React JumpStart

Student Workbook

Table of Contents

Disclaimers and Copyright Statement

Help us improve our courseware

Please send your comments and suggestions via email to info@watzthis.com

Credits

About the Author

Chris Minnick is a prolific published author, blogger, trainer, web developer and founder of WatzThis, Inc. Minnick has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies, including Microsoft, United Business Media, Penton Publishing, and Stanford University.

Since 2001, Minnick has trained thousands of Web and mobile developers. In addition to his in-person courses, Chris has written and produced online courses for Ed2Go.com, O'Reilly Media, and Pluralsight.

Minnick has authored and co-authored books and articles on a wide range of Internet-related topics including JavaScript, HTML, CSS, mobile apps, e-commerce, Web design, SEO, and security. His published books include JavaScript for Kids, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW eCommerce Certification Bible, and XHTML.

For 16 consecutive years, Chris was among the elite group of 20 software professionals and industry veterans chosen by Dr. Dobb's Journal to be a judge for the Jolt Product Excellence Awards.

In addition to his role with WatzThis?, Chris is a novelist, cheese-, beer-, and winemaker, swimmer, and musician.

Setup Instructions

Course Requirements

To complete the labs in this course, you will need:

- A computer with MacOS, Windows, or Linux.
- Access to the Internet.
- A modern web browser.
- Ability to install software globally (or certain packages pre-installed as specified below).

Classroom Setup

These steps must be completed in advance if the students will not have administrative access to the computers in the classroom. Otherwise, these steps can be completed during the course as needed.

☐ 1. Install node.js on each student's computer.

Go to **nodejs.org** and click the link to download the latest version from the LTS branch.

☐ 2. Install a code editor.

We recommend the free Atom editor (**atom.io**), Brackets (**brackets.io**), or Webstorm (A 30-day trial version is available from ***http://www.jetbrains.com/webstorm***).

☐ 3. Make sure Google Chrome is installed.

☐ 4. Install git on each student's computer.

If you're using MacOS, you already have git installed. If you're on Windows, git can be downloaded from **http://git-scm.com**. Select all the default options during installation.

Testing the Setup

☐ 1. Open a command prompt.
- Use Terminal on MacOS (/Applications/Utilities/Terminal).
- Use gitbash on Windows (installed with git).

☐ 2. Enter `cd` to navigate to the user's home directory (or change to a directory where student files should be created).

☐ 3. Enter the following:

```
git clone https://github.com/watzthisco/intro-to-react
```

The lab solution files for the course will download into a new directory called intro-to-react

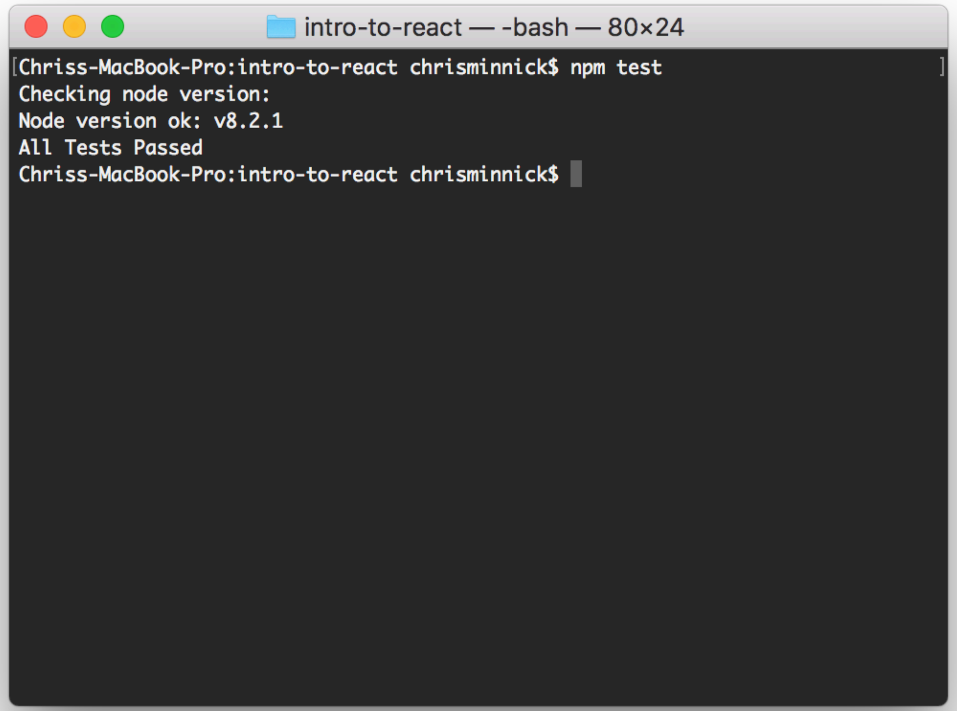
☐ 4. Enter `cd intro-to-react` to switch to the new directory.

☐ 5. Enter `npm install`

This step will take some time. If it fails, the likely problem is that your firewall is blocking ssh access to **github.com** and/or **registry.npmjs.org**.

