`) and it returns an array with 2 elements: the current state, and a function to change the state.

Class components have one big state object, and a function `this.setState` to change the whole thing at once (plus it shallow-merges the new value).

Function components come with no state at all, but the `useState` hook allows us to add little nuggets of state as we need them. So if all we need is a single boolean, we can create a bit of state to hold that.

Since we're creating these pieces of state in a sort of ad-hoc way, and there's no component-wide `setState` function, it makes sense that we'd need a function for updating each piece of state. So it's a pair: one value, one function.

Each `useState` can store one value, and the value can be any JS type – a number, boolean, object, array, etc.

The bracket syntax `[clicked, setClicked] = ...` to the left of the equal sign is *array destructuring*, and that's built in to JavaScript since ES6 (it's not a special hooks thing). It works similarly to object destructuring we've been using throughout the book, except that because array elements don't have names, you get to assign names to the items when you destructure them.

With `useState` it's common to name the returned values like `foo` and `setFoo`, but you can call them whatever you like. The first element is the current value, and the second element is a setter function.

Now I bet you have a lot of questions. Things like...

- How does React know what the old state was? When the component re-renders... won't the state get re-created every time?
- How can I store more complex state? I have to keep track of more than one value, you know!
- Why do hook names have to start with "use"? That seems fishy.
- If there's a rule about naming... does that mean I can make my own hooks?

Let's talk about those.

The "Magic" of Hooks

Ahh, the weird paradox of storing stateful information in a seemingly-stateless function component. This was the first question I had about hooks, and I had to figure out how they worked.

My first guess was some sort of compiler trickery... searching the code for `useWhatever` and replacing it with stateful logic somehow.

And then I heard about the call order rule (hooks must be called in the same order every time), and that just made me *more* confused. So here's how it actually works.

The first time React renders a function component, it creates an object to live alongside it – a bespoke object for that component instance, not a global one. This component's object is kept around as long as the component exists in the DOM.

Using that object, React can keep track of various bits of metadata that belong to a component.

Keep in mind here that React is the one calling your component. React components – even function ones – are not “self-rendering.” Remember that they don't return HTML directly, and that they instead return an object structure that React can convert into DOM nodes.

So, React has the ability to do some setup before it calls each component, and that's when it sets up this object to hold the behind-the-scenes “state” of the component.

One of the things in there is an *array of hooks*. It starts off empty. Every time you call a hook, like `useState`, React adds an item to that array.

The Call Order Matters

Let's say we have this component:

```
function AudioPlayer() {  
  const [volume, setVolume] = useState(80);  
  const [position, setPosition] = useState(0);  
  const [isPlaying, setPlaying] = useState(false);  
  
  // < pretend it returns something beautiful >  
}
```

Since it calls `useState` 3 times, React would put 3 entries in the array of hooks on the first render.

The *next* time this component is rendered, those same 3 hooks are called in the same order (of course, because code doesn't magically rewrite itself between calls). React can look into its array and say "Oh, I already have a `useState` hook in position 0, so instead of creating a new state, I'll return the existing one."

That's how React is able to create and maintain state across multiple function calls, even when the variables themselves go out of scope each time.

Step-by-step Example of Multiple `useState` Calls

Let's look at how this plays out in more detail. Here's the first render:

1. React has just created the component. It hasn't even called the function yet. It creates the metadata object, and the empty array of hooks. Let's imagine that object has a property called `nextHook` and it's set to 0. The first hook that executes will consume position 0.
2. React calls your component (which means it knows which metadata object to store the hooks in).
3. You call `useState`. React creates a new piece of state, puts it in position 0 of the hooks array, and returns your `[volume, setVolume]` pair with `volume` set to its initial value of 80. It also increments the `nextHook` index to 1.
4. You call `useState` again. React looks at position 1 of the array, sees that it's empty, and creates a new piece of state. Then it increments the `nextHook` index to 2, and returns `[position, setPosition]`.
5. You call `useState` a third time. React sees that position 2 is unfilled, creates the state, increments `nextHook` to 3, and returns `[isPlaying, setPlaying]`.

Now the array of hooks has 3 items in it, and the render is finished. What happens on the *next* render?

1. React needs to re-render the component. Hello, old friend. React has seen this component before, and it already has the associated metadata object.
2. React resets the `nextHook` index to 0, and calls your component.
3. You call `useState`. React looks into its array of hooks at index 0, and sees that it already has a hook in that slot! No need to create one. So it advances `nextHook` to index 1 and returns `[volume, setVolume]` with `volume` still set to 80.
4. You call `useState` again. This time, `nextHook` is 1, so React checks index 1 of the array – again, a hook already exists, so it increments `nextHook` and returns `[position, setPosition]`.
5. You call `useState` a third time. I think you know what happens by now.

So that's it. It's not magic; but it does rely on a few things being true.

Technically, the “array of hooks” is implemented as a linked list within React. If you already know how those work, awesome. If not, that’s a detail you can safely ignore!

Let’s talk about the ground rules now.

Rules of Hooks

It ain’t Fight Club, but we do have some rules to follow:

1. Only call hooks at the top level of your function. Don’t put them in loops, conditionals, or nested functions. In order for React to keep track of your hooks, the same ones need to be called in the same order every single time. If you called `useState` from inside an `if`, for instance, and it ran during the first render but got skipped during the second, React would be very confused.
2. Only call hooks from React function components, or from custom hooks (we’ll learn about those later). Don’t call them from outside a component (what would that even do?). Keeping all the calls inside components and custom hooks makes your code easier to follow too, because all the related logic is grouped together.
3. The names of custom hooks must start with “use”. Like `useState` or `useEffect` (well, not those two, those are taken).

The React team created some ESLint rules to catch problematic usage of hooks (those rules are baked in to Create React App projects) and the linter needs a way to identify what “a hook” looks like. Hence the naming prefix. Nothing magical going on there. The linter will be able to warn you if you violate rule 1 or 2, but only if you follow rule 3!

Update State Based on Previous State

Let’s look at another example: updating the value of state based on the previous value.

We’ll build a, uh, “step tracker.” Very easy to use. Just like a Fitbit. Every time you take a step, simply click the button. At the end of the day, it will tell you how many steps you took. I’m working on securing my first round of funding as you read this.

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

function StepTracker() {
  const [steps, setSteps] = useState(0);
```

```

function increment() {
  setSteps(steps => steps + 1);
}

return (
  <div>
    Today you've taken {steps} steps!
    <br />
    <button onClick={increment}>I took another step</button>
  </div>
);
}

ReactDOM.render(
  <StepTracker />,
  document.querySelector('#root')
);

```

This example looks a lot like the last one. This time I've imported `useState` directly from `React`, so we don't have to write out `React.useState`.

First, we're creating a new piece of state by calling `useState`, initializing it to 0. It returns the current value of steps (0) and a function for updating it. We have an `increment` function to increase the step counter.

You'll notice we're using the functional or "updater" form of `setSteps` here. We *could* just call `setSteps(steps + 1)` and it would work the same in this example, but I wanted to show you the updater form, because it'll be useful in case your update is happening in a closure which has closed over the old (stale) value of the state. Using the updater form ensures you are operating on the latest value of state.

Another thing we've done here is to extract the `increment` function, instead of inlining the arrow function on the button's `onClick` prop. We could have written button this way and it would've worked just the same:

```

<button onClick={() => setSteps(steps => steps + 1)}>
  I took another step
</button>

```

State as an Array

Remember that state can hold any value you want! Here’s an example of a list of random numbers. Clicking the button adds a new random number to the list:

```
function RandomList() {
  const [items, setItems] = useState([]);

  const addItem = () => {
    setItems([
      ...items,
      {
        id: items.length,
        value: Math.random() * 100
      }
    ]);
  };

  return (
    <>
      <button onClick={addItem}>Add a number</button>
      <ul>
        {items.map(item => (
          <li key={item.id}>{item.value}</li>
        ))}
      </ul>
    </>
  );
}
```

Notice we’re initializing the state to an empty array [], and take a look at the addItem function.

The state updater function (setItems, here) doesn’t “merge” new values with old – it overwrites the state with the new value. This is a departure from the way this.setState worked in classes.

So in order to add an item to the array, we’re using the ES6 spread operator ... to copy the existing items into the new array, and inserting the new item at the end.

State as an Object

Since the setter function returned by `useState` will *overwrite* the state each time you call it, it works differently from the class-based `this.setState`.

Recall that `this.setState` would shallow-merge the object you passed it, into the existing state, taking care not to clobber the other stuff in there.

The `useState` setter, instead, will clobber everything. It *replaces* the entire value with whatever you pass in. Here's an example where state is an object with a couple values:

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

const MultiCounter = () => {
  const [counts, setCounts] = useState({
    countA: 0,
    countB: 0
  });

  const incA = () => (
    setCounts(counts => ({
      ...counts,
      countA: counts.countA + 1
    })))
  );

  const incB = () => (
    setCounts(counts => ({
      ...counts,
      countB: counts.countB + 1
    })))
  );

  const badIncA = () => (
    setCounts({
      countA: counts.countA + 1
    })
  );
};
```

```
return (  
  <>  
    <div>A: {counts.countA}</div>  
    <div>B: {counts.countB}</div>  
    <button onClick={incA}>  
      Increment A  
    </button>  
    <button onClick={incB}>  
      Increment B  
    </button>  
    <button onClick={badIncA}>  
      Increment A Badly  
    </button>  
  </>  
>);  
}  
  
ReactDOM.render(  
  <MultiCounter />,  
  document.querySelector('#root')  
>);
```

Type this example in and see how it works. Click “Increment A” a few times, and “Increment B” a few times. Then try “Increment A Badly”.

The takeaway here is that if your state is a complex value like an object or array, you need to take care, when updating it, to copy in all the *other parts* that you don’t intend to change. The `...` spread operator is a big help for making copies of arrays and objects.

Exercises

1. Create a Room component with a “lightswitch” button and some text describing “The room is lit” or “The room is dark”. Clicking the button should toggle the light on and off, and update the text. Use the useState hook to store the lightswitch state.
2. Create a RandomList component that shows a button, and a list of random numbers. When you click the button, add another random number to the list. Store the array of numbers with useState. The initial state should be an empty array.
3. Create a component called AudioControls with 4 pieces of state: “volume”, “bass”, “mid, and”treble”, each storing a value between 1 and 100. The app should look something like this:



Display each value along with a label and a pair of +/- buttons for increasing and decreasing the value.

Make two separate versions of this component: In the first, store the values in their own individual useState variables. In the second, store the values together in one state variable, an object that looks like this:

```
{
  volume: <number>,
  bass: <number>,
  mid: <number>,
  treble: <number>
}
```

15 Thinking About State

What to Put in State

How do you decide what should go into state? Is there anywhere else to store persistent data?

As a general rule, data that is stored in state should be referenced inside render somewhere. Component state is for storing *UI state* – things that affect the visual rendering of the page. This makes sense because any time state is updated, the component will re-render.

If modifying a piece of data does not visually change the component, that data shouldn't go into state. Here are some things that make sense to put in state:

- User-entered input (values of text boxes and other form fields)
- Current or selected item (the current tab, the selected row)
- Data from the server (a list of products, the number of “likes” on a page)
- Open/closed state (modal open/closed, sidebar expanded/hidden)

Other stateful data that doesn't affect the visual output, like handles to timers and event handlers, should be stored on the component instance itself. You've got a `this` object available in class components – feel free to store those values on it.

Should Props Go in State?

You should avoid copying props into state. It creates a second source of truth for your data, which usually leads to bugs. If you ever find yourself copying a prop into state and then thinking, “Now how am I going to keep this updated?” – take a step back and rethink.

Components will automatically re-render when their parents do, and they'll receive fresh props each time, so there's no need to duplicate the props into state and then try to keep it up to date.

In object oriented code, it's common to instantiate an object with a set of values and have the object exist independently from then on. In those cases, you might need to “keep the object in sync” by manually updating its internal values to match the surrounding environment.

React, though, is more functional in nature. A component will receive fresh props from its parent every time it re-renders (and this applies to both classes and function components!). You don't need to worry that a component will get out of sync after the first render, and in fact, if you try to copy props into state to *avoid* a synchronization problem, you might actually end up causing one!


```
// Don't do this:
class BadExample extends Component {
  state = {
    data: this.props.data
  }

  // This componentDidMount function will run *after*
  // a render occurs
  componentDidMount(oldProps) {
    /*
     By duplicating the data, you then have to keep the
     local copy in sync with the updated props..
    */
    if(oldProps.data !== this.props.data) {
      // The component already rendered once with the
      // new value of this.props.data, and copying that
      // value into state will cause it to render *again*.
      this.setState({ data: this.props.data });
    }
  }

  render() {
    return (
      <div>The data: {this.state.data}</div>
    )
  }
}

// Do this instead:
class GoodExample extends Component {
  render() {
    return (
      <div>The data: {this.props.data}</div>
    )
  }
}
```

Initializing State from Props

So, is it *ever* ok to initialize state based on props? Yes. The original version of the React docs mentioned this:

However, it's not an anti-pattern if you make it clear that the prop is only seed data for the component's internally-controlled state.

Think of it this way: it's fine if the state needs a starting value which the component will then control. Ask yourself: does this component “own” the data? Does it need a “default value” for a piece of state? Those are good reasons to initialize state from a prop.

Thinking Declaratively

Getting used to React involves changing how you solve certain kinds of problems. It reminds me a little bit of learning to drive on the other side of the road.

The first time I experienced this, I was visiting Turks and Caicos. They drive on the left there. Being from the US where we drive on the right, this took a bit of reprogramming. I nearly died on the way out of the airport.

The weird thing was, even after I had learned to drive on the left in normal straight-and-level driving, my brain would revert to old habits whenever a different situation came up.

Turning into a parking lot? Habit took over and I drove into the wrong lane. Taking a left at a stop sign? Same problem. Taking a *right* at a stop sign? You'd think I would've learned by now – but no, to my brain, that was different somehow.

I tell this story because I had a similar experience when learning React, and I think many others do too.

Passing props to a component (as if that component were a function) makes sense pretty quickly. And JSX looks and works like HTML. Our brains are used to that.

The idea of passing data *down* and passing events *up* also makes sense pretty quickly, for simple cases. It's the “callback” pattern, commonly used elsewhere, so not all that foreign. Passing an `onClick` handler to a button is fairly normal.

But what happens when it's time to open a modal dialog? Or display a popup notification in the corner? Or animate an icon in response to an event? You might find, as I did, that these imperative, "event-based" things don't come naturally in the declarative world of React.

How to Develop "Declarative" Thinking

If you came from a framework or language where you primarily call functions to make things happen in a certain order ("imperative programming"), you need to adjust your mental model in order to work effectively with React. You'll adjust pretty quickly with practice – you just need a few new examples or "patterns" for your brain to draw from.

Here are a few.

Expanding/Collapsing an Accordion control

The old way: Clicking a toggle button opens or closes the accordion by calling its toggle function. The Accordion knows whether it is open or closed.

The declarative way: The Accordion can be displayed in either the "open" state, or the "closed" state, and we store that information as a flag inside the parent component's state (*not* inside the Accordion). We tell the Accordion which way to render by passing `isOpen` as a prop. When `isOpen` is true, it renders as open. When `isOpen` is false, it renders as closed.

```
<Accordion isOpen={true}/>  
// or  
<Accordion isOpen={false}/>
```

The biggest difference is that in the declarative React way, the expand/collapse state can be stored *outside* the Accordion and passed in as a prop. Instead of the Accordion instinctively (and internally) knowing whether it is open or closed, it is *told* to be open or closed by whichever component renders the Accordion.

