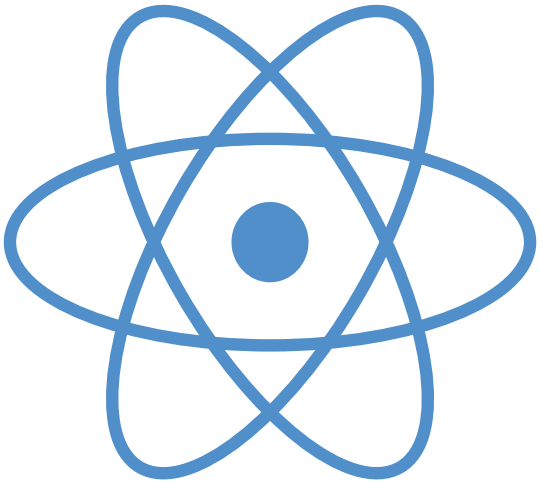


PURE REACT

BY DAVE CEDDIA

Contents

1 Introduction **4**

Why Just React? 6

How This Book Works 7

Environment Setup 10

Debugging Crash Course 13

2 Hello World **15**

3 JSX: What and Why **18**

What Is JSX? 18

HTML Inside JavaScript?! 18

4 Working With JSX **23**

Composing Components 23

Wrap JSX with Parentheses 24

"If" in JSX 28

Exercises 30

5 Example: Tweet Component **33**

6 Props **47**

Arguments to Components 47

Communicating With Parent Components 50

7 Example: Tweet With Props **52**

What Exactly Should Be Passed as a Prop? 56

Guidelines for Naming Props 57

8 PropTypes **62**

Documentation and Debugging In One 62

How Do I Validate Thee, Let Me Count the Ways 64

Example: Tweet with PropTypes 69

PropTypes as Documentation 72

Exercises 73

9 Children **75**

Different Types of Children 76

Dealing with the Children 76

PropTypes for Children 77

Versus Transclusion in Angular 78

A More Interesting Example 78

Exercises 80

10 Example: GitHub File List **82**

The “key” Prop 87

Exercises 96

11 State **100**

Example: A Counter 100

setState Is Asynchronous 102

Shallow vs Deep Merge 104

Handling Events 105

What to Put in State 106

Thinking Declaratively 106

Where to Keep State 110

"Kinds" of Components 113

12 Input Controls **114**

Controlled Inputs 114

Uncontrolled Inputs 117

13 The Component Lifecycle **118**

14 Example: Shopping Site **123**

Create ItemPage 131

Create Item 134

Create CartPage 143

Array.prototype.reduce 145

Object.keys 146

Exercises 152

15 Where To Go From Here **155**

1 Introduction

Welcome to Pure React.

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. Hundreds of 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."

But the opposite happens: instead of 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 here – when you learn React *cold*, you'll be able to go on and learn all of its friends with ease: Redux, Router, and the rest.

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 across on the web, 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 Webpack!"

It's too much at once. You might learn one or two core concepts with this approach, but it's a shaky foundation. Inevitably, when you sit down to write your own app, you get lost in the forest of libraries and concepts.

You can only get so far by copying and pasting pieces of code.

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

So here 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.

What We'll Cover

We will start where most programming books start, with Hello World. 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, composable components.

Finally, you'll learn about "state," how it differs from props, and how to organize it in an application. We'll look at using form controls and the Component Lifecycle.

Why Just React?

Without a solid understanding of React, simultaneously learning libraries like Redux and tools like 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 approach React by trying to learn Webpack + Babel + React + Redux + Routing + AJAX *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 the 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? I should 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, he will absolutely not allow you to rewrite it with all the best practices you've learned. 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:

> Build small, throwaway apps.

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

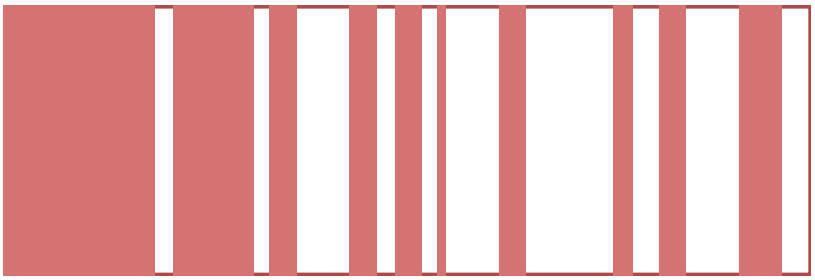
We'll start off with Hello World, of course. No self-respecting programming book would be complete without that.

As your skill set grows, low-fidelity copies of simple apps and sites like Reddit, Hacker News, and Slack make good projects. Designers call this "copywork," and it's great because it frees you from having to make *product decisions* like "where should the user go after login." 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 boring.

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 practice 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.

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 application!

Prerequisites

Tools

- Node.js (at least v4.0, >= 6 recommended)
- NPM (version >= 3 recommended) or yarn
- Google Chrome (or some other modern browser)
- React Developer Tools
- Your text editor or IDE of choice

You'll need Node and NPM (or Yarn). Head to https://nodejs.org and download the latest "Current" release. At the time of writing, this is v7.10.0.

Any modern browser should suffice. This book was developed against Chrome, so that's what I recommend. If you prefer another browser, I expect the JavaScript will work correctly but the CSS could require some tweaks.

Yarn

Yarn is a relatively new package manager for JavaScript, released in June of 2016. Compared to NPM, it is faster, and has the benefit of a *lock file*, which means it can reliably install the exact same packages every time you run `yarn` (Yarn's equivalent to `npm install`).

The commands are similar to NPM's:

- Install all packages: `npm install` or `yarn`
- Install a certain package: `npm install --save <package>` or `yarn add <package>`
- Start the development server: `npm start` or `yarn start`
- Run the tests: `npm test` or `yarn test`

Rather than padding the pages with things like "Run `npm start` (or `yarn start`)" throughout the book, I will only show the NPM commands from here on. However, feel free to use Yarn if you'd like. You can install it from https://yarnpkg.com.

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 DOM tree) and view the props and state assigned to each component. It is extremely useful for debugging.

Knowledge

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

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.

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.

Install Create React App

Run this command to install the tool globally (the `-g` means global):

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

If you get an error saying "Permission denied" (and you're on Linux or a Mac) you may need to re-run the command with "sudo", like this:

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

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 dev console, manually refresh the page, and fix any errors you see.
4. Check the command line console where you ran `npm start`. Look for any errors, and fix those.

To open the dev 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.

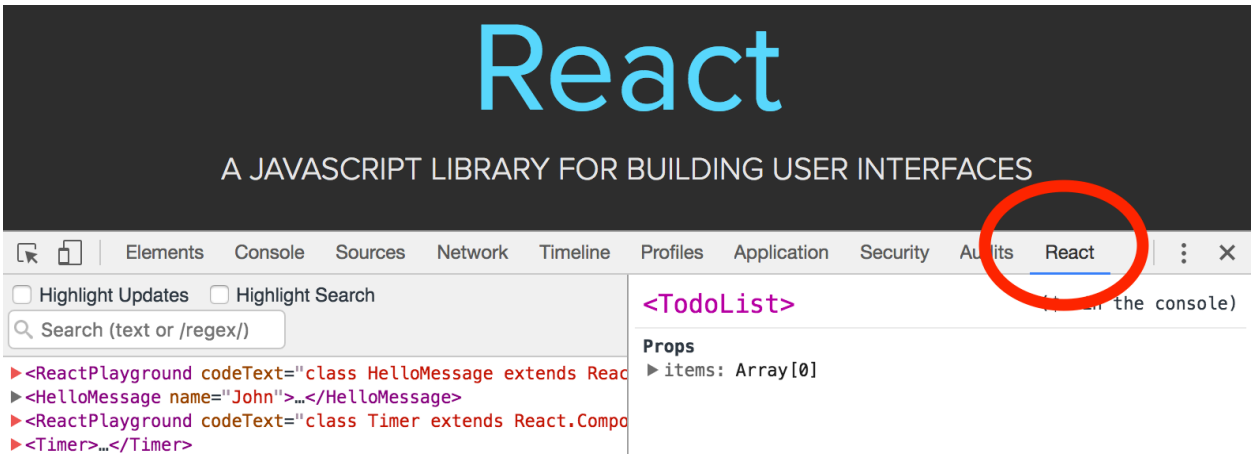
Still completely stuck? Stack Overflow has an abundance of React-related questions and answers, and the Reactiflux community at https://www.reactiflux.com/ is full of friendly people who can help. There is also a #react IRC channel on Freenode, which you connect to with an IRC client or visit http://irc.lc/freenode/reactjs.

Keep the Console Open!

The dev 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.

React Dev Tools

If you 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`, `npm`, and `create-react-app` installed. All of the tools are ready. Let's write some code!

Step 1

Use the `create-react-app` command to generate a new project. It 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 contains some files we won't need right now, so we'll delete them. You could also just ignore them.

```
$ rm src/App.* src/index.css src/logo.svg
```

Step 2

Open up the file `src/index.js`. Delete all of the contents, 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 `var React = require('react')`.

Step 3

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

```
$ 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).

```
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. They're often called "stateless function components."

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 Stack Overflow answers or old GitHub projects, so we'll go over how it works. But primarily we'll be writing functions where possible, and ES6 classes when we need to.

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, which we'll talk about below.

HTML Inside JavaScript?!

Before I even seriously looked into React, I knew it mixed HTML and JS, and I thought that was just *ugly*. I mean, that's undoing *years* of best-practices thinking, right? Since the Dark Days of jQuery we've known that directly setting innerHTML on elements from within JS was hackish and performed badly – not to mention full of security holes – so why is React making those same mistakes all over again?

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 that thing and converting it into HTML.

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

So what is React actually doing here? How does this work? And why isn't it terrible?

JSX Is Compiled to JavaScript

The JSX elements you write are actually compiled down to JavaScript by a tool called Babel. Babel is a "transpiler," a made-up word that means it transforms code into valid ES5 JavaScript that all browsers can understand. Each JSX element becomes a function call, where its arguments are its attributes ("props") and its contents ("children").

Here's an example of some simple JSX:

```
return <span>Hello!</span>;
```

And here is the JavaScript generated by the compiler:

```
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 (like `HelloWorld`, with no quotes).

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

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

You can also nest the calls:

```
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(/* ... */)
  );
}
```

Here is a slightly more complicated bit of JSX:

```
<span className='song-name'>
  {props.song.name}
</span>
```

And here is what it compiles to:

```
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. Nearly 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!

This is the number one concern most newcomers have: that pit-of-your-stomach feeling that mixing JS with HTML is just *wrong*. It might even make you *angry*.

Before I got into using React, I had the same apprehensions. It took some time (and writing a few small apps) before I began to understand the power of this unholy mixing. Don't worry though, because the exercises in this book will give you all the practice you need 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's a rite of passage for every new web developer, a piece of lore passed down through the generations about the Right Way to build web interfaces. "It's always been done this way."

There are some good reasons to separate the view from the logic. When combined, the code can turn into a tangled mess of conditional logic with duplicated HTML everywhere, like badly-written PHP.

If you have flashbacks to PHP or JSPs where database calls were mixed in with view code, 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, you've probably written the 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 JS code? 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 can add new functionality without changing both the template and its controller. Add a function in one file, call it from the other. Passing an extra argument? Gotta change it in two files.

If they were truly separated 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 `<span>Hello</span>` and `World` should render `<span>World</span>`. 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, "That all makes sense, yep." You'll never know if you internalized it 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, update 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, refresh it manually, and also make sure the app is running (run `npm start` if it's not).

You should see the same "Hello World!" as before. I know this seems simple, but there are some lessons here, I promise.

Wrap JSX with Parentheses

The following examples depend on the `Hello` and `World` components that you should have created, so make sure those exist before continuing.

First, a 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 removing the parens and watch what happens:

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

```
}
```

This will fail with an error in the browser console:

> HelloWorld(…): A valid React element (or null) must be returned. You may have returned undefined, an array or some other invalid object.

You'll also likely see a warning about "Unreachable code."

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>
  );
}
```

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:
  return (<Hello/> <World/>);

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

Returning two things at once makes no sense. 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:
return [<Hello/>, <World/>];

// Would turn into this JS
// (notice the brackets).
// It looks valid...
return [
  React.createElement(Hello, null),
  React.createElement(World, null)
];
```

Try it out! It should fail. In the dev console, you'll see:

> HelloWorld(…): A valid React element (or null) must be returned. You may have returned undefined, an array or some other invalid object.

So for now, the rule stands: you must return a single element. This is expected to change when React 16 is released.

JavaScript in JSX

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

```
function SubmitButton() {
  var 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 produces (aka returns) a single value. In contrast, *statements* do not produce values and can't be used inside JSX. Here are some examples of statements:

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

None of these things produces a value. `var 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 = var b = 5;
a = if(true) { 17; }
```

I'm hammering home this point because it's important, and it's a common tripping point for React beginners.

"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 (`?`). Use it like this:

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

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

```
function ValidIndicator() {
  var 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 may look a little strange. Remember that JavaScript needs to be inside single braces. Comments in JSX must go inside a JavaScript block like this:

```
function ValidIndicator() {
  var isValid = true;
  return (
    <span>
      {/* here is a comment */}
      {isValid && 'valid'}
      {!isValid && 'not valid'}
```

```
      {
       // Double-slash comments are
       // OK in multi-line blocks.
      }
    </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 `submitButton`.

