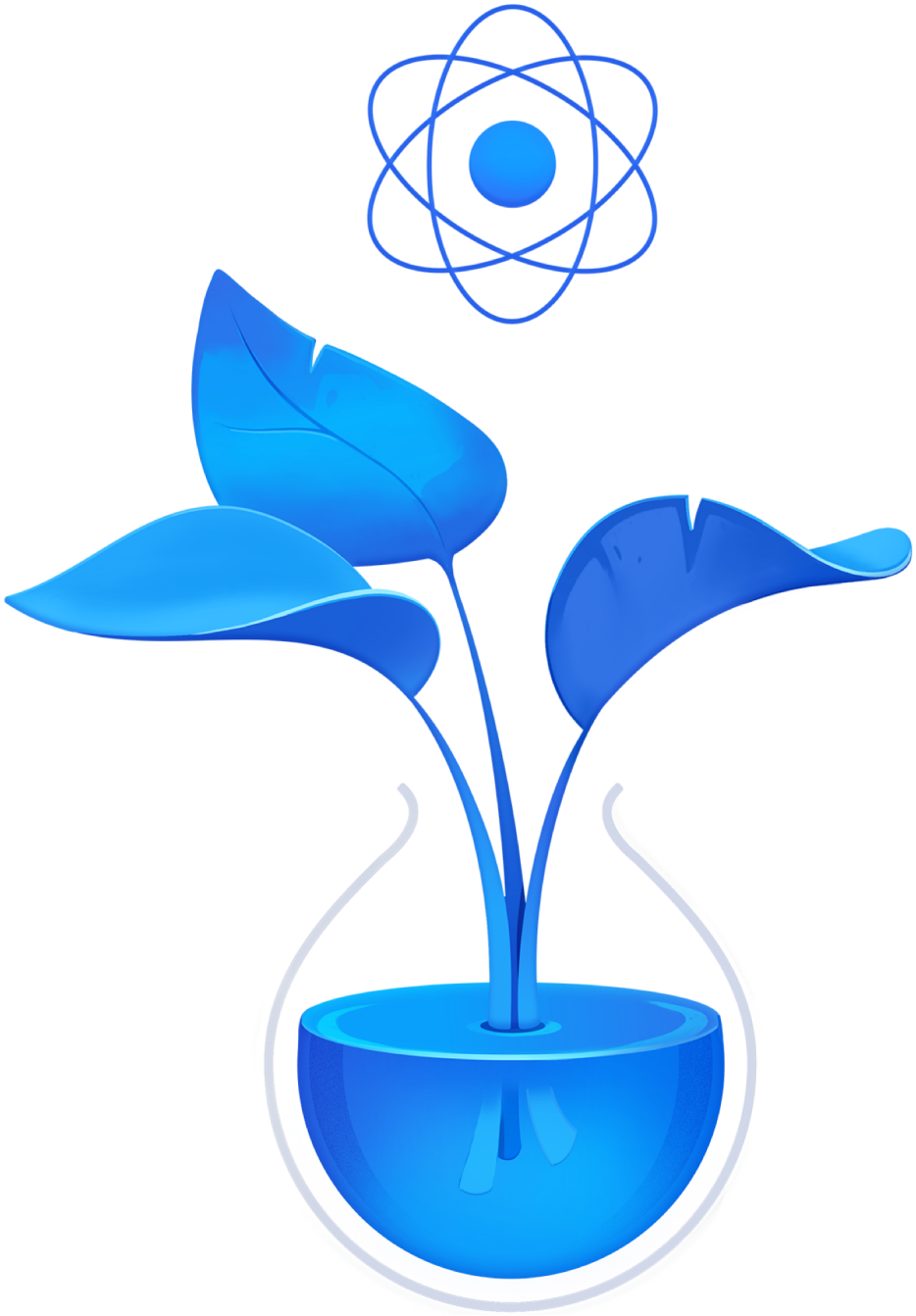


Pure React

DAVE CEDDIA

Contents

1 Welcome

Thanks for checking out this sample of *Pure React*!

Too often, sample chapters drop the reader into the middle of the book with little context.

Not this one, though! In these two chapters, you will:

- Set up your React development environment in just 2 minutes (no boilerplate required!)
- Write your first app with React
- Learn to carve a UI mockup into components, then code them

I hope you find it useful. If you have any questions as you work through it, don't hesitate to email me at dave@daveceddia.com.