Opening and Closing a Dialog

The old way: Clicking a button opens the modal. Clicking its Close button closes it.

The declarative way: Whether or not the Modal is open *is a state*. It's either in the "open" state or the "closed" state. So, if it's "open", we render the Modal. If it's "closed" we don't render the modal. Moreover, we can pass an `onClose` callback to the Modal – this way the *parent component* gets to decide what happens when the user clicks Close.

```
<div>
  {this.state.isModalOpen &&
    <Modal onClose={this.handleClose}/>}
</div>
```

Notifications

The old way: When an event occurs (like an error), call a notification library to display a popup, like `toastr.error("Oh no!")`.

The declarative way: Think of notifications as state. There can be 0 notifications, or 1, or 2... Store those in an array. Put a NotificationTray component somewhere near the root of the app, and pass it the messages to display. You can manage the array of messages in the root component's state, and pass an `addNotification` prop down to components that need to be able to surface notifications.

Animating a Change

Let's say you have a badge with a counter showing the number of logged-in users. It gets this number from a prop. What if you want the badge to animate when the number changes?

The old way: You might use jQuery to toggle a class that plays the animation, or use jQuery to animate the element directly.

The declarative way: You can respond when props change by implementing the `componentDidUpdate` lifecycle method and comparing the old value to the new one. If the value changed, you can set the "animating" state to `true`. Then in render, when "animating" is true, set a CSS class that triggers the animation. When "animating" is `false`, don't set that class. Here's what this might look like:

```

class Badge extends Component {
  componentDidUpdate(oldProps) {
    if(this.props.counter !== oldProps.counter) {
      // Set `animating` to true right now.
      // When the state change finishes, set a timer
      // to turn off the animation 200ms later.
      this.setState({ animating: true }, () => {
        setTimeout(() => {
          this.setState({ animating: false });
        }, 200);
      });
    }
  }

  render() {
    const animatingClass =
      this.state.animating ? 'animating' : '';
    return (
      <div className={`badge ${animatingClass}`}>
        {this.props.counter}
      </div>
    );
  }
}

```

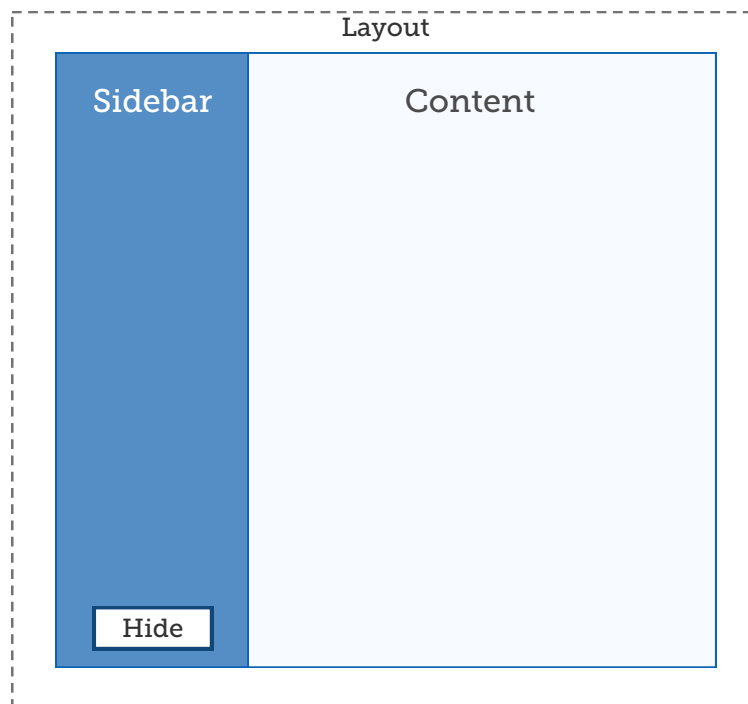
I just want you to know that this way of writing code may feel strange at first, and that's normal.

You'll find yourself wondering how to implement those old imperative patterns in a declarative way, and it'll take a little time to learn new patterns. But don't worry – it will become second nature once you do it a few times.

Where to Keep State

Whenever you can, it's best to keep components stateless. Components without state are easier to write, and easier to reason about. Sometimes this isn't possible, but often, pieces of data you initially think should go into internal state can actually be lifted up to the parent component, or even higher.

Think about a section of the page that can be shown or hidden, such as a sidebar:



When you click the “Hide” button, the sidebar should disappear, which means something needs to set a `showSidebar` flag to `false`. This flag should be stored in state somewhere. But where?

Option 1, the `showSidebar` flag could reside in the `Sidebar` component. This way the `Sidebar` “knows” whether it is open or not. This is similar to how *uncontrolled inputs* work, which we’ll see in the next chapter.

Option 2, the `showSidebar` flag can reside in the parent of `Sidebar`, which is the `Layout` component. Then parent can decide whether to render the `Sidebar` or not:

```
class Layout extends React.Component {
  state = {
    showSidebar: false
  }

  toggleSidebar = () => {
    this.setState({
      showSidebar: !this.state.showSidebar
    })
  }
}
```

```

    });
  }

  render() {
    const { showSidebar } = this.state;
    return (
      <div className="layout">
        {showSidebar &&
          <Sidebar
            onHide={this.toggleSidebar}>
              some sidebar content
            </Sidebar>
          <Content
            isSidebarVisible={showSidebar}
            onShowSidebar={this.toggleSidebar}>
              some content here
            </Content>
          </div>
        );
      }
    }

    const Content = ({
      children,
      isSidebarVisible,
      onShowSidebar
    }) => (
      <div className="content">
        {children}
        {!isSidebarVisible && (
          <button onClick={onShowSidebar}>
            Show
          </button>
        )}
      </div>
    );

    const Sidebar = ({
      onHide,

```

```
    children
  }) => (
    <div className="sidebar">
      {children}
      <button onClick={onHide}>
        Hide
      </button>
    </div>
  );
```

This way Sidebar doesn't internally "know" whether it's visible. It is either rendered, or not rendered.

Keeping the state in a parent component might feel unnatural. It might even seem at first glance like this would make Sidebar hard to reuse, since you need to pass a callback function that Sidebar can call when it needs to hide.

On the contrary, having fewer components containing state means fewer places to look when a bug appears. And if you need to do anything with that state, such as save it to `localStorage` to persist across page reloads, the logic only needs to exist in one component. Less duplicated code means less time tracking down bugs and less time keeping all the duplicates in sync.

Browsers come with `localStorage`, a place you can use to save data between sessions. Closing and reopening the browser or refreshing the page doesn't clear data in `localStorage`. You could save the Sidebar state with `localStorage.setItem('showSidebar', true)` and later retrieve it with `localStorage.getItem('showSidebar')`.

You can see that if the Sidebar component knew how to save its own state to `localStorage`, and there were multiple Sidebars in the app, they would need a way to differentiate themselves. A single `showSidebar` key in `localStorage` would lead to conflicts, and the Sidebars don't know where in the app they reside.

Perhaps the parents could be responsible for passing down a unique "id" or "location" to each Sidebar. However, that would split the concern of "where to save" between both the parent and the Sidebar. That makes *two* components that need to know about saving, rather than one. A better design would keep all of the saving-related logic in one place, and that is easier to do if the related state is stored alongside it.

“Kinds” of Components

Architecturally, you can segment components into two kinds: *Presentational* (a.k.a “Dumb”) and *Container* (a.k.a. “Smart”).

Presentational components are stateless. They simply accept props and render some elements based on those props. A stateless component will generally contain less logic, and will be easier to debug and test. They are, in essence, pure functions. They always return the same result for a given set of props, and they don’t *change* anything. Ideally, most of your components will be presentational.

Sidebar, as written above, is a presentational component. So is Content, and so is the Tweet component we made a while back. They accept data and render it, and if events need to be handled, they call back up to the parent. Other kinds of common presentational components include buttons, navigation bars, links, images, etc.

Container components are stateful. They maintain state for themselves and any child components, and pass it down to them via props. They usually pass down handler functions to children, and respond to callbacks by updating their internal state. Container components are also responsible for asynchronous communication, such as AJAX calls to the server.

In an ideal world, you’d try to organize your app so that the components at the very top level (and maybe one level below that) are containers, and everything under them is presentational. In the real world this is difficult to achieve because you might have nested inputs that contain their own state, or more complicated requirements. That’s okay though – perfection is not the goal.

16 Input Controls

Now that you’ve got a handle on how state works, we can talk about handling user input.

Input controls in React come in two flavors: *controlled* and *uncontrolled*.

Controlled Inputs

The reason they’re called “controlled” is because you are responsible for controlling their state. You need to pass in a value, and keep that value updated as the user types. It’s a little more work, but after you write a few of them it will become second nature.

Instead of the “magical” 2-way binding that some other frameworks have, React makes things explicit. Here’s what a controlled input looks like:

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

const InputExample = () => {
  const [text, setText] = useState('');

  const handleChange = event => {
    setText(event.target.value);
  };

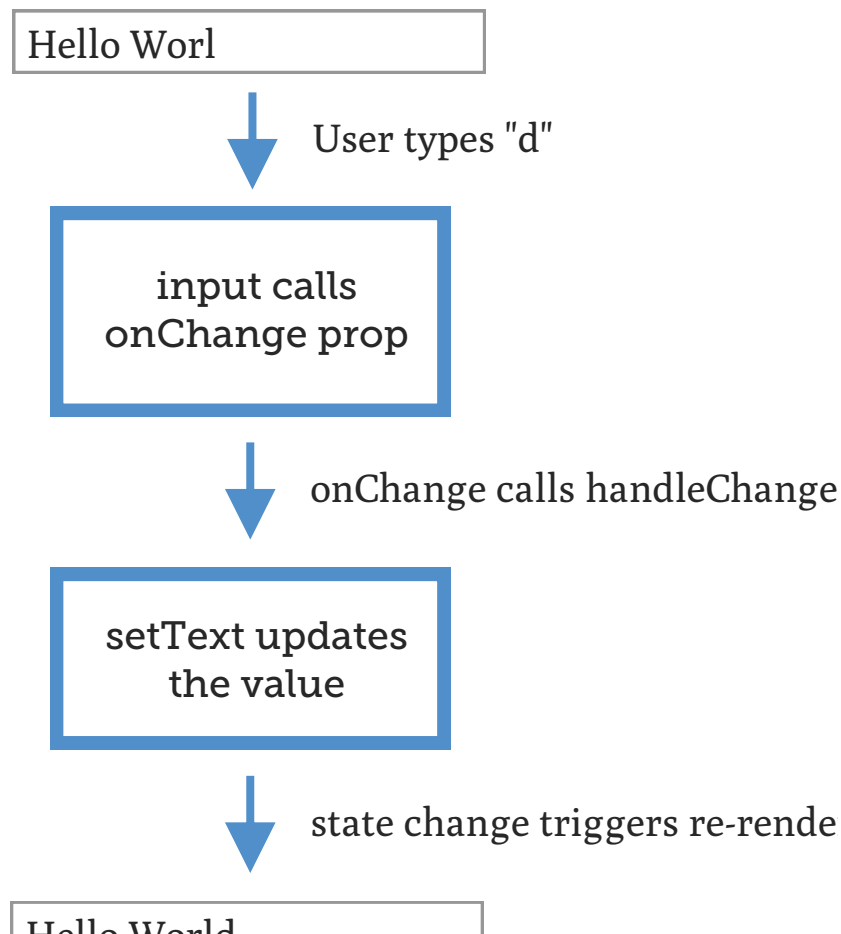
  return (
    <input
      type="text"
      value={text}
      onChange={handleChange}
    />
  );
}
```

At the top of the function we create a piece of state to hold the input’s value, and store that in `text`.

Then there’s the `handleChange` function, updates the text with the value from the event. It gets passed in as the input’s `onChange` prop, so that every time you press a key, the `handleChange` function will get called with the input’s current value (the whole thing, not just the most recent key).

Notice that we're also passing a `value` prop to the input, effectively telling it what to display.

Imagining we had an input with the text "Hello Worl" in it (missing the 'd'), here's the sequence of events that happens when you type the final letter:



Now, maybe your first reaction is that this seems like a lot of code for a simple input. You're not wrong, but this method does provide a lot of power.

In the example above, try removing or commenting out the call to `setText`, then try to type in the box. Predictably, nothing happens because the value is stuck at the initial value.

Now, what happens if we ignore the event data entirely?

Try changing `setText` to ignore the event data, and instead always setting the text to the same value, like this:

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

const TrickInput = () => {
  const [text, setText] = useState(
    'try typing something'
  );

  const handleChange = event => {
    setText('haha nope');
  };

  return (
    <input
      type="text"
      value={text}
      onChange={handleChange}
    />
  );
}
```

You'll see that the input thwarts your attempts to change it, and also laughs at you.

This technique is useful if you need to do some kind of custom validation or formatting, because you can do both in the `handleChange` function. Don't want the user to type numbers? Strip out the numbers before updating the state:

```
const NoNumbersInput = () => {
  // ...
  const handleChange = event => {
    let text = event.target.value;
    setText(
      text.replace(/[0-9]/g, '')
    );
  };
  // ...
}
```

Try typing some numbers in the box. Try pasting in a string that contains numbers. Flawless! No flickering or any weird behavior.

You could format the input as a phone number, or a credit card number, or make changes in response to the cursor position. It's pretty powerful.

But maybe you don't care about all that. You just want the input to manage its own state. Uncontrolled inputs will do that, with a few caveats.

Uncontrolled Inputs

When an input is *uncontrolled*, it manages its own internal state. So you can put an input on the page like this (without a `value` prop):

```
const EasyInput = () => (  
  <input type="text" />  
);
```

Type in it, do whatever you like. It works normally.

You can pass a `defaultValue` prop, and the component will be initialized with that value. After that initial render though, if you need to change the value, the best way to do that is to convert it to a controlled input.

When you want to get a value out of it, you have two options.

First, you can pass an `onChange` prop and respond to the events. The downside with this is that you can't easily extract a value at will. You need to listen to the changes, and keep track of the "most recent value" somewhere, probably in state. So it doesn't save much code over using a controlled input.

Refs

Alternatively, you can use a *ref*. A ref gives you access to the input's underlying DOM node, so you can pull out its value directly.

In function components, we can call the `useRef` hook to create an empty ref, and then pass that into a `ref` prop on the input.

Here's an example using a ref on an input to extract its value when a button is clicked:

```
import React, { useRef } from 'react';

const RefInput = () => {
  const input = useRef();

  const showValue = () => {
    alert(`Input contains: ${input.current.value}`);
  };

  return (
    <div>
      <input type="text" ref={input} />
      <button onClick={showValue}>
        Alert the Value!
      </button>
    </div>
  );
};
```

The ref prop is a special one. Pass it a ref object, and when the component mounts, React will save the DOM node (or the component instance, if it's a class component) into the ref's current property.

Note that refs can only be used to refer to regular elements (div, input, etc) and stateful (class) components. Function components don't have a backing instance so there's no way to refer to them. That doesn't mean that you can't write stateless components that *use* refs, just that you can't attach a ref prop to a stateless component.

The object returned by `useRef` is more than just a way to hold a DOM reference, though. It can hold *any* value specific to this component instance, and it persists between renders. So that means `useRef` can be used to create generic instance variables, just like you can do with a React class component with `this.whatever = value`.