Note: If you get an error at this point that says npm was not recognized as a command, you'll need to add npm to the system path. This may be helpful: https://stackoverflow.com/questions/27864040/fixing-npm-path-in-windows-8

- ☐ 6. When everything is done, enter `npm run test`
- ☐ 7. If you get an error, delete the **node_modules** folder (by entering `rm -r node_modules`) and run `npm install` again, followed by `npm run test`.
- ☐ 8. A series of things will happen and then a message will appear and tell you that the test passed.



```
Chriss-MacBook-Pro:intro-to-react chrisminnick$ npm test
Checking node version:
Node version ok: v8.2.1
All Tests Passed
Chriss-MacBook-Pro:intro-to-react chrisminnick$
```

Introduction and Git Repo Info

Most of the labs in this course build on the labs that came before. So, if you don't complete a lab or can't get a certain lab to work, it's possible that you can get stuck and won't be able to move forward until the error is corrected.

To help you check your work and to make it possible to come into the class at any point, the git repository for this course contains finished versions of every lab.

The url for the course repository is:
`https://github.com/watzthisco/intro-to-react`

You can find the finished code for each lab inside the **solutions** directory.

Lab 01: Get Started with Create React App

- ☐ 1. Open your terminal application (**Terminal** on MacOS or **git-bash** on Windows)
- ☐ 2. Change to your home directory or your **intro-to-react** project directory.

```
cd intro-to-react
```

- ☐ 3. Use Create React App to make a new React project.

```
npx create-react-app react-sample
```

If this produces an error, you most likely need to upgrade the version of node and npm on your computer (see the setup instructions).

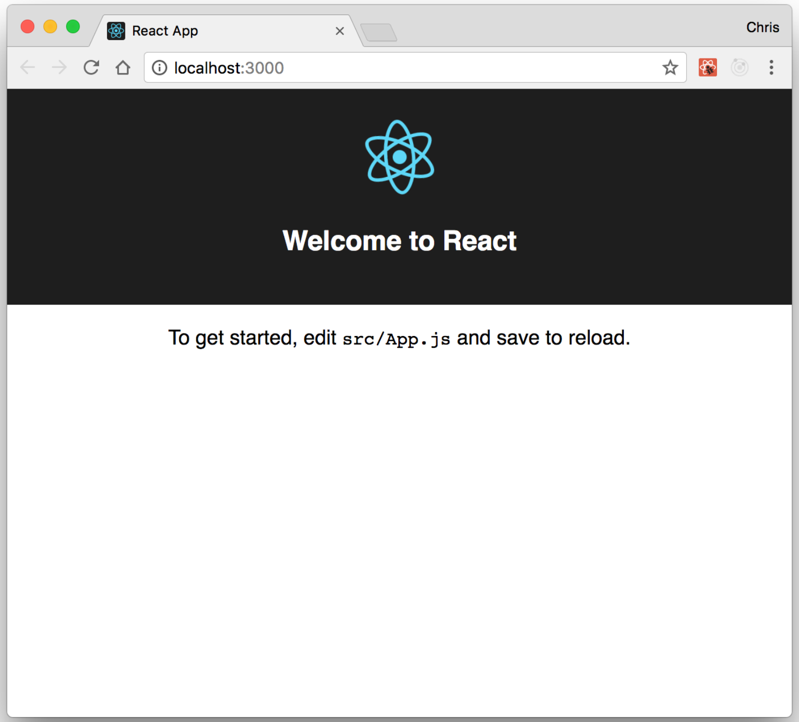
- ☐ 4. Go into the new directory.

```
cd react-sample
```

- ☐ 5. Test that everything was installed and works.

```
npm start
```

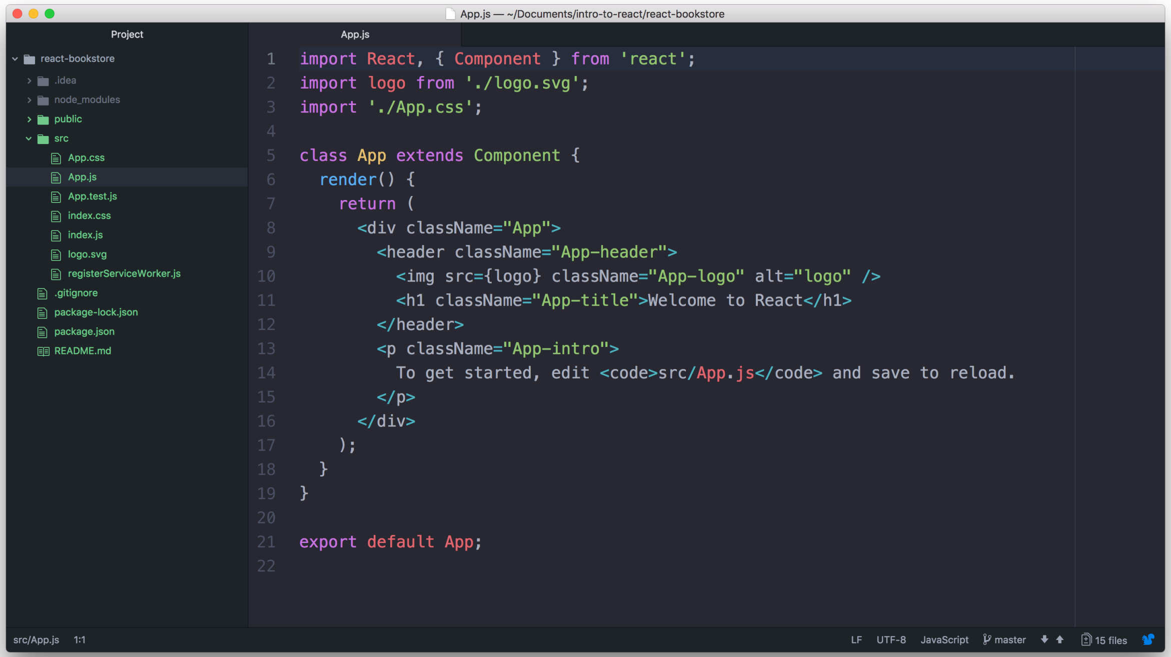
If the app was successfully created, a browser will open and you should see the following Welcome to React page.