If you want the full book, you can buy it at https://daveceddia.com/pure-react Enjoy!

Environment Setup

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

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

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

Prerequisites

Tools

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

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

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

Create React App

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

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

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

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

Yarn

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

React Developer Tools

The React Developer Tools can be installed from here:

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

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

Knowledge

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

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

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

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

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

Project Directory

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

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

2 Hello World

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

Step 1

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

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

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

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

Step 2

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

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

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

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

```
  );
}

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

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

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

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

Step 3

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

```
$ npm start
```

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

How the Code Works

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

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

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

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

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

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

In the full book, all the nitty-gritty details about JSX and how it works are explained next. But since this sample is short on space (and you want to get to the fun part), in the next section we will turn some rough sketches into real working components.

3 Example: Tweet Component

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

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

1. Sketch

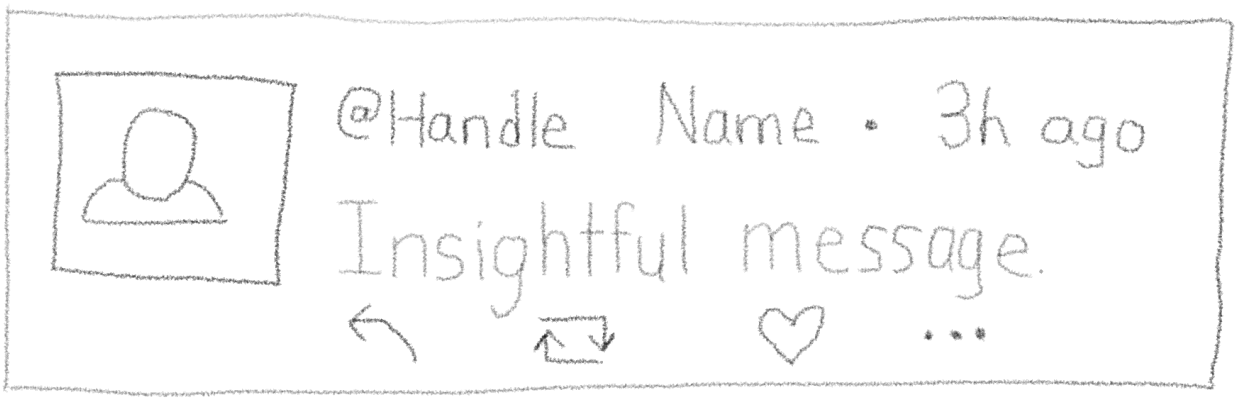