The only thing is, assigning a ref counts as a "side effect" so you can't change it during a render – only inside the body of a `useEffect` hook. In the next chapter, we'll learn about `useEffect`.

Exercises

1. Create an app with labelled, controlled text inputs for First Name and Last Name, and have it display “Hello, {firstName} {lastName}!” in real time, as you type into the text boxes.
2. Create another version of the app from Exercise 1, with labelled text inputs for First Name and Last Name, but make them *uncontrolled* inputs instead. Create refs to the inputs, and add a Submit button that, when clicked, will update the display to reflect the values in the inputs.
3. Though we haven’t explicitly talked about other input types like select, radio, checkbox, and textarea, they work much the same as the plain old text inputs you’ve used so far.

The select and textarea both use the familiar value and onChange props. The radio and checkbox differ a bit. Checkboxes and radio buttons both use a checked prop instead of value to determine if they’re active or not, so instead of reading event.target.value you’ll need to use event.target.checked.

With radio buttons, it’s often helpful to give them a static value prop, and store that value in state to keep track of which radio button is selected. Then, you can set their checked prop based on that value. Here’s a snippet of code to show you what I mean:

```
setLetter = (event) => {
  this.setState({
    letter: event.target.value
  });
}

render() {
  const { letter } = this.state;
  return (
    <form>
      <input
        type="radio"
        value="a"
        checked={letter === 'a'}
        onChange={this.setLetter}
      />
      <input
        type="radio"
        value="b"
        checked={letter === 'b'}
        onChange={this.setLetter}
      />
      <input
```

```
    type="radio"
    value="c"
    checked={letter === 'c'}
    onChange={this.setLetter}
  />
</form>
);
}
```

Create an app that involves radio buttons, a checkbox, a select dropdown, and a textarea. You can model it after this form for ordering a pizza, or make up something on your own.

React App

localhost:3000

Order Your Pizza

Size

☐ Small ☐ Medium ☒ Large

Topping

Pepperoni

☐ Gluten free

Special instructions:

extra crispy

Send Order

17 The useReducer Hook

The word “reducer” evokes images of Redux for many – but you don’t have to understand Redux to use the useReducer hook.

Before we get into how you can take advantage of useReducer to manage complex state in your components, let’s talk about what a reducer is.

What’s a Reducer?

A “reducer” is a fancy word for a function that takes 2 values and returns 1 value.

If you have an array of things, and you want to *combine* those things into a single value, the “functional programming” way to do that is to use Array’s reduce function. For instance, if you have an array of numbers and you want to add them all together, you can write a reducer function and pass it to reduce, like this:

```
const adder = (total, number) => {  
  return total + number;  
};  
let numbers = [1, 2, 3];  
let sum = numbers.reduce(adder, 0);
```

The adder function here is called a *reducer*.

If you haven’t seen this before it might look a bit cryptic. What this does is call the adder for each element in the array, passing in the previous total and the current element as number. Whatever you return becomes the new total. The second argument to reduce (0 in this case) is the initial value for total. In this example, the function provided to reduce (a.k.a. the “reducer” function) will be called 3 times:

- Called with (0, 1), returns 1.
- Called with (1, 2), returns 3.
- Called with (3, 3), returns 6.
- reduce returns 6, which gets stored in sum.

Ok, but, what about useReducer?

I spent half a page explaining Array's reduce function because, well, useReducer takes the same arguments, and basically works the same way. You pass a reducer function and an initial value (initial state). Your reducer receives the current state and an action, and returns the new state. We could write one that works just like the summation reducer:

```
useReducer((state, action) => {
  return state + action;
}, 0);
```

So... what triggers this? How does the action get in there? Good question.

useReducer returns an array of 2 elements, similar to the useState hook. The first is the current state, and the second is a dispatch function. Here's how it looks in practice:

```
const [sum, dispatch] = useReducer((state, action) => {
  return state + action;
}, 0);
```

Notice how the "state" can be any value, same as with useState. It doesn't have to be an object. It could be a number, or an array, or anything else.

Let's look at a complete example of a component using this reducer to increment a number:

```
import React, { useReducer } from 'react';

function Counter() {
  // First render will create the state, and it will
  // persist through future renders
  const [sum, dispatch] = useReducer((state, action) => {
    return state + action;
  }, 0);

  return (
    <>
```

```

    {sum}

    <button onClick={() => dispatch(1)}>
      Add 1
    </button>
  </>
);
}

```

Create a project and give it a try.

You can see how clicking the button dispatches an action with a value of 1, which gets added to the current state, and then the component re-renders with the new (larger!) state.

I'm intentionally showing an example where the "action" doesn't have the form `{ type: "INCREMENT_BY", value: 1 }` or some other such thing, because the reducers you create don't *have to* follow the typical patterns from Redux. The world of Hooks is a new world: it's worth considering whether you find old patterns valuable and want to keep them, or whether you'd rather change things up.

A More Complex Example

Let's look at an example with a reducer that does a bit more. We'll create a component to manage a shopping list.

Create a new project if you like, or re-use the one from the last example.

First we need to import two hooks:

```
import React, { useReducer, useRef } from 'react';
```

Then create a component that sets up a ref and a reducer. The ref will hold a reference to a form input, so that we can extract its value. (We could also manage the input with state, passing the value and onChange props as usual, but this is a good chance to show off the useRef hook and save a few lines of code!)

```
const reducer = (state, action) => {
  switch (action.type) {
    // do something with the action
  }
};

function ShoppingList() {
  const inputRef = useRef();
  const [items, dispatch] = useReducer(reducer, []);

  return (
    <>
      <form onSubmit={handleSubmit}>
        <input ref={inputRef} />
      </form>
      <ul>
        {items.map((item, index) => (
          <li key={item.id}>
            {item.name}
          </li>
        ))}
      </ul>
    </>
  );
}
```

Notice that our “state” in this case is an array. We’re initializing it to an empty array (the second argument to `useReducer`) and will be returning an array from the reducer function soon.

We’re also writing the reducer function outside the component. You don’t have to do it this way – you could instead write it inline, and pass it to `useReducer` directly. But putting it outside the component makes it clear that it shouldn’t depend on bits of state or props. Any values the reducer needs should be passed in as part of an action. The other benefit of writing the reducer outside the component is that it can be easier to reuse.

We’ve wrapped the input with a form so that pressing Enter will trigger the submit function. Now we need to write the `handleSubmit` function that will add an item to the list, and we need to handle the action in the reducer.

```

const reducer = (state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name
        }
      ];
    default:
      return state;
  }
};

function ShoppingList() {
  const inputRef = useRef();
  const [items, dispatch] = useReducer(reducer, []);

  function handleSubmit(e) {
    e.preventDefault();
    dispatch({
      type: 'add',
      name: inputRef.current.value
    });
    inputRef.current.value = '';
  }

  return (
    /* ... same ... */
  );
}

```

We’ve filled out the reducer function with two cases: one for when the action has `type === 'add'`, and the `default` case for everything else.

When the reducer gets the “add” action, it returns a new array that includes all the old elements, plus the new element at the end.

We're using the length of the array as a sort of auto-incrementing (ish) ID. This works for our purposes here, but it's not a great idea for a real app because it could lead to duplicate IDs, and bugs. (better to use a library like `uuid` (<https://www.npmjs.com/package/uuid>) or let the server generate a unique ID)

The `handleSubmit` function is called when the user presses Enter in the input box, and so we need to call `preventDefault` to avoid a full page reload when that happens. Then it calls `dispatch` with an action. In this app, we're deciding to give our actions a more Redux-y shape – an object with a `type` property and some associated data. We're also clearing out the input.

Remove an Item

Now let's add the ability to remove an item from the list.

We'll add a "delete" `<button>` next to the item, which will dispatch an action with `type === "remove"` and the index of the item to remove.

Then we just need to handle that action in the reducer, which we'll do by filtering the array to remove the doomed item.

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'add':
      // ... same as before ...
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index !== action.index);
    default:
      return state;
  }
};

function ShoppingList() {
  const inputRef = useRef();
  const [items, dispatch] = useReducer(reducer, []);

  function handleSubmit(e) { /* same */ }

  return (
    <>
```

```

    <form onSubmit={handleSubmit}>
      <input ref={inputRef} />
    </form>
    <ul>
      {items.map((item, index) => (
        <li key={item.id}>
          {item.name}
          <button
            onClick={() => dispatch({ type: 'remove', index })}
          >
            X
          </button>
        </li>
      ))}
    </ul>
  </>
);
}

```

So... is Redux Dead?

Many peoples' first thought upon seeing the useReducer hook went something like... "well, React has reducers built in now, and it has Context to pass data around, so Redux is dead!" I wanted to give some thoughts on that here, because I bet you might be wondering.

I don't think useReducer will kill Redux any more than the Context API killed Redux (it didn't). I *do* think that useReducer further expands React's capabilities in terms of state management, so the cases where you truly need Redux might be... reduced.

Redux provides a *global* store where you can keep app data centralized. useReducer is localized to a specific component. Nothing would stop you from building your own mini-Redux with useReducer and useContext, though!

Redux still does more than Context + useReducer combined – it has the Redux DevTools for nice debugging, and middleware for customizability, and a whole ecosystem of helper libraries. The other big benefit is that Redux includes performance optimizations by default, whereas with the Context API you need to take care to avoid rendering too often.

You can pretty safely argue that Redux is used in plenty of places where it is overkill (including almost every example that teaches how to use it), but I think it still has sticking power.

Exercises

1. Add a button to clear the shopping list. You'll need to dispatch a new action ("clear") and handle it in the reducer.
2. Make a "room" with a light that has 4 levels – off, low, medium, high – and change the level each time you press a button. Create a second button to turn the lights off.
3. Make a "keypad" with 6 buttons that must be pressed in the correct order to unlock it. Each correct button press advances the state. An incorrect button should reset it.

18 The *useEffect* Hook

You’ve learned how adding lifecycle methods to class components allow you to “make things happen” at specific times – say, after a component mounts, or after it re-renders.

The generic name for these actions is “side effects”, a term that comes from functional programming. In the ideal world, you could implement your UI as a pure function of props and state: given a specific set of state *like this*, the app should look *like that*. That’s the core idea behind React, and it works well most of the time.

Sometimes though, you need to do something that doesn’t fit within that box. That could be kicking off a request to fetch data, or focusing an input control when the page loads. Basically, anything that doesn’t fit the paradigm of stateful updates can be handled by a side effect.

With the *useEffect* hook, you can respond to lifecycle events directly inside function components. Namely, three of them: *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount*. All with one function!

Remember that every change to props or state will cause a component to re-render. On every render, *useEffect* will have a chance to run. By default, your effects will execute on every render, but we’ll see how to limit how often they run.

Think of *useEffect* as “if-this-then-that” for your React components.

Let’s create a project where we can play around with *useEffect*. Create an empty project and open up the *index.js* file.

```
$ create-react-app useeffect-hook
$ cd useeffect-hook
$ rm src/*
$ touch src/index.js
```

Then type out this example:

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

const LogEffect = () => {
```

```
const [text, setText] = useState('');

useEffect(() => {
  console.log('latest value:', text);
});

return (
  <input
    value={text}
    onChange={e => setText(e.target.value)}
  />
);
};

ReactDOM.render(
  <LogEffect />,
  document.querySelector('#root')
)
```

Start up the example with `npm start` and open up the console. You'll see that the app logged "latest value:" even though you haven't typed anything into the box! That's because `useEffect` runs after the initial render. It *always* runs after the initial render, and there's no way to turn that off. (this is similar to `componentDidMount`)

Now try typing in the box, and you'll see that it logs a message for every character you type. That's because the effect is running on (after) every render.

Limit When an Effect Runs

Often, you'll only want an effect to run in response to a *specific* change. Maybe when a prop's value changes, or a change occurs to state. That's what the second argument of `useEffect` is for: it's a list of *dependencies*.

Here's an example that says, "If the `blogPostId` prop changes, then download that blog post and display it":

```
useEffect(() => {  
  fetch(`/posts/${blogPostId}`)  
    .then(content =>  
      setContent(content)  
    )  
}, [blogPostId])
```

This one says, “If the username changes, then save it to localStorage”:

```
useEffect(() => {  
  localStorage.setItem('username', username)  
}, [username])
```

This one says, “As soon as the user enters the right passcode, then show the secret”:

```
useEffect(() => {  
  if(passcode === '1234') {  
    setShowSecret(true);  
  }  
}, [passcode])
```

Notice how that code will continue to show the secret even if the user changes the passcode? It works like unlocking a safe. If they type “1234”, the secret will appear. If they then hit backspace and change it to “123”, the secret would still be shown.

Compare that to this version, which would hide the secret if they changed the passcode from “1234” to anything else:

```
useEffect(() => {  
  setShowSecret(passcode === '1234');  
}, [passcode])
```

The key similarity with all of these examples is that they include the array of dependencies as the second argument. This array should contain all the values that, if they change, should cause the effect to be recomputed.

Focusing an Input Automatically

Let's look at how you can focus an input control upon first render, using `useEffect` combined with the `useRef` hook.

```
import React, { useEffect, useState, useRef } from "react";
import ReactDOM from "react-dom";

function App() {
  // Store a reference to the input's DOM node
  const inputRef = useRef();

  // Store the input's value in state
  const [value, setValue] = useState("");

  useEffect(
    () => {
      // This runs AFTER the first render,
      // so the ref is already set.
      console.log("render");
      inputRef.current.focus();
    },
    // The effect "depends on" inputRef
    [inputRef]
  );

  return (
    <input
      ref={inputRef}
      value={value}
      onChange={e => setValue(e.target.value)}
    />
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

At the top, we’re creating an empty ref with `useRef`. Passing it to the input’s `ref` prop takes care of setting it up once the DOM is rendered. And, importantly, the value returned by `useRef` will be stable between renders – it won’t change.

So, even though we’re passing `[inputRef]` as the 2nd argument of `useEffect`, it will effectively only run once, right after the component is mounted. This is basically “`componentDidMount`” (except the precise timing of it, which we’ll talk about later).

Try typing out the example yourself. Notice that the input is auto-focused upon page load. Then try typing in the box. Each character triggers a re-render, but if you look at the console, you’ll see that “render” is only printed once. The `useEffect` is looking at the value of `inputRef` each time, seeing that it hasn’t changed since the previous render, and then deciding not to run your effect function.

Here’s another way to think of this dependency array: it should contain every variable that the effect function uses from the surrounding scope. So if it uses a prop? That goes in the array. If it uses a piece of state? That goes in the array.

Only Run on Mount and Unmount

As we’ve seen, if you pass *no array*, then you’re telling `useEffect` to run every render. What if you pass an *empty array* `[]`?

The empty array says “this effect depends on nothing,” and so it will only run *once*, after the first render.

So if we changed our component above to call `useEffect` like this:

```
useEffect(() => {
  console.log('mounted');
  return () => console.log('unmounting...');
}, []) // <-- add this empty array here
```

Then it will print “mounted” after the initial render, remain silent throughout its life, and print “unmounting...” on its way out.

This comes with a warning, though: passing the empty array is prone to bugs. It’s easy to forget to add an item to it if you add a dependency, and if you *miss* a dependency, then that value will be stale the next time `useEffect` runs and it might cause some strange problems. Just make sure that when you say empty array `[]`, you really mean it.

Unmount and Cleanup

In that last example, we're returning a function from the effect, and that function will be called to clean up the effect.

Not every effect will need to clean up after itself, but some do. Effects that start a timer or interval, or start a request that needs to be cancelled, or add an event listener that needs to be removed are all examples of times when you'll want to return the cleanup function.

Cleanup always happens when the component is unmounted, and it'll also happen every time before the effect runs. That's useful if you, say, want to subscribe to some events based on a prop, but then re-subscribe when that prop changes:

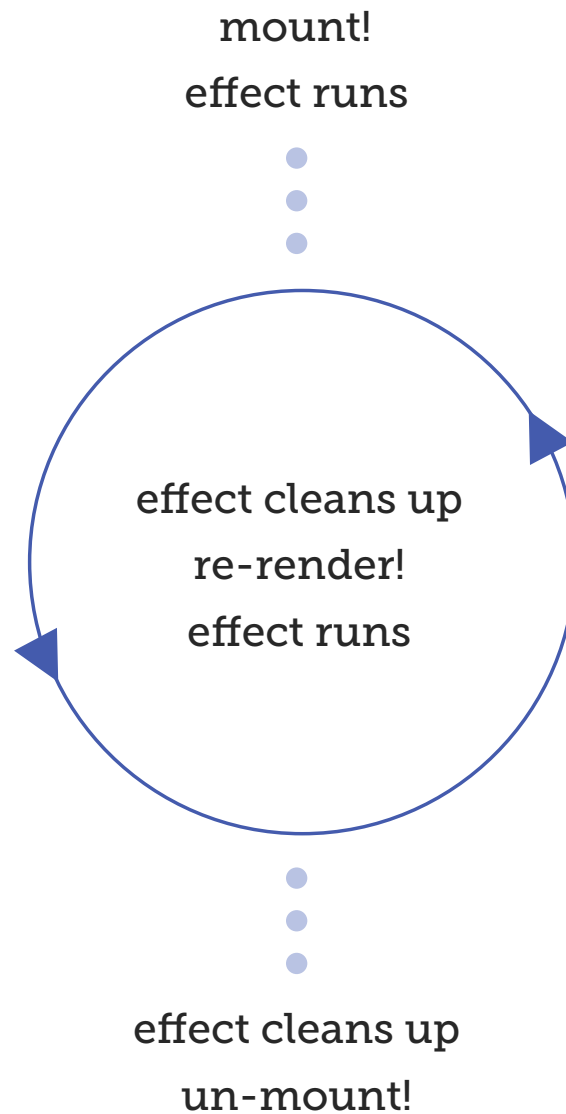
```
useEffect(() => {  
  // on mount, and every time the blogPostId  
  // changes, set up a subscription  
  subscribeToNewComments(blogPostId);  
  
  // on unmount, and every time before  
  // subscribing anew, unsubscribe from  
  // the last blogPostId  
  return () => unsubscribeFromComments(blogPostId);  
}, [blogPostId])
```

This is a nice clean way of keeping subscriptions in sync with a certain piece of data.

You might think that there's a bug, that it will unsubscribe from the *new* blogPostId. But that's not what happens, because the function being returned is a *closure* that latches on to the value of blogPostId at the time it is created.

If, during the first time through this effect the blogPostId is 42, then it will simultaneously subscribe to comments for post 42, and return a function that will *later* be able to unsubscribe from comments for post 42. The returned function holds on to the value 42.

Here's how the “cycle of effects” works, between the initial render, subsequent re-renders, and finally being unmounted:



Fetch Data With `useEffect`

Let's look at another common use case: fetching data and displaying it. In a class component, you'd put this code in the `componentDidMount` method. To do it with hooks, we'll pull in `useEffect`. We'll also need `useState` to store the data.

It's worth mentioning that when the data-fetching portion of React's new Suspense feature is ready, that'll be the preferred way to fetch data. Fetching from `useEffect` has one big gotcha (which we'll go over) and the Suspense API is going to be much easier to use. For now, though, `useEffect` gets the job done!

Here's a component that fetches posts from Reddit and displays them:

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

function Reddit() {
  // Initialize state to hold the posts
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    // Fetch the data when the component mounts
    fetch("https://www.reddit.com/r/reactjs.json")
      .then(res => res.json())
      .then(json => {
        // Save the posts into state
        setPosts(json.data.children.map(c => c.data))
      })
  }); // <-- we didn't pass the 2nd arg. what will happen?

  // Render as usual
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

ReactDOM.render(
  <Reddit />,
  document.querySelector("#root")
);
```

You'll notice that we aren't passing the second argument to `useEffect` here. This is bad. Don't do this.

Passing no 2nd argument causes the `useEffect` to run every render. Then, when it runs, it fetches the data and later *updates the state*. Once the state is updated, the component re-renders, which triggers the `useEffect` again. You can see the problem: we're going to end up with an infinite loop!

To fix this, we need to pass an array as the 2nd argument. What should be in the array?

Go ahead, think about it for a second.

...

...

...

The only variable that `useEffect` depends on is `setPosts`. Therefore we should pass the array `[setPosts]` here. Because `setPosts` is a setter returned by `useState`, it won't be recreated every render, and so the effect will only run once.

Fun fact: When you call `useState`, the setter function it returns is only created once! It'll be the exact same function instance every time the component renders, which is why it's safe for an effect to depend on one. This fun fact is also true for the `dispatch` function returned by `useReducer`.

Re-fetch When Data Changes

Let's expand on the example to cover another common problem: how to re-fetch data when something changes, like a user ID, or in this case, the name of the subreddit.

First we'll change the `Reddit` component to accept the subreddit as a prop, fetch the data based on that subreddit, and only re-run the effect when the prop changes:

```
// 1. Destructure the `subreddit` from props:
function Reddit({ subreddit }) {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
```

```

    // Fetch the data when the component mounts
    fetch(
      `https://www.reddit.com/r/${subreddit}.json`
    )
      .then(res => res.json())
      .then(json =>
        // Save the posts into state
        setPosts(json.data.children.map(c => c.data))
      )
  }, [subreddit, setPosts]);

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

// 4. Pass "reactjs" as a prop:
ReactDOM.render(
  <Reddit subreddit='reactjs' />,
  document.querySelector("#root")
);

```

This is still hard-coded, but now we can customize it by wrapping the `Reddit` component with one that lets us change the subreddit. Add this new `App` component, and render it at the bottom:

```

function App() {
  // 2 pieces of state: one to hold the input value,
  // another to hold the current subreddit.
  const [inputValue, setValue] = useState("reactjs");
  const [subreddit, setSubreddit] = useState(inputValue);

  // Update the subreddit when the user presses enter
  const handleSubmit = e => {
    e.preventDefault();

```

```

    setSubreddit(inputValue);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <input
          value={inputValue}
          onChange={e => setValue(e.target.value)}
        />
      </form>
      <Reddit subreddit={subreddit} />
    </>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));

```

The app is keeping 2 pieces of state here – the current input value, and the current subreddit. Submitting the input will “commit” the subreddit, which will cause `Reddit` to re-fetch the data from the new selection. Wrapping the input in a form allows the user to press Enter to submit.

Type carefully! There’s no error handling. If you type a subreddit that doesn’t exist, the app will blow up. (You’ll add some error handling in the exercises!)

We could’ve used just 1 piece of state here – to store the input, and send the same value down to `Reddit` – but then the `Reddit` component would be fetching data with every keypress.

The `useState` at the top might look a little odd, especially the second line:

```

const [inputValue, setValue] = useState("reactjs");
const [subreddit, setSubreddit] = useState(inputValue);

```

We’re passing an initial value of “reactjs” to the first piece of state, and that makes sense. That value will never change.

But what about that second line? What if the initial state *changes*? (and it will, when you type in the box)

Remember that `useState` is stateful. It only uses the initial state *once*, the first time it renders. After that it's ignored. So it's safe to pass a transient value, like a prop that might change or some other variable.

Making Visible DOM Changes

The `useEffect` function is like the swiss army knife of hooks. It can be used for a ton of things, from setting up subscriptions to creating and cleaning up timers to changing the value of a ref.

One thing it's *not* good for is making DOM changes that are visible to the user. The way the timing works, an effect function will only fire *after* the browser is done with layout and paint – too late, if you wanted to make a visual change.

For those cases, React provides the `useLayoutEffect` hook. It works the same as `useEffect` in terms of the arguments it takes. The only difference is that it will run at the same time as `componentDidMount` would have – that is, it runs synchronously between when browser has updated the DOM and before those changes are painted to the screen.

Most of the time, `useEffect` is the one you want. And because `useEffect` runs after layout and paint, a slow effect won't make the UI janky.

But if your effect needs to measure DOM elements or change them in some visible way, then write that in a `useLayoutEffect`.

Exercises

1. Render an input box and store its value with `useState`. Then set the `document.title` in an effect, keeping the page's title in sync with the input.
2. Add a click handler to the document, and print a message every time the user clicks. (don't forget to clean up the handler!)
3. The Reddit example from this chapter is lacking error handling, and if you enter an invalid subreddit name, the app will break. Add code to intercept errors, handle them gracefully, and display an error message.

19 The Context API

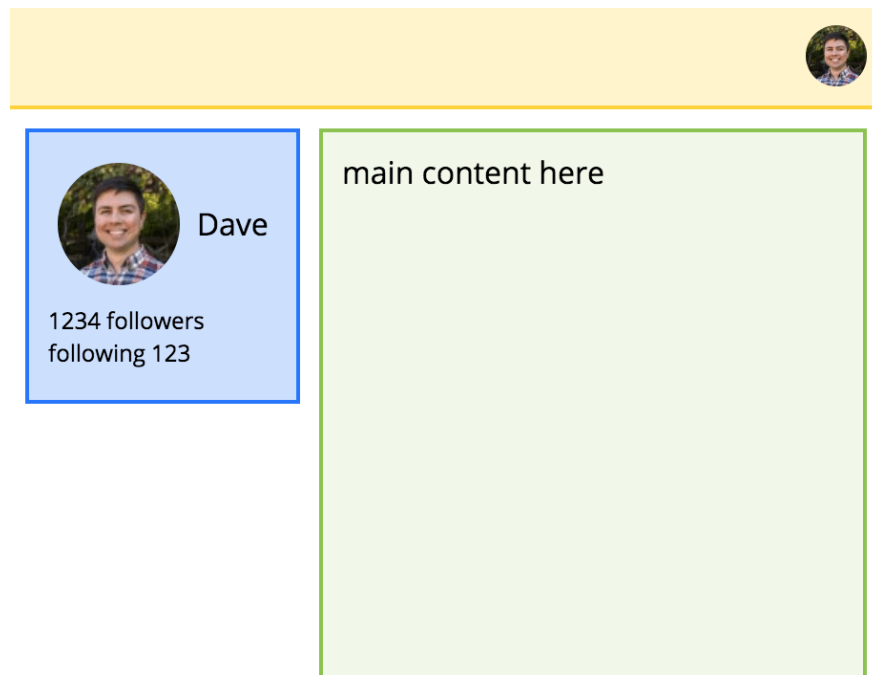
React 16.3 added a new Context API – *new* in the sense that the *old* context API was a behind-the-scenes feature that most people either didn't know about, or avoided using because the docs said to avoid using it.

Today, though, the Context API is a first-class citizen in React, open to all (not that it wasn't before, but it's, like, official now).

The purpose of Context is to make it easier to pass deeply-nested props. Context helps work around the situation sometimes called “prop drilling” where one component high up in the tree needs to send data down to a grandchild (or great-great-great grandchild) by threading props through a bunch of intermediate components.

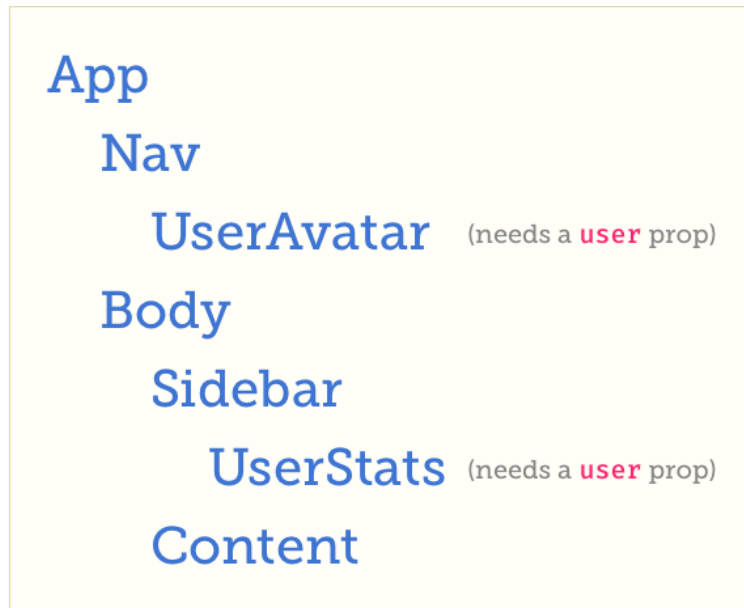
Before: A Prop Drilling Example

Let's look at an example. We'll start with a plain React version of the app (no Context) and then add Context to see how it helps.



This app has the user's information displayed in two places: in the nav bar at the top-right, and in the sidebar next to the main content.

The component structure looks like this:



We need to store the user's info high enough in the tree that it can be passed down to the components that need it. The data needs to be kept at least *one level above* all of the components that need access to it. In this case, the keeper of user info has to be App.

In order to get the user info down to the components that need it, App needs to pass it along to Nav and Body. They, in turn, need to pass it down *again*, to UserAvatar (hooray!) and Sidebar. Finally, Sidebar has to pass it down to UserStats.

Let's look at how this looks in code.