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.).

Early versions of React kept a "whitelist" of all the built-in element names so it could tell them apart from custom ones. 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 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 `<br>` or `<input>` or maybe even `<li>`.

```
// 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.getElementById('root')
);
```

You can delete or ignore the `src/App*` files and the logo. If you do not explicitly `import` them, they do not 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 (replace `.` with a space character) and take note of the output (hint: leading and trailing spaces are removed, and so are newlines):

a. Single lines

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

c. Empty newlines

```
<div>
Empty

Newlines

Here
</div>
```

d. Spaces with newlines

```
<div>
 Non-breaking
 Spaces 
</div>
```

f. 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:
  //var username = "root";
  //var username = undefined;
  //var username = null;
  //var 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?

5 Example: Tweet Component

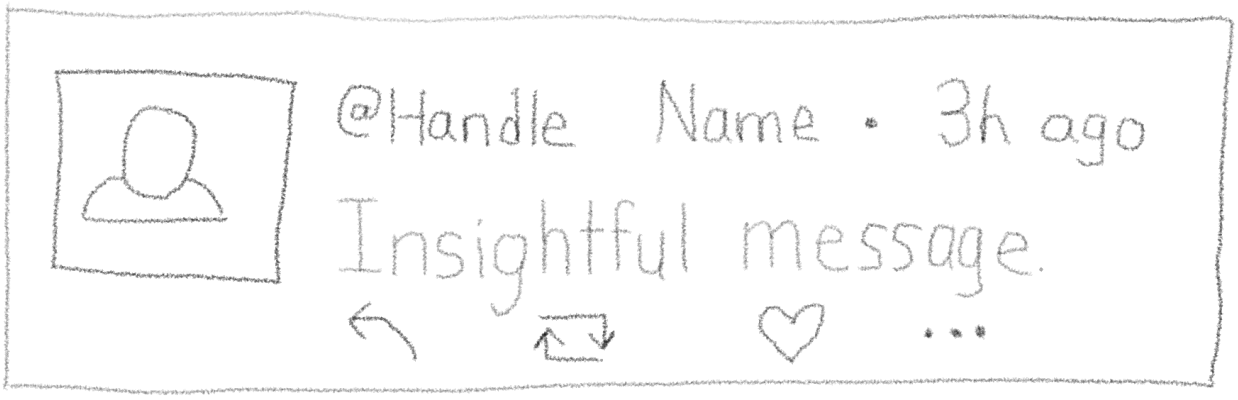
1. Sketch

This is your introduction to "thinking in components." Let's start with something concrete: a humble sketch. This could come from a designer (wireframes or mockups) or it could be your own creation.

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 sketch: you can tell when you're "done." Without a sketch, 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.

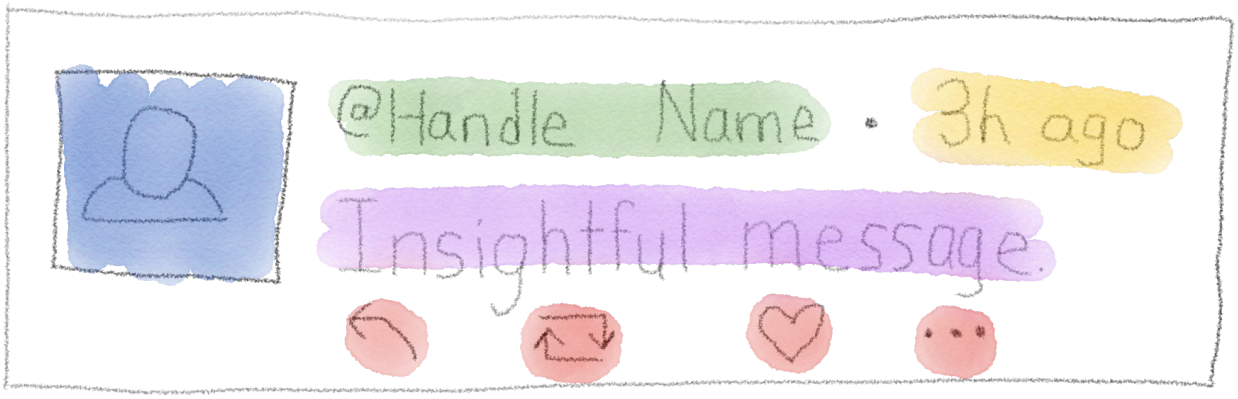
2. Carve into Components

The next step is to break this sketch into components. Draw boxes around the "parts," and think about reusability.

Imagine you had 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.

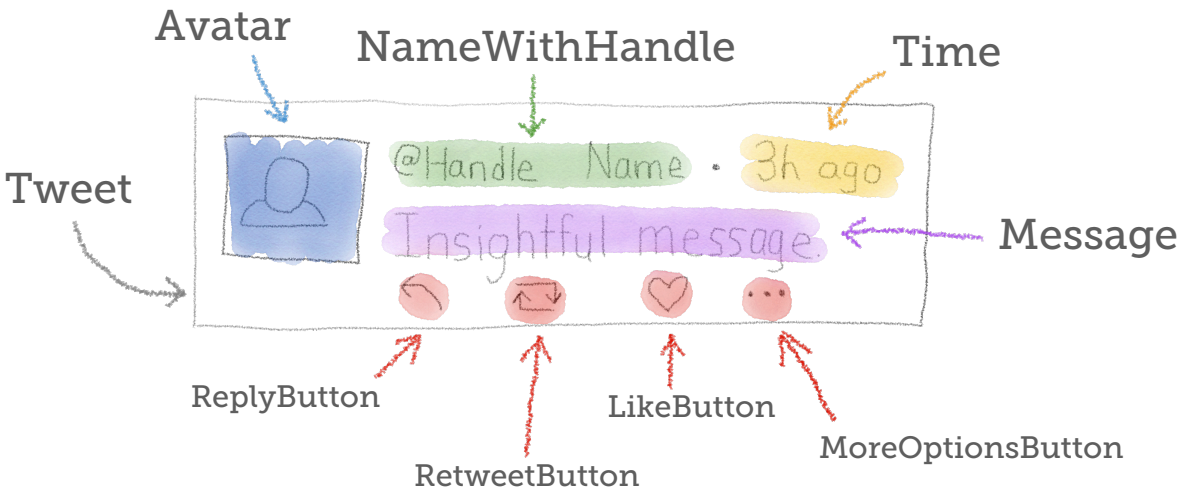
Another strategy: make every "thing" a component. Buttons, bits of text, images, and so on.

Try it yourself, then compare with this:



3. Name the Components

Now that we've broken the sketch into pieces, we can name them.



Each of these will become a component, with `Tweet` being the "parent" of the others:

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

4. Build

We know the component hierarchy now – let's build it.

Top-Down, or Bottom-Up?

There are two ways to approach this.

1. Start at the top. Build the Tweet component first, then build its children. Build Avatar, then NameWithHandle, and so on.
2. Start at the bottom (the "leaves" of the tree). Build Avatar, then NameWithHandle, 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, 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).

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

For example, if we were building Twitter, 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 build bottom-up.

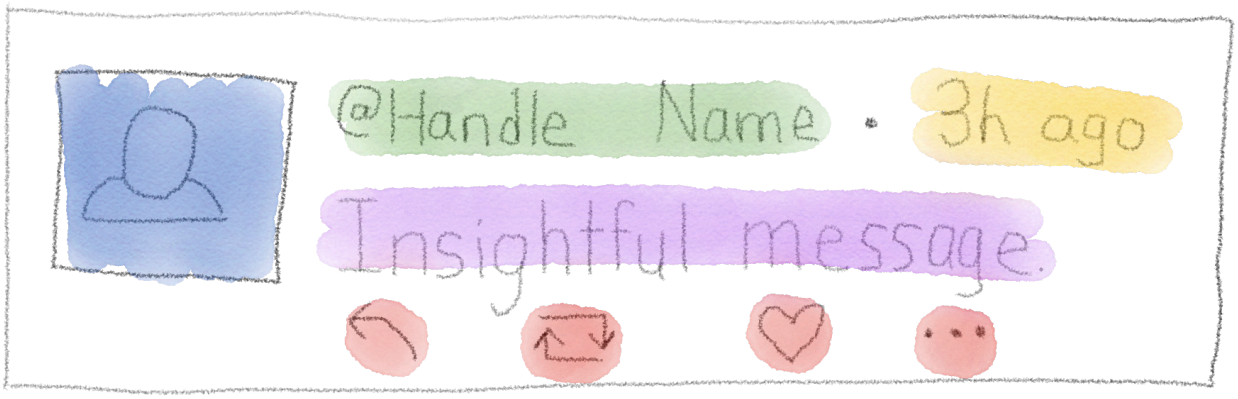
There's another case where building from the bottom is a good idea, and that's if you have an existing app written in another framework like Angular or Backbone and you want to rewrite in it React. Starting at the top makes little sense here, because it'll have a ripple effect across your entire code base.

Starting at the bottom is manageable and controlled. You can build the "leaf nodes" of your app – the small, contained 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
```

It gave us a few files we won't use here, so we can safely delete them:

```
$ rm src/App.* src/logo.svg
```

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/4.7.0/css/font-awesome.min.css">
```

We also have a generated `index.css` file in the `src` directory. Open it up 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:

```
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 running:

```
$ npm start
```

And the page should render something like this:

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

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

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 in the browser – open the dev 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`.

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.

Back to our outline. Let's build the `Avatar` component. Add the component function to `index.js`:

```
function Avatar() {
  return (
    <img
      src="https://www.gravatar.com/avatar/nothing"
      className="avatar"
      alt="avatar" />
  );
}
```

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`.

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

Next we need to change `Tweet` to render `Avatar`:

```
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 `NameWithHandle`:

```
function Message() {
  return (
    <div className="message">
      This is less than 140 characters.
    </div>
  );
}

function NameWithHandle() {
  return (
    <span className="name-with-handle">
      <span className="name">Your Name</span>
      <span className="handle">@yourhandle</span>
    </span>
  );
}
```

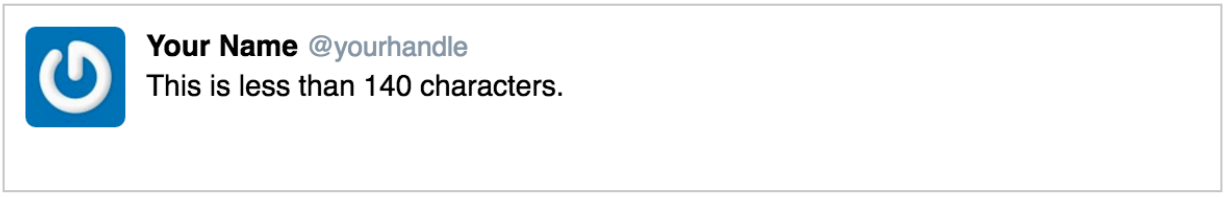
If you refresh now, nothing will have changed because we still need to update `Tweet` to use these new components:

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

```
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:

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

// This can be rewritten as an
// anonymous function:

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

// ...which can be turned into an
// arrow function:

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

// It can be further simplified to:

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. The function itself is the `() => { ... }` part.

Notice how there's no `return` in the last 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.

You'll continue to use arrow functions as you continue through the book, so don't worry if they look foreign now. You'll get used to them.

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.

ES6 adds `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.

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

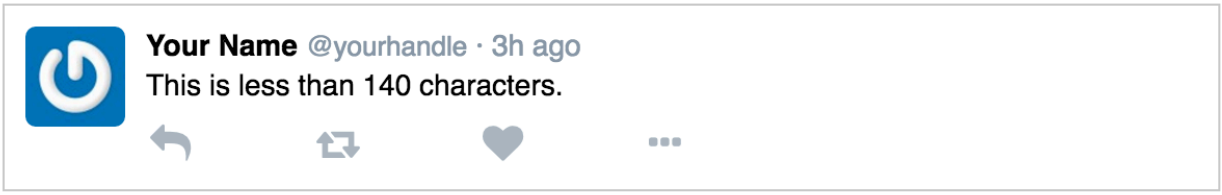
```
function Tweet() {
  return (
    <div className="tweet">
      <Avatar/>
      <div className="content">
        <NameWithHandle/><Time/>
        <Message/>
        <div className="buttons">
          <ReplyButton/>
          <RetweetButton/>
          <LikeButton/>
          <MoreOptionsButton/>
        </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 coming up next: parameterizing components with props.

6 Props

Arguments to Components

Where HTML elements have "attributes," React components have "props" (short for "properties").

We've already seen that React components can be written as functions, so it's natural to assume that we could pass arguments to those functions. Props are the *arguments* to your components.