Welcome to React

To get started, edit `src/App.js` and save to reload.

- ☐ 6. Open your new project in the code editor of your choice.
- ☐ 7. Find **App.js** inside the **src** directory and open it for editing.



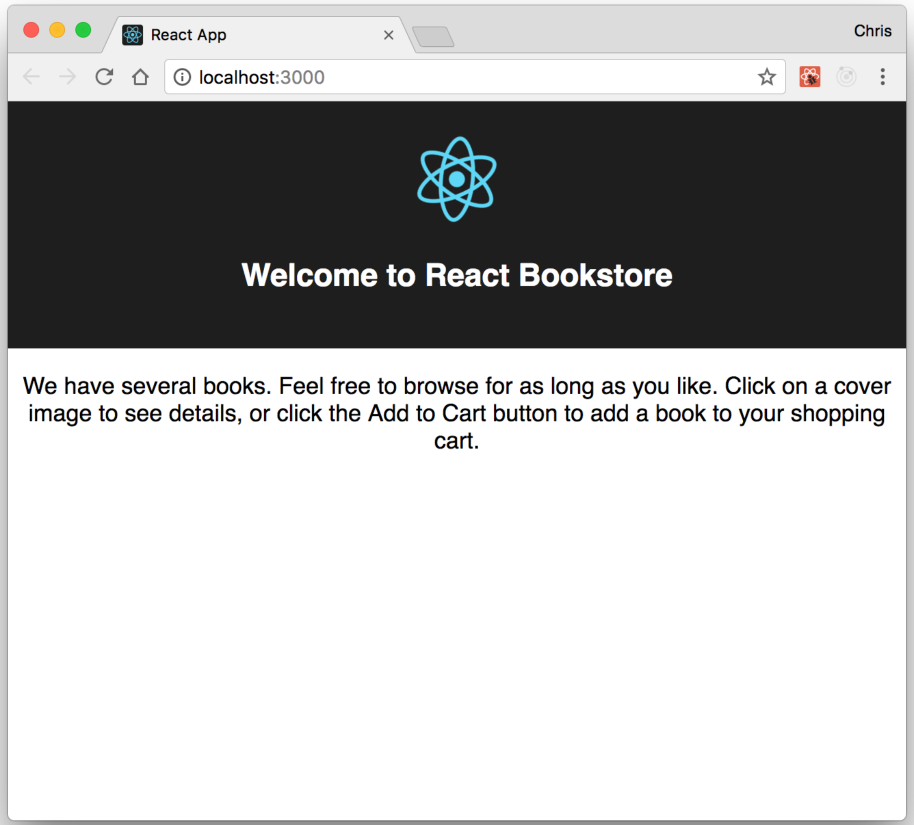
```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

- ☐ 8. Change the contents of the h1 element to say `Welcome to React Bookstore`.
- ☐ 9. Change the contents of the paragraph under the header to welcome visitors to the bookstore:

  ```
  We have several books. Feel free to browse for as long as
  you like. Click on a cover image to see details, or click
  the Add to Cart button to add a book to your shopping cart.
  ```

- ☐ 10. Return to your web browser and notice that the text has been automatically refreshed (if your app is still running) in your terminal emulator.

  If it's doesn't refresh, return to your Terminal emulator and enter **npm start**.



Welcome to React Bookstore

We have several books. Feel free to browse for as long as you like. Click on a cover image to see details, or click the Add to Cart button to add a book to your shopping cart.

Lab 02: Your First Component

React components let you divide your user interface into independent and reusable pieces. The simplest components simply output some piece of HTML, given some input. Such components can be written without a Class. All that's required is a simple JavaScript function.

In this lab, you'll create a functional component to hold the contents of the page footer.

- ☐ 1. Create a new file named **Footer.js** in the **src** directory
- ☐ 2. Type the code below into **Footer.js**

```
import React from 'react';

function Footer(){

      return (<footer>this is the footer.</footer>);

}

export default Footer;
```

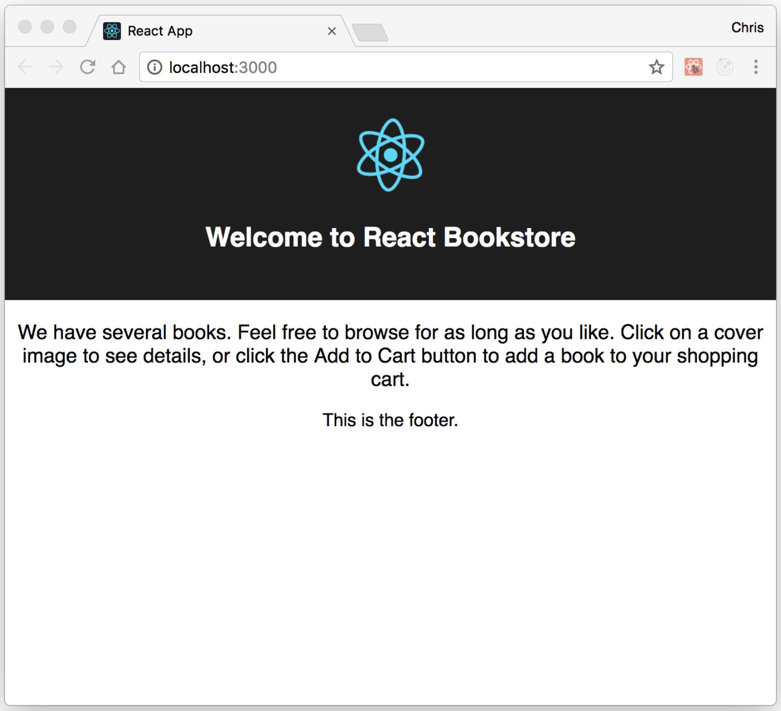
- ☐ 3. Add the following to **App.js** (under the other import statements)