Spending a few seconds putting pen (or pencil) to paper can save you time later. Even if you can't draw (it doesn't need to be pretty).

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

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

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

Here is the sketch we'll be building from:



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

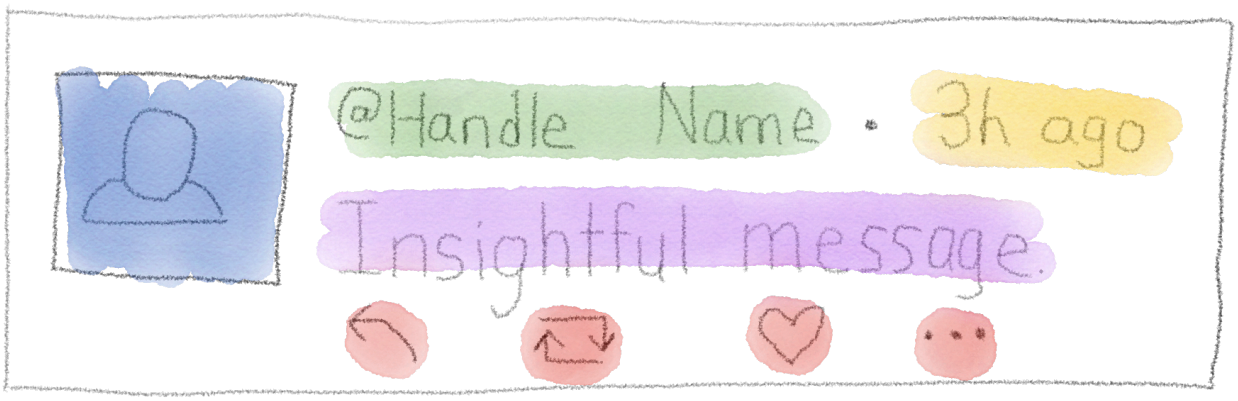
2. Carve into Components

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

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

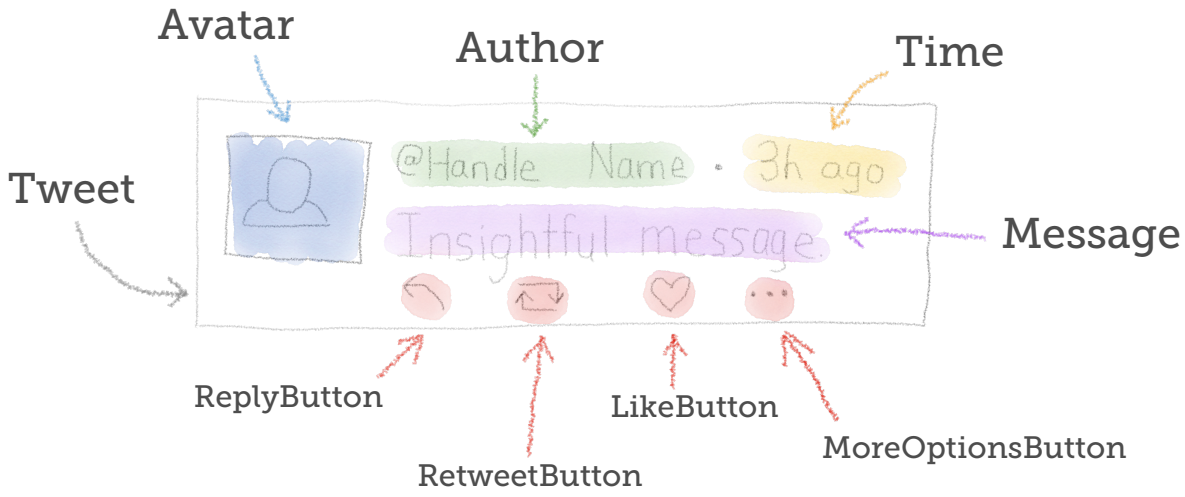
Another strategy is to make every "thing" a component. Things like buttons, chunks of related text, images, and so on.

Try it yourself, then compare with this:



3. Name the Components

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



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

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

4. Build

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

Top-Down, or Bottom-Up?

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

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

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

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

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

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

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

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

Rewriting an Existing App

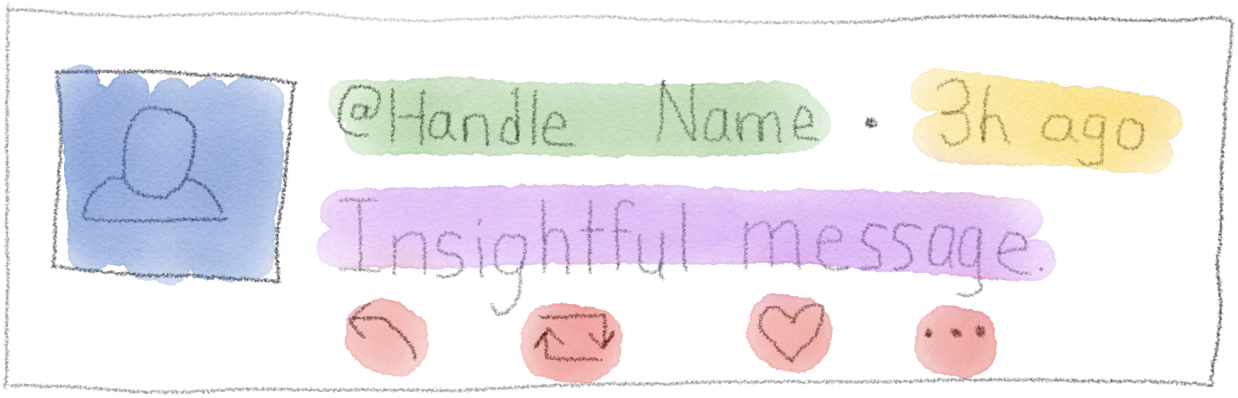
Another situation where building from the bottom is preferable is when you are converting an app to React. If you have an existing app written in another framework like AngularJS or Backbone and you want to rewrite in it React, starting at the top makes little sense, because it'll have a ripple effect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

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

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

Build the Tweet Component

Here's our blueprint again:



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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
$ npm start
```

And the page should render something like this:

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

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

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

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

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

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

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

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

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

Next we need to include `Avatar` in the `Tweet` component:

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

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

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

It's getting better:

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

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

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

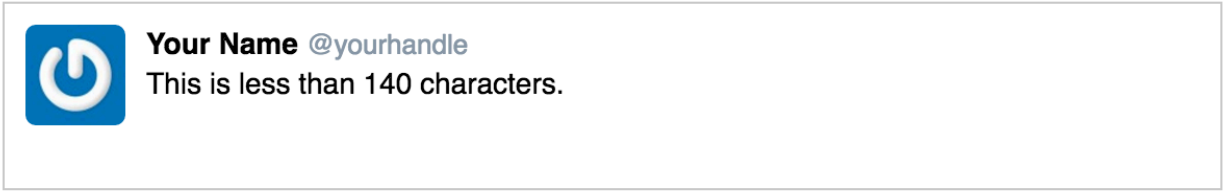
If you refresh after adding these, nothing will have changed because we still need to update `Tweet` to use these new components, so do that next:

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

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

```
.name {
  font-weight: bold;
  margin-bottom: 0.5em;
  margin-right: 0.3em;
}

.handle {
  color: #8899a6;
  font-size: 13px;
}
```



It's looking more like a real tweet now!

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

```
const Time = () => (
  <span className="time">3h ago</span>
);

const ReplyButton = () => (
  <i className="fa fa-reply reply-button"/>
);

const RetweetButton = () => (
  <i className="fa fa-retweet retweet-button"/>
);

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

const MoreOptionsButton = () => (
  <i className="fa fa-ellipsis-h more-options-button"/>
);
```

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

```
// Here's a normal function component:
function LikeButton() {
  return (
    <i className="fa fa-heart like-button"/>
  );
}

// It can be rewritten as an
// anonymous function and stored in a
// (constant) variable:
```

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

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

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

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

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

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

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

Arrow functions are a nice concise way of writing components.

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

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

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

The let and const Keywords

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

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

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

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

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

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

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

Add the Buttons and the Time

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

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

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

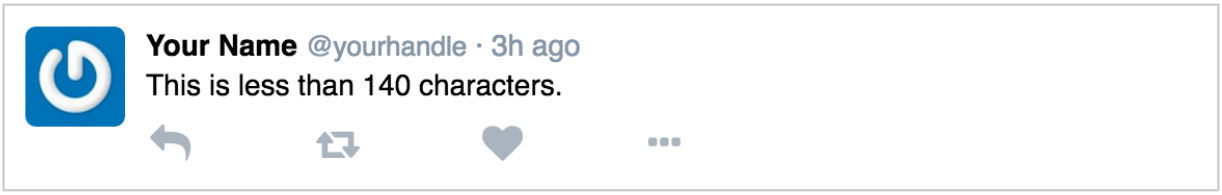
Finally, add a few more styles to cover the time and buttons:

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

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

.buttons {
  margin-top: 10px;
  margin-left: 2px;
  font-size: 1.4em;
  color: #aab8c2;
}
.buttons i {
  width: 80px;
}
```

And here we have a fairly respectable-looking tweet!



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

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

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

4 It Doesn't Have To End Here

Thanks for reading this sample chapter from *Pure React* by Dave Ceddia. I hope you enjoyed it!

The full 242-page book contains more examples like this, 42 exercises to reinforce your learning, and covers topics including:

- An introduction to React 16.13 (the latest one)
- Easy project setup with Create React App (you'll be running code within minutes)
- Debugging strategies for when things go wrong
- Master JSX syntax, including "if"s, loops, and dynamic child components
- Use props to make reusable components and communicate between them
- How PropTypes can save you debugging time and help "future you" remember how to use the components you wrote
- Use the "children" prop to render dynamic content
- How to write React in modern ES6 Javascript, with just-in-time notes about ES6 syntax
- How input controls work in React
- Where and how to properly use state in your app
- Learn the lifecycle of a component (and how to achieve the same results with the `useEffect` hook)
- Learn to use the React Context API to pass data deeply through your app without prop drilling
- Build stateful function components with React Hooks!
- How to use each hook individually, and in combination
- `useState` for simple state
- `useEffect` for side effects (and you'll learn what a "side effect" is)
- `useReducer` for more complex state
- `useContext` to easily pull data out of Context
- `useRef` for inputs and more

The complete book can be purchased at purereact.com.

You may also enjoy my blog at daveceddia.com where I have a number of free articles about React covering things from getting started to avoiding common gotchas to learning test-driven development.