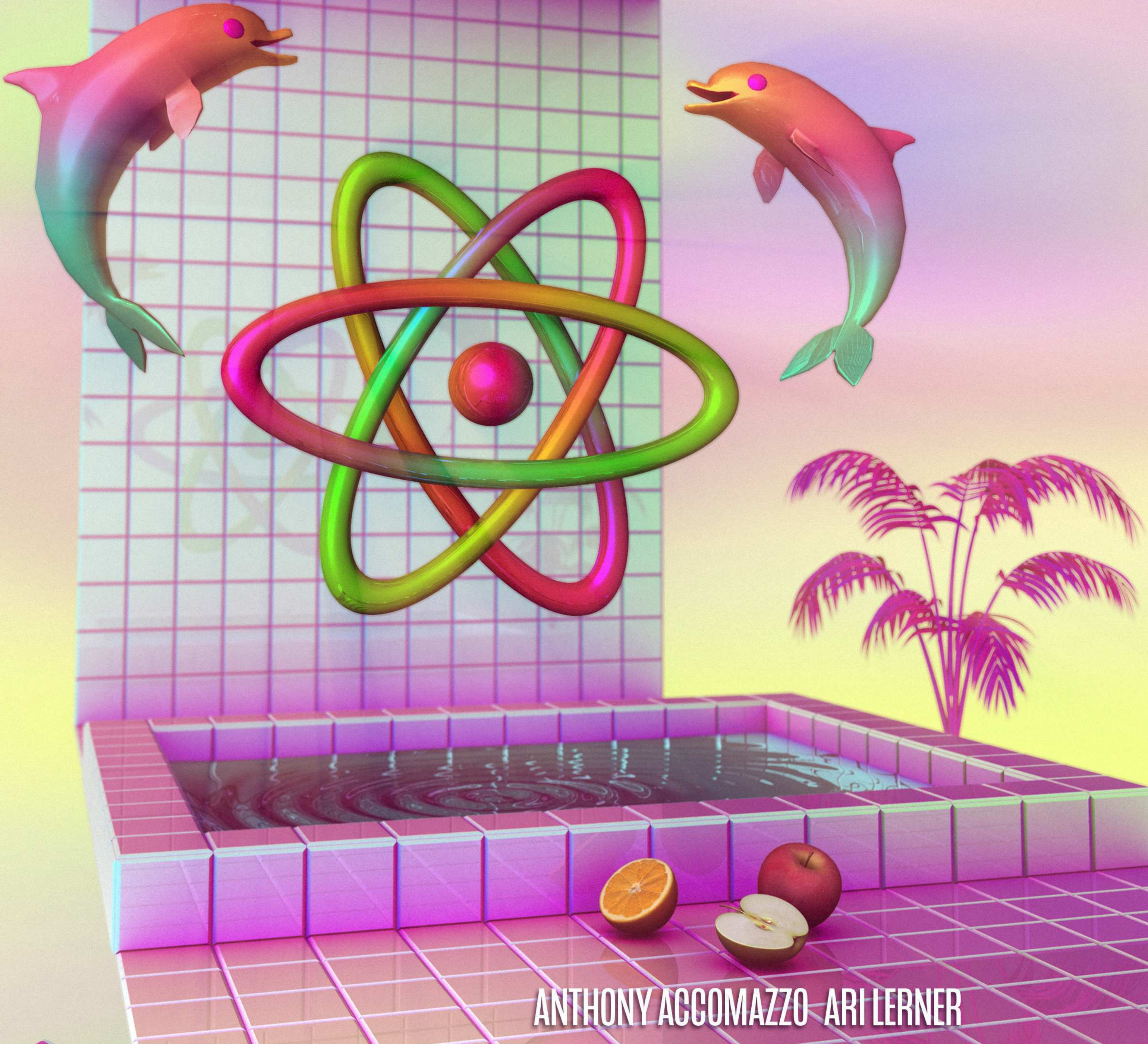
FULLSTACK REACT

The Complete Guide to ReactJS and Friends



ANTHONY ACCOMAZZO ARI LERNER
CLAY ALLSOPP DAVID GUTTMAN
TYLER MCGINNIS NATE MURRAY

FULLSTACK.io

Fullstack React

The Complete Book on ReactJS and Friends

Anthony Accomazzo, Ari Lerner, David Guttman, Nate Murray,
Clay Allsopp and Tyler McGinnis

Contents

Book Revision

Revision 13 - Covers up to React (15.3.0, 2016-08-02)

Prerelease

This book is a prerelease version and a work-in-progress.

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: react@fullstack.io[1].

Chat With The Community!

We're experimenting with a community chat room for this book using Gitter. If you'd like to hang out with other people learning React, come join us on Gitter[2]!

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow @fullstackio[3]

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: react@fullstack.io[4].

[1] mailto:react@fullstack.io?Subject=Fullstack%20React%20book%20feedback
[2] https://gitter.im/fullstackreact/fullstackreact
[3] https://twitter.com/fullstackio
[4] mailto:react@fullstack.io?Subject=react%202%20testimonial

Your first React Web Application

Building Product Hunt

In this chapter, you're going to get a crash course on React by building a simple voting application inspired by Product Hunt[5]. You'll become familiar with how React approaches front-end development and all of the fundamentals necessary to build an interactive React app from start to finish. Thanks to React's core simplicity, by the end of the chapter you'll already be well on your way to writing a variety of fast, dynamic interfaces.

We'll focus on getting our React app up and running fast. We take a deeper look at concepts covered in this section throughout the course.

Getting started

Sample Code

All the code samples are included with this course. Still, we highly recommend implementing the code yourself along with us throughout each section. Playing around with examples and sample code will help solidify and strengthen concepts.

Provided with the sample code is a very simple Node.js web server that we'll be using to run our application.

Code editor

As you'll be writing code throughout this course, you'll need to make sure you have a code editor you're comfortable working with. If you don't already have a preferred editor, we recommend installing Atom[6] or Sublime Text[7].

Node.js and NPM

For all the projects in this course, we'll need to make sure we have a working Node.js[8] development environment along with NPM.

[5]http://producthunt.com

[6]http://atom.io

[7]https://www.sublimetext.com/

[8]http://nodejs.org

There are a couple different ways you can install Node.js so please refer to the Node.js website for detailed information: https://nodejs.org/download/[9]

> If you're on a Mac, your best bet is to install Node.js directly from the Node.js website instead of through another package manager (like Homebrew). Installing Node.js via Homebrew is known to cause some issues.

The Node Package Manager (npm for short) is installed as a part of Node.js. To check if npm is available as a part of our development environment, we can open a Terminal window and type:

```
$ npm -v
```

If a version number is not printed out and you receive an error, make sure to download a Node.js installer that includes npm.

Browser

Lastly, we highly recommend using the Google Chrome Web Browser[10] to develop React apps. We'll use the Chrome developer toolkit throughout this course. To follow along with our development and debugging we recommend downloading it now.

Previewing the application

We'll be building a basic React app that will allow us to touch on React's most important concepts at a high-level before diving into them in subsequent sections. Let's begin by taking a look at a working implementation of the app.

Open up the sample code that came with the course, changing to the `voting_app/` directory in the terminal:

```
$ cd voting_app/
```

> If you're not familiar with `cd`, it stands for "change directory." If you're on a Mac, do the following to open terminal and change to the proper directory:
>
> 1. Open up `/Applications/Utilities/Terminal.app`.
> 2. Type `cd`, without hitting enter.
> 3. Tap the spacebar.
> 4. In the Finder, drag the `voting_app/` folder on to your terminal window.
> 5. Hit Enter.
>
> Your terminal is now in the proper directory.

[9] https://nodejs.org/download/

[10] https://www.google.com/chrome/

Throughout the book, a codeblock starting with a `$` signifies a command to be run in your terminal.

Windows Users: We'll be using Linux/Mac-style commands on the command line throughout this book. We'd highly recommend installing Cygwin[11] as it will let you run commands just as we have printed here.

First, we'll need to use `npm` to install all our dependencies:

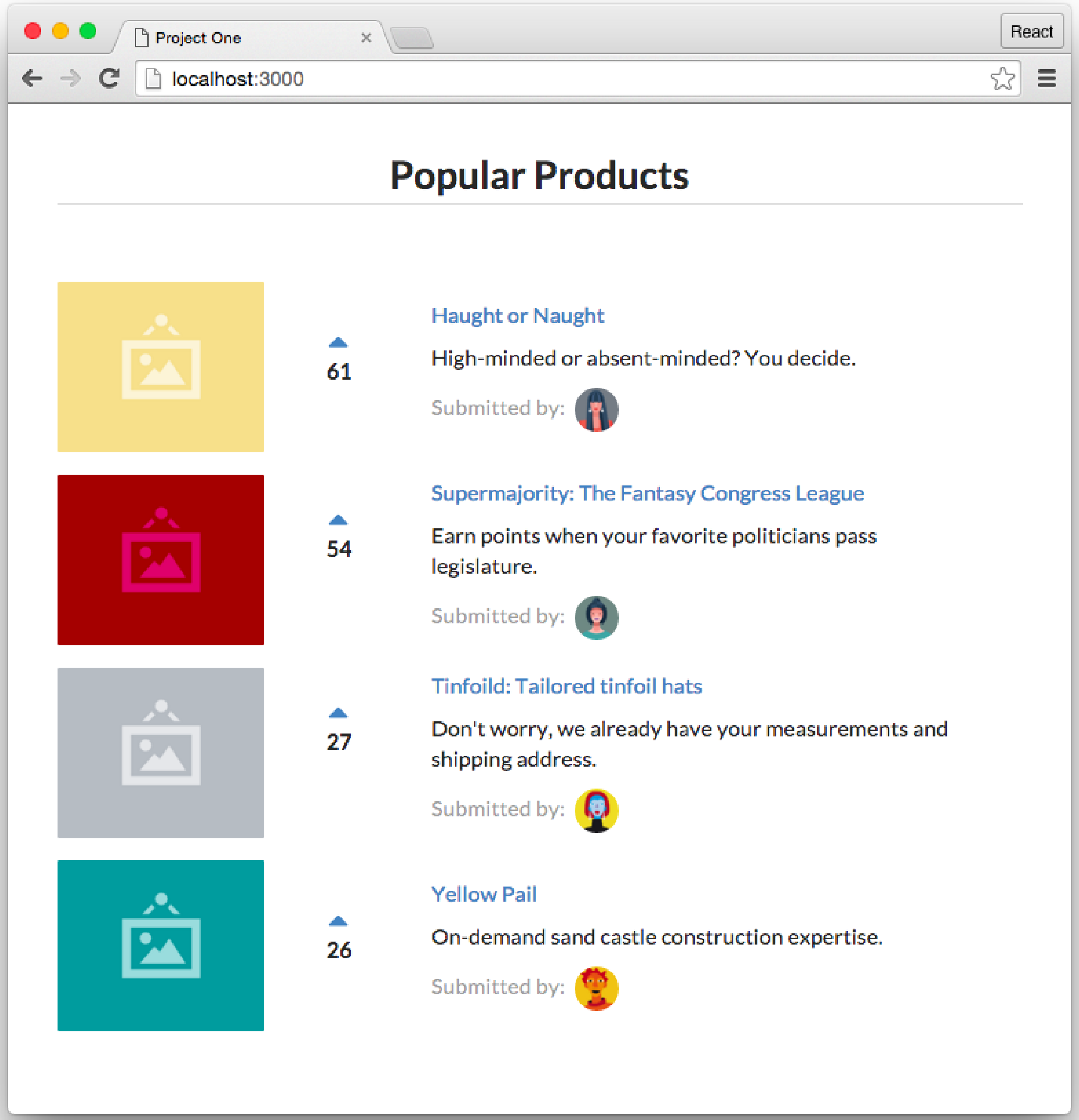
```
$ npm install
```

With our dependencies installed, we can boot the server using the `npm run` script `server`:

```
$ npm run server
```

Heading to our browser, we can view the running application at the URL: http://localhost:3000[12].