```
import Footer from './Footer.js';
```

- ☐ 4. Add the following inside the **<div>** in **App.js** (after the **<p>** element).

```
<Footer />
```

- ☐ 5. Start the app (if it's not already running) and view it in your browser.



Welcome to React Bookstore

We have several books. Feel free to browse for as long as you like. Click on a cover image to see details, or click the Add to Cart button to add a book to your shopping cart.

This is the footer.

Lab 03: Create More Components

In this lab, you'll make your React application more modular by turning the main parts of the view into components.

☐ 1. Comment out the service worker import and function call in `src/index.js`.

This file manages the caching of your app when used in a production environment, but it can cause confusion during development, so it's best to just leave it out for now.

☐ 2. Using what you learned from creating **Footer.js**, make **Header.js** and **Content.js** and to replace code in **App.js**.

At the end of this lab, your page should look the same as it does at the beginning when opened in a browser.

Your finished return statement in **App.js** should match this:

```
return (
  <div className="App">
    <Header />
    <Content />
    <Footer />
  </div>
);
```

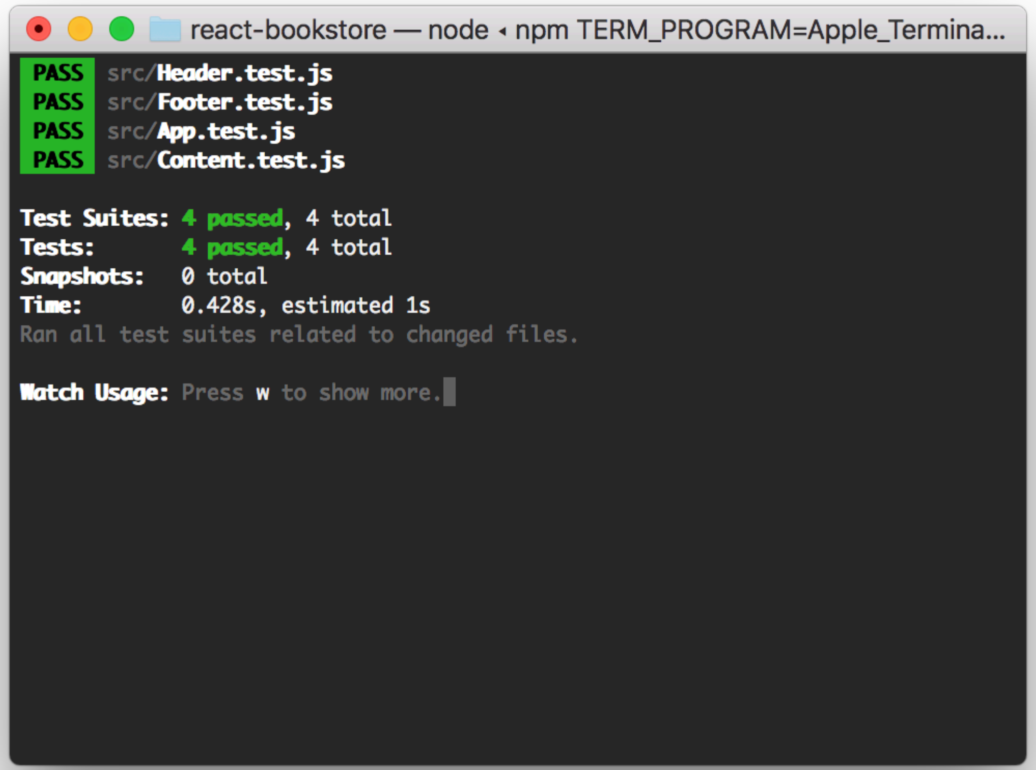
Lab 04: Testing React

Create React App generates a simple smoke test (also known as a build verification test, or BVT) for whether the sample component (**App.js**) renders. In this lab, you'll use the sample test file to create a smoke test for the new components you created in the previous lab.

- ☐ 1. Make copies of **App.test.js** for testing **Footer.js**, **Header.js**, and **Content.js**
- ☐ 2. Modify the contents of the new files to test the new components.
- ☐ 3. Run your tests by entering the following in the command line

```
npm test
```

- ☐ 4. Make sure that all the tests pass.



```
 PASS  src/Header.test.js
 PASS  src/Footer.test.js
 PASS  src/App.test.js
 PASS  src/Content.test.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.428s, estimated 1s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Lab 05: Static Version

The first step in creating a React UI is to create a static version. In this lab, you'll start with a mockup of the ReactBookstore application and you'll create components to make a mockup of the catalog.

- ☐ 1. If you haven't already done it, clone the class github repository, as described in the Setup Instructions at the beginning of this book.

```
git clone https://github.com/watzthisco/intro-to-react
```

- ☐ 2. Open **intro-to-react/labs/lab05/**.