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

const UserAvatar = ({ user, size }) => (
  <img
    className={`user-avatar ${size || ''}`}
    alt="user avatar"
    src={user.avatar}
  />
);
```



```

const UserStats = ({ user }) => (
  <div className="user-stats">
    <div>
      <UserAvatar user={user} />
      {user.name}
    </div>
    <div className="stats">
      <div>{user.followers} Followers</div>
      <div>Following {user.following}</div>
    </div>
  </div>
);

const Nav = ({ user }) => (
  <div className="nav">
    <UserAvatar user={user} size="small" />
  </div>
);

const Content = () => (
  <div className="content">main content here</div>
);

const Sidebar = ({ user }) => (
  <div className="sidebar">
    <UserStats user={user} />
  </div>
);

const Body = ({ user }) => (
  <div className="body">
    <Sidebar user={user} />
    <Content user={user} />
  </div>
);

class App extends React.Component {
  state = {
    user: {

```

```

    avatar:
      'https://www.gravatar.com/avatar/5c3dd2d257ff0e14dbd2583485dbd44b',
    name: 'Dave',
    followers: 1234,
    following: 123
  }
};

render() {
  const { user } = this.state;

  return (
    <div className="app">
      <Nav user={user} />
      <Body user={user} />
    </div>
  );
}
}

ReactDOM.render(
  <App />,
  document.querySelector('#root')
);

```

Here, App initializes the state to contain the “user” object as static data. In a real app this data might come from a server. (And of course, the App component could just as easily be written with hooks and `useState`.)

I want to be clear that “prop drilling” is not discouraged by any means; it’s a perfectly valid pattern and core to the way React works. But deep drilling can be a bit annoying to write, and the annoyance is amplified when you have to pass down a lot of props instead of just one.

There’s a bigger downside to this prop drilling strategy though: it creates coupling between components that would otherwise be decoupled. In the example above, Nav needs to accept a “user” prop and pass it down to UserAvatar, even though Nav does not have any need for the user otherwise.

Tightly-coupled components that accept and forward props are more difficult to reuse because whenever you transplant one to a new location, you’ve gotta wire it up with its parents (and potentially, ancestors even further up the tree).

Let’s look at how we might improve it.

The “Slots” Pattern

If you can find a way to *coalesce* your app structure and take advantage of the `children` prop, it can lead to cleaner code without having to resort to deep prop drilling, or Context, or anything else.

The `children` prop is a great solution for components that need to be generic placeholders, like `Nav`, `Sidebar`, and `Body` in this example. Be aware, too, that you can pass JSX elements into *any* prop, not just the one named “children” – so if you need more than one “slot” to plug components into, keep that in mind.

Here’s the same example as above, rewritten so that `Nav` and `Sidebar` accept a `children` prop and render it as-is. Notice the `Body` component too – it doesn’t take a prop named “children” but it has two “slots” of sorts that it renders to the page.

Written this way, the top-level `App` component can simply render what it needs to, using the data it already has in scope, without having to pass data down more than one level.

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

const UserAvatar = ({ user, size }) => (
  <img
    className={`user-avatar ${size || ''}`}
    alt="user avatar"
    src={user.avatar}
  />
);

const UserStats = ({ user }) => (
  <div className="user-stats">
    <div>
```

```

    <UserAvatar user={user} />
    {user.name}
  </div>
  <div className="stats">
    <div>{user.followers} Followers</div>
    <div>Following {user.following}</div>
  </div>
</div>
);

// Accept children and render it/them
const Nav = ({ children }) => (
  <div className="nav">{children}</div>
);

const Content = () => (
  <div className="content">main content here</div>
);

const Sidebar = ({ children }) => (
  <div className="sidebar">{children}</div>
);

// Body needs a sidebar and content, but written this way,
// they can be ANYTHING
const Body = ({ sidebar, content }) => (
  <div className="body">
    <Sidebar>{sidebar}</Sidebar>
    {content}
  </div>
);

class App extends React.Component {
  state = {
    user: {
      avatar:
        'https://www.gravatar.com/avatar/5c3dd2d257ff0e14dbd2583485dbd44b',
      name: 'Dave',
      followers: 1234,
    },
  },

```

```

        following: 123
      }
    };

    render() {
      const { user } = this.state;

      return (
        <div className="app">
          <Nav>
            <UserAvatar user={user} size="small" />
          </Nav>
          <Body
            sidebar={<UserStats user={user} />}
            content={<Content />}
          />
        </div>
      );
    }
  }

ReactDOM.render(
  <App />,
  document.querySelector("#root")
);

```

If your app is more complex, though, it might be tough to figure out how to adapt the children pattern this way.

Let's see how you might replace the prop drilling with the Context API.

Using the React Context API

Context is like an electrical bus running behind every component: to receive the power (data) passing through it, you need only plug in. (fun fact: if you've used Redux and wondered how it passes data around behind the scenes, Context is the secret sauce!)

There are 3 important pieces to the Context API:

- The `React.createContext` function creates the context
- The `Provider` (returned by `createContext`) establishes the “electrical bus” running through a component subtree
- The `Consumer` (also returned by `createContext`) taps into the “electrical bus” to extract the data

The `Provider` accepts a `value` prop which can be whatever you want. It’ll most likely be an object containing your data and any actions you want to be able to perform on the data.

The `Consumer` taps into the data and makes it available to the component that needs it.

Here’s the same example rewritten to use Context to pass the data around.

```
import React from "react";
import ReactDOM from "react-dom";
import "./styles.css";

// First, we create a new context.
// createContext returns an object with 2 properties:
//   { Provider, Consumer }

// We'll use the Provider and Consumer below,
// but rather than pull them out individually,
// we can store the whole object in a variable.
// As long as it's named with UpperCase, we can use
// its properties in JSX expressions.
const UserContext = React.createContext();

// Components that need the data can tap into the context
// by rendering the Consumer. It uses the "render props"
// pattern -- rendering a function as a child (more on that below).
const UserAvatar = ({ size }) => (
  <UserContext.Consumer>
    {user => (
      <img
        className={`user-avatar ${size} || ""`}
        alt="user avatar"
        src={user.avatar}
      />
    )}
  />
)
```

```

    />
  })
</UserContext.Consumer>
);

// Notice that we don't need the 'user' prop any more!
// The Consumer fetches the user from context
const UserStats = () => (
  <UserContext.Consumer>
    {user => (
      <div className="user-stats">
        <div>
          <UserAvatar user={user} />
          {user.name}
        </div>
        <div className="stats">
          <div>{user.followers} Followers</div>
          <div>Following {user.following}</div>
        </div>
      </div>
    )}
  </UserContext.Consumer>
);

// The components that once had to launder the `user` prop
// are now nice and simple.

const Nav = () => (
  <div className="nav">
    <UserAvatar size="small" />
  </div>
);

const Content = () => (
  <div className="content">
    main content here
  </div>
);

```

```
const Sidebar = () => (  
  <div className="sidebar">  
    <UserStats />  
  </div>  
);  
  
const Body = () => (  
  <div className="body">  
    <Sidebar />  
    <Content />  
  </div>  
);  
  
// Inside App, we make the context available  
// using the Provider  
class App extends React.Component {  
  state = {  
    user: {  
      avatar:  
        "https://www.gravatar.com/avatar/5c3dd2d257ff0e14dbd2583485dbd44b",  
      name: "Dave",  
      followers: 1234,  
      following: 123  
    }  
  };  
  
  render() {  
    return (  
      <div className="app">  
        <UserContext.Provider value={this.state.user}>  
          <Nav />  
          <Body />  
        </UserContext.Provider>  
      </div>  
    );  
  }  
}
```

Let's go over how this works.

Remember there's 3 pieces: the context pair itself (the object returned by `React.createContext`), and the two components that talk to it (Provider and Consumer).

Provider and Consumer are a Pair

The Provider and Consumer are bound together. Inseparable. And they only know how to talk to *each other*. If you created two separate contexts, say "Context1" and "Context2", then Context1's Provider and Consumer would not be able to communicate with Context2's Provider and Consumer.

Context Holds No State

An important fact of Context is that it *does not hold any of its own state*. It is merely a conduit for your data. Think of it like passing props (because that's what it is!). You're not so much putting data *into* Context as you are passing data *through* Context.

When you pass a value prop to the Provider, that exact value gets passed down to any Consumers that know how to look for it (only the Consumers that are bound to the same context as the Provider), and the Provider doesn't "save" that value anywhere. To change the value, just pass something new into the value prop.

The Default Value

When you create the context, you can pass in a "default value" like this:

```
const Ctx = React.createContext(yourDefaultValue);
```

This default value is what the Consumer will receive when it is placed in a tree with no Provider above it. If you don't pass one, the default will just be undefined.

This is a *default* value, not an *initial* value. The difference is subtle, but important. I'll say it one more time: a context doesn't *retain* anything; it merely distributes the data you pass as the value prop on the Provider.

The “Render Props” Pattern

The context Consumer expects its child to be a single function. This is called the “render props” pattern:

```
<UserContext.Consumer>
  {user => (
    <div>{user.name}</div>
  )}
</UserContext.Consumer>
```

The name “render props” comes from the idea that one of the *props* you’re passing (the `children` prop, in this case) is a function that is being called to *render* the output.

This syntax will look weird if you’re not used to it. If you stare at it for a second, you’ll realize that the arrow function inside the braces `{user => (...)}` is a JavaScript expression, the same kind we’ve seen intermixed with JSX throughout this book.

Notice, too, that the render prop function isn’t called immediately. This delays execution until the `UserContext.Consumer` decides to call that function, which means that it can effectively “wait” until the user is available.

The render props pattern is nice because it lets you say, “I know what I want to render here, but I don’t have all the data yet.” Passing that function lets *you* decide what the output will be, but lets the *component* (`UserContext.Consumer` in this case) decide when and where to render that output.

The Consumer will call your function at render time, passing in the value that it got from the Provider somewhere above it (or the context’s default value, or `undefined` if you didn’t pass a default).

If you forget to pass a child function to the Consumer, and instead pass regular JSX, you’ll get an error saying “render is not a function”.

If you need to wrap the content with some other element, put those wrapping elements outside and around the Consumer, instead of inside. You don’t need to worry about the Consumer introducing any wrapper divs or anything. It’s invisible as far as the DOM structure is concerned.

Provider Accepts One Value

Provider takes just a single value, as the `value` prop. But remember that the value can be anything. In practice, if you want to pass multiple values down, you'll create an object with all the values and pass down that object.

That's pretty much the nuts and bolts of the Context API.

Advanced Context Patterns

Since creating a context gives us two components to work with (Provider and Consumer), we're free to use them however we want. Here are a couple ideas.

Turn the Consumer into a Higher-Order Component

Not fond of the idea of adding the `UserContext.Consumer` around every place that needs it? Well, it's your code! You can do what you want.

If you'd rather receive the value as a prop, you could write a little wrapper around the Consumer like this:

```
function withUser(Component) {
  return function ConnectedComponent(props) {
    return (
      <UserContext.Consumer>
        {user => <Component {...props} user={user}/>}
      </UserContext.Consumer>
    );
  }
}
```

And then you could rewrite, say, `UserAvatar` to use this new `withUser` function:

```
const UserAvatar = withUser(({ size, user }) => (
  <img
    className={`user-avatar ${size || ""}`}

```

```

    alt="user avatar"
    src={user.avatar}
  />
));

```

The `withUser` function here is following the “higher-order component” pattern (a.k.a. HoC): it takes a component as an argument and wraps it up by returning a new component (the `ConnectedComponent` function here). That new component can do whatever you want – inject new props, change the value of props before passing them along, or whatever else. Principally, it should render the Component that was passed to the HoC (`withUser`, here), but beyond that it can do whatever you want.

The higher-order component pattern was very popular for a while, and then fell out of favor. It’s used less often in new code these days, because most people opt to write a custom hook instead of introducing the extra level of nesting that comes with an HoC.

I’m explaining it here because you might see the pattern in code you come across, and because it can still be a useful pattern sometimes. I think the HoC pattern also does a good job demonstrating the idea that components are real actual JavaScript functions and can be composed in the same ways.

It can be helpful sometimes to take a step back and look at React code through the lens of “regular JavaScript code.” Looking at the code this way helps avoid the feeling that you must follow “React patterns” in everything you do (you don’t!). Remember that “React code” is really 99% JavaScript at the end of the day.

Hold State in the Provider

The context’s Provider is just a conduit, remember. It doesn’t retain any data. But that doesn’t stop you from making your *own* wrapper to hold the data.

In the example above, I left `App` holding the data, so that the only new thing you’d need to understand was the Provider + Consumer components. But maybe you want to make your own “store” of sorts. You could create a component to hold the state and pass it through context:

```

class UserStore extends React.Component {
  state = {
    user: {
      avatar:

```

```

        "https://www.gravatar.com/avatar/5c3dd2d257ff0e14dbd2583485dbd44b",
        name: "Dave",
        followers: 1234,
        following: 123
      }
    };

    render() {
      return (
        <UserContext.Provider value={this.state.user}>
          {this.props.children}
        </UserContext.Provider>
      );
    }
  }

  // ... other components are unchanged ...

const App = () => (
  <div className="app">
    <Nav />
    <Body />
  </div>
);

ReactDOM.render(
  <UserStore>
    <App />
  </UserStore>,
  document.querySelector("#root")
);

```

Now your user data is nicely contained in its own component whose *sole* concern is user data. Awesome. App can be stateless once again. I think it looks a little cleaner, too.

The useContext Hook

One drawback of the Context API is how the Consumer adds a lot of nesting with its render props pattern. It's especially noticeable if a component needs to pull data from a few different contexts:

```
const Something = () => (
  <UserContext.Consumer>
    {user => (
      <LanguageContext.Consumer>
        {lang => (
          <ThemeContext.Consumer>
            {theme => (
              <div>ugh</div>
            )}
          </ThemeContext.Consumer>
        )}
      </LanguageContext.Consumer>
    )}
  </UserContext.Consumer>
);
```

With the introduction of React Hooks, we can simplify this a ton by using the `useContext` hook to access the data. Here's the same component:

```
const Something = () => {
  const user = useContext(UserContext);
  const lang = useContext(LanguageContext);
  const theme = useContext(ThemeContext);

  return <div>yay!</div>;
};
```

The only thing to watch out for is that `useContext` requires that you pass it the *entire context object*, not just the context's Consumer. Other than that, there's nothing else to it. The `useContext` hook can be used anywhere you'd normally use the Consumer.

Pass Actions Down Through Context

Remember that the object being passed down through the Provider can contain whatever you want. Which means it can contain functions. You might even call them “actions.”

Here’s another example: a simple Room with a lightswitch to toggle the background color – err, I mean lights.

The state is kept in the store, which also has a function to toggle the light. Both the state and the function are passed down through context. This time I’m using hooks, just to show that Context works with both hooks and classes.

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

// Plain empty context
const RoomContext = React.createContext();

// A component whose sole job is to manage
// the state of the Room
function RoomStore({ children }) {
  const [isLit, setLit] = useState(false);

  const toggleLight = () => {
    setLit(!isLit);
  };

  // Pass down the state and the onToggleLight action
  return (
    <RoomContext.Provider
      value={{
        isLit,
        onToggleLight: toggleLight
      }}
    >
      {children}
    </RoomContext.Provider>
  );
}
```

```

// Receive the state of the light, and the function to
// toggle the light, from RoomContext
const Room = () => {
  const { isLit, onToggleLight } = useContext(RoomContext);

  return (
    <div className={`room ${
      isLit ? 'lit' : 'dark'
    }`} >
      The room is {isLit ? 'lit' : 'dark'}.
      <br />
      <button onClick={onToggleLight}>
        Flip
      </button>
    </div>
  );
};

const App = () => (
  <div className="app">
    <Room />
  </div>
);