Passing Props

This bit of JSX is passing a prop called `name` with a string value of `"Dave"`:

```
<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.

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.

Here, we're passing a string for `className`, a number for the `age` prop, and an actual JavaScript expression for 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 `var`
- Use `if`, `for`, `while`, etc.
- Define functions with `function`

Remember that JSX evaluates to JavaScript, and 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(var 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={var x = true; x && 'is true'}/>
// gets compiled to:
React.createElement(Broken, {
  value: var 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 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". It can extract multiple keys, 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 an Angular background this is a change. Angular's 2-way binding mechanism allowed modifying scope variables (Angular's version of props) and would automatically propagate those changes to the parent component.

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?

If a child needs to send data to its parent, the parent can inject 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 a prop named `onAction`, which it can call whenever it needs to send up data or notify the parent of an event. You'll notice that the built-in `button` element accepts an `onClick` prop, which it'll call when the button is clicked. We'll look more deeply at event handling later on.

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

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 will 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 into 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 } destructuring
function Tweet({ tweet }) {
  return (
    <div className="tweet">
      <Avatar/>
      <div className="content">
        <NameWithHandle/><Time/>
        <Message/>
        <div className="buttons">
          <ReplyButton/>
          <RetweetButton/>
          <LikeButton/>
          <MoreOptionsButton/>
        </div>
      </div>
```

```
    </div>
  );
}

// ...

var 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 }) {
  var 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.

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 doing string concatenation, as in `"https://www.gravatar.com/avatar/" + hash`.

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!

NameWithHandle and Time

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

```
<NameWithHandle/><Time/>
```

With these lines:

```
<NameWithHandle 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 to install Moment:

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

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 `NameWithHandle` and `Time` components. You'll notice that `Time` uses the new `moment` library.

```
function NameWithHandle({ author }) {
  const { name, handle } = author;
  return (
    <span className="name-with-handle">
      <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 `<span>`, and then you could remove the `return` and the braces. I'll leave it up to you.

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:

- If `Time` expects a tweet object and pulls out the timestamp property, it won't be usable with anything other than a tweet object. 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" a tweet with a timestamp.
- 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 could 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:

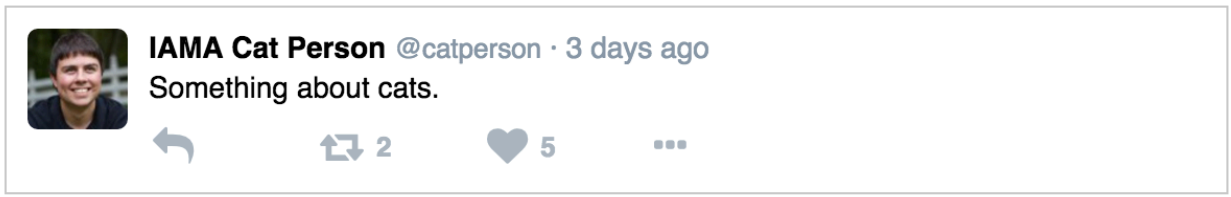
```
.buttons i {
  width: 80px;
}
```

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 `<span>` element or `null`.

Notice that we had to introduce 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).

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 likes: 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 since its argument needed to be replaced with a destructuring to extract `count`. Otherwise it looks the same. Then, using it in `RetweeetButton` 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 `<span>` with class "like-count", but the contents could 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).

React has a secret weapon though: 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. This is a change that came with React 15.5. You should already have this package installed if you're using Create React App, but if not, a quick `npm install --save prop-types` will fix that.

Next, notice that propTypes are set as a property on the function itself. This style works whether the component is a function, or 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 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 console if you forget a required prop:

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

> Warning: Failed propType: Required prop `message` was not specified in `Comment`.

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

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

> Warning: Failed propType: 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 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 propTypes 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 for 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

React's propTypes allow you to express many different validations. By default, a propType validation is *optional*. If you want to make it required, add `.required` to the end.

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
}
```

This PropTypes `shape` is requiring an object that has `name` and `age` keys, so 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

Any PropType validation can be made required by adding `.required` to the end of it. These are all required:

```
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 `.required` to your PropType 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.getElementById('root')
);

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

// This will also work:
ReactDOM.render(
  <CustomTest myCustomProp="abc"/>,
  document.getElementById('root')
```

```
);
```

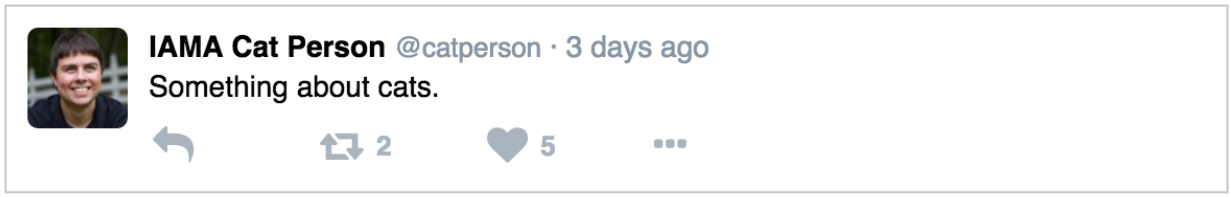
Example: Tweet with PropTypes

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

Recall that the structure looks like this:

- Tweet
  - Avatar
  - NameWithHandle
  - 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 PropType 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 dev console like this:

> Warning: Failed propType: 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? Nice! 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.

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

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

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

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

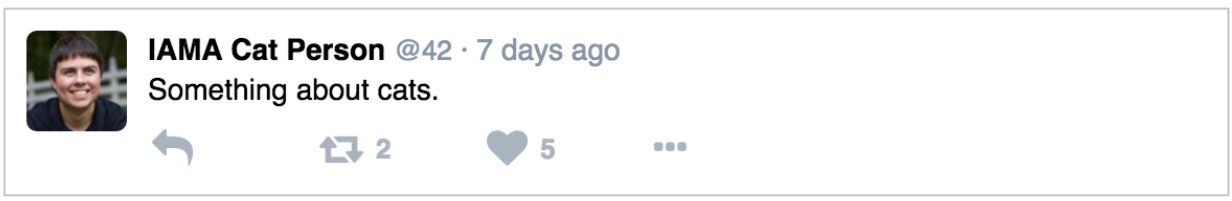
Here, the `author` prop is required, and it must have a `name` and `handle` as strings.

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

React complains, as expected:

> Warning: Failed propType: Invalid prop 'author.handle' of type 'number' supplied to 'NameWithHandle', 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! This 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 `React.PropTypes.object`, or use a more complex `shape`. It's up to you! (I suggest trying to figure out the `shape` yourself).

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 difficult 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 `NameWithHandle`, 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

One last thing about 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. You won't have to scan through the code to decipher which props are required.

Exercises

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

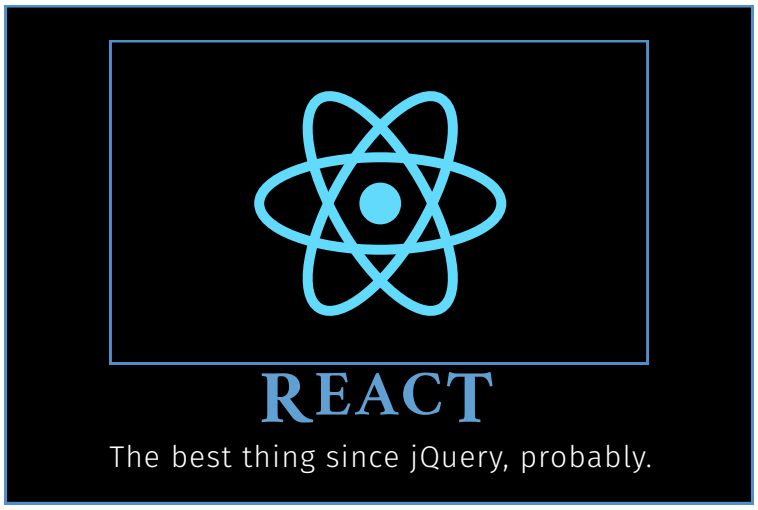
1. Create an `AddressLabel` component that takes a `person` object as a prop and renders their name and address like so:

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

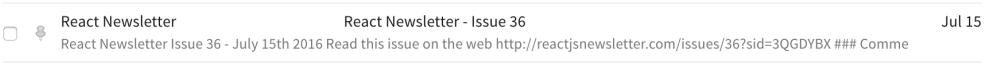
2. Create an `Envelope` component that takes `toPerson` and `fromPerson` as props and uses your `AddressLabel` from Exercise 1 to display the return address and the recipient address. Make sure to include a `Stamp` too!

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 can express nested 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.

You have to explicitly render the children somewhere in your component. If you don't, they are ignored. For instance, if the `IconButton` component looked like this:

```
function IconButton() {
  return (
    <button>
      <i class="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, we can place the text after the icon:

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

Different Types of Children

The `children` prop is always pluralized as `children` no matter whether there's a single child or multiple children. Moreover, the type of `children` will change depending on what it contains.

When there are multiple children, `children` is an array of ReactElements.

However, when there is only one child, it is a *single ReactElement*. This might seem a little weird: wouldn't it be easier to deal with if `children` was always an array?

Well, yes, it would. But `children` is used so often, and the single-child use case is common enough that the React team decided to optimize 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 exception 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 `oneOfType` validator like this:

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

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

Versus Transclusion in Angular

If you've used Angular, you may be familiar with the concept of "transclusion," a made-up word that Angular uses to mean "take the child elements of a directive and insert them where `ng-transclude` is."

React's `children` prop is a similar concept, but more powerful and a bit easier to wrap your head around.

A More Interesting Example

In the example above, we simply 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>
  );
}
```

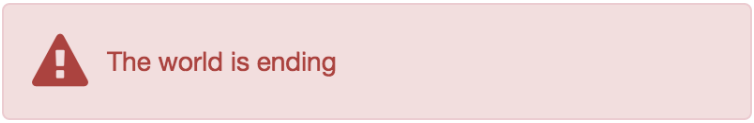
The code converts children into an array, then walks backward from the end as it inserts a new element between every existing element.

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.

2. Every React component has a `type` property, which means you can inspect the type of a child component by iterating over the children with `React.Children.forEach`.

Create the `Nav` and `NavItem` components we looked at earlier. The `NavItem` should take a `url` prop and the link text as a child, then render an anchor tag (`a`) element with that text, and pointing to that url.

Next, modify the `Nav` component to check that every one of its children is a `NavItem`, and throw an error if any other type of component is found.

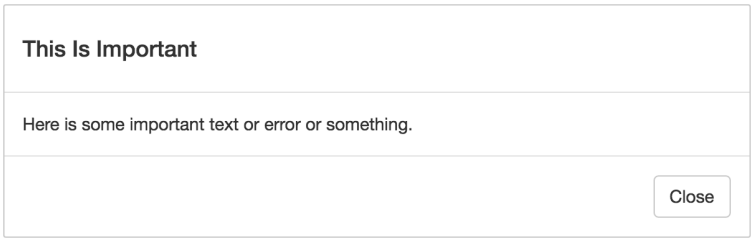
As a test, the example code from earlier should work correctly:

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

And this code should throw an error:

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

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 component that replicates GitHub's file list. It's what you see at the top of every GitHub repository:

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 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 "div" elements in the design.

Name the Components

Once you've decided which parts will be components, give them names. For the highlights above, I came up with these ones (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 that it has 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 that should 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 Running

Here's something to strive for when you're building components: Make sure it always works!

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/App.* src/logo.svg
```

Copy the `index.html` file from the `tweet` project, from the "public" directory. 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? Try to do it from memory. Look back at the Tweet example if your memory is fuzzy.

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 is how you render lists of things in React.

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 the reason 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 the prop.

In practice, 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 idea.
- **Permanent**: An element's key must not change between renders. This means `key={Math.random()}` is a bad idea.
- **Unique**: No two elements should have the same key.

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 using a hashing function, or generate unique IDs with a library like shortid.

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 might cause tough-to-diagnose rendering bugs.

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 assigned directly to the top-level components in the array, not their children.

```
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: 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. Try to write the code yourself before looking ahead. This is a weird one.

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 `getFileName`, which, if you look closely, is *not* a component.

```
function getFileName(file) {
  return [
    <FileIcon file={file} key={0}/>,
    <td className="file-name" key={1}>{file.name}</td>
  ];
```

```
}
```

This is just a function that takes a `file` and returns an array of elements. Components cannot return arrays without wrapping them with an element. Why write it this way?

The problem is that we're inside a table row (`<tr>`) and we need two `<td>`s next to each other, but a true component cannot return a bare array without a wrapper.

You could avoid this problem entirely by writing `FileIcon` and `FileName` as separate components that each return a `<td>` cell (rather than nesting `FileIcon` inside `FileName` like we're doing here). That is arguably a better solution than this! I left it this way to show this edge case in case you run into a situation where it's unavoidable. Avoid this kind of thing if you can.

At the time of writing, React 16 (a.k.a. React Fiber) is just around the corner, and promises to solve this problem of returning multiple elements from a component. You can track the progress of Fiber at http://isfiberreadyyet.com/.

Notice that we're passing hardcoded values for `key`, here. Didn't I just say not to do that?

Yeah, this is normally a bad idea, but because this array will *always* have 2 elements in that specific order, it won't cause a problem here. The `key` is required because we're returning an array, and elements in an array must have a `key`.

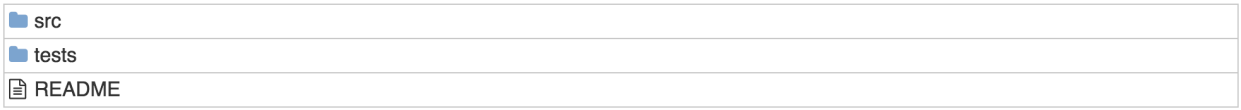
Here's the updated `FileListItem` that uses the `getFileName` function:

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

Alright, with that weirdness out of the way, let's add some CSS and see how it looks.