  You'll see three folders: **data**, **images**, and **mockup**.

- ☐ 3. Open **data/products.js** in your code editor.

  This is a file in JavaScript Object Notation (JSON) containing 100 great books. We'll be building a store and shopping cart using this data.

- ☐ 4. Open **labs/lab05/mockup** and look at the **mockup.jpg** image.

  This image shows what the final store and shopping cart should look like.

- ☐ 5. Figure out how you might divide the user interface shown in **mockup.png** into a hierarchy of components. Make a quick drawing on paper, or in MS Paint, or however you like. Check out **mockup-components.png** if you want to see one way it can be done.

Hint 1: If two components need to access the same piece of data, they should have a common parent that holds this data.

Hint 2: Look for repeating elements that can be made into components.

- ☐ 6. Create a new project using create-react-app.

```
npx create-react-app react-bookstore
```

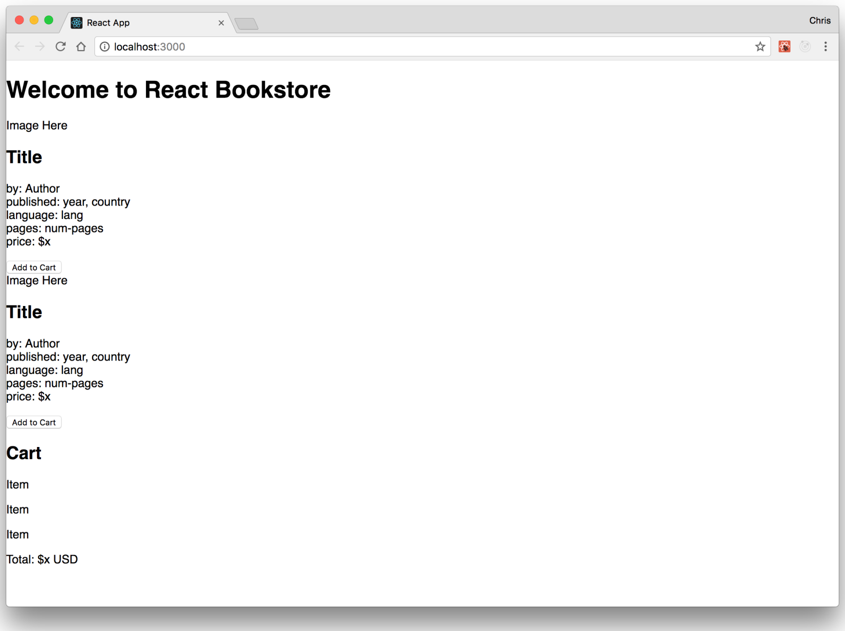
- ☐ 7. Move the **data** directory from the **intro-to-react/labs/lab05** directory into the **src** directory inside your new **react-bookstore** project.
- ☐ 8. Move the **images** directory into the **public** directory.
- ☐ 9. Replace the default **App.js** component in your new project with the following.

NOTE: The components referenced in this code don't exist yet. You'll be creating them in the next step.

```
import React from 'react';
import Cart from './Cart';
import ProductList from './ProductList';

function App() {
    return (
        <div>
            <header>
```

```
                <div>
                    <h1>Welcome to React Bookstore</h1>
                </div>
            </header>

            <div>
                <div>
                    <ProductList />
                </div>
                <div>
                    <Cart />
                </div>
            </div>

            <footer>

            </footer>
        </div>
    );
}

export default App;
```

- ☐ 10. Based on the simple test you saw in the last lab, make test suites and smoke tests for all the components referenced in the App component that don't currently have tests.
- ☐ 11. Make the necessary sub-components to pass the tests you wrote in step 10.
- ☐ 12. Run **npm test** and make sure that your tests pass.
- ☐ 13. Run **npm start** to verify that your code builds. Your UI should now look something like this:



Welcome to React Bookstore

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Cart

Item

Item

Item

Total: $x USD

Lab 06: Styling React

In this lab, you'll use Bootstrap to apply some global layout styles to the react-bookstore project, and you'll learn how to use style modules to add styles to individual components.

☐ 1. Install Bootstrap inside your **react-bookstore** project.

```
npm install --save bootstrap
```