[11]https://www.cygwin.com/
[12]http://localhost:3000



Popular Products

61
Haught or Naught
High-minded or absent-minded? You decide.
Submitted by:

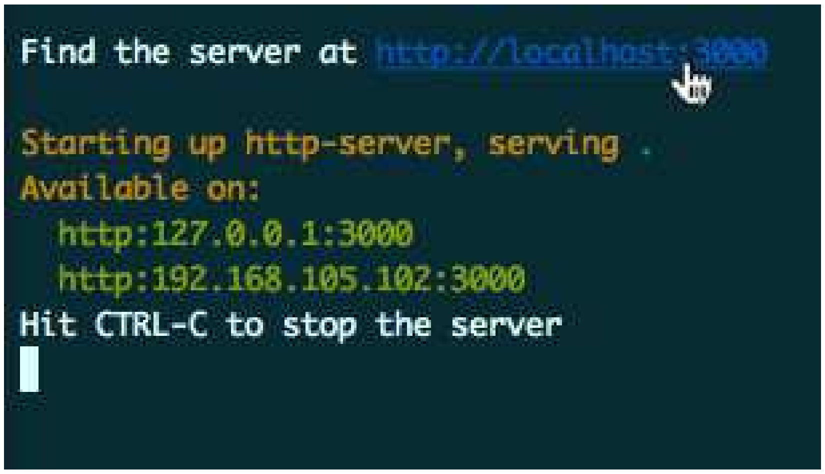
54
Supermajority: The Fantasy Congress League
Earn points when your favorite politicians pass legislature.
Submitted by:

27
Tinfoild: Tailored tinfoil hats
Don't worry, we already have your measurements and shipping address.
Submitted by:

26
Yellow Pail
On-demand sand castle construction expertise.
Submitted by:

Completed version of the app

Mac users can click on links inside of terminal. Just hold command and double-click on the link:



```
Find the server at http://localhost:3000

Starting up http-server, serving .
Available on:
  http:127.0.0.1:3000
  http:192.168.105.102:3000
Hit CTRL-C to stop the server
```

Clicking a link in the console

This demo app is a site like Product Hunt[13] or Reddit[14]. These sites have lists of links that users can vote on.

In our app we can up-vote products and all products are sorted, instantaneously, by number of votes.

The app we build in this section has a slightly different style than the completed version we just looked at. This is because the completed version has some additional HTML structure that's purely for aesthetics. Feel free to use the HTML in `app-complete.js` to style your component after completing this section.

To quit a running Node server, hit CTRL+C.

Prepare the app

In the terminal, run `ls -1p` to see the project's layout:

```
$ ls -1p
```

[13] http://producthunt.com
[14] http://reddit.com

```
README.md
app-complete.js
app.js
data.js
images/
index.html
node_modules/
package.json
style.css
vendor/
```

We'll be working with `index.html` and `app.js` for this project. `app-complete.js` is the completed application that we will be building towards.

All projects include a handy `README.md` that have instructions on how to run them.

To get started, we'll ensure `app-complete.js` is no longer loaded in `index.html`. We'll then have a blank canvas to begin work inside `app.js`.

Open up `index.html` in your favorite text editor. It should look like this:

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

  <head>
    <meta charset="utf-8">
    <!-- Disable browser cache -->
    <meta http-equiv="cache-control" content="max-age=0" />
    <meta http-equiv="cache-control" content="no-cache" />
    <meta http-equiv="expires" content="0" />
    <meta http-equiv="expires" content="Tue, 01 Jan 1980 1:00:00 GMT" />
    <meta http-equiv="pragma" content="no-cache" />
    <title>Project One</title>
    <link rel="stylesheet" href="vendor/semantic-ui/semantic.min.css" />
    <link rel="stylesheet" href="style.css" />
    <script src="vendor/babel-core-5.8.25.js"></script>
    <script src="vendor/react.js"></script>
    <script src="vendor/react-dom.js"></script>
  </head>

  <body>
```

```
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
    <script src="./data.js"></script>
    <script src="./app.js"></script>
    <!-- Delete the line below to get started. -->
    <script type="text/babel" src="./app-complete.js"></script>
  </body>

</html>
```

We'll go over all of the dependencies being loaded under the `<head>` tag later. The main HTML document is these few lines here:

```
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
```

For this project, we're using Semantic UI[15] for styling.

Semantic UI is a CSS framework, much like Twitter's Bootstrap[16]. It provides us with a grid system and some simple styling. You don't need to know Semantic UI in order to use this book. We'll provide all of the styling code that you need. At some point, you might want to check out the docs Semantic UI docs[17] to get familiar with the framework and explore how you can use it in your own projects.

The `class` attributes here are just concerned with style and are safe to ignore. Stripping those away, our core markup is quite succinct:

```
<div>
  <h1>Popular Products</h1>
  <div id="content"></div>
</div>
```

We have a title for the page (`h1`) and a `div` with an `id` of `content`. This `div` is where we will ultimately mount our React app. We'll see shortly what that means.

The next few lines tell the browser what JavaScript to load. To start building our own application, let's remove the `./app-complete.js` script tag completely. The comments in the code indicate which line to remove:

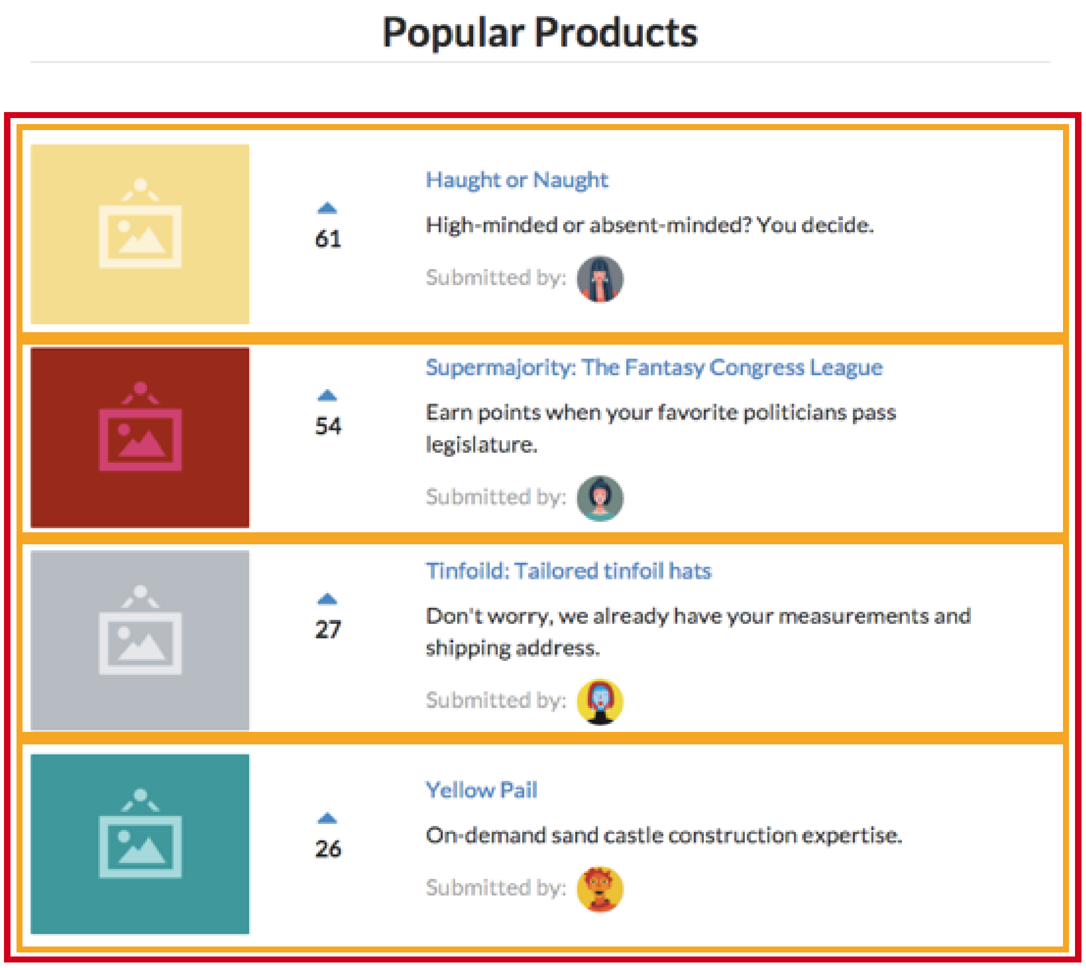
[15] http://semantic-ui.com/
[16] http://getbootstrap.com/
[17] http://semantic-ui.com/introduction/getting-started.html

```
<script src="./data.js"></script>
<script src="./app.js"></script>
<!-- Delete the line below to get started. -->
~~<script type="text/babel" src="./app-complete.js"></script>~~
```

After we save our updated `index.html` and reload the web browser, we'll see that our app has disappeared.

Our first component

Building a React app is all about **components**. An individual React component can be thought of as a *UI* component in an app. Let's take a look at the components in our app:


Popular Products
61
Haught or Naught
High-minded or absent-minded? You decide.
Submitted by:
54
Supermajority: The Fantasy Congress League
Earn points when your favorite politicians pass legislature.
Submitted by:
27
Tinfoild: Tailored tinfoil hats
Don't worry, we already have your measurements and shipping address.
Submitted by:
26
Yellow Pail
On-demand sand castle construction expertise.
Submitted by:

The app's components

We have a hierarchy of one parent component and many sub-components:

1. `ProductList` (**red**): Contains a list of product components
2. `Product` (**orange**): Displays a given product

Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable. Furthermore, as we'll see in this chapter and throughout this course, React's paradigm for component data flow and interactivity is rigidly defined. We'll see how this well-defined system is beneficial.

The classic UI paradigm is to manipulate the DOM with JavaScript directly. As complexity of an app grows, this can lead to all sorts of inconsistencies and headaches around managing state and transitions. In React, when the inputs for a component change, the framework simply re-renders that component. This gives us a robust UI consistency guarantee:

With a given set of inputs, the output (how the component looks on the page) will always be the same.

Let's start off by building the `ProductList` component. We'll write all of our React code for this section inside the file `app.js`. Let's open the `app.js` file and insert the component:

```
const ProductList = React.createClass({
  render: function () {
    return (
      <div className='ui items'>
        Hello, friend! I am a basic React component.
      </div>
    );
  },
});
```

ES6: Prefer `const` and `let` over `var`

Both the `const` and `let` statements declare variables. They are introduced in ES6.

`const` is a superior declaration in cases where a variable is never re-assigned. Nowhere else in the code will we re-assign the `ProductList` variable. Using `const` makes this clear to the reader. It refers to the "constant" state of the variable in the context it is defined within.

If the variable will be re-assigned, use `let`.

If you've worked with JavaScript before, you're likely used to seeing variables declared with `var`:

```
var ProductList = ...
```

We encourage the use of `const` and `let` instead of `var`. In addition to the restriction introduced by `const`, both `const` and `let` are *block scoped* as opposed to *function scoped*. Typically this separation of scope helps avoid unexpected bugs.

`React.createClass()`

To create a component, we use the function `React.createClass()`. This is how all components are defined in React. We pass in a single argument to this function: a JavaScript object.

This *class definition object* (the argument we pass to the `React.createClass()` method) in our case has just one key, `render`, which defines a rendering function. `render()` is the only required method for a React component. React uses the return value from this method to determine what exactly to render to the page.

The `createClass()` method is one way to create a React component. The other main way of defining one is by using the ES6 classical implementation:

```
class ProductList extends React.Component {}
```

We'll primarily be using the `createClass()` method throughout this course.

If you have some familiarity with JavaScript, the return value may be surprising:

```
return (
  <div className='ui items'>
    Hello, friend! I am a basic React component.
  </div>
);
```

The syntax of the return value doesn't look like traditional JavaScript. We're using **JSX** (JavaScript eXtension syntax), a syntax extension for JavaScript written by Facebook. Using JSX enables us to write the markup for our component views in a familiar, HTML-like syntax. In the end, this JSX code compiles to vanilla JavaScript. Although using JSX is not a necessity, we'll use it in this course as it pairs really well with React.

If you don't have much familiarity with JavaScript, we recommend you follow along and use JSX in your React code too. You'll learn the boundaries between JSX and JavaScript with experience.

JSX

React components ultimately render HTML which is displayed in the browser. As such, the `render()` method of a component needs to describe how the view should be represented as HTML. React builds our apps with a fake representation of the Document Object Model (DOM, for short, refers to the browser's HTML tree that makes up a web page). React calls this the *virtual DOM*. Without getting deep into details for now, React allows us to describe a component's HTML representation in JavaScript.

JSX was created to make this JavaScript representation of HTML more HTML-like. To understand the difference between HTML and JSX, consider this JavaScript syntax:

```
React.createElement('div', {className: 'ui items'},
  'Hello, friend! I am a basic React component.'
)
```

Which can be represented in JSX as:

```
<div className='product'>
  Hello, friend! I am a basic React component.
</div>
```

The code readability is slightly improved in the latter example. This is exacerbated in a nested tree structure:

```
React.createElement('div', {className: 'ui items'},
  React.createElement('p', null, 'Hello, friend! I am a basic React component.')
)
```

In JSX:

```
<div className='ui items'>
  <p>
    Hello, friend! I am a basic React component.
  </p>
</div>
```

JSX presents a light abstraction over the JavaScript version, yet the legibility benefits are huge. Readability boosts our app's longevity and makes it easier to onboard new developers.

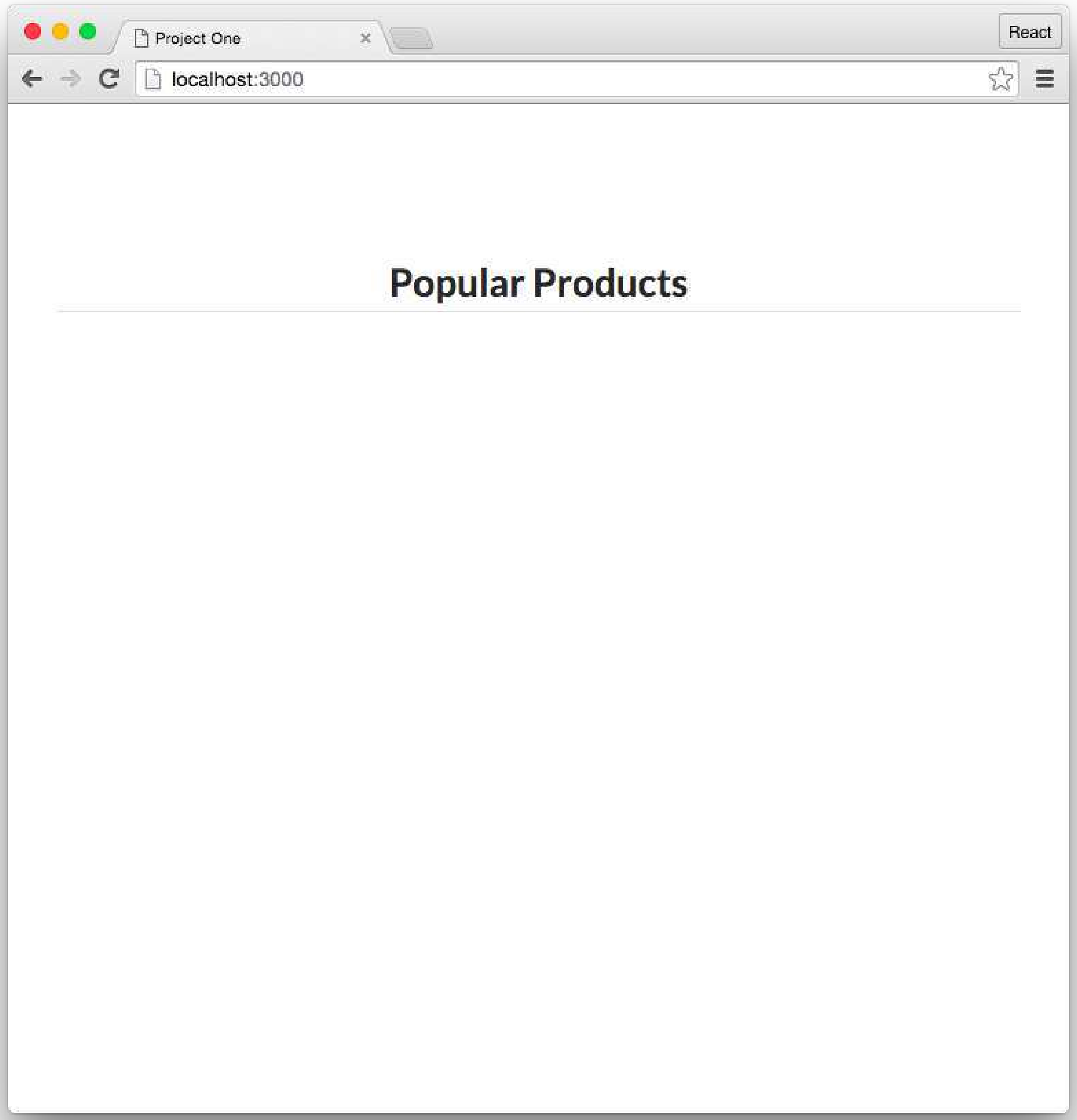
Even though the JSX above looks exactly like HTML, it's important to remember that JSX is actually just compiled into JavaScript (ex: `React.createElement('div')`).

During runtime React takes care of rendering the actual HTML in the browser for each component.

The developer console

Our first component is written and we now know that it uses a special flavor of JavaScript called JSX for improved readability.

After editing and saving our `app.js`, let's refresh the page in our web browser and see what changed:



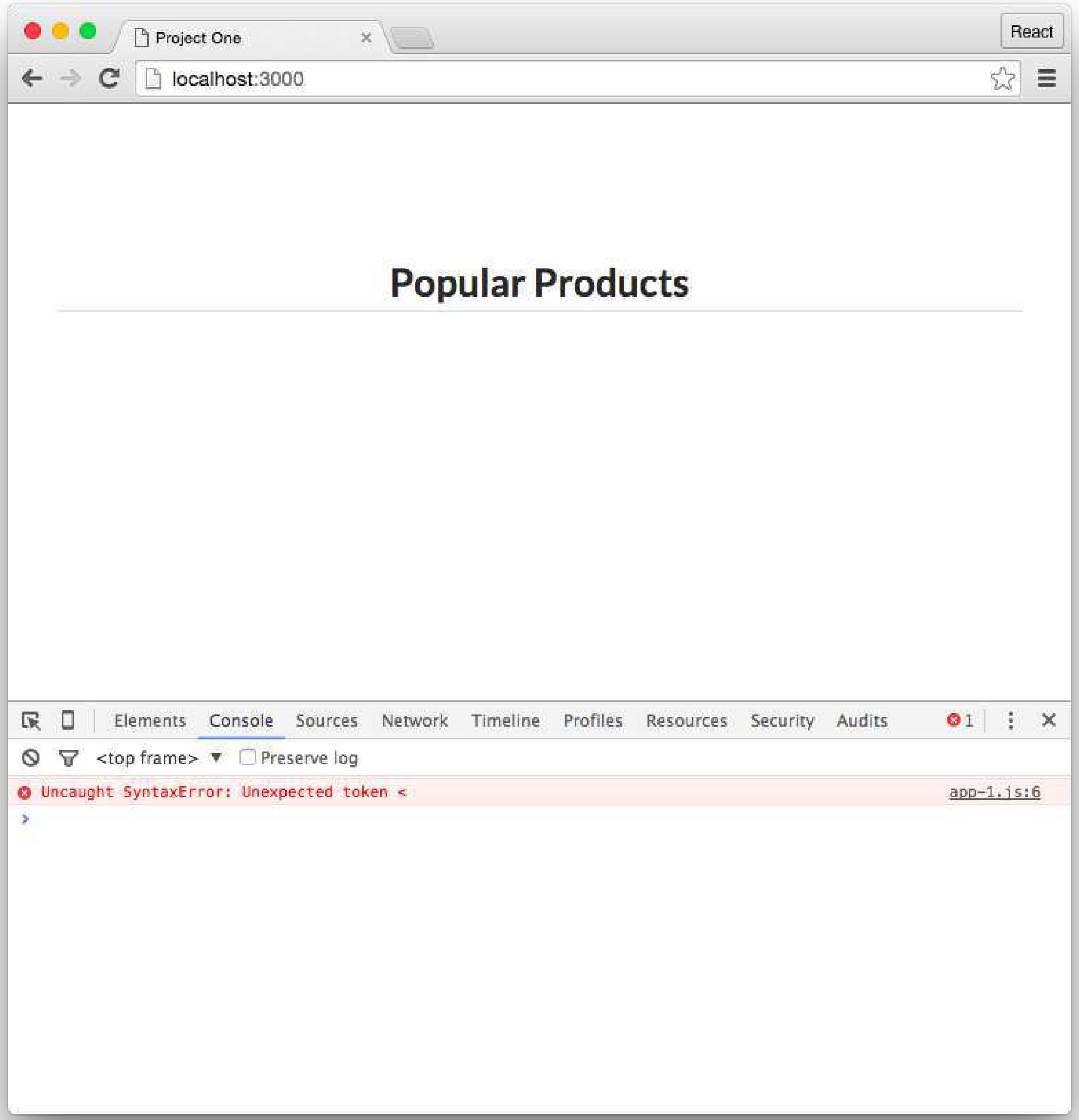
Nothing?

Every major browser comes with a toolkit that helps developers working on JavaScript code. A central part of this toolkit is a console. Think of the console as JavaScript's primary communication medium back to the developer. If JavaScript encounters any errors in its execution, it will alert you in this developer console.

> To open the console in Chrome, navigate to View > Developer > JavaScript Console.
>
> Or, just press `Command` + `Option` + `J` on a Mac or `Control` + `Shift` + `L` on Windows/Linux.

Opening the console, we are given a cryptic clue:

```
Uncaught SyntaxError: Unexpected token <
```



```
Uncaught SyntaxError: Unexpected token <                app-1.js:6
```

Error in the console

This `SyntaxError` prevented our code from running. A `SyntaxError` is thrown when the JavaScript engine encounters tokens or token order that doesn't conform to the syntax of the language when parsing code. This error type indicates some code is out of place or mistyped.

The issue? Our browser's JavaScript parser exploded when it encountered the JSX. The parser doesn't know anything about JSX. As far as it is concerned, this `<` is completely out of place.

We know that JSX is an extension to standard JavaScript. Let's empower our browser's plain old

JavaScript interpreter to use this extension.

Babel

Babel is a JavaScript transpiler. For those familiar with ES6 JavaScript, Babel turns ES6/ES7 code into ES5 code so that our browser can use lots of these features with browsers that only understand ES5.

For our purposes, a handy feature of Babel is that it understands JSX. Babel compiles our JSX into vanilla JS that our browser can then interpret and execute. Let's tell the browser's JS engine we want to use babel to compile and run our JavaScript code.

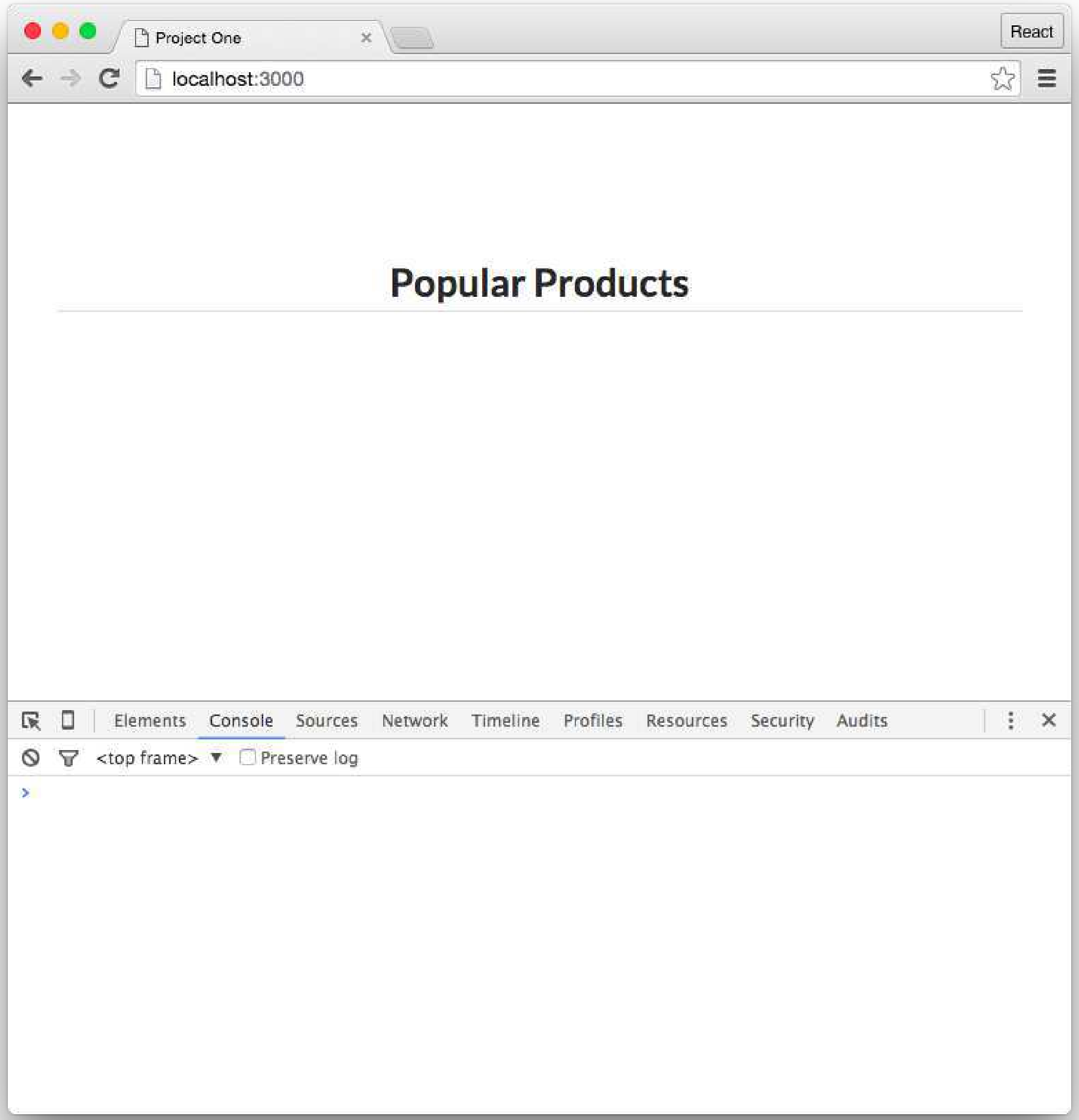
The sample code's `index.html` already imports Babel in the `head` tags of `index.html`:

```
<head>
  <!-- ... -->
  <script src="vendor/babel-core-5.8.25.js"></script>
  <!-- ... -->
</head>
```

All we need to do is tell our JavaScript runtime that our code should be compiled by Babel. We can do this by setting the `type` attribute when we import the script in `index.html` from `text/javascript` to `text/babel`. Open `index.html` and change this line:

```
<script src="./data.js"></script>
<script type="text/babel" src="./app.js"></script>
```

Save `index.html` and reload the page.



Still nothing. However, the console reveals we no longer have any JavaScript compilation errors. Babel successfully compiled our JSX into JS and our browser was able to run that JS without any issues.

If your console encounters an error, check that the file has been saved before reloading the browser. If an error still persists, check the code previously added to make sure there aren't any syntactical mistakes.

So what's happening? We've defined the component, but we haven't yet told React to do anything with it yet. We need to tell the React framework that we want to run our app.

`ReactDOM.render()`

We're going to instruct React to render this `ProductList` inside a specific DOM node.

Add the following code below the component inside `app.js`:

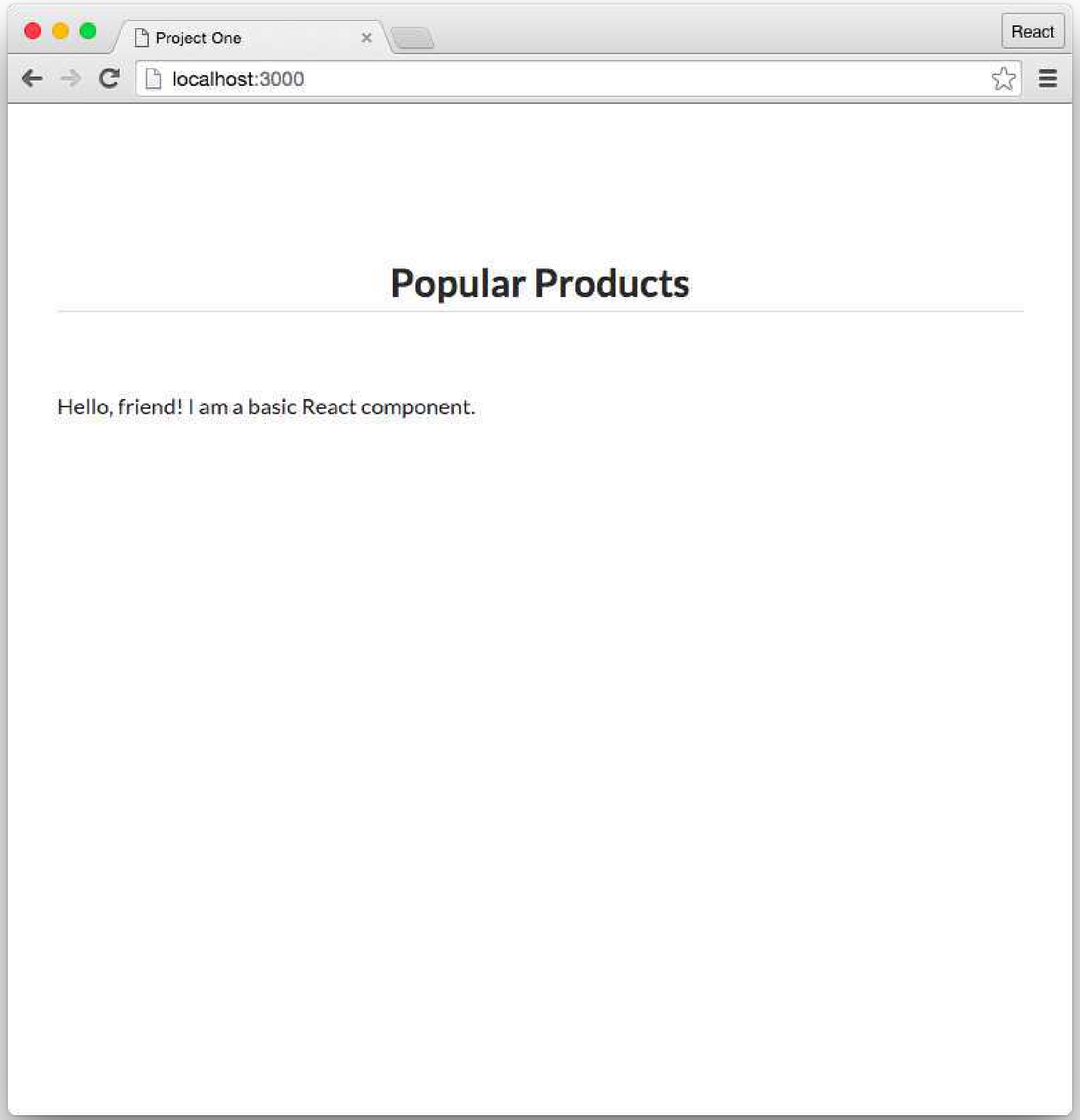
```
ReactDOM.render(
  <ProductList />,
  document.getElementById('content')
);
```

We pass in two arguments to the `ReactDOM.render()` method. The first argument is *what* we'd like to render. Here, we're passing in a reference to our React component `ProductList` in JSX. The second argument is *where* to render it. We'll send a reference to the browser's DOM element.