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

```
}

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



CommitMessage

Let's create the `CommitMessage` component next. This one is very straightforward, and so is the CSS. We'll use `CommitMessage` inside `FileListItem`.

```
const FileListItem = ({ file }) => (
  <tr className="file-list-item">
    {getFileName(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
};

.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 --save 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.

Alright, let's use `Time` inside `FileListItem`. First we need to import it, so add this line to the top of `index.js`:

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

Since we're importing our own file instead of something from `node_modules`, the path needs to be relative (`./time`) rather than just the module name (`time`).

Then, we can update `FileListItem`. Note that `Time` doesn't render a `<td>` so we have to wrap it in one:

```
const FileListItem = ({ file }) => (
  <tr className="file-list-item">
    {getFileName(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're rendering 3 components in 3 different ways, including the weird `getFileName` to hack around a limitation in React. It would be much better if the component looked 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. You forget that table cells must be direct children of their rows, adhere too closely to the mockup, and take yourself down a path that results in bad code.

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

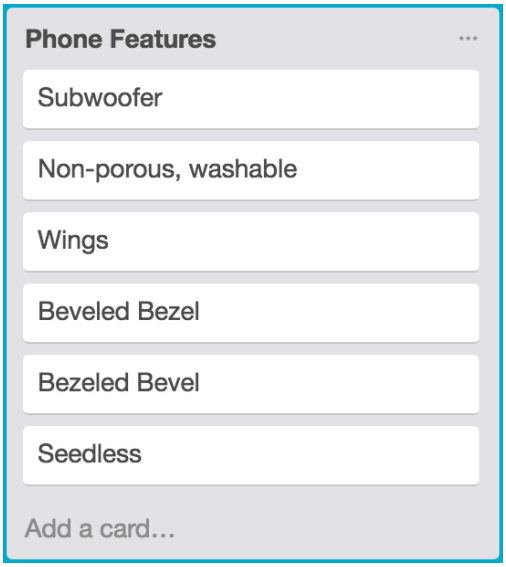
1. Refactor the GitHub file listing example so that none of the components return a table cell (`<td>`). Every component should return a `<span>` or `<div>` instead. This makes them more reusable, and also should greatly improve the code inside `FileList`. Change the CSS if necessary.
2. In real applications, you'll want to have only one component per file most of the time. Sometimes a file will contain a few related components, when those components are always used together (as in the `Nav`/`NavItem` example from earlier), but otherwise they should be separated out. 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.

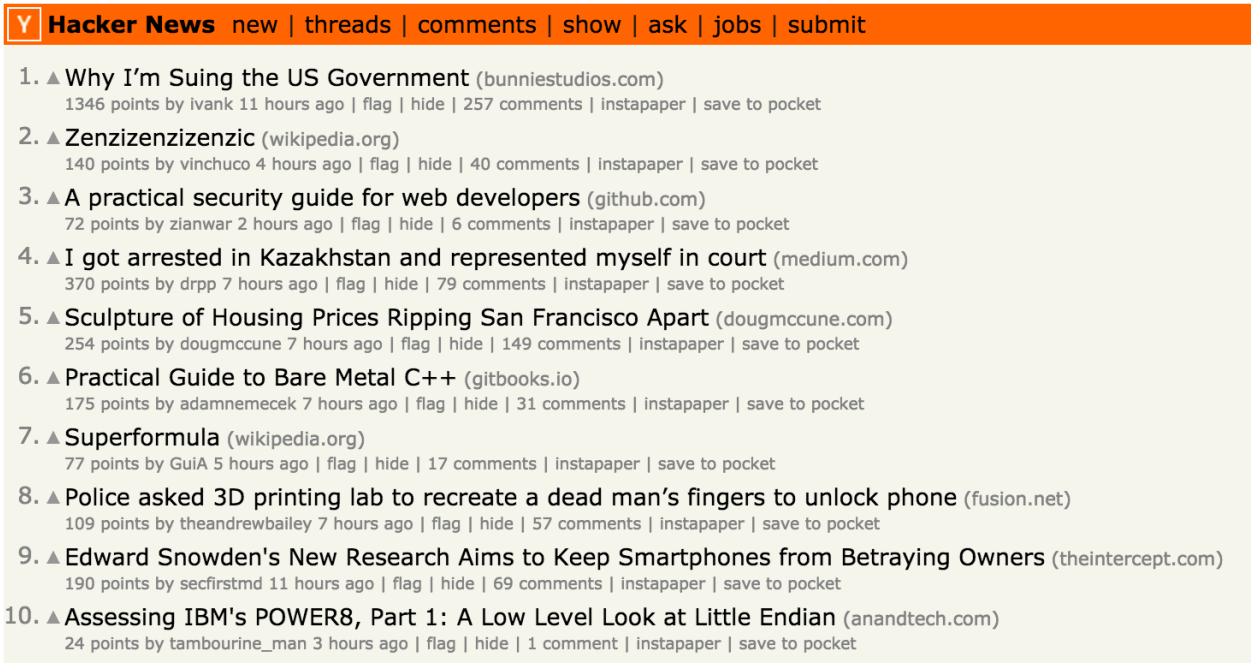
(screenshot from https://trello.com)



5. Hacker News

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

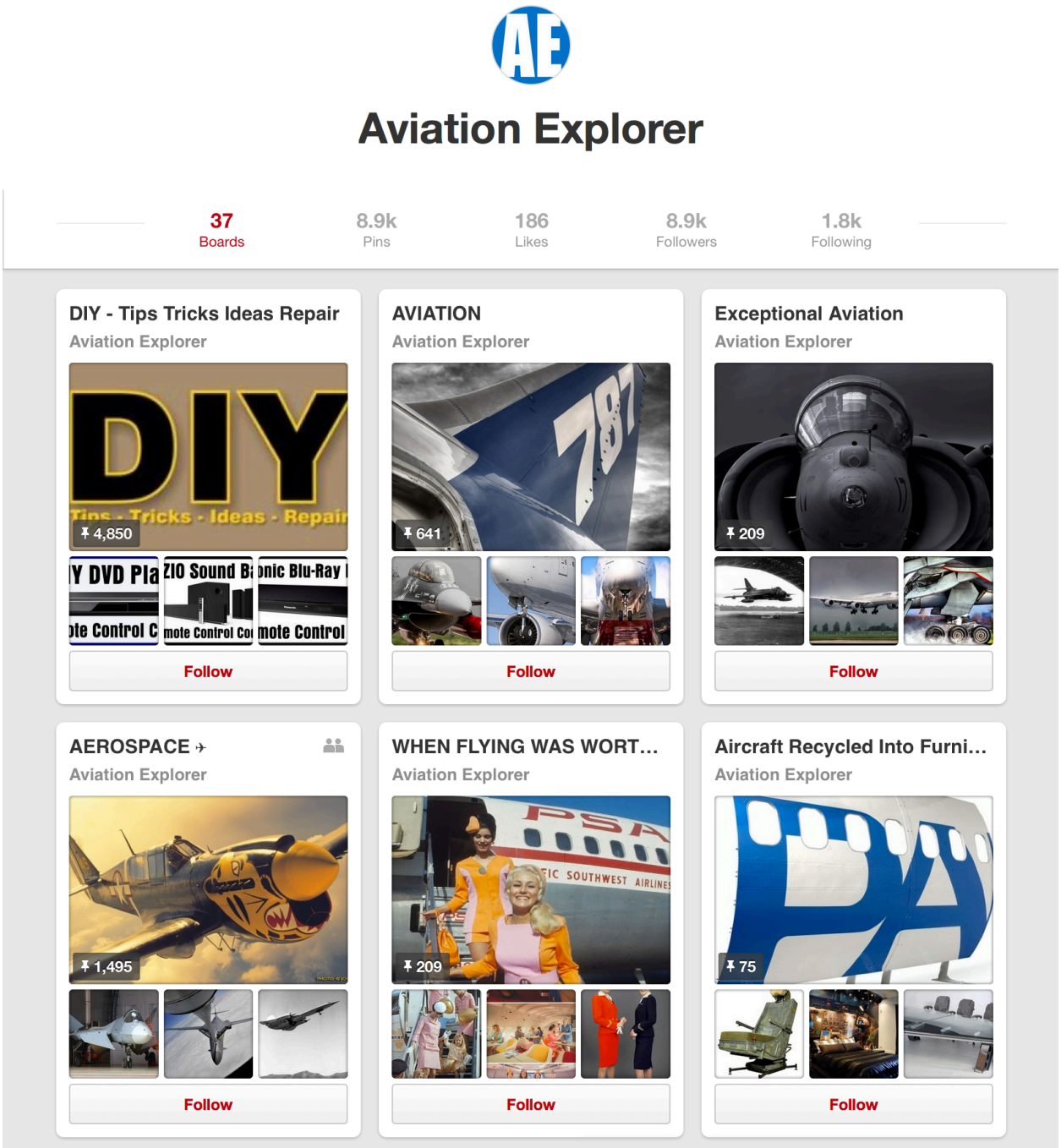
(screenshot from https://news.ycombinator.com/news)



6. Pinterest

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

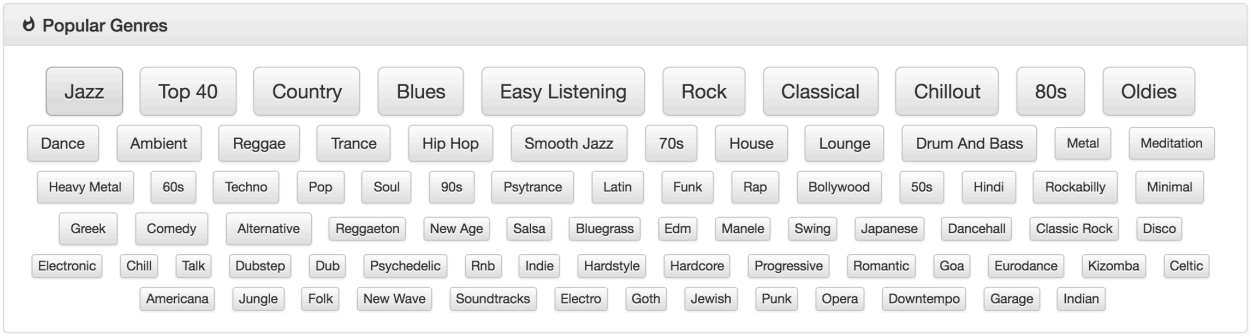
(sceenshot from https://www.pinterest.com/aviationhd/)



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

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? This is where *state* comes in.

To see how state is useful, let's look back at an example from the Props chapter. Here is a `Parent` component that contains a `Child` component. `Parent` passes down a function which `Child` calls whenever a button is clicked:

Example: A Counter

```
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 its `handleAction` function is called.

To do this, `Parent` needs 3 new things: an *initial state* where the counter is set to 0, a call to `this.setState` to increment the counter, and a way to display the current count.

Remember, though, that function components are *stateless*. To make `Parent` stateful, it needs to be transformed into an ES6 `class` component.

Here's the new version of `Parent`, renamed to `CountingParent` – make sure to update the call to `ReactDOM.render`!

```
class CountingParent extends React.Component {
  state = {
    actionCount: 0
  }

  handleAction = (action) => {
    console.log('Child says', action);

    // actionCount is incremented, and
    // the new count replaces the existing one
    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 instantiated, its initial state is set up via the `state = {...}` *property initializer*. Although this syntax is not officially part of ES6, it is supported by Babel and Create React App, and widely used in the React community. (An alternative is to set up initial state in a *constructor*, which we'll see later).

Next, you click the button, like it tells you to. 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 two `CountingParent` components on the page, they'll each have counters that will start at 0 and increment independently. You can prove this to yourself by creating a component that contains a few `CountingParents`:

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

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`.

setState Is Asynchronous

I 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)`, it will very likely print the old state instead of the one you just set.

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

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

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`. The function receives the current state and props as arguments, and it is expected to return the desired new 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. It works because calling `setState` "queues" the updates 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" – 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.

Shallow vs Deep Merge

When you call `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 curren state. Here's how that works.

The "shallow merging" behavior means that if you have a state like this:

```
{
  score: 7,
  user: {
    name: "somebody",
    age: 26
  },
  products: [ /*...*/ ]
}
```

And then you do `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.

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.

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 Facebook 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. You also can't access it asynchronously (say, after a timeout). If you *need* to access it asynchronously, you can call `event.persist()` and React will keep it around for you.

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, like handles to timers, should be stored on the component instance itself. You've got a `this` object available in class components classes, feel free to use it!

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 funny 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 – our brains are used to that. It looks and works like HTML.

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 Growl-style 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 jQuery or Angular or any other framework where you call functions to make things happen ("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 is either in 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}/>
```

This example is pretty simple. Hopefully nothing too mind-bending. The biggest change is that in the declarative React way, the expand/collapse state is stored *outside* the Accordion and passed in as a prop.

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 it.