☐ 2. Link to **bootstrap.css** inside of **index.js.**

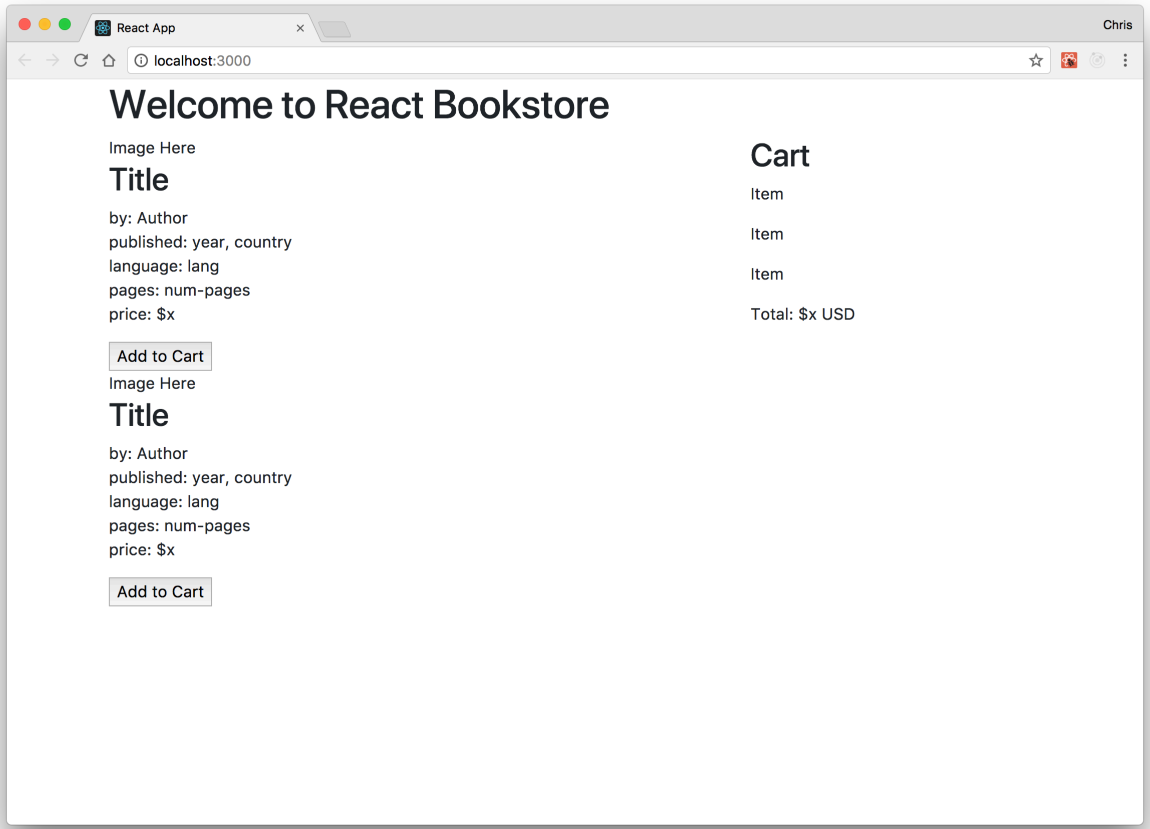
```
import 'bootstrap/dist/css/bootstrap.css';
```

☐ 3. Add Bootstrap style classes to **App.js**.

```
import React from 'react';
import Cart from './Cart';
import ProductList from './ProductList';

function App() {
    return (
        <div className="container">
            <header className="row">
                <div className="col-md-12">
                    <h1>Welcome to React Bookstore</h1>
                </div>
            </header>
            <div className="row">
                <div className="col-md-8">
                    <ProductList/>
                </div>
                <div className="col-md-4">
                    <Cart/>
                </div>
            </div>
            <footer>
            </footer>
        </div>
    );
}

export default App;
```

☐ 4. Run **npm test** and make sure that your tests pass.

☐ 5. Run **npm start** to verify that your code builds. Your UI should now look like this:



Welcome to React Bookstore

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Cart

Item

Item

Item

Total: $x USD

☐ 6. Return to your terminal application and press **Ctrl-C** to stop the server.

☐ 7. Modify your **ProductList** component to make each product an item in an unordered list.

```
<ul>
      <li><Product /></li>
      <li><Product /></li>
</ul>
```

☐ 8. Create a new file in the **src** directory named **ProductList.css.js**.

This will be our first style module.

☐ 9. Inside **ProductList.css.js**, create a default export containing two style objects, **productList** and **productListItem**.

```
export default {
    productList: {
        padding: 0,
        display: 'flex',
        flexWrap: 'wrap',
        justifyContent: 'space-between',
        alignItems: 'stretch'
    },
        productListItem: {
            listStyle: 'none',
            width: '32%'
        }
}
```

This two style objects will control the layout of the products inside the product list.

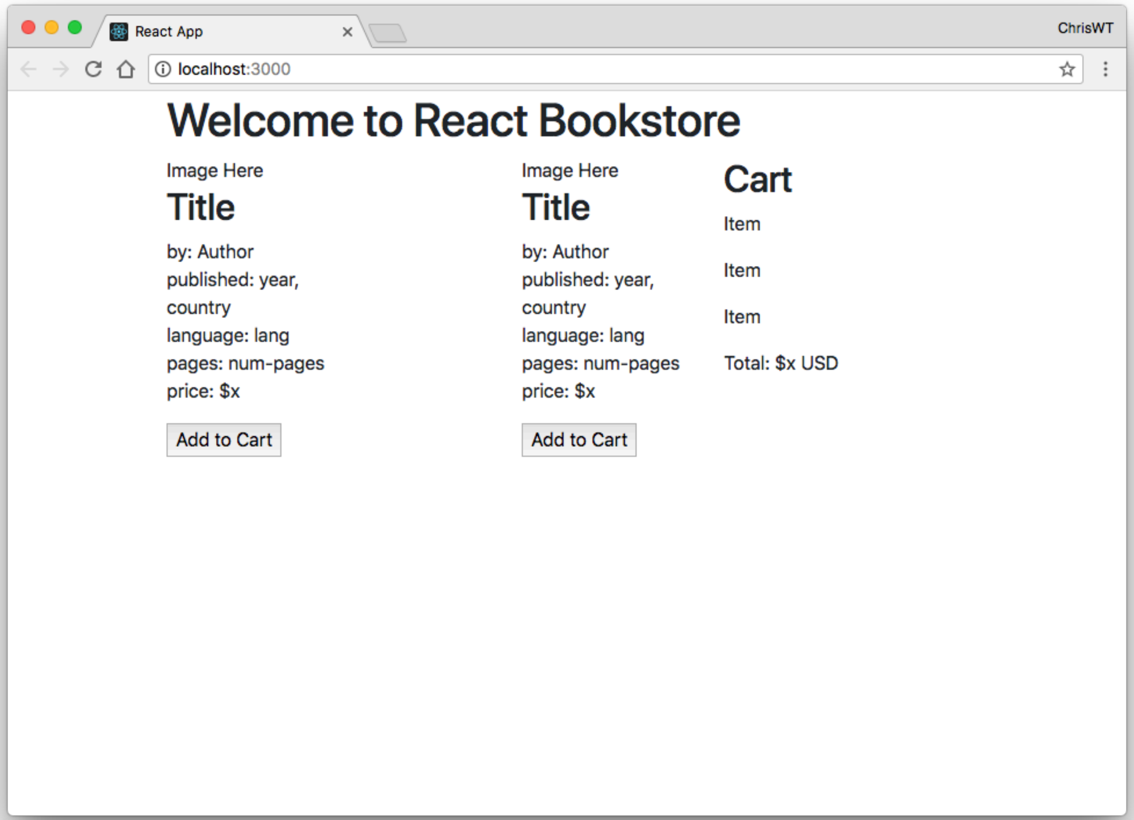
☐ 10. Import the style module into **ProductList.js** and give the module the name **styles**.

```
import styles from './ProductList.css.js';
```