```
ReactDOM.render([what], [where]);
```

In our code, we have a difference in casing between the different types of React element declarations. We have HTML DOM elements like `<div>` and a React component called `<ProductList />`. In React, native HTML elements *always* start with a lowercase letter whereas React component names *always* start with an uppercase letter.

With `ReactDom.render()` now at the end of `app.js`, save the file and refresh the page in the browser:



Popular Products

Hello, friend! I am a basic React component.

We successfully implemented a React component in JSX, ensured it was being compiled to JS, and rendered it in the DOM in the web browser.

Our second component

Currently, our only component is `ProductList`. We'll want `ProductList` to render a list of products (its "sub-components" or "child components"). In HTML, we could render this entirely in the JSX that `ProductList` returns. Although this works, we don't get any benefit from encoding our entire app in a single component. Just like we can embed HTML elements in the JSX of our components, we can embed other React components.

Let's build a child component, `Product`, that will contain a product listing. Just like with the `ProductList` component, we'll use the `React.createClass()` function with a single key of `render`:

```
const Product = React.createClass({
  render: function () {
      return (<div></div>)
  }
});
```

For every product, we'll add an image, a title, a description, and an avatar of the post author. The relevant code might look something like:

```
const Product = React.createClass({
  render: function () {
    return (
      <div className='item'>
        <div className='image'>
          <img src='images/products/image-aqua.png' />
        </div>
        <div className='middle aligned content'>
          <div className='description'>
            <a>Fort Knight</a>
            <p>Authentic renaissance actors, delivered in just two weeks.</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img
              className='ui avatar image'
              src='images/avatars/daniel.jpg'
            />
          </div>
        </div>
      </div>
    );
  },
});
```

We've used a bit of SemanticUI styling in our code here. As we discussed previously, this JSX code will be compiled to JavaScript in the browser. As it runs in the browser as JavaScript, we cannot use any reserved JavaScript words in JSX. Setting the `class` attribute on an HTML element is how we add a Cascading StyleSheet (CSS, for short) to apply styles to the class. In JSX, however, we cannot use `class`, so React changes the key from `class` to `className`.

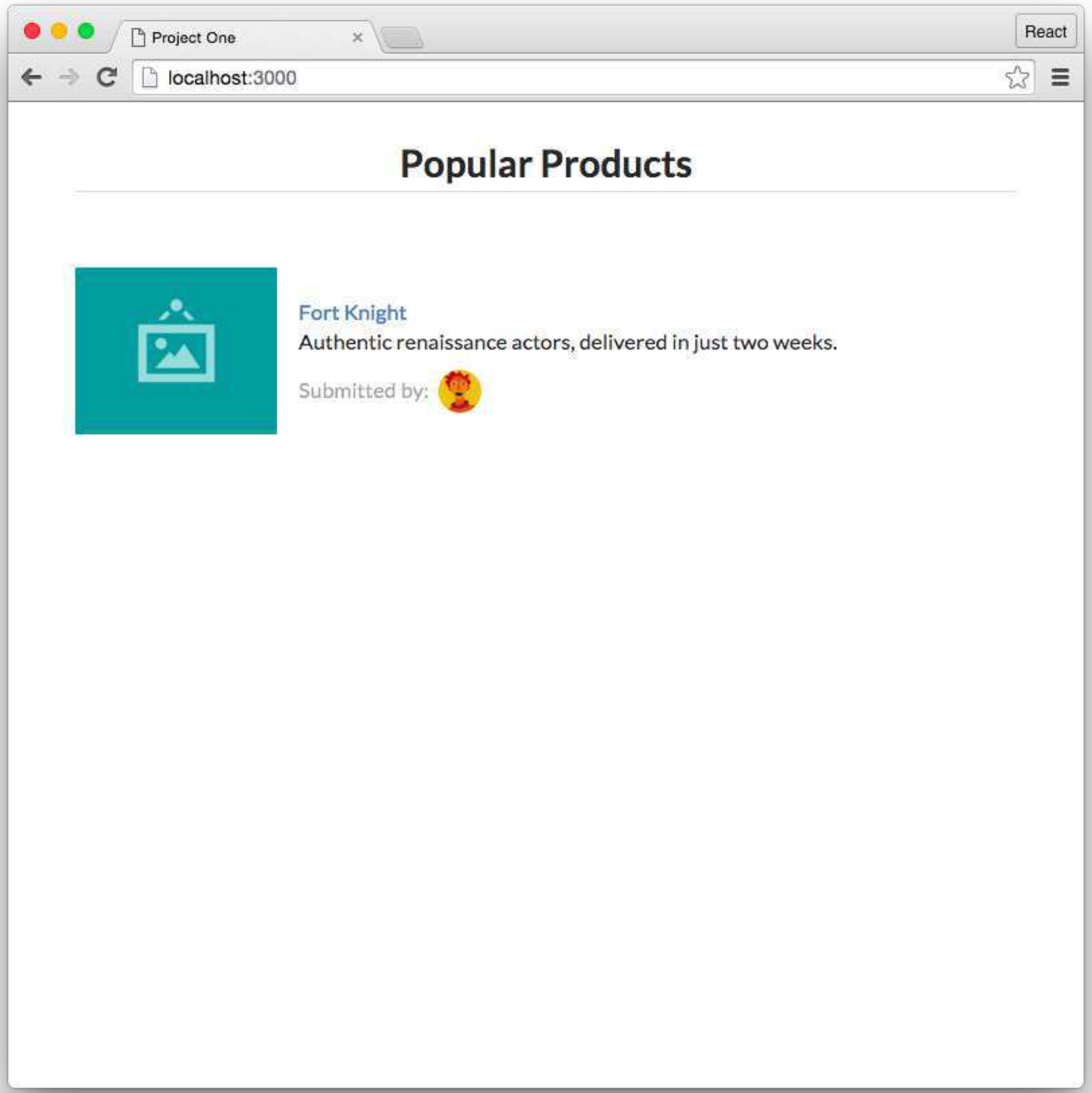
Structurally, the `Product` component is similar to the `ProductList` component. Both have a single `render()` method which returns information about an eventual HTML structure to display.

> Remember, the JSX components return is *not* actually the HTML that gets rendered, but is the *representation* that we want React to render in the DOM. This course looks at this in-depth in later sections.

To use the `Product` component, we can modify our parent `ProductList` component to list the `Product` component:

```
const ProductList = React.createClass({
  render: function () {
    return (
      <div className='ui items'>
        <Product />
      </div>
    );
  },
});
```

Save `app.js` and refresh the web browser.



Popular Products

Fort Knight
Authentic renaissance actors, delivered in just two weeks.
Submitted by:

With this update, we now have two React components being rendered in our webapp. The `ProductList` parent component is rendering the `Product` component as a child nested underneath its root `div` element.

At the moment, the child `Product` component is static. We hardcoded an image, the name, the description, and author details. To use this component in a meaningful way, we'll change it to be data-driven and therefore dynamic.

Making `Product` data-driven

Attributes in `Product` like title and description are currently hard-coded. We will need to tweak our `Product` component to make it data-driven. Having the component be driven by data will allow us to dynamically render the component based upon the data that we give it. Let's familiarize ourselves with the product data model.

The data model

In the sample code, we've included a file called `data.js` which contains some example data for our products. In the future, we might fetch this data over a network request (we cover this in later sections). The `data.js` file contains a JavaScript object called `Data` that contains an array of JavaScript objects, each representing a product object:

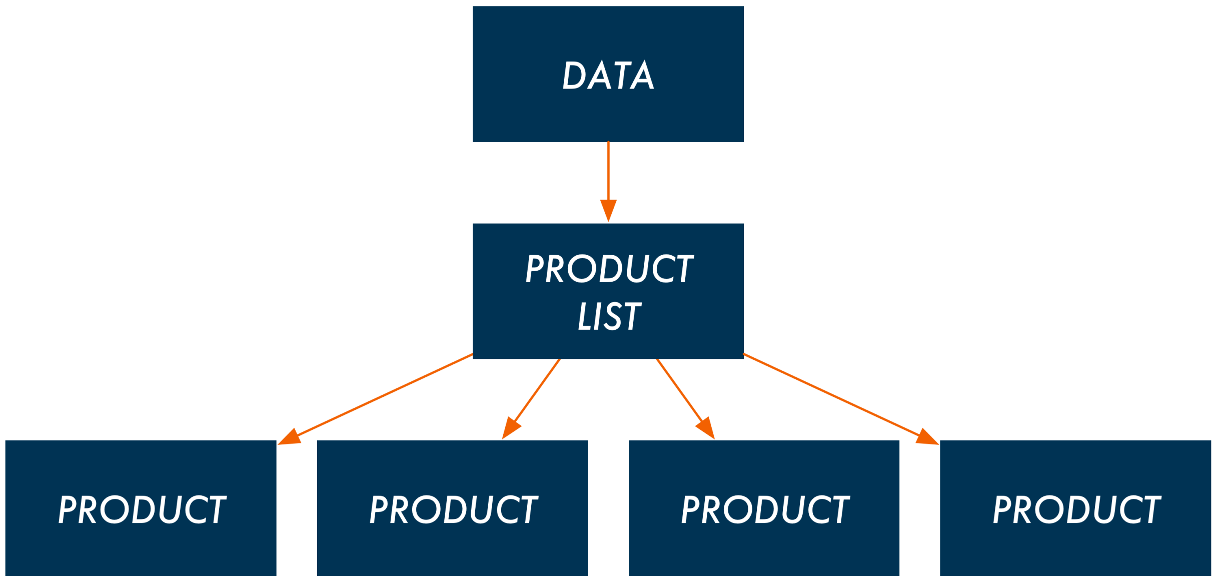
```
      id: 1,
      title: 'Yellow Pail',
      description: 'On-demand sand castle construction expertise.',
      url: '#',
      votes: generateVoteCount(),
      submitter_avatar_url: 'images/avatars/daniel.jpg',
      product_image_url: 'images/products/image-aqua.png',
    },
    {
```

Each product has a unique `id` and a handful of properties including a `title` and `description`. `votes` are randomly generated for each one with the included function `generateVoteCount()`.

We can use the same attribute keys in our React code.

Using props

We want to modify our `Product` component so that it no longer uses static, hard-coded attributes, but instead is able to accept data passed down from its parent, `ProductList`. Setting up our component structure in this way enables our `ProductList` component to dynamically render any number of `Product` components that each have their own unique attributes. Data flow will look like this:



The way data flows from parent to child in React is through **props**. When a parent renders a child, it can send along props the child can depend upon.

Let's see this in action. First, let's modify `ProductList` to pass down props to `Product`. Using our `Data` object instead of typing the data in, let's pluck the first object off of the `Data` array and use that for the single product:

```
const ProductList = React.createClass({
  render: function () {
    const product = Data[0];
    return (
      <div className='ui items'>
        <Product
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitter_avatar_url={product.submitter_avatar_url}
          product_image_url={product.product_image_url}
        />
      </div>
    );
  },
});
```

Here, the `product` variable is a JavaScript object that describes the first of our products. We pass the product's attributes along individually to the `Product` component using the syntax `[prop_-name]=[prop_value]`. The syntax of assigning attributes in JSX is exactly the same as HTML and XML.

There are two interesting things here. The first is the braces (`{}`) around each of the property values:

```
id={product.id}
```

In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is an expression. To pass in a string instead of an expression, for instance, we can do so like this:

```
id='1'
```

Using the `'` as a delimiter for the string instead of the `{}`.

> JSX attribute values **must** be delimited by either braces or quotes.
>
> If type is important and we want to pass in something like a Number or a `null`, use braces.

Now the `ProductList` component is passing props down to `Product`. Our `Product` component isn't using them yet, so let's modify the component to use these props:

```
const Product = React.createClass({
  render: function () {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.product_image_url} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a>
              <i className='large caret up icon'></i>
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a href={this.props.url}>
              {this.props.title}
            </a>
          </div>
          <div className='extra'>
```

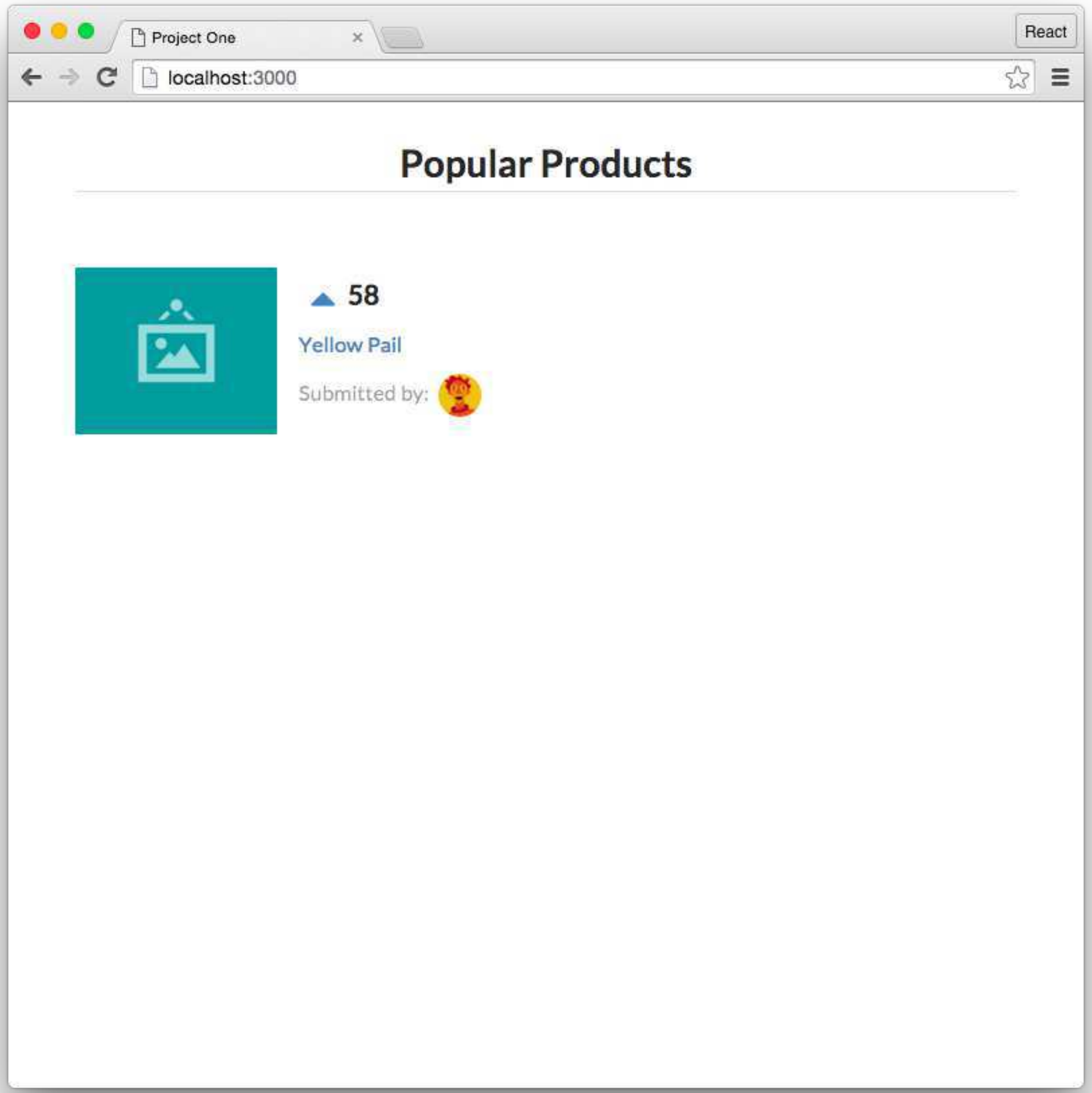
```
            <span>Submitted by:</span>
            <img
              className='ui avatar image'
              src={this.props.submitter_avatar_url}
            />
          </div>
        </div>
      </div>
    );
  },
});
```

In React, we can access all component props through the `this.props` in a component object. We can access all of the various props we passed along with the names assigned by `ProductList`. Again, we're using braces as a delimiter.

> `this` is a special keyword in JavaScript. The details about `this` are a bit nuanced, but for the purposes of the majority of this book, `this` will be bound to the React component class and we'll discuss when it differs in later sections. We use `this` to call methods on the component.
>
> For more details on `this`, check out this page on MDN[18].

With our updated `app.js` file saved, let's refresh the web browser again:

[18]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this



The `ProductList` component now shows a single product listed, the first object pulled from `Data`. Now that `Product` is rendering itself based on the props that it receives from `ProductList`, our code is poised to render any number of unique products.

Rendering multiple products

Modify the `ProductList` component again. This time, we're going to make an array of `Product` components, each representing an individual object in the `Data` array. Rather than passing in a single product, let's map over all of our products:

```
const ProductList = React.createClass({
  render: function () {
    const products = Data.map((product) => {
      return (
        <Product
          key={'product-' + product.id}
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitter_avatar_url={product.submitter_avatar_url}
          product_image_url={product.product_image_url}
        />
      );
    });
    return (
      <div className='ui items'>
        {products}
      </div>
    );
  },
});
```

Before `render`'s return method, we create a variable called `products`. We use JavaScript's `map` function to iterate over each one of the objects in the `Data` array.

The function passed to `map` simply returns a `Product` component. Notably, we're able to represent the `Product` component instance in JSX inside of `return` without issue because it compiles to JavaScript. This `Product` is created exactly as before with the same props. As such, the `products` variable ends up just being an array of `Product` components, which in this case is four total.

Array's `map` method takes a function as an argument. It calls this function with each item inside of the array (in this case, each object inside `Data`) and builds a **new** array by using the return value from each function call.

Because the `Data` array has four items, `map` will call this function four times, once for each item. When `map` calls this function, it passes in as the first argument an item. The return value from this function call is inserted into the new array that `map` is constructing. After handling the last item, `map` returns this new array. Here, we're storing this new array in the variable `products`.

Note the use of the `key={'product-' + product.id}` prop. React uses this special property to create unique bindings for each instance of the `Product` component. The `key` prop is not used by our `Product` component, but by the React framework. It's a special property that we discuss deeper in our advanced components section. For the time being, it's enough to note that this property needs to be unique per React instance in a map.

ES6: Arrow functions

Up until this point, we've been using the traditional JavaScript function declaration syntax:

```
render: function () { ... }
```

We've been using this syntax to declare **object methods** for our React component classes.

Inside of the `render()` method of `ProductList`, we pass an **anonymous arrow function** to `map()`. Arrow functions were introduced in ES6. Throughout the book, whenever we're declaring anonymous functions we will use arrow functions. This is for two reasons.

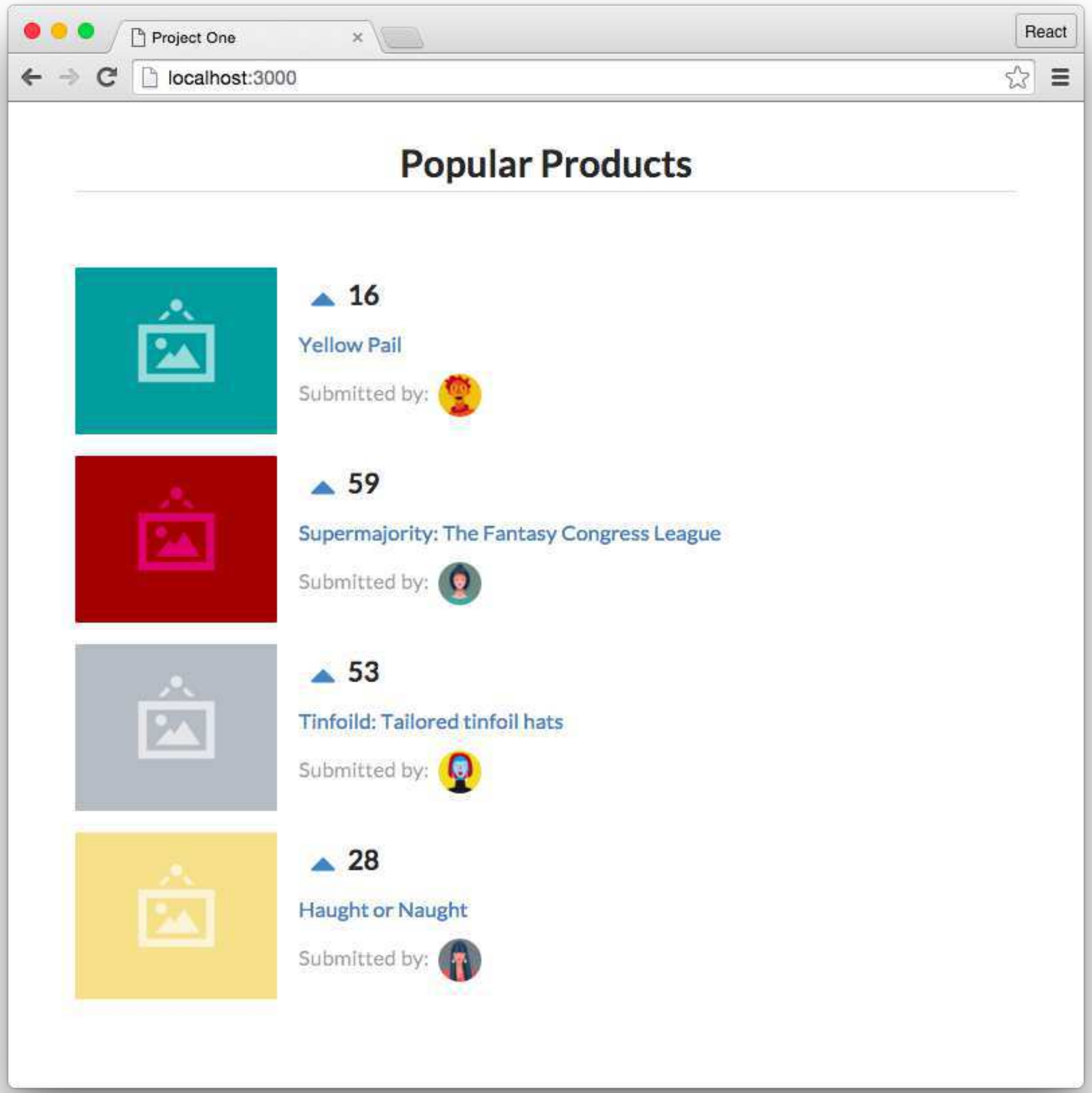
The first is that the syntax is much terser. Compare this declaration:

```
const timestamps = messages.map(function(m) { return m.timestamp });
```

To this one:

```
const timestamps = messages.map(m => m.timestamp);
```

Of greater benefit, though, is how arrow functions bind the `this` object. We cover the semantics of this difference later in the sidebar titled "Arrow functions and `this`."



Popular Products

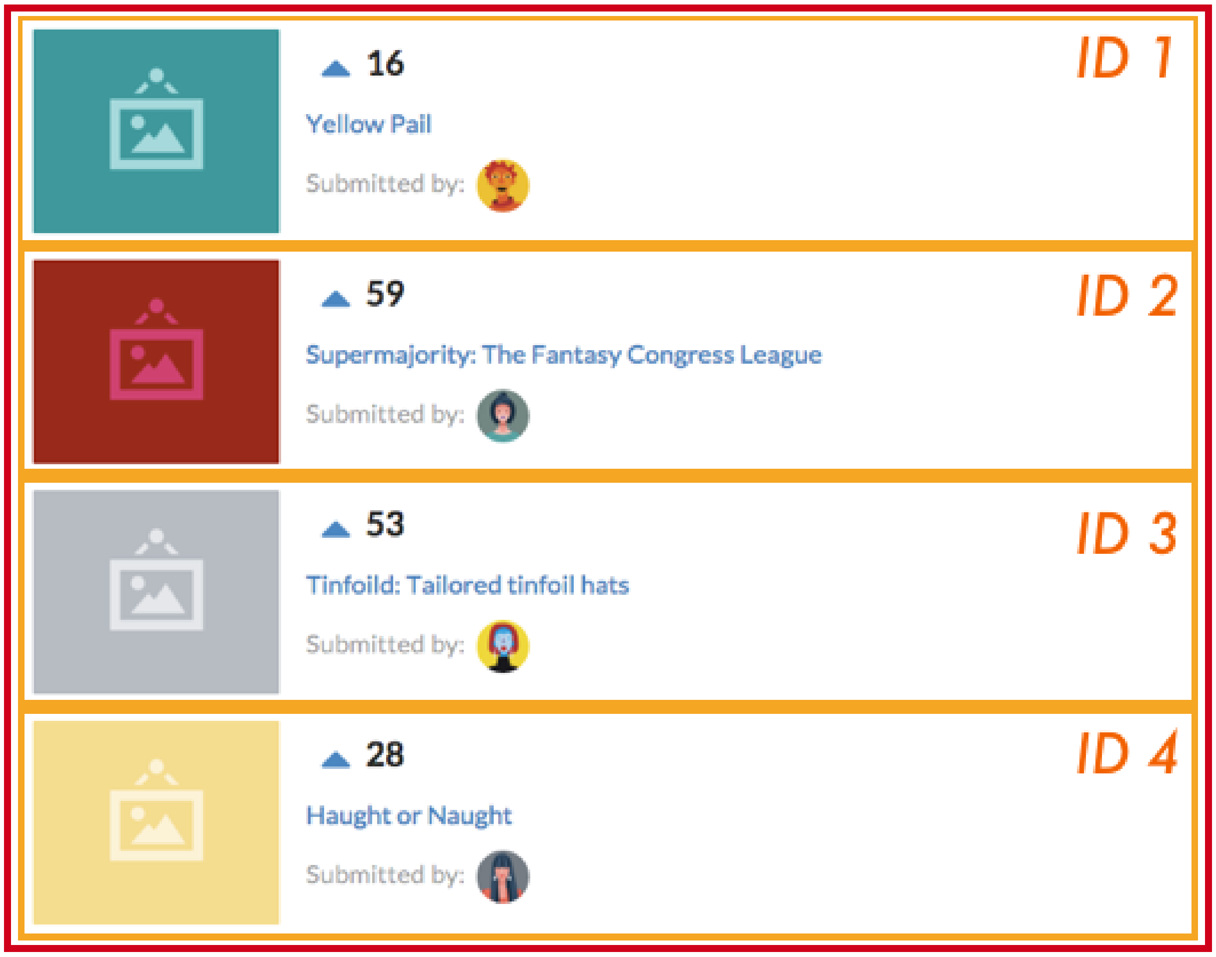
16
Yellow Pail
Submitted by:

59
Supermajority: The Fantasy Congress League
Submitted by:

53
Tinfoild: Tailored tinfoil hats
Submitted by:

28
Haught or Naught
Submitted by:

After we save the updates to our `app.js` and refresh the page, we'll see that we have five total React components at work. We have a single parent component, our `ProductList` component which contains four child `Product` components, one for each product object in our `Data` variable (from `data.js`).



`Product` components (orange) inside of the `ProductList` component (red)

We added an 'up vote' caret icon in our `Product` component above. If we click on one of these buttons, we'll see that nothing happens. We've yet to hook up an event to the button.

Although we have a dynamic React app running in our web browser, this page still lacks interactivity. While React has given us an easy and clean way to organize our HTML thus far and enabled us to drive HTML generation based on a flexible, dynamic JavaScript object, we've still yet to tap into its true power in enabling super dynamic, interactive interfaces.

The rest of this course digs deep into this power. Let's start with something simple: the ability to up-vote a given product.

React the vote (your app's first interaction)

When the up-vote button on each one of the `Product` components is clicked, we expect it to update the `votes` attribute for that `Product`, increasing it by one. In addition, this vote should be updated in the `Data` variable. While `Data` is just a JavaScript variable right now, in the future it could just as easily be a remote database. We would need to inform the remote database of the change. While we cover this process in-depth in later sections of this course, we'll get used to this practice by updating `Data`.

The `Product` component can't modify its `votes`. **`this.props` is immutable**.

Remember, while the child has access to *read* its own `props`, it doesn't own them. In our app, the parent component, `ProductList`, owns them. React favors the idea of *one-way data flow*. This means that data changes come from the "top" of the app and are propagated "downwards" through its various components.

A child component does not own its `props`. Parent components own the `props` of the child component.

We need a way for the `Product` component to let `ProductList` know that a click on its up-vote event happened and then let `ProductList`, the owner of both that product's props as well as the store (`Data`), handle the rest. It can update `Data` and then data will flow downward from `Data`, through the `ProductList` component, and finally to the `Product` component.

Propagating the event

Fortunately, propagating an event from a child component to a parent is easy. We can pass down *functions* as props too. We can have the `ProductList` component give each `Product` component a function to call when the up-vote button is clicked. Functions passed down through props are the canonical manner in which children communicate events with their parent components.

Let's start by modifying `Product` to call a function when the up-vote button is clicked.

First, add a new component function to `Product`, `handleUpVote()`. We'll anticipate the existence of a new prop called `onVote()` that's a function passed down by `ProductList`:

```
const Product = React.createClass({
  handleUpVote: function () {
    this.props.onVote(this.props.id);
  },
  render: function () {
  // ...
```

We're setting up an expectation that the `ProductList` component sends a function as a prop to the `Product` component. We'll call this function `onVote()`. `onVote()` accepts a single argument. The argument we're passing in is the `id` of the product (`this.props.id`). The `id` is sufficient information for the parent to deduce which `Product` component has produced this event. We'll implement the function shortly.

We can have an HTML element inside the `Product` component call a function when it is clicked. We'll set the `onClick` attribute on the `a` HTML tag that is the up-vote button. By passing the name of the function `handleUpVote()` to this attribute, it will call our new `handleUpVote()` function when the up-vote button is clicked:

```
const Product = React.createClass({
  handleUpVote: function () {
    this.props.onVote(this.props.id);
  },
  render: function () {
    return (
      // ...
          <div className='header'>
            <a onClick={this.handleUpVote}>
              <i className='large caret up icon'></i>
            </a>
            {this.props.votes}
          </div>
          // ...
```

Let's define the function in `ProductList` that we pass down to `Product` as the prop `onVote()`. This is the function that the `Product` component's `handleUpVote()` calls whenever the up-vote button is clicked:

```
const ProductList = React.createClass({
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
    const products = Data.map((product) => {
      return (
        <Product
          key={'product-' + product.id}
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitter_avatar_url={product.submitter_avatar_url}
          product_image_url={product.product_image_url}
          onVote={this.handleProductUpVote}
        />
      );
    });
    return (
      <div className='ui items'>
        {products}
      </div>
    );
  },
});
```

First we define a `handleProductUpVote()` function on `ProductList`. We've set it up to accept a `productId`, as anticipated.

This function is then passed down as a prop, `onVote`, just the same as any other prop.

ES6: Arrow functions and `this`

Inside `ProductList`, we use array's `map()` method on `Data` to setup the variable `products`. We pass an anonymous arrow function to `map()`. Inside this arrow function, we call `this.handleProductUpVote`. Here, `this` is bound to the React object.

We introduced arrow functions earlier and mentioned that one of their benefits was how they bind the `this` object.

The traditional JavaScript function declaration syntax (`function () {}`) will bind `this` in anonymous functions to the global object. To illustrate the confusion this causes, consider the following example:

```
function printSong() {
  console.log("Oops - The Global Object");
}

const jukebox = {
  songs: [
    {
      title: "Wanna Be Startin' Somethin'",
      artist: "Michael Jackson",
    },
    {
      title: "Superstar",
      artist: "Madonna",
    },
  ],
  printSong: function (song) {
    console.log(song.title + " - " + song.artist);
  },
  printSongs: function () {
    // `this` bound to the object (OK)
    this.songs.forEach(function (song) {
      // `this` bound to global object (bad)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Oops - The Global Context"
// > "Oops - The Global Context"
```

The method `printSongs()` iterates over `this.songs` with `forEach()`. In this context, `this` is bound to the object (`jukebox`) as expected. However, the anonymous function passed to `forEach()` binds its internal `this` to the global object. As such, `this.printSong(song)` calls the function declared at the top of the example, *not* the method on `jukebox`.

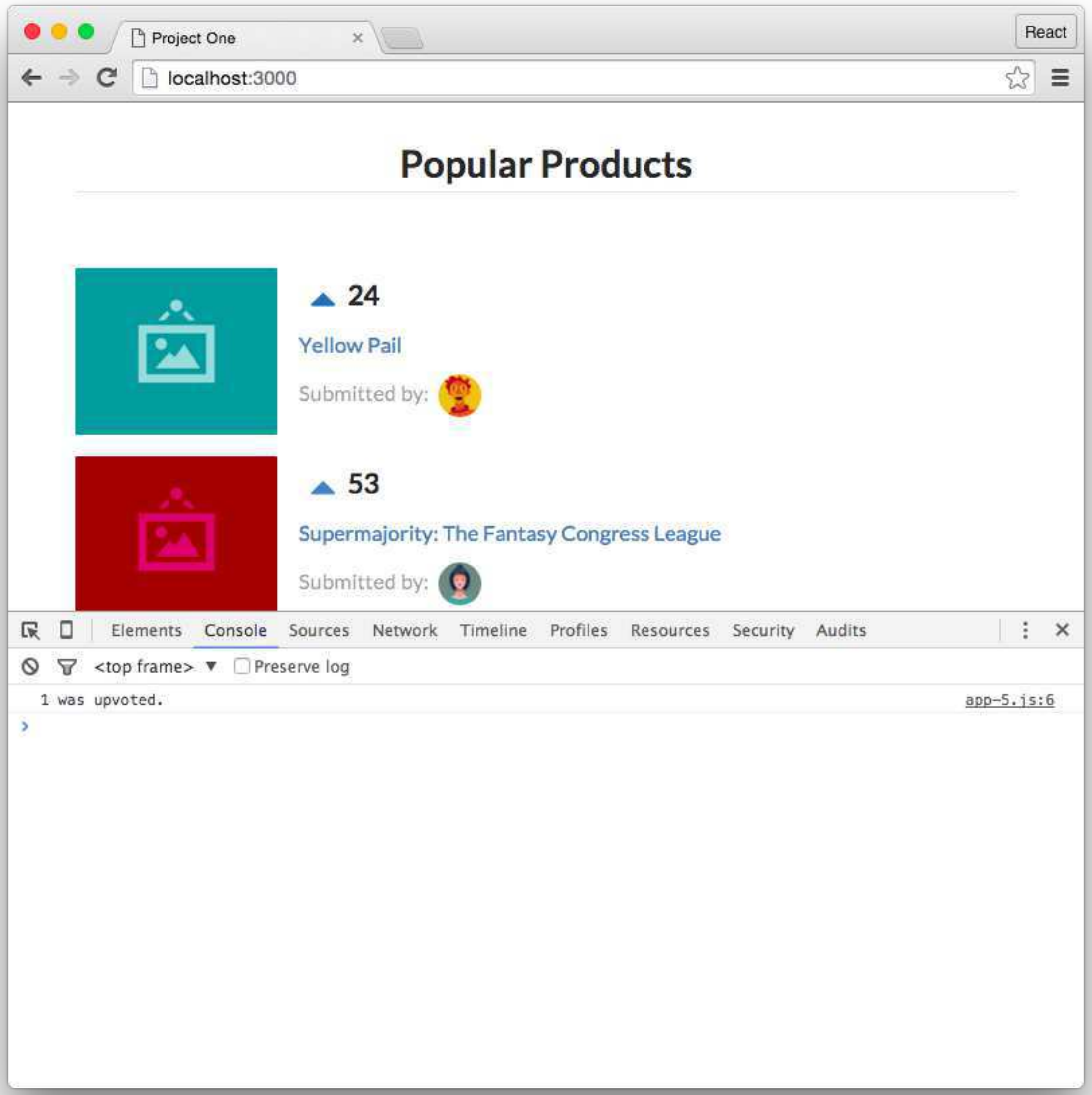
JavaScript developers have traditionally used workarounds for this behavior, but arrow functions solve the problem by **capturing the `this` value of the enclosing context**. Using an arrow function for `printSongs()` has the expected result:

```
// ...
  printSongs: function () {
    this.songs.forEach((song) => {
      // `this` bound to same `this` as `printSongs()` (`jukebox`)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Wanna Be Startin' Somethin' - Michael Jackson"
// > "Superstar - Madonna"
```

For this reason, throughout the book we will use arrow functions for all anonymous functions.

Saving our updated `app.js`, refreshing our web browser, and clicking an up-vote will log some text to our JavaScript console:



```
1 was upvoted.                                                    app-5.js:6
```

The events are being propagated up to the parent. Finally, we need `ProductList` to update the store, `Data`.

It's tempting to modify the `Data` JavaScript object directly. We could hunt through `Data` until we find the corresponding product and then update its vote count. However, React will have no idea this change occurred. So while the vote count will be updated in the store, this update will not be reflected to the user. This would be equivalent to just making a call to a server to update the vote count but then doing nothing else. The front-end would be none the wiser.

In order to move forward, there's one more critical concept for React components we need to cover: state.

Using state

Whereas props are immutable and owned by a component's parent, state is mutable and owned by the component. `this.state` is private to the component and can be updated with `this.setState()`. As with props, when the state updates the component will re-render itself.

Every React component is rendered as a function of its `this.props` and `this.state`. This rendering is deterministic. This means that given a set of props and a set of state, a React component will always render a single way. As we mentioned earlier, this approach makes for a powerful UI consistency guarantee.

As we are mutating the data for our products (the number of votes), we should consider this data to be stateful. We should treat `Data` as some external store, which is used to update `this.state` for `ProductList`.

Let's modify the `ProductList` component's `render` function so that it uses state as opposed to accessing `Data` directly. In order to tell React that our component is *stateful*, we'll define the function `getInitialState()` and return a non-falsey value:

```
const ProductList = React.createClass({
  getInitialState: function () {
    return {
      products: [],
    };
  },
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
    const products = this.state.products.map((product) => {
      return (
        <Product
        // ...
```

Like `render()`, `getInitialState()` is a special method on a React component. It is one of several lifecycle methods available. It is executed exactly once during the component lifecycle and defines the initial state of the component.

Now, instead of mapping over `Data` to produce the variable `products`, we are reading from `this.state`. In `getInitialState()`, `this.state` is initialized to the JavaScript object:

```
{
  products: [],
}
```

But is never actually updated to anything meaningful. Indeed, if we were to save and refresh now, all of our products would be missing again. We need to update the state using `Data`.

> We cover all of the component lifecycle methods in-depth in a later section.

Setting state with `this.setState()`

Let's use another lifecycle method, `componentDidMount()`, to set the state for `ProductList` to `Data`:

```
const ProductList = React.createClass({
  getInitialState: function () {
    return {
      products: [],
    };
  },
  componentDidMount: function () {
    const products = Data.sort((a, b) => {
      return b.votes - a.votes;
    });
    this.setState({ products: products });
  },
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
  // ...
```

In our `ProductList` component, the `componentDidMount()` function uses the native JavaScript Array's `sort()` method to ensure that we sort products based upon the number of votes in descending order. This sorted array of products is then used in the special component method `this.setState()`. As anticipated, this method updates `this.state` and re-renders the component.

As we'll need to update and sort the state after we click on an up-vote button, we can define this functionality as a single function so we don't need to duplicate this functionality. We'll call this function `updateState()`:

```
const ProductList = React.createClass({
  getInitialState: function () {
    return {
      products: [],
    };
  },
  componentDidMount: function () {
    this.updateState();
  },
  updateState: function () {
    const products = Data.sort((a, b) => {
      return b.votes - a.votes;
    });
    this.setState({ products: products });
  },
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
  // ...
```

Never modify state outside of `this.setState()`. This function has important hooks around state modification that we would be bypassing.

We discuss state management in detail throughout the book.

Array's `sort()` method takes an optional function as an argument. If the function is omitted, it will just sort the array by each item's Unicode code point value. This is rarely what a programmer desires. If the function is supplied, elements are sorted according to the functions return value.

On each iteration, the arguments `a` and `b` are two elements in the array. Sorting depends on the return value of the function:

1. If the return value is less than `0`, `a` should come first (have a lower index).
2. If the return value is greater than `0`, `b` should come first.
3. If the return value is equal to `0`, leave order of `a` and `b` unchanged with respect to each other.

`sort()` mutates the original array it was called on. Later on in the course, we discuss why mutating arrays or objects can be a dangerous pattern.

If we save and refresh now, we see that the products are back.

Updating state

With state management in place, we need to modify `handleProductUpVote()` inside `ProductList`. When `handleProductUpVote()` is invoked from inside the `Product` component, it should update `Data` and then trigger a state update for `ProductList`:

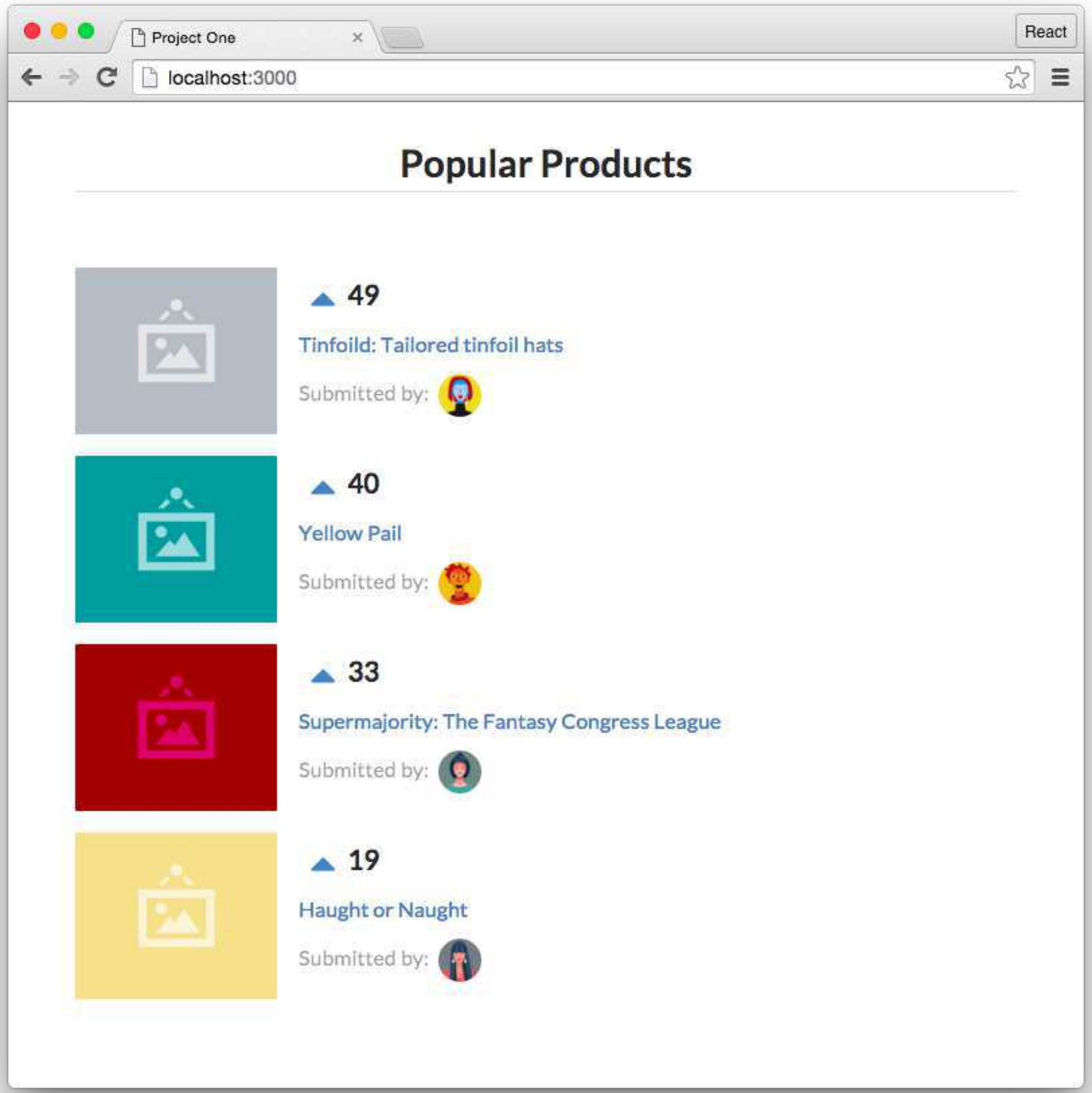
```
const ProductList = React.createClass({
  // ...
  handleProductUpVote: function (productId) {
    Data.forEach((el) => {
      if (el.id === productId) {
        el.votes = el.votes + 1;
        return;
      }
    });
    this.updateState();
  },
  // ...
```

We use Array's `forEach()` to traverse `Data`. When a matching `object` is found (based upon the object's `id`), its `votes` attribute is incremented by 1. After `Data` is modified, we'll call `this.updateState()` to update the state to the current products value. The component's `state` is then updated and React intelligently re-renders the UI to reflect these updates.

Array's `forEach()` method executes the provided function once for each element present in the array in ascending order.

Handling these updates on a remote store would be a fairly trivial update. Instead of modifying `Data` inside of `handleProductUpVote()`, we would make a call to a remote service. The server's response would trigger a callback that would then make a subsequent call to the service asking for the most recent data, updating the state with the data from the response. We'll be exploring server communication in this course's next project.

Save `app.js`, refresh the browser, and cross your fingers:



At last, the vote counters are working! Try up-voting a product a bunch of times and notice how it quickly jumps above products with lower vote counts.

Technically, we can perform the same operation in `this.updateState()` within `getInitialState()` as well, bypassing the brief period where the state is empty. However, this is usually bad practice for a variety of reasons.

We look into why in our advanced components section, but briefly, our `state` will be set by a call to a remote server. As React is still performing the initial render of the app, we'd have to halt it in its tracks, block on a web request, and then update the state when the result is returned. Instead, by just giving React a valid blank "scaffold" for the state, React will render everything first then make parallel, asynchronous calls to whichever servers to fill in the details. Much more efficient.

Again, the style of this app differs slightly from the demo we saw at the beginning of this section. If you'd like to add some additional style to your components, refer to the HTML structure in `app-complete.js`.

Congratulations!

We have just completed our first React app. Not so bad, eh?

There are a ton of powerful features we've yet to go over, yet all of them build upon the core fundamentals we just covered:

1. Think about and organize your app into components
2. JSX and the `render` method
3. Data flow from parent to children through props
4. Event flow from children to parent through functions
5. State vs props
6. How to manipulate state
7. Utilizing React lifecycle methods

Onward!

Chapter Exercises

1. Add down-voting capability to each Product. You can insert a down arrow with this JSX snippet:

```
<i className='large caret down icon'></i>
```

2. Add a "sort direction" button to the top of `ProductList`, above all the products. It should enable the user to toggle sorting products by ascending or descending.