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 `componentWillReceiveProps` *lifecycle method* and comparing the old value to the new one (you'll learn more about lifecycle methods in Chapter 14). If it changed, you can set the "animating" state to `true`. Then in `render`, when "animating" is true, add a CSS class that does the animation. When "animating" is `false`, don't add that class. Here's what this might look like:

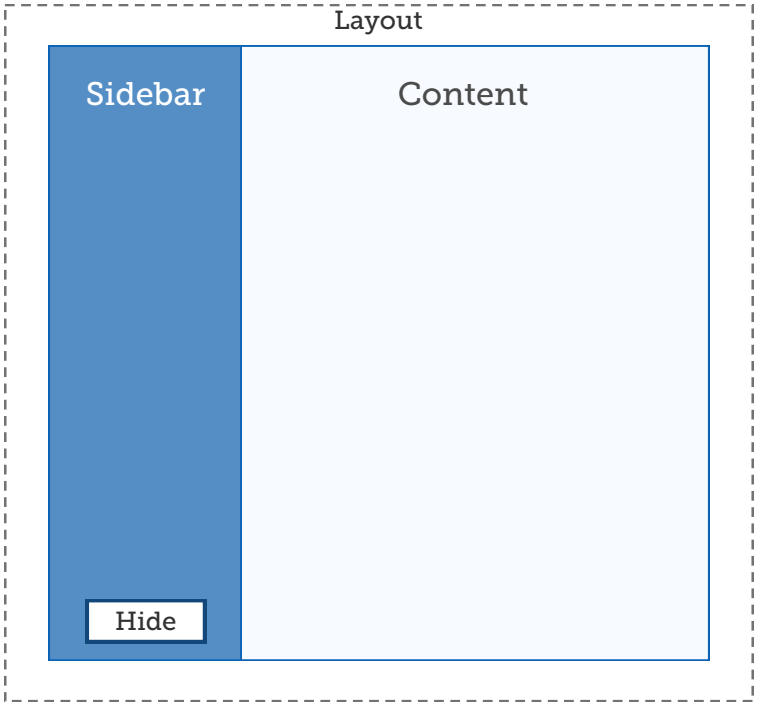
```
class Badge extends Component {
  componentWillReceiveProps(nextProps) {
    if(this.props.counter !== nextProps.counter) {
      // Set animating to true right now.
      // When the state change finishes, set a timer
      // to turn animating off in 200ms.
      this.setState({ animating: true }, () => {
        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 to warn you 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 crystal clear once you do it a few times.

Where to Keep State

Whenever you can, it's best to keep components stateless. Sometimes this isn't possible, but often, pieces of data you initially think should go into internal state can actually be pulled up to the parent component, or even higher.

Think about a section of the page that can be shown or hidden, maybe a sidebar like this:



When you click the "Hide" button, the sidebar should disappear, which means setting 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 go in the parent `Layout` component, and the 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, for instance, save it to localStorage to persist across page reloads, that logic only needs to exist in one component instance.

Browsers come with `localStorage`, a place you can use to save data between sessions. Closing and reopening the browser or refreshing the page won't clear the data. 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 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 accepts props and render some elements based on those props. Being stateless means that the component will 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 `Child`, 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.

The `Parent` component, above, is a presentational component. Perhaps surprisingly, `Page` is actually presentational. Presentational components can contain Container components, and Containers can contain Presentational components – there aren't any strict rules for nesting.

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. That's okay though – perfection is not the goal.

12 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*.

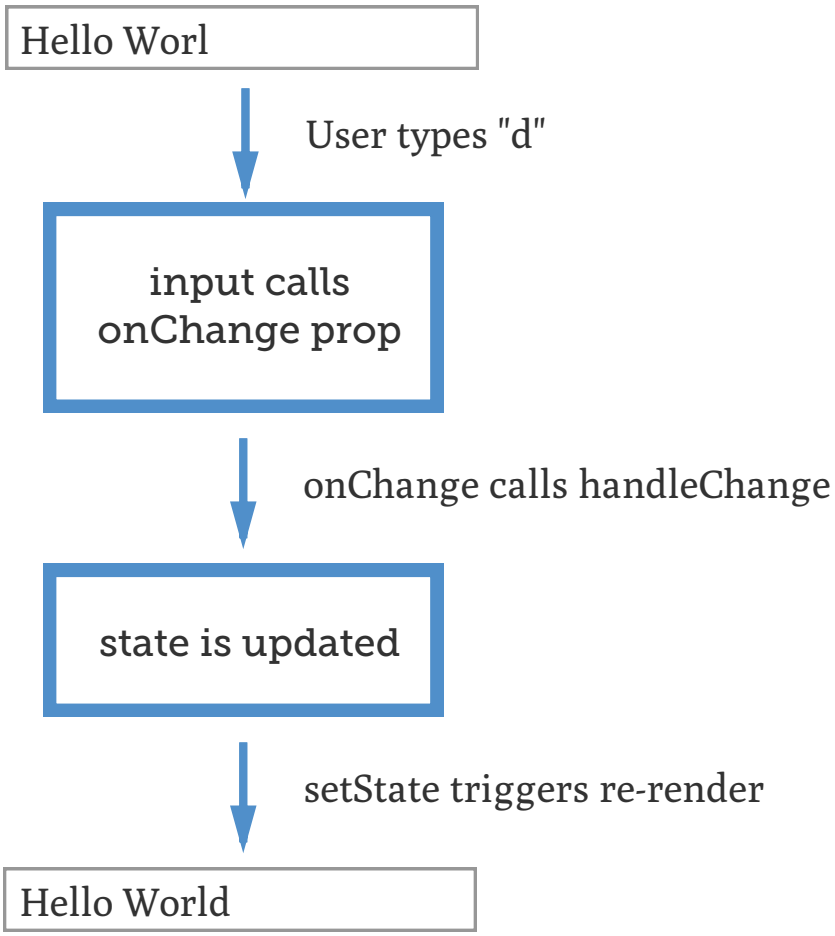
Controlled Inputs

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.

Remember how state works: a value and a callback function are passed as props, and the parent component is responsible for reacting to changes and passing an updated value to the child.

```
class ControlledInput extends React.Component {
  state = { text: '' };

  handleChange = (event) => {
    this.setState({
      text: event.target.value
    });
  };

  render() {
    return (
      <input type="text"
        value={this.state.text}
        onChange={this.handleChange} />
    );
  }
}
```

It works like this:



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 `this.setState`, then try to type in the box. Nothing happens, because the value is stuck at the initial value.

Then, try changing `this.setState` to ignore the event data, instead always setting the text to the same value, like this:

```
class TrickInput extends React.Component {
  state = {
    text: 'try typing something'
  };

  handleChange = (event) => {
    this.setState({
      text: 'haha nope'
```

```
    });
  };

  render() {
    return (
      <input type="text"
        value={this.state.text}
        onChange={this.handleChange} />
    );
  }
}
```

You'll see that the input thwarts your attempts to change it, and mocks you too.

This is very powerful though. If you want to do some kind of custom validation, or formatting, you can do that in the `handleChange` function. Don't want the user to type numbers? Strip out the numbers before updating the state:

```
class NoNumbersInput extends React.Component {
  // ...

  handleChange = (event) => {
    let text = event.target.value;
    text = text.replace(/[0-9]/g, '');
    this.setState({ text });
  };

  // ...
}
```

A little side note – that syntax `this.setState({ text })` is just ES6 shorthand for `this.setState({ text: text`
With ES6, if the key is the same as the variable name, you don't have to write it twice.

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. Like I said, 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, but there are 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. If you want to get a value out of it, you can pass an `onChange` prop and respond to the events. The problem with this is that you can't easily extract a value at will. You still need to listen to the changes, and keep track of the "most recent value" somewhere.

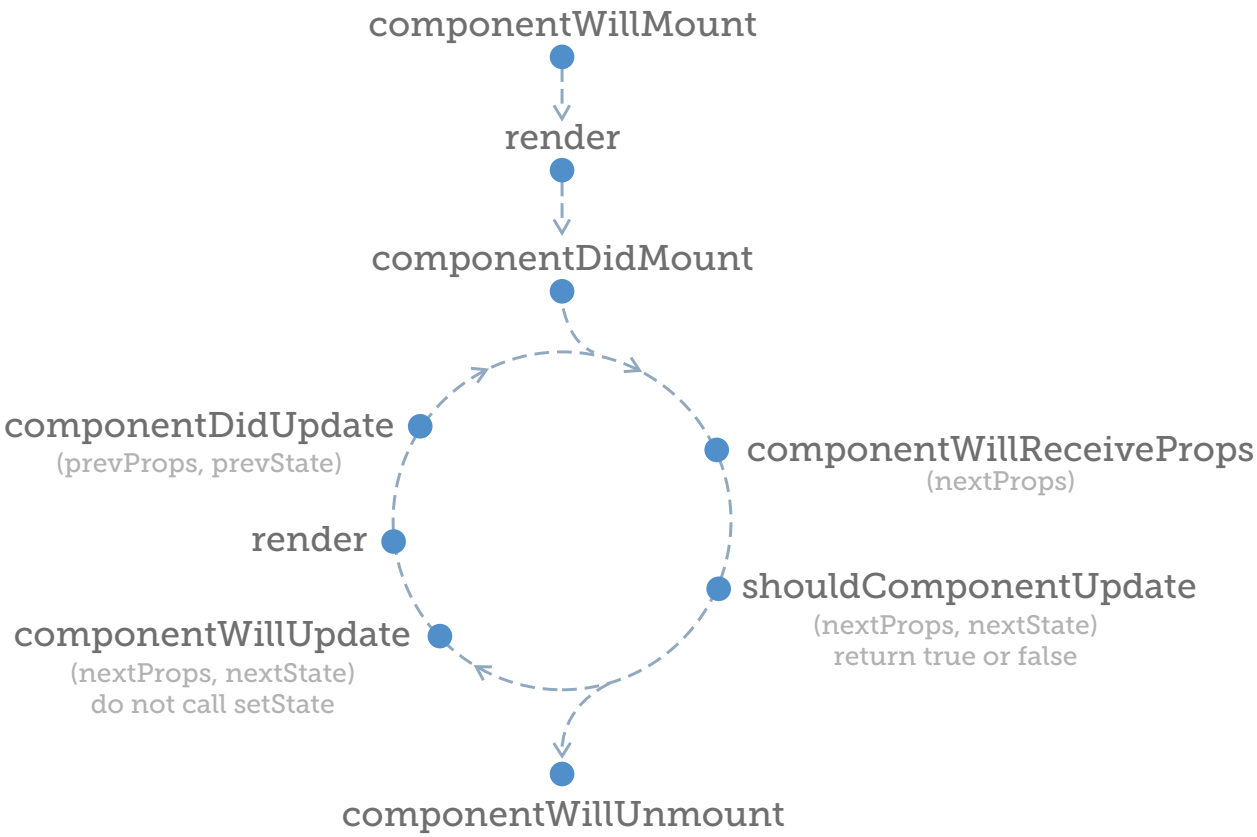
Uncontrolled inputs can be given a `defaultValue` prop, and the component will be initialized with that value. After that initial render though, if you want to change the value you'll need to use a controlled input.

13 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:



Usage

Tapping in to 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.

Here's an example that uses every lifecycle method. Run it, click the button, and watch the console to see the order in which they're called.

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

class LifecycleDemo extends React.Component {
  state = {counter: 0};

  constructor(props) {
    super(props);
    console.log('Constructing...');
    console.log('State already set to:', this.state);
  }

  componentWillMount() {
    console.log('About to mount...');
  }

  componentDidMount() {
    console.log('Mounted.');
  }

  componentWillReceiveProps(nextProps) {
    console.log('Current props:', this.props);
    console.log('Next props:', nextProps);
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('Deciding to update');
    return true;
  }

  componentWillUpdate(nextProps, nextState) {
    console.log('About to update...');
```

```
  }

  componentDidUpdate() {
    console.log('Updated.');
  }

  componentWillUnmount() {
    console.log('Goodbye cruel world.');
  }

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

  render() {
    console.log('Rendering...');
    return (
      <div>
        <span>Counter: {this.state.counter}</span>
        <button onClick={this.handleClick}>
          Click to increment
        </button>
      </div>
    );
  }
}

ReactDOM.render(
  <LifecycleDemo/>,
  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, as shown above, then it will already be set by the time the constructor executes.

componentWillMount: Called immediately *before* your component mounts for the first time. Children are mounted before the parent, so each child's `componentWillMount` will be called called before the parent.

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.

Updating

These are called, in order, before and after each render. None of them are called during the initial render.

componentWillReceiveProps(nextProps): This is passed the `nextProps` that will be received. Old props are still available as `this.props`, and you can call `this.setState` if necessary.

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): Rendering is a foregone conclusion at this point. Use this to do any prep work before `render` is called. You *cannot* call `this.setState` from within this method.

render: You know this one well. It fits in the lifecycle right between `componentWillUpdate` and `componentDidUpdate`.

componentDidUpdate: Render is done. You can use this opportunity to operate on the DOM if you need to. Prior to this, DOM nodes could still be in flux.

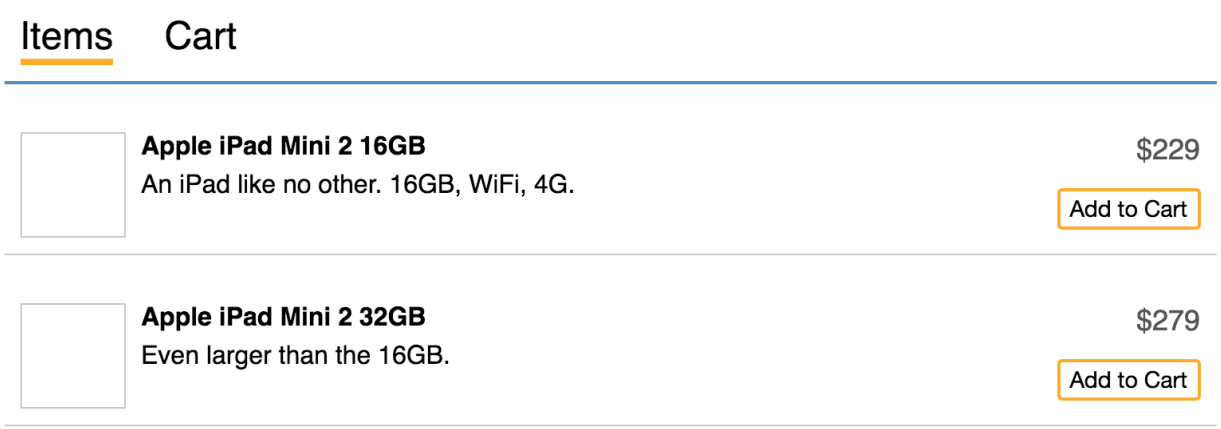
Unmounting

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.

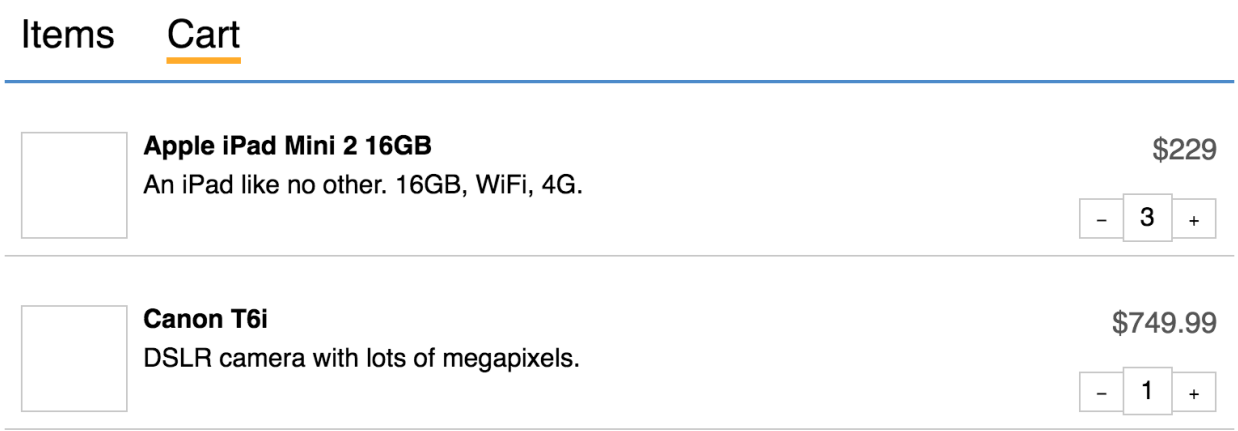
14 Example: Shopping Site

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:

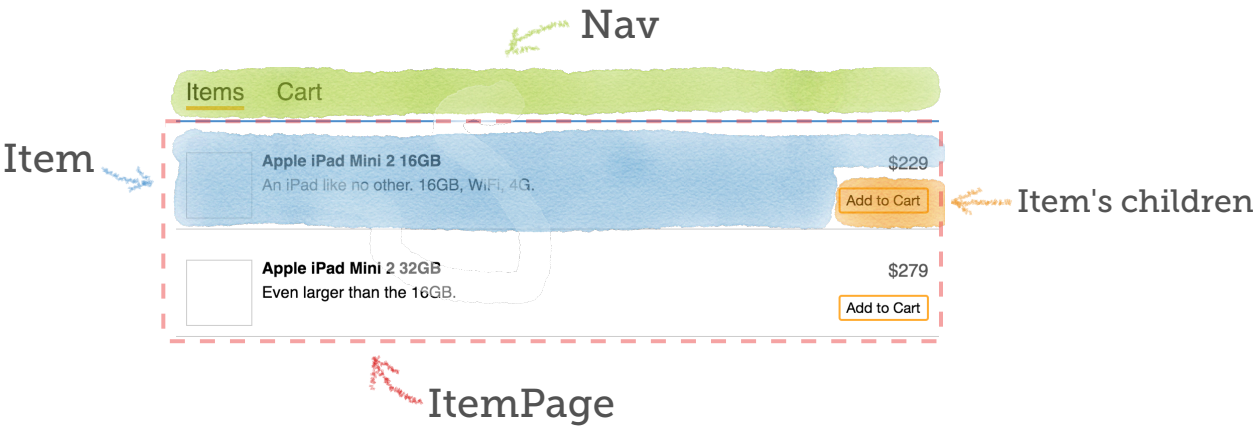


And it will have a shopping cart showing the selected items, with +/- buttons and a count for each one:

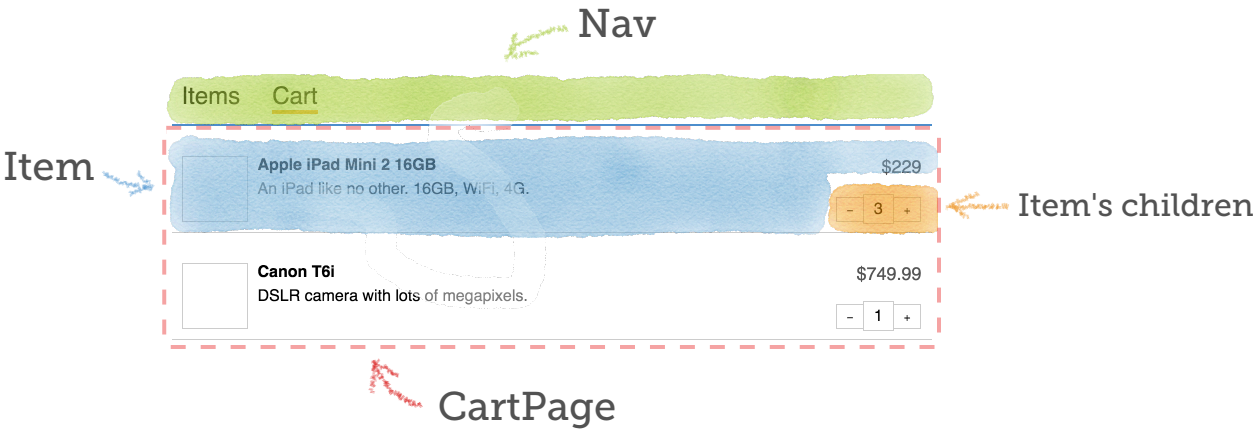


Break Into Components

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 larger `render` functions with 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:

```
$ create-react-app shopper && cd shopper
```

Open up `src/App.js` and replace its contents with this to get started:

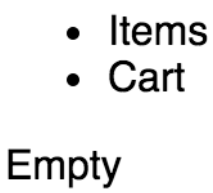
```
import React from 'react';
import Nav from './Nav';
import './App.css';

class App extends React.Component {
  renderContent() {
    return (
      <span>Empty</span>
    );
  }

  render() {
    return (
      <div className="App">
        <Nav/>
        <main className="App-content">
          {this.renderContent()}
        </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"><a>Items</a></li>
      <li className="App-nav-item"><a>Cart</a></li>
    </ul>
  </nav>
);

export default Nav;
```

Start up the app and see what we have:

```
$ npm start
```



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-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:

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.

Initialize the state at the top, and add a function to select a tab:

```
class App extends Component {
  state = {
    activeTab: 0
  };

  handleTabChange = (index) => {
    this.setState({
      activeTab: index
    });
  }
```

```
  // ...
}
```

Now, we need to pass the `handleTabChange` function down into `Nav`, along with the `activeTab` so it can render correctly. We'll also update `renderContent` to display the active page.

```
class App extends Component {
  // ...

  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0: return <span>Items</span>;
      case 1: return <span>Cart</span>;
    }
  }

  render() {
    let {activeTab} = this.state;
    return (
      <div className="App">
        <Nav activeTab={activeTab} onTabChange={this.handleTabChange} />
        <main className="App-content">
          {this.renderContent()}
        </main>
      </div>
    );
  }
}
```

Update the `Nav` component in `src/Nav.js` to use the new props:

```
const Nav = ({ activeTab, onTabChange }) => (
  <nav className="App-nav">
    <ul>
      <li className={`App-nav-item ${activeTab === 0 && 'selected'}`}>
        <a onClick={() => onTabChange(0)}>Items</a>
      </li>
```

```
      <li className={`App-nav-item ${activeTab === 1 && 'selected'}`}>
        <a onClick={() => onTabChange(1)}>Cart</a>
      </li>
    </ul>
  </nav>
);
```

Then add some styles to the bottom of `src/App.css`:

```
.App-nav-item a {
  cursor: pointer;
}
.App-nav-item a:hover {
  color: #999;
}
.App-nav-item.selected {
  border-bottom-color: #FFAA3F;
}
```

Here's what it should look like now:

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(0)}` looks better.

But remember, 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.

Creating Functions in Render: Bad?

Another note about this: it's generally a bad idea to create new functions to pass into props like the code above, but it's worth understanding why, so you can apply the rule when it makes sense.

The reason it's bad to create new functions is for performance: not because functions are expensive to create, but because passing a new function down to a *pure* component every time it renders means that it will always see "new" props, and therefore always re-render (needlessly).

Avoid creating a new function in render when (a) the child component receiving the function prop is *pure* and (b) you expect the parent component to re-render often.

Function components and classes that extend `Component` are not pure. They will *always* re-render, even if their props haven't changed. If a class extends `PureComponent` instead, though, it *is* pure and it will skip re-rendering when its props are unchanged. Avoiding needless re-renders is the easiest way to improve performance in a React app.

The `Nav` component above will probably not re-render very often, but for the sake of example, how would you get rid of the arrow functions? Here's one way to do it:

```
const Nav = ({ activeTab, onTabChange }) => (
  <nav className="App-nav">
    <ul>
      <li className={`App-nav-item ${activeTab === 0 && 'selected'}`}>
        <NavLink index={0} onClick={onTabChange}>Items</NavLink>
      </li>
      <li className={`App-nav-item ${activeTab === 1 && 'selected'}`}>
        <NavLink index={1} onClick={onTabChange}>Cart</NavLink>
      </li>
    </ul>
  </nav>
);

class NavLink extends React.Component {
  handleClick = () => {
    this.props.onClick(this.props.index);
  }

  render() {
    return (
```

```
      <a onClick={this.handleClick}>
        {this.props.children}
      </a>
    );
  }
}
```

Here, the `handleClick` function will only be created *once*, when the component is rendered the first time. This fixes the problem (`render` no longer creates a function), but at the expense of more complex code (a whole new component!). It's a tradeoff, and one worth considering when you run into this with your own projects.

Create ItemPage

Let's turn our attention to the list of items for sale. We'll create a new component to display this data. Create a file called `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>
  );
}
ItemPage.propTypes = {
  items: PropTypes.array.isRequired
};

export default ItemPage;
```

Nothing new here: we're just iterating over a list of items and rendering them out. Since the individual items are going to 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 Mini 2 16GB",
    description: "An iPad like no other. 16GB, WiFi, 4G.",
    price: 229.00
  },
  {
    id: 1,
    name: "Apple iPad Mini 2 32GB",
    description: "Even larger than the 16GB.",
    price: 279.00
  },
  {
    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};
```

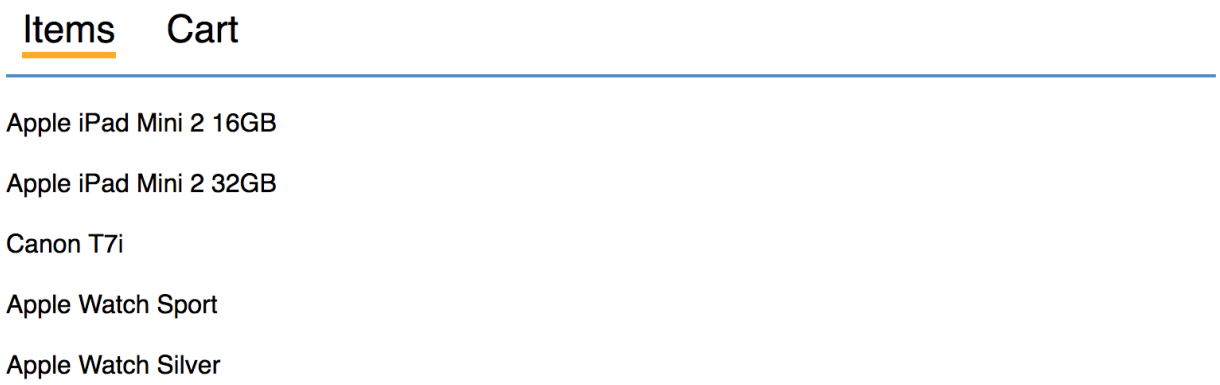
Then we will pass those items into the new `ItemPage` component. Back in `App.js`, import the `ItemPage` and the static `items` data.