☐ 11. Attach the styles to the appropriate elements.

```
<ul style={styles.productList}>
  <li style={styles.productListItem}><Product /></li>
  <li style={styles.productListItem}><Product /></li>
</ul>
```

☐ 12. Run **npm start** and preview the styled list in your browser.



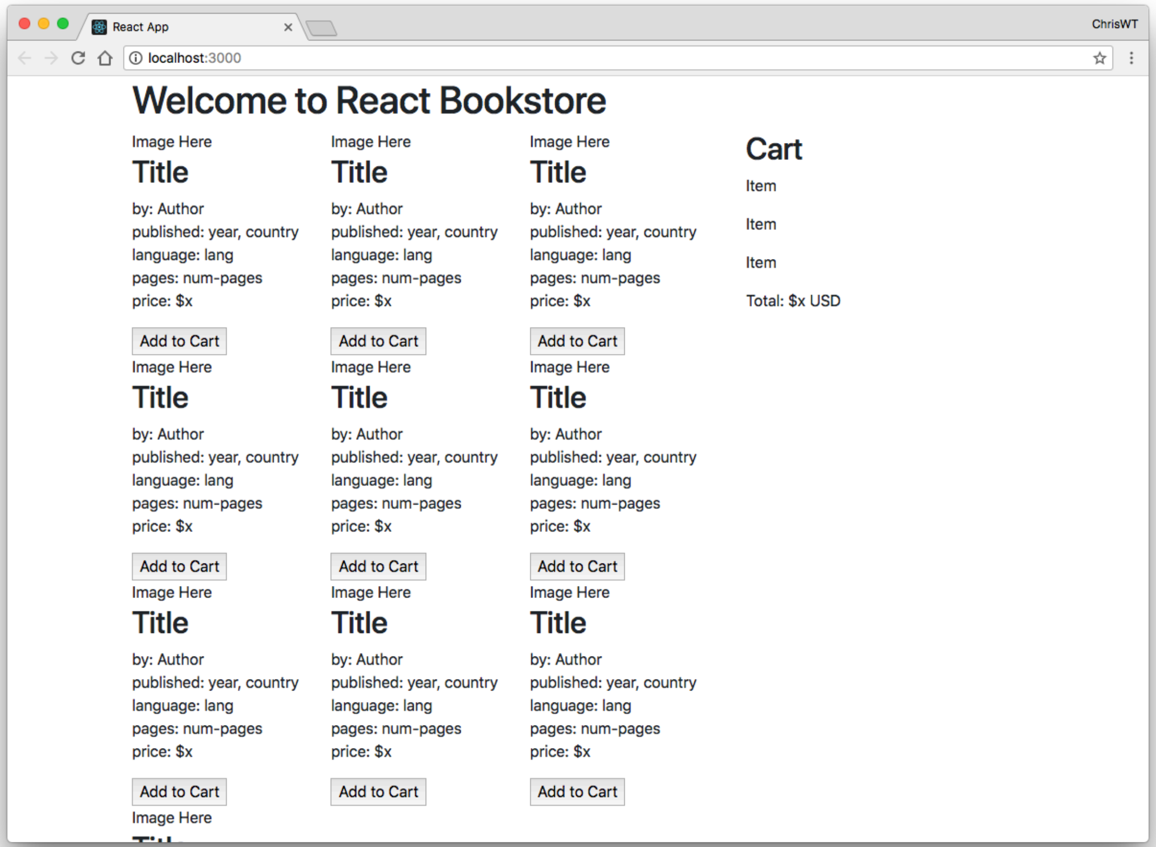
Welcome to React Bookstore

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Cart
Item
Item
Item
Total: $x USD

☐ 13. Add as many additional `<Product />` elements to the list as you want by copying and pasting additional lines in the return statement.



Welcome to React Bookstore

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here

Cart
Item
Item
Item
Total: $x USD

☐ 14. Create empty style modules for `Product`, `Cart`, and `CartItem` and import them into each module using the same pattern shown above.

☐ 15. Make an empty style object for each component and attach this empty style object to the outermost element in each component's return.