// Wrap the whole app in the RoomStore
// (you could also do this inside `App`)
ReactDOM.render(
  <RoomStore>
    <App />
  </RoomStore>,
  document.querySelector('#root')
);

```


Type out this example! Click the button and watch the “lights” turn on and off.

20 Example: Shopping Site

In this example we'll build a simple shopping site that will demonstrate some of the things we've covered.

Here's what it will look like when we're done. It'll have a page full of items to buy:


Items Cart


	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 Add to Cart
---	---	--------------------------------------

	Apple iPad Mini 2 32GB Even larger than the 16GB.	\$279 Add to Cart
---	---	--------------------------------------

And it will have a shopping cart showing the selected items, with +/- buttons and a count for each one:

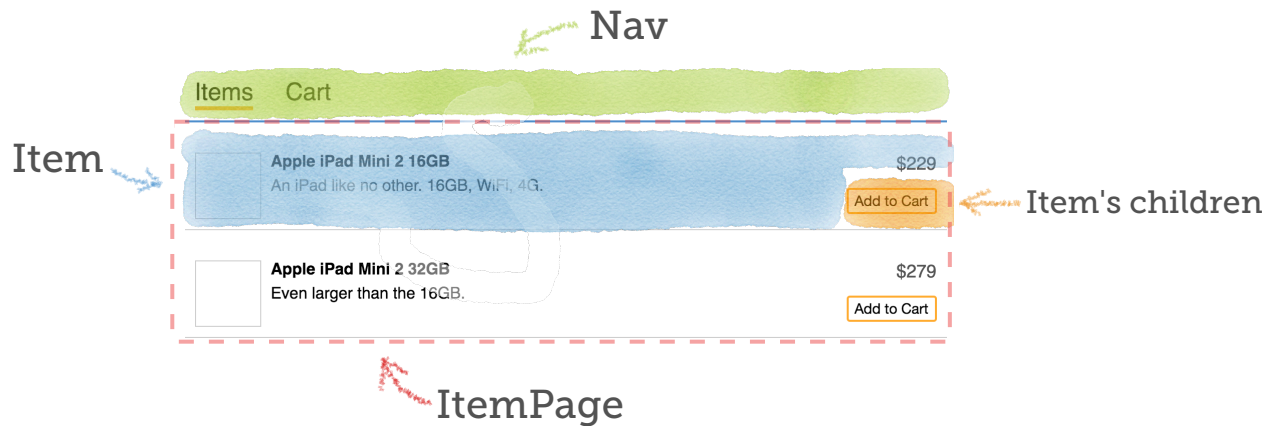
Items Cart

	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 <div>- 3 +</div>
---	---	---------------------------

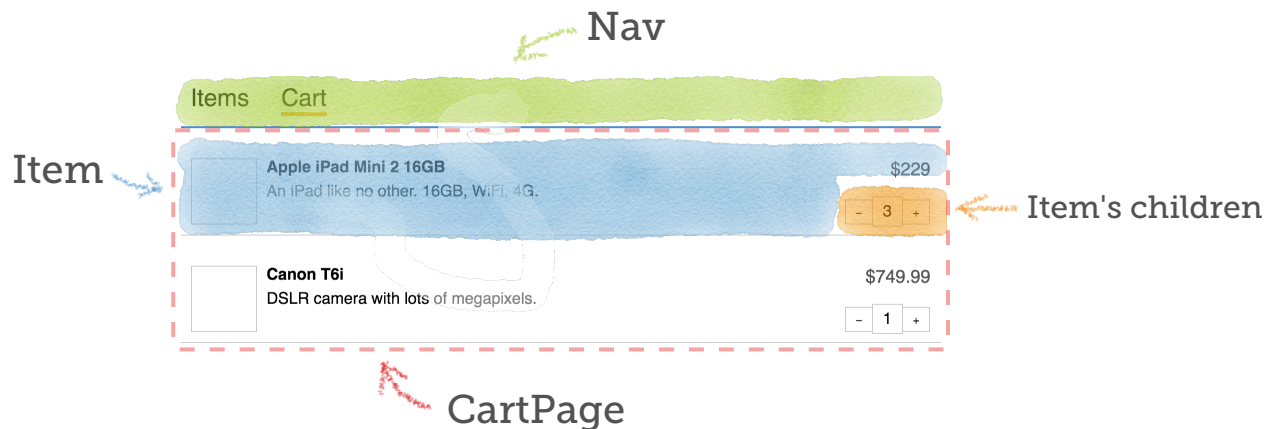
	Canon T6i DSLR camera with lots of megapixels.	\$749.99 <div>- 1 +</div>
---	--	------------------------------

Break Into Components

Following the same steps as before, let's start by breaking down the app into its components and giving them names. Here is the Items page:



And here is the Cart page:



There's also a top-level "App" component which contains everything else.

You'll notice that the two pages are very similar. Compared to previous examples, this one will have less-granular components. The components will have more "stuff" in them. You can always break things down later, and sometimes it's better to avoid abstracting into tiny pieces early on.

Here are the components we'll build:

- App
 - ItemPage
 - CartPage
 - Item
 - Nav

Start Building

We'll start by generating a new project, and leave the generated files in place this time.

```
$ create-react-app shopper && cd shopper
```

Open up `src/App.js` and replace its contents with this to get started:

```
import React from 'react';
import Nav from './Nav';
import './App.css';

const App = () => {
  return (
    <div className="App">
      <Nav />
      <main className="App-content">
        <span>Empty</span>
      </main>
    </div>
  );
};

export default App;
```

Then create `src/Nav.js` and start it off with this:

```
import React from 'react';

const Nav = () => (
  <nav className="App-nav">
    <ul>
      <li className="App-nav-item">
        <button>Items</button>
      </li>
      <li className="App-nav-item">
        <button>Cart</button>
      </li>
    </ul>
  </nav>
);

export default Nav;
```

Nav items would normally be links with an anchor tag like `Items`. In this app, though, we're going to stay at the `/` route, so it doesn't make sense to make these links. We *could* make them `<a>` tags without an `href` attribute, but since that's not good for accessibility (and Create React App will warn you if you do it), we're using `button` tags here instead. We will style the buttons as links with CSS in just a second.

Start up the app and see what we have:

```
$ npm start
```

-
-

Empty

It's working, but it needs styling. Open up `src/App.css`, and replace the contents with this:

```
.App {
  max-width: 800px;
  margin: 0 auto;
}
.App-nav {
  margin: 20px;
  padding: 10px;
  border-bottom: 2px solid #508fca;
}
.App-nav ul {
  padding: 0;
  margin: 0;
}
.App-nav button {
  font-size: 18px;
  background: transparent;
  border: none;
  cursor: pointer;
}
.App-nav-item {
  list-style: none;
  display: inline-block;
  margin-right: 32px;
  font-size: 24px;
  border-bottom: 4px solid transparent;
}
.App-content {
  margin: 0 20px;
}
```

That's a little better:

Items Cart

Empty

Next, we need to keep track of the active tab, and choose whether to render `ItemPage` or `CartPage`. To do this, we'll store the active tab in state, and make the `Nav` trigger a state change. The `App` component is the best home for this state, since `App` will need it to know which page to render.

Import the `useState` hook, and create the `activeTab` state at the top of the `App` component. We'll also pass the `activeTab` and `setActiveTab` function into the `Nav` component so that `Nav` will be able to highlight the active tab and change it.

```
import React, { useState } from 'react';
import Nav from './Nav';
import './App.css';

const App = () => {
  const [activeTab, setActiveTab] = useState('items');

  return (
    <div className="App">
      <Nav
        activeTab={activeTab}
        onTabChange={setActiveTab}
      />
      <main className="App-content">
        <span>Empty</span>
      </main>
    </div>
  );
};
```

Now we'll work on replacing the "Empty"-ness with the actual content, depending on the selected tab.

Below the App component, in the same file, create a Content component that will decide what to render. Then replace the `Empty` with this new component, passing in the active tab that it needs.

```
const App = () => {
  const [activeTab, setActiveTab] = useState('items');

  return (
    <div className="App">
      <Nav
        activeTab={activeTab}
        onTabChange={setActiveTab}
      />
      <main className="App-content">
        <Content tab={activeTab} />
      </main>
    </div>
  );
};

// add this:
const Content = ({ tab }) => {
  switch (tab) {
    default:
    case 'items':
      return <span>the items</span>;
    case 'cart':
      return <span>the cart</span>;
  }
};
```

Update the Nav component in `src/Nav.js` to use the new props. We're going to make the classNames dynamic so that we can highlight the active tab, and to keep the code tidy, we'll do that computation in its own function.

Because we're adding another statement to the Nav function, we need to change it into the long-form arrow function with braces and a return.

```

const Nav = ({ activeTab, onTabChange }) => {
  const itemClass = tabName =>
    `App-nav-item ${
      activeTab === tabName ? 'selected' : ''
    }`;

  return (
    <nav className="App-nav">
      <ul>
        <li className={itemClass('items')}>
          <button onClick={() => onTabChange('items')}>
            Items
          </button>
        </li>
        <li className={itemClass('cart')}>
          <button onClick={() => onTabChange('cart')}>
            Cart
          </button>
        </li>
      </ul>
    </nav>
  );
};

```

Looking at the code factored this way, you might notice that the `` items look very similar. Could you extract them into a `NavItem` component? Give it a try! Do you think that makes the code easier to read, or harder?

Then add some styles to the bottom of `src/App.css`:

```

.App-nav-item button:hover {
  color: #999;
}
.App-nav-item.selected {
  border-bottom-color: #ffaa3f;
}

```


Here's what it should look like now:

Items Cart

Cart

Arrow Function onClick

You'll notice that we're passing an arrow function to the `onClick` handlers here. At first glance this may look strange. You might think `onClick={onTabChange('items')}` looks better.

But remember, the values of props are evaluated *before* they're passed down. When written without the arrow function, `onTabChange` would be called *when Nav renders*, instead of when the link is clicked. That's not what we want. Wrapping it in an arrow function delays execution until the user clicks the link.

You're bound to forget to wrap a click handler in an arrow function in one of your own projects some day. I've done it many times, and I *still* do it sometimes.

If you ever find yourself wondering, "Why is this handler running early?!" – check how it's being passed. This mistake can even lead to an infinite loop, if your handler causes a state change that causes a re-render that causes another state change...

Is It Bad to Create Functions During Render?

You may have heard that creating functions on every render is bad. Some people will say that you should not do this:

```
<button onClick={() => doStuff()}>Do Stuff</button>
```

And that instead, you should try to write code like this:

```
<button onClick={doStuff}>Do Stuff</button>
```

The reasoning goes that creating functions on every render causes a performance problem.

Let's talk about where this concern comes from and why you don't need to worry in most cases.

Optimizing Components

React components, by default, will re-render every time their internal state is changed or their parent re-renders. This re-render happens even if the props being passed to the component are exactly the same.

An easy way to optimize a component, then, is to skip re-rendering it when the props are exactly the same as the last render. (A state change will always cause a re-render though, no matter what.)

There are a few ways to write this optimization.

In a class component, you can implement the `shouldComponentUpdate(prevProps)` function to check the previous props against the current props manually. If you return `false` from this function, the component won't be rendered.

Alternatively, you can change the class to extend `React.PureComponent` instead of `React.Component`, and that will automatically check the previous props against the current props.

For function components, you can wrap them in a call to `React.memo` like this:

```
const Message = React.memo(({ text }) => (  
  <div>{text}</div>  
));
```

That achieves the same result as extending `React.PureComponent`. It'll check each prop against its previous value, and skip the render if they all match.

In the Nav example above, the `<button>` elements receiving the `onClick` prop aren't even React *components*. They're just plain old React elements. So optimization is moot. Plain DOM nodes will render extremely fast, no matter what you pass them.

A Re-created Function Won't Match The Last One

So, back to the concern about creating functions during render.

Even though the actual creation of functions is very fast, the side effect of it is that “Pure” components will be forced to re-render every time. If you *did* put those optimizations in place, then passing a new function to that component every time it renders will break the optimization.

That sounds bad! So why is it usually okay to create functions during render?

First, if the component isn't optimized, then it's going to re-render whether you pass it brand new functions or not. Since components aren't optimized by default, it's less likely to be a concern.

Second, though – and this is the important one – rendering is *fast* in most cases. React is fast by default.

Even if a component has a lot of nested JSX to render, that'll be pretty quick. Slowdowns come when you're rendering long lists of items, or triggering renders very frequently (such as every frame of an animation, or every time the mouseMove event occurs). When you notice your app feels sluggish, that's a great time to look into which components are causing it and apply optimizations where needed.

Create ItemPage

Let's turn our attention back to the app, and to the list of items for sale. We'll create a new component to display this data. Create a file called `src/ItemPage.js` and type this in:

```
import React from 'react';
import PropTypes from 'prop-types';
import './ItemPage.css';

function ItemPage({ items }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          {item.name}
        </li>
      )}
    </ul>
  )
}
```

```
    </ul>
  );
}
ItemPage.propTypes = {
  items: PropTypes.array.isRequired
};

export default ItemPage;
```

Since we're using `PropTypes`, you'll need to install that package in this project with `npm install prop-types`.

Here we're just iterating over a list of items and rendering them out. Since the individual items will eventually be more involved than just a "name", we'll extract that out into its own component shortly.

This component relies on `ItemPage.css`, which doesn't exist yet. Create that file, and add this CSS to style the list:

```
.ItemPage-items {
  margin: 0;
  padding: 0;
}
.ItemPage-items li {
  list-style: none;
  margin-bottom: 20px;
}
```

We're going to need some items to sell. We'll use static data here, same as before.

I like to start with static data even when I have a server that can return real data. You can go from *nothing* to *something* pretty quickly when there are no moving parts. And honestly, it's just more *fun* when there's something on the screen that I can tweak.

Create a file called `static-data.js` (under `src`) and paste in this code.

```
let items = [
  {
    id: 0,
    name: "Apple iPad",
    description: "An iPad like no other. WiFi, 4G, lots of storage.",
    price: 329.00
  },
  {
    id: 1,
    name: "Apple iPad Pro",
    description: "Even more expensive than the regular iPad.",
    price: 799.00
  },
  {
    id: 2,
    name: "Canon T7i",
    description: "DSLR camera with lots of megapixels.",
    price: 749.99
  },
  {
    id: 3,
    name: "Apple Watch Sport",
    description: "A watch",
    price: 249.99
  },
  {
    id: 4,
    name: "Apple Watch Silver",
    description: "A more expensive watch",
    price: 599.99
  }
];

export {items};
```

Then we will pass those items into the new `ItemPage` component.

Back in `App.js`, import the `ItemPage` and the static `items` data.