```
import ItemPage from './ItemPage';
import {items} from './static-data';
```

And then update the `renderContent` function to use the new ItemPage:

```
class App extends Component {
  // ...
  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0: return <ItemPage items={items}/>
      case 1: return <span>Cart</span>;
    }
  }
  // ...
}
```

Now you should see some items rendering!



Create Item

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 `li` from this component. We don't want to force users of `Item` to put it inside a `<ul>`. The parent can put this item into an `li` if it chooses to.

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>
      <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 `<li>` we're now rendering the `Item` component instead of just the item's name. Also new is the `onAddToCart` prop and its associated propType.

At this point, the code won't work, and there will be a warning about the missing `onAddToCart` prop.

To complete the chain of passed-down props, we need to pass something to `ItemPage`'s `onAddToCart` prop from inside the `App` component. While we're at it, we may as well actually add the items to a cart! That will require a new `cart` property inside state.

Let's walk through it. First, in `App.js`, add a `cart` property to the initial state, and initialize it to an empty array:

```
class App extends Component {
  state = {
    activeTab: 0,
    cart: []
  }

  // ...
}
```

Then update `renderContent` to pass a function as the `onAddToCart` prop:

```
class App extends Component {
  // ...

  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0:
```

```
        return (
          <ItemPage
            items={items}
            onAddToCart={this.handleAddToCart} />
        );
      case 1:
        return <span>Cart</span>;
    }
  }

  // ...
}
```

The function `this.handleAddToCart` doesn't exist yet, so create that next. It will accept an item, and add the item to the cart:

```
class App extends Component {
  // ...

  handleAddToCart = (item) => {
    this.setState({
      cart: [...this.state.cart, item.id]
    });
  }

  // ...
}
```

What this is doing is setting the `cart` state to a copy of the current cart, plus one new item.

There's some new ES6 syntax here: `...this.state.cart` is the *spread* operator, and it expands the given array into its individual items. Here's an example of the spread operator:

```
// With arrays:
var a = [1, 2, 3];
var b = [a, 4];    // => [[1, 2, 3], 4]
var c = [...a, 4]; // => [1, 2, 3, 4]

// With objects (technically not ES6)
```

```
var o1 = {a: 1, b: 2};
var o2 = {...o1, c: 3}; // => {a: 1, b: 2, c: 3}
```

Note that the spread operator for arrays is officially part of ES6, but the one for objects is not. However, object spread is supported by Babel, which Create React App is using under the hood to turn our code into browser-compatible ES5.

Updating State Immutably

You might be wondering… why not just modify state directly and then call `setState`, like this?

```
this.state.cart.push(item.id);
this.setState({cart: this.state.cart});
```

This is a bad idea for two reasons (even though it would work in this example).

You should *never* modify ("mutate") state or its child properties directly. Don't do `this.state.something = x`, and don't do `this.state = x`. React relies on you to call `this.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.

Mutating state directly can lead to odd bugs, and components that are hard to optimize.

Here's an example. Recall that one of the common ways to optimize a React component is to make it "pure," meaning that it only re-renders when its props change. This can be done automatically by extending `React.PureComponent` instead of `React.Component`, or manually by implementing the `shouldComponentUpdate` lifecycle method to compare `nextProps` with current props. If the props look the same, it skips the render, and saves some time.

Here is a simple component that renders a list of items. Notice that it extends `React.PureComponent`:

(If you have the example code, this code is under the "impure-state" folder.)

```
class ItemList extends React.PureComponent {
  static propTypes = {
    items: PropTypes.array.isRequired
  }

  render() {
```

```
    return (
      <ul>
        {this.props.items.map(item =>
          <li key={item.id}>{item.value}</li>
        )}
      </ul>
    );
  }
}
```

Now, here is a tiny app that renders the `ItemList` and allows you to mutably or immutably add items to the list:

```
class App extends Component {
  // Initialize items to an empty array
  state = {
    items: []
  }

  // Initialize a counter that will increment
  // for each item ID
  nextItemId = 0;

  makeItem() {
    // Create a new ID and use
    // a random number as the value
    return {
      id: this.nextItemId++,
      value: Math.random()
    };
  }

  // The Right Way:
  // copy the existing items and add a new one
  addItemImmutably = () => {
    this.setState({
      items: [
        ...this.state.items,
        this.makeItem()
      ]
```

```
    });
  }

  // The Wrong Way:
  // mutate items and set it back
  addItemMutably = () => {
    this.state.items.push(this.makeItem());
    this.setState({ items: this.state.items });
  }

  render() {
    return (
      <div>
        <button onClick={this.addItemImmutably}>
          Add item immutably (good)
        </button>
        <button onClick={this.addItemMutably}>
          Add item mutably (bad)
        </button>
        <ItemList items={this.state.items}/>
      </div>
    );
  }
}
```

Try it out. Click the immutable Add button a few times and notice how the list updates as expected. Then click the mutable Add button and notice how the new items don't appear, even though state is being changed.

This happens because `ItemList` is *pure*, and because pushing a new item on the `this.state.items` array does not replace the underlying array. `ItemList` will notice that its props haven't changed and it will not re-render.

Finally, click the immutable Add button again, and watch how the `ItemList` re-renders with all the missing (mutably-added) items.

That is a long way of saying that you shouldn't mutate state, even if you immediately call `setState`. Optimized components (ones that implement `shouldComponentUpdate`) might not re-render if you do.

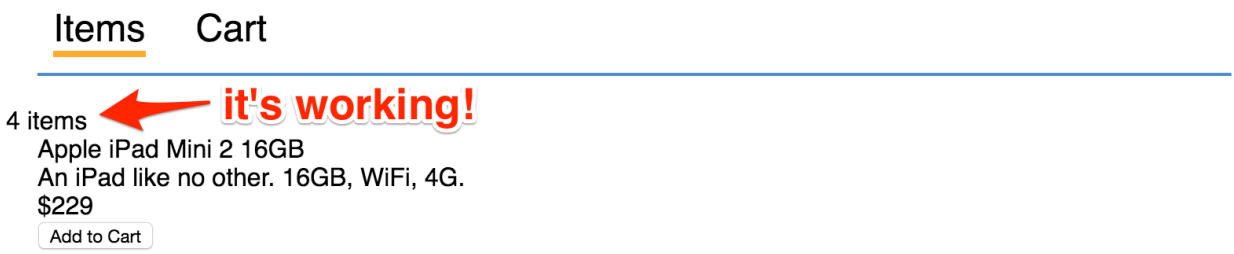
Instead, you should create *new* objects and arrays when you call `setState`, which is what we did above with the spread operator.

Back to the Code

Let's 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 `App.js`'s `render` method:

```
<div>
  {this.state.cart.length} items
</div>
```

Now try it again. The item count should increase each time you click "Add to Cart." Once you verify that it's working, you can remove that debugging code.

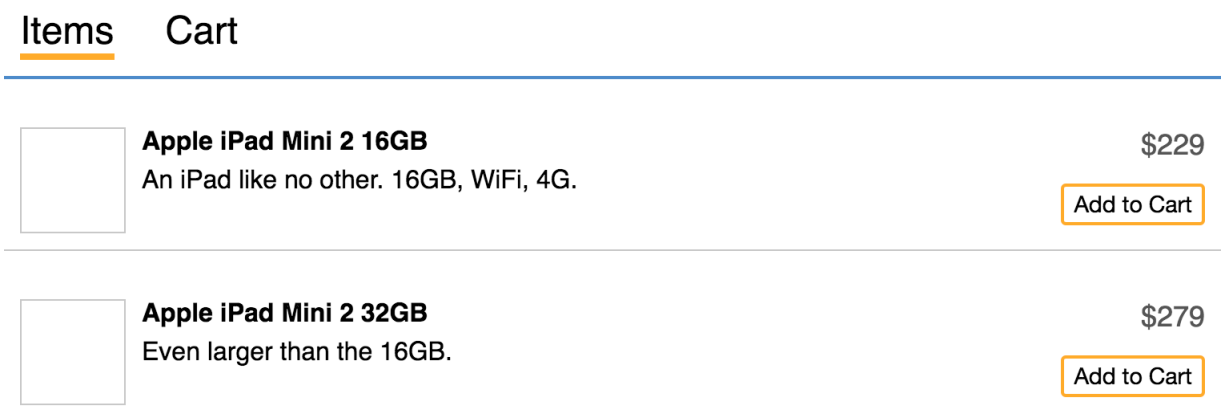


Let's add some CSS to make it look a little better. In `Item.css` (which you created earlier, but is currently empty), add this CSS:

```
.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:



Create CartPage

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;
```

This simply renders the items that it's given. You'll notice that we aren't passing an `onAddToCart` handler to `Item`, which will cause a prop warning. We'll fix that shortly.

Pass Items to CartPage

The cart needs items to display. The trouble is, our `cart` state just contains item indices, not actual items. Additionally, there can be duplicate indices when an item is added multiple times. We'll need to massage the `cart` state into an array of items suitable to pass into `CartPage`.

In `App.js`, import `CartPage` at the top:

```
import CartPage from './CartPage';
```

Then add a `renderCart` function, and update `renderContent` to call it.

```
class App extends Component {
  // ...

  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0:
        return (
          <ItemPage
            items={items}
            onAddToCart={this.handleAddToCart} />
        );
      case 1:
        return this.renderCart();
    }
  }

  renderCart() {
    // Count how many of each item is in the cart
    let itemCounts = this.state.cart.reduce((itemCounts, itemId) => {
      itemCounts[itemId] = itemCounts[itemId] || 0;
      itemCounts[itemId]++;
```

```
      return itemCounts;
    }, {});

    // Create an array of items
    let cartItems = Object.keys(itemCounts).map(itemId => {
      // Find the item by its id
      var item = items.find(item =>
        item.id === parseInt(itemId, 10)
      );

      // Create a new "item" that also has a 'count' property
      return {
        ...item,
        count: itemCounts[itemId]
      }
    });

    return (
      <CartPage items={cartItems} />
    );
  }

  // ...
}
```

There are a couple things in that code you may not have seen before.

Array.prototype.reduce

The `reduce` function works like a summation operation. It takes an optional initial value (`{}` here), and then calls the given function with the accumulated total and the current array item. The value returned from the function becomes the new `total`, and it moves on to the next item in the array. When `reduce` is done, it returns the final `total`. We're using `reduce` to build up an object where the keys are `itemIds` and the values are the number of each item in the cart.

Here's a smaller, simpler example of `reduce` in action:

```
let a = [1, 2, 3, 4];
let total = a.reduce((sum, value) => {
```

```
  return sum + value;
}, 0);

// The arrow function is called 4 times:
//   (0, 1) => returns 1
//   (1, 2) => returns 3
//   (3, 3) => returns 6
//   (6, 4) => returns 10
// Then reduce returns the last return value (10)
```

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];
}
```

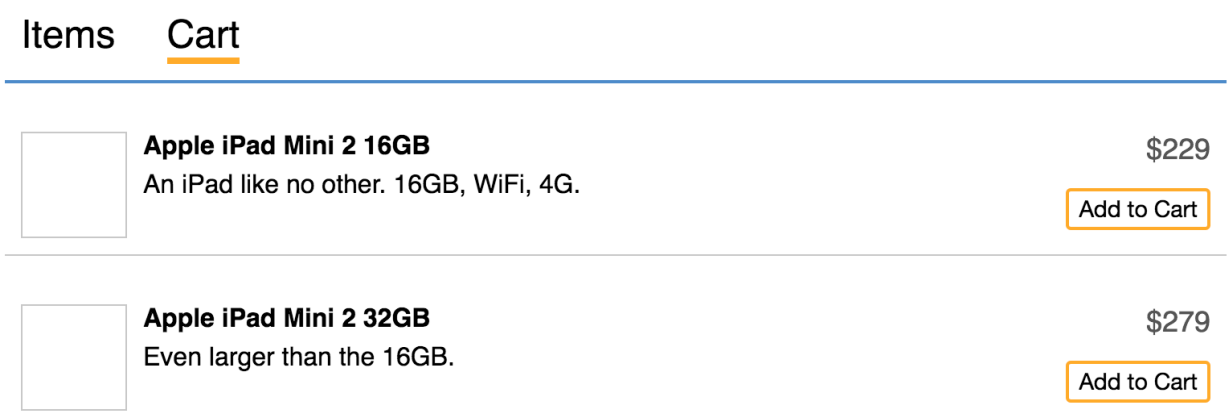
Any time you want to iterate over the values of an array to create an aggregate result, consider using `reduce`.

Object.keys

`Object.keys` returns an array of the keys in an object. For instance `Object.keys({a: 1, b: 2})` would return `['a', 'b']`. Objects don't have a built-in iterator `forEach` function like arrays do, so `Object.keys` makes it easier to iterate over the keys of an object.

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. We still have the "Add to Cart" buttons, and we're getting propType warnings about `onAddToCart`. We'll fix both of those next.



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 `CartPage`? 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 more conditional logic, making `Item` harder and 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 a `<button>`, we now have `{children}` instead. Also, `Item` no longer needs to know about the `onAddToCart` function, so we've removed its propType.

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 those functions, so head let's back to `App.js` and create those. At the bottom of `renderCart`, we need to pass the extra props to `CartPage`:

```
class App extends Component {
  // ...
  renderCart() {
    // ...
    return (
```

```
        <CartPage
          items={cartItems}
          onAddOne={this.handleAddToCart}
          onRemoveOne={this.handleRemoveOne} />
      );
    }
    // ...
}
```

We already have the `handleAddToCart` function that takes an item and adds its id to the cart, so we can just reuse that here.