```
import ItemPage from './ItemPage';
import { items } from './static-data';
```

And then update the `Content` component to use the new `ItemPage`:

```
const Content = ({ tab }) => {
  switch (tab) {
    default:
    case 'items':
      return <ItemPage items={items} />;
    case 'cart':
      return <span>the cart</span>;
  }
};
```

Now you should see some items rendering!

Items Cart

Apple iPad Mini 2 16GB

Apple iPad Mini 2 32GB

Canon T7i

Apple Watch Sport

Apple Watch Silver

Who Owns the Data?

In your own apps, you'll get to decide which components "own" which pieces of data. Here, we're saying that `App` owns the list of items and the state of the cart, and it passes them down to the relatively dumb `ItemPage` (and soon, `CartPage`). When it came time to use *real* data from a server, we'd fetch the data from within `App`.

In this case, because the items are needed by both pages, we pulled the data up to App. But sometimes it'll make more sense for the page-level components to own the data. For instance, if you had an ItemDetail page that wanted to display a list of reviews for an item, it would probably make the most sense for ItemDetail to “own” that review data.

Create the Item Component

Let's now create the real Item component to use in ItemList. It'll take an item prop, and we'll also want to be able to add it to the cart, so it should take an onAddToCart prop too.

This component will be stateless. Its responsibilities are to display an item, and to let its parent know when the “Add to Cart” button is clicked.

Learning from the GitHub example, we won't return an from this component. We don't want to force users of Item to put it inside a . By returning a plain <div>, the parent can put this item into an (or not) if it chooses to. This keeps the code more flexible.

With that in mind, create src/Item.js and src/Item.css. In Item.js, write out the component:

```
import React from 'react';
import PropTypes from 'prop-types';
import './Item.css';

const Item = ({ item, onAddToCart }) => (
  <div className="Item">
    <div className="Item-left">
      <div className="Item-image" />
      <div className="Item-title">
        {item.name}
      </div>
      <div className="Item-description">
        {item.description}
      </div>
    </div>
    <div className="Item-right">
      <div className="Item-price">
        ${item.price}
      </div>
    </div>
  </div>
)
```

```

    <button
      className="Item-addToCart"
      onClick={onAddToCart}
    >
      Add to Cart
    </button>
  </div>
</div>
);
Item.propTypes = {
  item: PropTypes.object.isRequired,
  onAddToCart: PropTypes.func.isRequired
};

export default Item;

```

Now let's wire it in to the ItemPage component. In ItemPage.js, update the code to look like this:

```

import React from 'react';
import PropTypes from 'prop-types';
import Item from './Item';
import './ItemPage.css';

function ItemPage({ items, onAddToCart }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          <Item
            item={item}
            onAddToCart={() => onAddToCart(item)} />
        </li>
      )}
    </ul>
  );
}

ItemPage.propTypes = {
  items: PropTypes.array.isRequired,

```



```

    onAddToCart: PropTypes.func.isRequired
  };

  export default ItemPage;

```

We're importing `Item`, and then inside the `` we're now rendering the `Item` component instead of just the item's name. Also new is the `onAddToCart` prop and its associated `propTypes`.

At this point, the code won't work, and there will be a warning about the missing `onAddToCart` prop.

Now that `ItemPage` requires an `onAddToCart` prop, we need to pass that from inside the `App` component. While we're at it, we may as well actually add the items to a cart! We can store that in state, and we will add a new `cart` array to hold the items.

Let's walk through it.

First, in `App.js`, create a new piece of state called `cart`, initialized to an empty array, to hold the items. We'll write a function called `addToCart` that accepts an item, and adds it to the cart. (Later we'll need to group the items together)

```

const App = () => {
  const [activeTab, setActiveTab] = useState('items');
  const [cart, setCart] = useState([]);

  const addToCart = item => {
    setCart(prevCart => [...prevCart, item]);
  };

  // ...
}

```

We also need to pass the `addToCart` function down to the `Content` component so that it can pass it along to `ItemPage`. To do that we'll add an `onAddToCart` prop to `Content`:

```

const App = () => {
  // ...

```

```

return (
  <div className="App">
    <Nav
      activeTab={activeTab}
      onTabChange={setActiveTab}
    />
    <main className="App-content">
      <Content
        tab={activeTab}
        onAddToCart={addToCart}
      />
    </main>
  </div>
);
}

```

Then we'll update Content to pass the onAddToCart prop through to ItemPage:

```

const Content = ({ tab, onAddToCart }) => {
  switch (tab) {
    default:
    case 'items':
      return (
        <ItemPage
          items={items}
          onAddToCart={onAddToCart}
        />
      );
    case 'cart':
      return <span>the cart</span>;
  }
};

```

Let's try it now: clicking "Add to Cart" should update the cart. But we have no way to verify the cart is filling up!

Just for debugging, add this bit of code inside of the App component somewhere:

```
<div>
  {this.state.cart.length} items
</div>
```

Now try it again. The item count should increase each time you click “Add to Cart.”



We won’t need that debugging code for long, but leave it in for a minute.

Most of this is similar to things you’ve seen before, but I want to call your attention to the `addToCart` function:

```
const addToCart = item => {
  setCart(prevCart => [...prevCart, item]);
};
```

What this is doing is setting the cart state to a copy of the current cart, plus one new item. We’re using the “updater” form of `setCart`, where we pass it a function, and it calls that function with the previous cart value. It implicitly returns the new array, which replaces the old cart and then re-renders.

There’s some new ES6 syntax here: `...prevCart` is the *spread* operator, and it expands the given array into its individual items. Here’s an example of the spread operator:

```
// With arrays:
const a = [1, 2, 3];
const b = [a, 4];    // => [[1, 2, 3], 4]
const c = [...a, 4]; // => [1, 2, 3, 4]
```

```
// With objects:
const o1 = {a: 1, b: 2};
const o2 = {...o1, c: 3}; // => {a: 1, b: 2, c: 3}
```

Updating State Immutably

You might be wondering... why not just modify the cart directly and then call `setCart`, like this?

```
const addToCart = item => {
  cart.push(item);
  setCart(cart);
}
```

This won't work, because it *mutates* the cart instead of creating a new one. After the `setCart` call, React will look at the cart that was passed in and go, "that looks the same as the old one. No need to re-render!" The app will look as if nothing had happened.

Try it out! Replace the `addToCart` function with this broken version, and watch how the item count doesn't go up.

Once you've seen what happens, you can remove the bit of debugging code we added earlier, the `<div>{cart.length} items</div>`.

You should never modify ("mutate") state or its child properties directly. Don't do `state.something = x`, and don't do `state = x`. React relies on you to call the setter (e.g. `setState`) when you want to make a change, so that it will know something changed and trigger a re-render. If you circumvent `setState` the UI will get out of sync with the data.

This applies to class component as well: avoid writing `this.state = stuff` or `this.state.something = stuff`.

Mutating state directly can lead to odd bugs, and components that are hard to optimize.

Instead, you should create *new* objects and arrays when you call `setState`, which is what we did above with the spread operator.

Back to the Code

Let's add some CSS to make it look a little better. In `Item.css` (which you created earlier, but is currently empty), add this CSS:

```
.Item {
  display: flex;
  justify-content: space-between;
  border-bottom: 1px solid #ccc;
  padding: 10px;
}
.Item-image {
  width: 64px;
  height: 64px;
  border: 1px solid #ccc;
  margin-right: 10px;
  float: left;
}
.Item-title {
  font-weight: bold;
  margin-bottom: 0.4em;
}
.Item-price {
  text-align: right;
  font-size: 18px;
  color: #555;
}
.Item-addToCart {
  margin-bottom: 5px;
  font-size: 14px;
  border: 2px solid #FFAA3F;
  background-color: #fff;
  border-radius: 3px;
  cursor: pointer;
}
.Item-addToCart:hover {
  background-color: #FFDDB2;
}
.Item-addToCart:active {
  background-color: #ED8400;
}
```



```

    color: #fff;
    outline: none;
  }
  .Item-addToCart:focus {
    outline: none;
  }
  .Item-left {
    flex: 1;
  }
  .Item-right {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
  }

```

That's a bit better:

Items Cart

	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 Add to Cart
	Apple iPad Mini 2 32GB Even larger than the 16GB.	\$279 Add to Cart

Create the CartPage Component

Now that we can add items to the cart, we'll build the component that renders the cart itself.

Create `src/CartPage.css` and `src/CartPage.js`, and open up `CartPage.js`. Initially, the code will be very similar to the code from `ItemPage`.

```

import React from 'react';
import PropTypes from 'prop-types';

```

```
import Item from './Item';
import './CartPage.css';

function CartPage({ items }) {
  return (
    <ul className="CartPage-items">
      {items.map(item =>
        <li key={item.id} className="CartPage-item">
          <Item item={item} />
        </li>
      )}
    </ul>
  );
}
CartPage.propTypes = {
  items: PropTypes.array.isRequired
};

export default CartPage;
```

CartPage accepts a list of items and renders the items as an unordered list. You'll notice that we aren't passing an `onAddToCart` handler to `Item`, which will cause a prop warning. We'll fix that shortly.

Group and Count the Items

The cart needs items to display. The trouble is, our cart state just contains item indices, not actual items. On top of that, there can be duplicate indices when an item is added multiple times.

We'll need to group the items in the cart state into an array of items with counts before we can pass it into `CartPage`. Let's put that into a function called `summarizeCart` near the top of `App.js`, outside of the `App` component:

```
const summarizeCart = cart => {
  const groupedItems = cart.reduce((summary, item) => {
    summary[item.id] = summary[item.id] || {
      ...item,
      count: 0
    }
  }, {});
  return groupedItems;
};
```

```

    };

    summary[item.id].count++;

    return summary;
  }, {}));

  return Object.values(groupedItems);
};

```

In the code above, we're using Array's built-in `reduce` function to group together the items by their IDs, and keep a count of how many of each item is in the cart. The first time we encounter an item we create a new object that contains everything from the item *plus* a new property called `count`. The next time we see that item, we only increment the count.

Once we have the `groupedItems` object, we can use `Object.values` to extract an array of the *values* of the object (ignoring the keys, which in our case, are item IDs).

The Array `reduce` function can be pretty confusing at first. That's totally normal. If you're not sure what's going on here, try inserting a `console.log` or two and watch the function play out in the console!

The “reducer” concept works the same as with the `useReducer` hook we saw earlier in the book. In case that hasn't fully clicked yet, here's a re-explanation of how reducers work, with a different example from before.

```

const letters = ['R', 'e', 'a', 'c', 't'];

// `reduce` takes 2 arguments:
//   - a function to do the reducing (you might say, a "reducer")
//   - an initial value for accumulatedResult
const word = letters.reduce(
  function(accumulatedResult, arrayItem) {
    return accumulatedResult + arrayItem;
  },
  ''); // <-- notice this empty string argument: it's the initial value

console.log(word) // => "React"

```


In this example, the reducer will get called 5 times (because there are 5 elements in the array). The calls go like this:

- first called with (' ', 'R') => returns 'R'
 - the empty string ' ' comes from the 2nd argument to reduce, and the 'R' is the first element of the array
- then ('R', 'e') => returns 'Re'
 - the 'R' comes from the previous return value, and 'e' is the next element of the array
- then ('Re', 'a') => returns 'Rea'
 - the 'Re' is the previous return value, and 'd' is the third array element
- then ('Rea', 'c') => returns 'Reac'
 - by now you are sensing a pattern
- then ('Reac', 't') => returns 'React'
 - the pattern is all too clear now

The last return value, 'React', is returned as the final result and stored in the word variable.

The reduce function is a functional-style shorthand for code like this:

```
const a = [1, 2, 3, 4];
let total = 0;
for(let i = 0; i < a.length; i++) {
  total += a[i];
}
```

Any time you want to iterate over the values of an array to create an aggregate result, consider using reduce.

Pass the Grouped Items to CartPage

Now that we have the summarizeCart function, we can use it to convert the cart array into something that CartPage can display. We'll pass the summarized cart as the cart prop to Content, and then it can be passed along to CartPage.

```
const App = () => {
  // ...
```

```

return (
  <div className="App">
    <Nav
      activeTab={activeTab}
      onTabChange={setActiveTab}
    />
    <main className="App-content">
      <Content
        tab={activeTab}
        onAddToCart={addToCart}
        cart={summarizeCart(cart)}
      />
    </main>
  </div>
);
};

const Content = ({ tab, onAddToCart, cart }) => {
  switch (tab) {
    default:
    case 'items':
      return (
        <ItemPage
          items={items}
          onAddToCart={onAddToCart}
        />
      );
    case 'cart':
      return <CartPage items={cart} />;
  }
};



```

Modifying the Cart

At this point, the “Cart” tab should be working. Click “Add to Cart” a few items, then click over to the “Cart” tab, and the items should be there.

However, the “Add to Cart” buttons are out of place on the Cart page, and we’re still getting propTypes warnings about `onAddToCart`. We’ll fix both of those next.

Items Cart

	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 Add to Cart
	Apple iPad Mini 2 32GB Even larger than the 16GB.	\$279 Add to Cart

Even though the cart is working, it’s missing important features. There is no way to add or remove items, no total price, and we don’t even know how many of each item is in the cart.

Remember how we reused the `Item` component inside `CartItem`? It would be nice if we could render a customized `Item` that had Add and Remove buttons in place of the “Add to Cart” button.

There are a few ways to go about this.

One idea: we could make a copy of the `Item` component, rename it `CartItem`, and customize it as necessary. This is quick and easy, but duplicates code.

Another idea: we could make `Item` take a prop that tells it to be a “regular item” or a “cart item.” Using that prop, `Item` would decide whether to render an “Add to Cart” button or “Add/Remove” buttons. This would work, but it’s messy – `Item` would have multiple responsibilities. If it became necessary to reuse `Item` in a third situation, we might end up adding even more conditional logic, making `Item` harder to understand and maintain.

A third idea is to pass children to `Item` and let it decide where to put them. This is the idea we’ll implement. It makes `Item` fairly reusable. If you want an “Add to Cart” button, pass that in. If you want Add/Remove buttons instead, pass those in. Let’s see how this works.

Refactoring Item

The new Item and its propTypes looks like this:

```
const Item = ({ item, children }) => (
  <div className="Item">
    <div className="Item-left">
      <div className="Item-image" />
      <div className="Item-title">{item.name}</div>
      <div className="Item-description">{item.description}</div>
    </div>
    <div className="Item-right">
      <div className="Item-price">${item.price}</div>
      {children}
    </div>
  </div>
);
Item.propTypes = {
  item: PropTypes.object.isRequired,
  children: PropTypes.node
};
```

Where once there was an “Add to Cart” <button>, we now have {children} instead. Also, Item no longer needs to know about the onAddToCart function, so we’ve that prop and its corresponding propTypes.

If you refresh at this point, you’ll see that the “Add to Cart” buttons are gone from the Cart page *and* the Items page. To fix that, open up ItemPage.js and pass a <button> as a child of the Item:

```
function ItemPage({ items, onAddToCart }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          <Item item={item}>
            <button
              className="Item-addToCart"
              onClick={() => onAddToCart(item)}>
```

```

        Add to Cart
      </button>
    </Item>
  </li>
)}
</ul>
);
}

```

There we go, back to normal. Now for the new part: open `CartPage.js`, and inside `Item`, pass in the plus/minus buttons and the item count. We're also going to need handler functions for when an item is added or removed.

```

function CartPage({ items, onAddOne, onRemoveOne }) {
  return (
    <ul className="CartPage-items">
      {items.map(item =>
        <li key={item.id} className="CartPage-item">
          <Item item={item}>
            <div className="CartItem-controls">
              <button
                className="CartItem-removeOne"
                onClick={() => onRemoveOne(item)}>&ndash;</button>
              <span className="CartItem-count">{item.count}</span>
              <button
                className="CartItem-addOne"
                onClick={() => onAddOne(item)}>+</button>
            </div>
          </Item>
        </li>
      )}
    </ul>
  );
}

CartPage.propTypes = {
  items: PropTypes.array.isRequired,
  onAddOne: PropTypes.func.isRequired,

```

```
onRemoveOne: PropTypes.func.isRequired
};
```

This is all stuff you've seen before.

The code won't work until we pass the `onAddOne` and `onRemoveOne` functions, so head let's back to `App.js` and create those. We'll update `Content` to accept-and-forward the props to `CartPage`:

```
const Content = ({ tab, onAddToCart, onRemoveItem, cart }) => {
  switch (tab) {
    default:
    case 'items':
      return (
        <ItemPage
          items={items}
          onAddToCart={onAddToCart}
        />
      );
    case 'cart':
      return (
        <CartPage
          items={cart}
          onAddOne={onAddToCart}
          onRemoveOne={onRemoveItem}
        />
      );
  }
};
```

We already have the `addToCart` function that takes an item and adds it to the cart, so we can just reuse that here.

Now we need to create a corresponding `removeItem` function and pass that down to `Content`. It will take an item, find an occurrence of that item in the cart, and then update the cart state to remove that occurrence.

```

const App = () => {
  // ...

  const removeItem = item => {
    let index = cart.findIndex(i => i.id === item.id);
    if (index >= 0) {
      setCart(cart => {
        const copy = [...cart];
        copy.splice(index, 1);
        return copy;
      });
    }
  };

  return (
    <div className="App">
      <Nav
        activeTab={activeTab}
        onTabChange={setActiveTab}
      />
      <main className="App-content">
        <Content
          tab={activeTab}
          onAddToCart={addToCart}
          onRemoveItem={removeItem}
          cart={summarizeCart(cart)}
        />
      </main>
    </div>
  );
}

```

We need to avoid mutating the state directly, so we're making a copy of the cart and then using the slice function to remove a single item at the index we found.

This is a little more work than if we were to allow mutations but it is a good habit to get into. It'll be especially important if you start using Redux in the future. Redux relies heavily on immutability.

To wrap up the add/remove buttons, let's give them a bit of style. Add this CSS to `CartPage.css`:



```

.CartPage-items {
  margin: 0;
  padding: 0;
}
.CartPage-items li {
  list-style: none;
}
.CartItem-count {
  padding: 5px 10px;
  border: 1px solid #ccc;
}
.CartItem-addOne,
.CartItem-removeOne {
  padding: 5px 10px;
  border: 1px solid #ccc;
  background: #fff;
}
.CartItem-addOne {
  border-left: none;
}
.CartItem-removeOne {
  border-right: none;
}

```

Here it is in all its glory:

Items Cart

	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 <div>- 2 +</div>
	Canon T6i DSLR camera with lots of megapixels.	\$749.99 <div>- 2 +</div>

Exercises

1. It would be nice if the shopping cart told us how many thousands of dollars we were about to spend on Apple products, rather than letting us guess. Add a “Total” to the bottom of the Cart page similar to this:

	Apple iPad Mini 2 32GB Even larger than the 16GB.	\$279 <div>- 4 +</div>
---	---	---------------------------

Total: \$2032

2. The Cart page is completely blank when the cart is empty. Modify it to display “Your cart is empty” when there are no items in the cart, something like this:

Items Cart


Your cart is empty.

Why not add some expensive products to it?

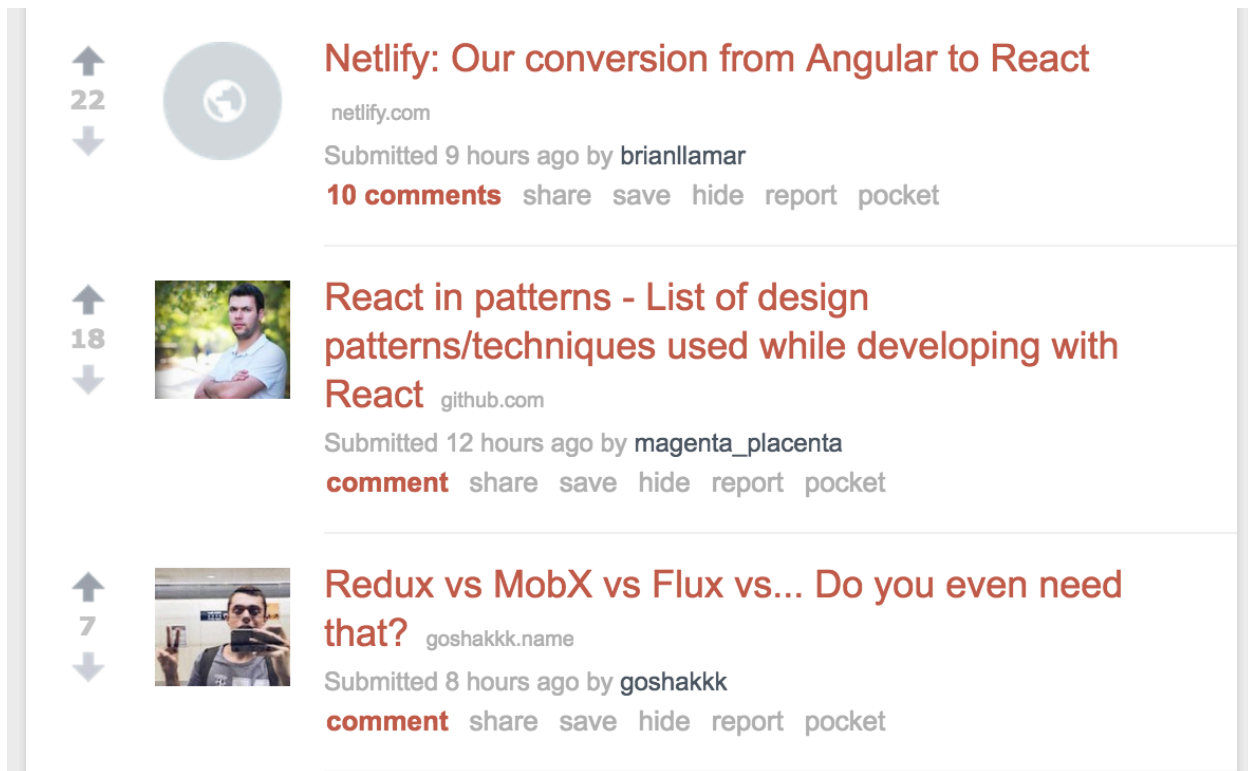
3. Display a summary of the shopping cart in the top-right of the nav bar, as shown below. Include the total number of items in the cart and the total price. Make sure to account for the fact that each item in the cart array could have a count greater than 1. As a bonus, make the cart summary clickable, and switch to the Cart page on click. Add a shopping cart icon too if you like (this one is from Font Awesome).

Items Cart

 2 items (\$458)

	Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G.	\$229 <div>- 2 +</div>
---	---	---------------------------

4. Build a simplified version of Reddit, modeled after this screenshot:



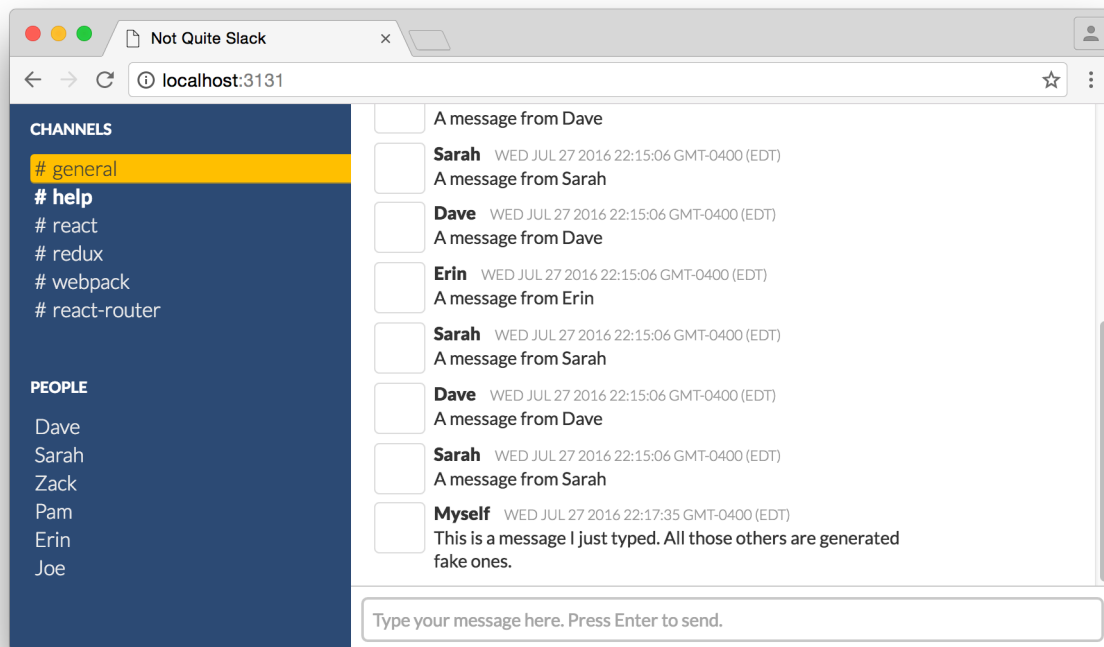
Here are the features it should have:

- Display the number of votes
- Functioning Upvote and Downvote controls
- Display the number of comments
- Sort the stories by number of upvotes
- The title should be a real link
- Other links do not need to be functional (share, save, hide, etc.)

Follow the same process we've done throughout this book: outline the components in the screenshot, decide which pieces of state you need to keep and which components should own that state, then start building components.

Reddit's API is public and you don't need a key. You can use static data from e.g. <http://www.reddit.com/r/reactjs.json> – save this to a file and import it as we've done in a few of the examples.

5. Build a simplified version of Slack, modeled after this UI:



Here are the features it should have:

- A list of Channels
- A list of Users
- Maintain state for the currently-selected Channel or User
- Click a Channel or User to select it
- Only one Channel or User can be selected at a time
- Each Channel and User has its own history of messages
- Generate some fake messages for each channel
- Show an input box at the bottom, only when a Channel or User is selected
- Typing in the input box and hitting ENTER should add that message to the selected Channel/User and clear the input.

21 Where To Go From Here

First of all, congratulations for making it to the end of the book. I hope that you had as much fun reading it as I did writing it.

Now, with that said, this is not the end. This is only the *beginning* of your React journey.

At this point you have a solid grasp of React's fundamentals, and you've learned some ES6 too. You've learned about classes and hooks. You've done a few API calls. From here, you can go on to learn about React Router, Redux or MobX, and whatever else you need.

Here's a suggested order in which to tackle them. I think it's important that Redux be learned *last*, but the others can be reordered as you see fit.

API Calls and CRUD Operations

We talked about how to make GET calls to an API, but in most apps you'll also want to POST, PUT, PATCH, and DELETE things too. These are the standard CRUD operations (CRUD stands for Create, Read, Update, Delete). These things aren't part of React, so you'll need to look up how to do them within your particular HTTP library – whether that's `fetch` or `axios` or one of the others.

Depending on your backend server, you might also want to learn GraphQL, which is a query language for fetching exactly the data you need from the server.

With a REST API, you might end up making multiple calls to get the data you need: one to get the list of products, then one call per product to get detail info, then another call for each to get the reviews for each product, and so on.

This all depends on how your backend API is structured, but in a lot of cases, a REST API will require more calls to get the same amount of data. You can end up *underfetching* or *overfetching*. With GraphQL, you can describe exactly which data you need (including child data & related data) and get *all* of it with a single call. GraphQL is pretty exciting.

Testing

Testing is an important part of building software. Whether you decide to use test-driven development (TDD) or not, having tests around your code gives you confidence that everything works as it should.

There are a handful of tools to learn for testing React. The most important is Jest.

Jest handles running the tests as well as running assertions and creating mocks and spies. Every app made with Create React App already has Jest set up for you, and they even come with a passing test in `App.test.js`. Use the `npm test` command to run the tests.

On top of Jest, you'll want some kind of library for rendering components and making assertions about them. [React Testing Library](#) is a good choice. It encourages you to write tests the way your *users* would use the app, which results in tests that are less brittle and leaves your code easier to change.

Another popular option, though less popular these days, is Enzyme. It's another library for testing React components, created by Airbnb. It allows you to render components and interact with them in your tests, and make assertions about props and state and the contents of your components. Enzyme allows you access to the internals and the implementation details (the props and state), which can lead you into writing brittle tests that break whenever you change your components.

If you're starting from scratch, I recommend React Testing Library. But if you're working within a project that already has a lot of Enzyme tests, that's fine too.

Webpack

Build tools are a major stumbling block in the beginning, which is why setting up Webpack wasn't part of this book. If you don't need to customize your build yet, postpone learning Webpack until later.

I recommend reading the article [Webpack – The Confusing Parts](#) as an introduction to Webpack and its way of thinking.

You should know that the `create-react-app` tool we've been using has a production build mode, which can delay (or eliminate entirely!) your need to configure Webpack by hand. To use it, run `npm run build`, which will create a production-optimized build with minified files and a production build of React. If you don't need to customize anything, Create React App is great out of the box, and perfectly suited to write real production React apps.

It also has an "eject" command (run `npm run eject`) which extracts the generated Webpack config and exposes it for you to customize. It is one of the most well-commented Webpack configs I've seen.

There's also a middle ground: customizing some of the decisions Create React App made without ejecting. This is possible with the [react-app-rewired](#) package, and I recommend taking a look at that if you want to make a small tweak to the config but don't want to fully eject.

ES6

You’ve already seen some ES6 throughout the book: arrow functions, `let/const`, classes, destructuring, and `import/export`. This is most of the ES6 you’ll use on a regular basis, but there are a number of other nice additions, like promises, `for...of` loops, dynamic object keys, iterators, and more.

You can learn these features as you go, but it’s a good idea to at least do a quick survey of everything so you know what’s available. This [Overview of ES6 Features](#) is a great resource.

Routing

Now that you have a fairly good handle on how React works, React Router’s concepts will build upon them. It makes use of components to lay out the routing for your application, effectively mapping your components to URLs. Check out [React Router’s documentation](#) to get started.

Some people conflate React Router and Redux in their head – they’re not actually related or dependent on each other, although there is a library that links the router’s state with Redux. You can (and should!) learn to use React Router before diving into Redux.

Redux

Dan Abramov, the creator of Redux, [will tell you](#) not to add Redux to your project too early, and for good reason – you don’t actually need Redux until you encounter the problems it solves, and it’s a dose of complexity that can be disastrous early on.

The concepts behind Redux are fairly simple, but there is a gap between understanding how the pieces work and knowing how to use Redux in a real app. It will take some practice to truly understand Redux. It’s also influenced by functional programming concepts, which can take some getting used to if you’re not familiar with them.

So, before you commit to adding Redux to a larger project, repeat what you did throughout this book: build small projects. Take code that uses plain React state and convert it to use Redux. Build a bunch of little Redux experiments to really internalize how it works.

If you want a complete course on Redux, take a look at my course [Pure Redux](#).

TypeScript

When I wrote this book, typed versions of JavaScript were in their infancy. We had Flow and TypeScript, and it wasn't clear which, if any, would stick around. Today, the landscape is quite different. TypeScript has emerged as the "winner" (though some people still use Flow) and has gained pretty wide adoption over the last few years.

TypeScript is a typed superset of JavaScript. That means that all valid JS is also valid TS. With it, you can annotate your functions and objects with type information – similar in spirit to React's `PropTypes` – and then the TypeScript compiler can verify everything is correct.

If you already enjoy statically-typed languages, you'll be right at home with TypeScript. If not, I suggest trying it out on a few small projects. There's a bit of a learning curve but it can be really nice to have the assurance that you're passing around the right types, and having your IDE autocomplete methods is icing on the cake. Create React App supports TypeScript out of the box, by passing the `--typescript` flag when you create a new project.

Here are a couple places to learn more:

- [Learn X in Y minutes: TypeScript](#) – a quick rundown of TypeScript syntax
- [TypeScript Docs](#) – the official documentation has a quick-start tutorial and pointers to further resources

On Boilerplate Projects

I recommend building your project brick-by-brick, adding pieces as you need them. That way you will understand how all of the moving parts work, which means you can fix them when they break or extend them when you outgrow the default settings.

It's easy to think that learning "Pure React" is only for beginners, or that you'll be required to use a boilerplate project for any substantial project. I don't think anything could be further from the truth.

Part of the power of React is being able to build your own toolset the way you like it. Even if you abdicate control and use a boilerplate project, you still have to learn the conventions of that particular project. I suggest steering clear of boilerplate projects most of the time.

One of the allures of a boilerplate is that it prescribes a project structure for you. If the structure of files on disk is keeping you from making progress, read my article [How to Structure Your React Project](#).

Thank You

Thanks so much for not only purchasing, but *finishing* this book. I would love to know your thoughts – what was good, what needs improvement, or any suggestions for future topics. Feel free to reach me at dave@davedceddia.com or contact me on Twitter @dceddia.