We need to create the `handleRemoveOne` function. It will take an item, find an occurrence of that item in the cart, and then update the `cart` state to remove that occurrence.

```
class App extends Component {
  // ...
  handleRemoveOne = (item) => {
    let index = this.state.cart.indexOf(item.id);
    this.setState({
      cart: [
        ...this.state.cart.slice(0, index),
        ...this.state.cart.slice(index + 1)
      ]
    });
  }
  // ...
}
```

We're using the array *spread* operator twice here, in order to avoid mutating the state. We're taking the left half of the array (up to the item we want to remove) and concatenating it with the right half of the array (everything after the item being removed).

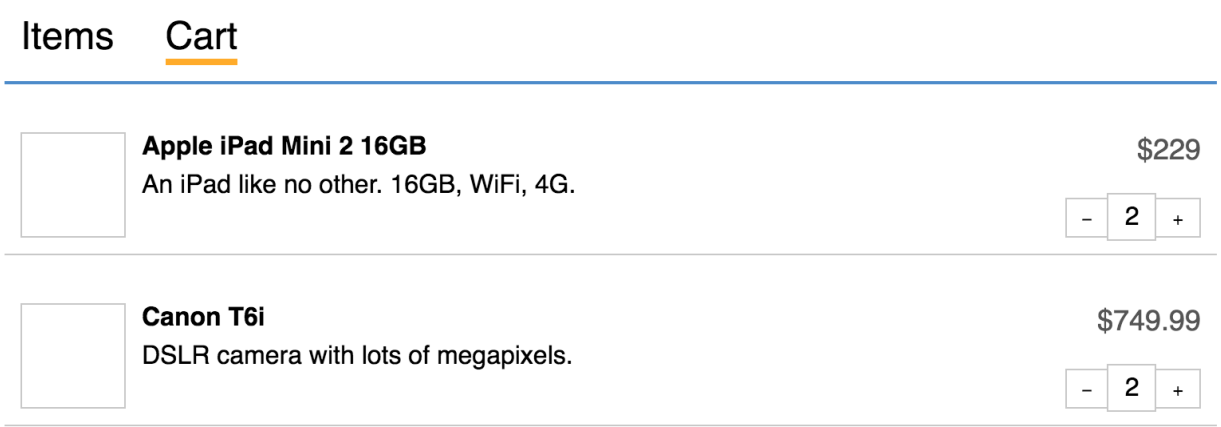
This is a little more work than if we were to allow mutations but it is a good habit to get into. It'll be especially important if you start using Redux in the future. Redux relies heavily on immutability.

To wrap up the add/remove buttons, let's give them a bit of style. Add this CSS to `CartPage.css`:

```
.CartPage-items {
```

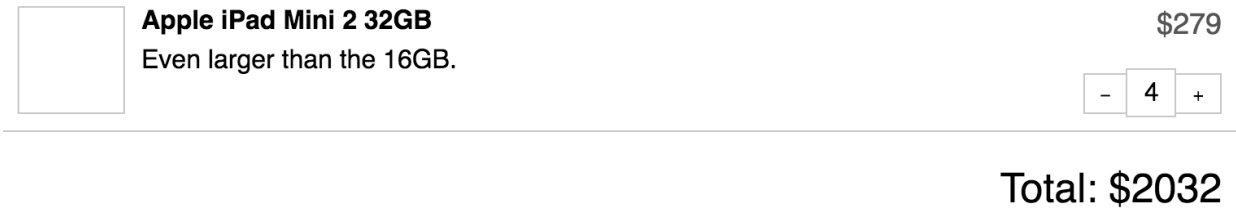
```
  margin: 0;
  padding: 0;
}
.CartPage-items li {
  list-style: none;
}
.CartItem-count {
  padding: 5px 10px;
  border: 1px solid #ccc;
}
.CartItem-addOne,
.CartItem-removeOne {
  padding: 5px 10px;
  border: 1px solid #ccc;
  background: #fff;
}
.CartItem-addOne {
  border-left: none;
}
.CartItem-removeOne {
  border-right: none;
}
```

Here it is in all its glory:

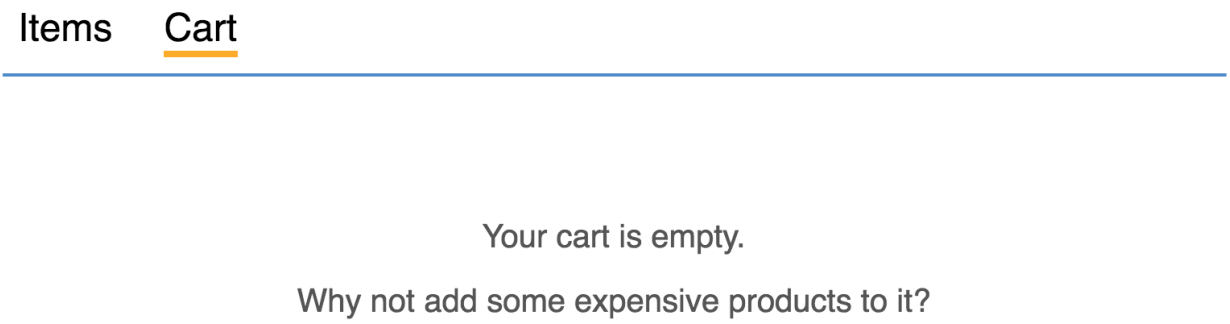


Exercises

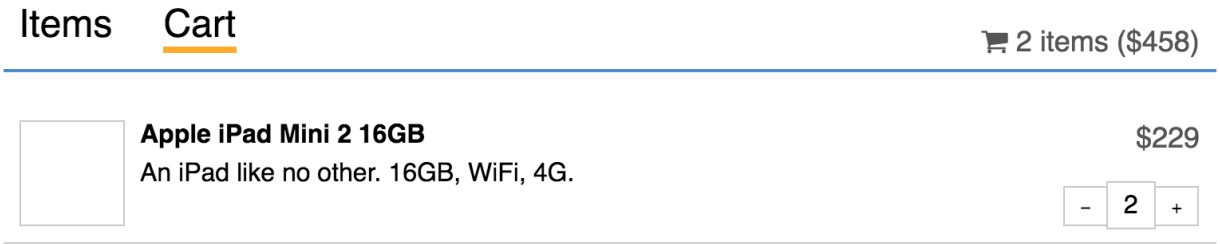
1. It would be nice if the shopping cart told us how many thousands of dollars we were about to spend on Apple products, rather than letting us guess. Add a "Total" to the bottom of the Cart page similar to this:



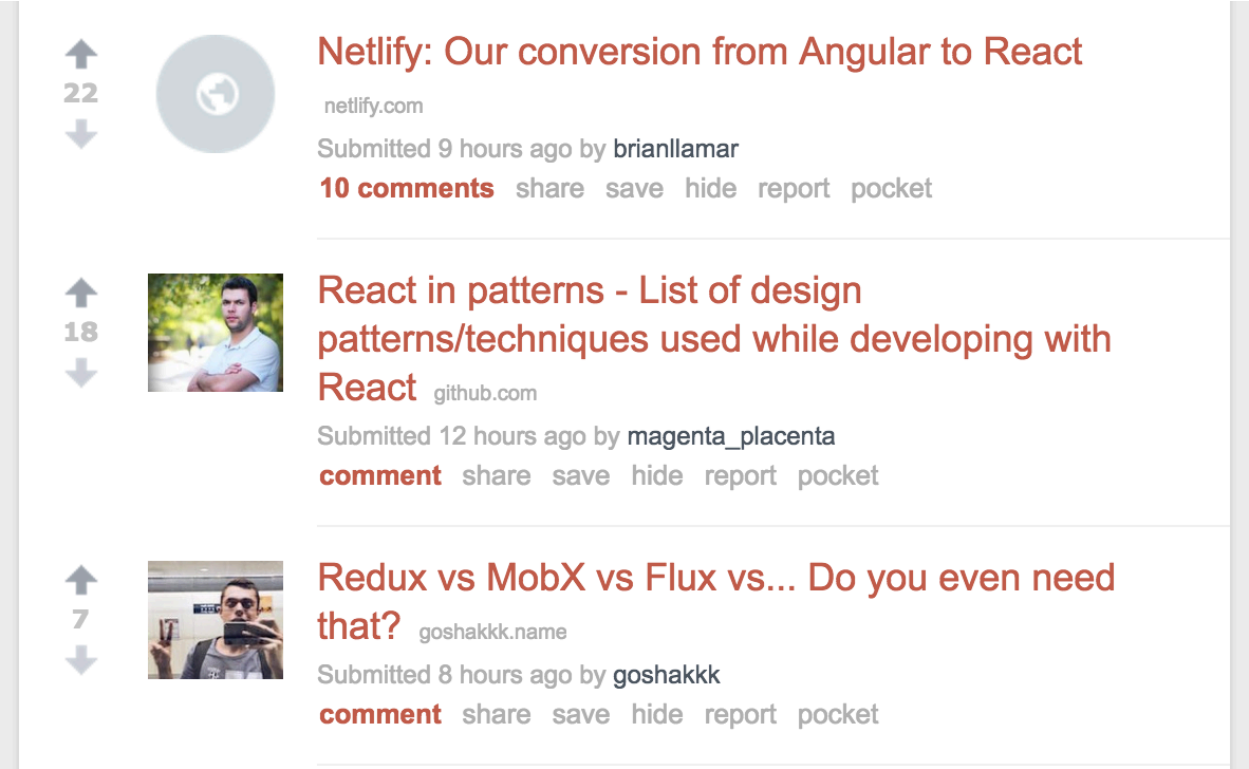
2. The Cart page is completely blank when the cart is empty. Modify it to display "Your cart is empty" when there are no items in the cart, something like this:



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).



4. Build a simplified version of Reddit, modeled after this screenshot:



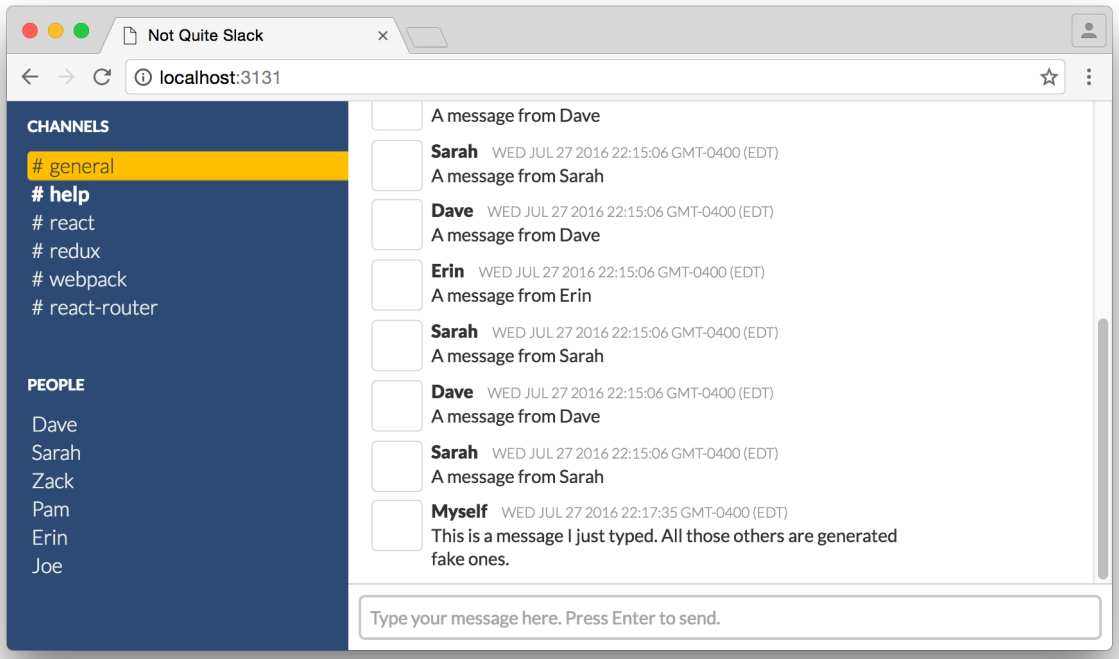
Here are the features it should have:

- 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.

15 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. From here, you can go on to learn about AJAX, Webpack, 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.

AJAX

Learning how to connect your app to a server is fun, because you get to see your app come alive with real data.

React doesn't have any built-in ways to talk to servers, so you'll need to use an HTTP library. There are a number of them available. I like fetch because it will eventually be part of the JavaScript standard library, but axios and superagent are also good choices.

How and where should you make HTTP requests? A common practice is to fetch data within the `componentDidMount` lifecycle method. When the response comes back, call `this.setState` to save that data, and use props to pass it down to the components that need it.

As with all state, it's best to keep it as high up in the component hierarchy as possible. For instance, if a `Page` contains multiple `List` components, don't have each `List` fetch its own data – fetch that data in the `Page` component, and pass it down to the `Lists` as props.

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 new tools to learn for testing React. You'll need a way to make assertions (Chai), probably want a library for setting up spies (Sinon), and will definitely want to use Enzyme, a library for testing React components created by Airbnb.

I put together an article to get you started with testing React here.

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. Egghead.io has a nice quick introduction to Webpack as well, at https://egghead.io/lessons/javascript-intro-to-webpack.

You should know that the `create-react-app` tool we've been using has a production build mode (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 build and exposes it for you to customize.

ES6

You've already seen some of this 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 on Github 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.

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 fall into the trap of thinking that learning "Pure React" is only for beginners, and that you'll basically 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 stack 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 this article: https://daveceddia.com/react-project-structure/

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@daveceddia.com or contact me on Twitter @dceddia.