For example, here's what the return statement for the `Cart` component might look like.

```
return (
<div style={styles.cart}>
     <h2>Cart</h2>
     <CartItem />
     <CartItem />
     <CartItem />
     Total: $x USD
</div>
);
```

Lab 07: Props and Containers

At this point, you should have a static and partially styled version of the application, built using the following React components:

```
App
ProductList
Product
Cart
CartItem
```

In this lab, we'll reorganize our project to use container components and to pass data to the presentational components via the props object.

- ☐ 1. Create two sub-directories inside **src**: one named **components**, and one named **containers**
- ☐ 2. Move **ProductList.js**, **Product.js**, **Cart.js**, and **CartItem.js**, along with their test suites and css modules, into the **components** directory.
- ☐ 3. Move **App.js** and **App.test.js** into the containers folder and update references as necessary.

Note: Make sure to update the import of `App` in **index.js**

- ☐ 4. Import the data file into **App.js.**

Note: Because the data module uses a default export, you can import it using any name that you like. I've used `productsData` below, but you can use anything that makes sense to you.

```
import productsData from '../data/products';
```

- ☐ 5. Pass `productsData` to the `ProductList` component as a prop called `products`.

```
<ProductList products={productsData} />
```

- ☐ 6. Update `ProductList` to loop over the products array and generate a `Product` for each element in the array, passing appropriate data to the `Product` components as props.

```
return(
    <ul style={styles.productList}>
        {this.props.products.map(product => (
            <li key={product.id}
                style={styles.productListItem}>
                <Product {...product} />
            </li>
        ))}
    </ul>
);
```

- ☐ 7. Inside the render method of `Product`, deconstruct `this.props` into individual constants (to save yourself from having to type '`this.props`' repeatedly in the return statement.

```
const { title, author, published, country, lang, pages,
image, url, price } = this.props;
```

☐ 8. Update `Product` to make use of the props passed to it to display data about each product.

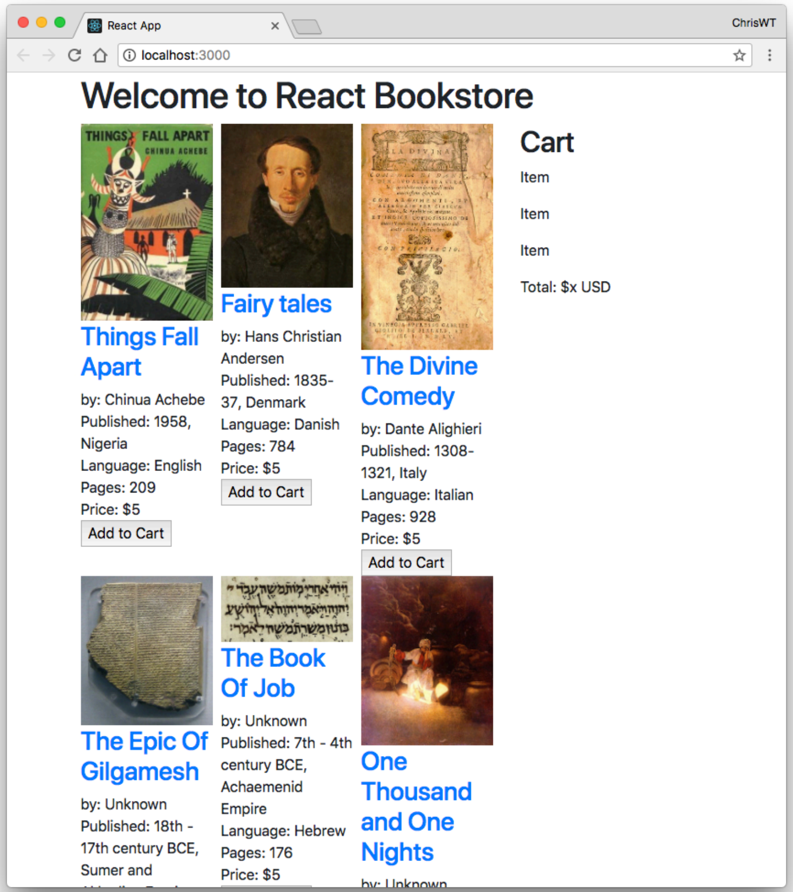
☐ 9. Create a new style rule in **Product.css.js** called thumbnail and set properties to format the book thumbnail images.

```
thumbnail: {
    maxWidth: "100%",
    height: "auto"
}
```

☐ 10. Add a style attribute to the `img` element in `Product`. Your `img` element should look something like the following:

```
<img style={styles.thumbnail} src={image ? "images/" +
image:"images/default.jpg"} alt="{title}" />
```

☐ 11. Run `npm start`



Welcome to React Bookstore

Things Fall Apart
by: Chinua Achebe
Published: 1958, Nigeria
Language: English
Pages: 209
Price: $5
Add to Cart

Fairy tales
by: Hans Christian Andersen
Published: 1835-37, Denmark
Language: Danish
Pages: 784
Price: $5
Add to Cart

The Divine Comedy
by: Dante Alighieri
Published: 1308-1321, Italy
Language: Italian
Pages: 928
Price: $5
Add to Cart

Cart
Item
Item
Item
Total: $x USD

The Epic Of Gilgamesh
by: Unknown
Published: 18th - 17th century BCE, Sumer and

The Book Of Job
by: Unknown
Published: 7th - 4th century BCE, Achaemenid Empire
Language: Hebrew
Pages: 176
Price: $5

One Thousand and One Nights
by: Unknown

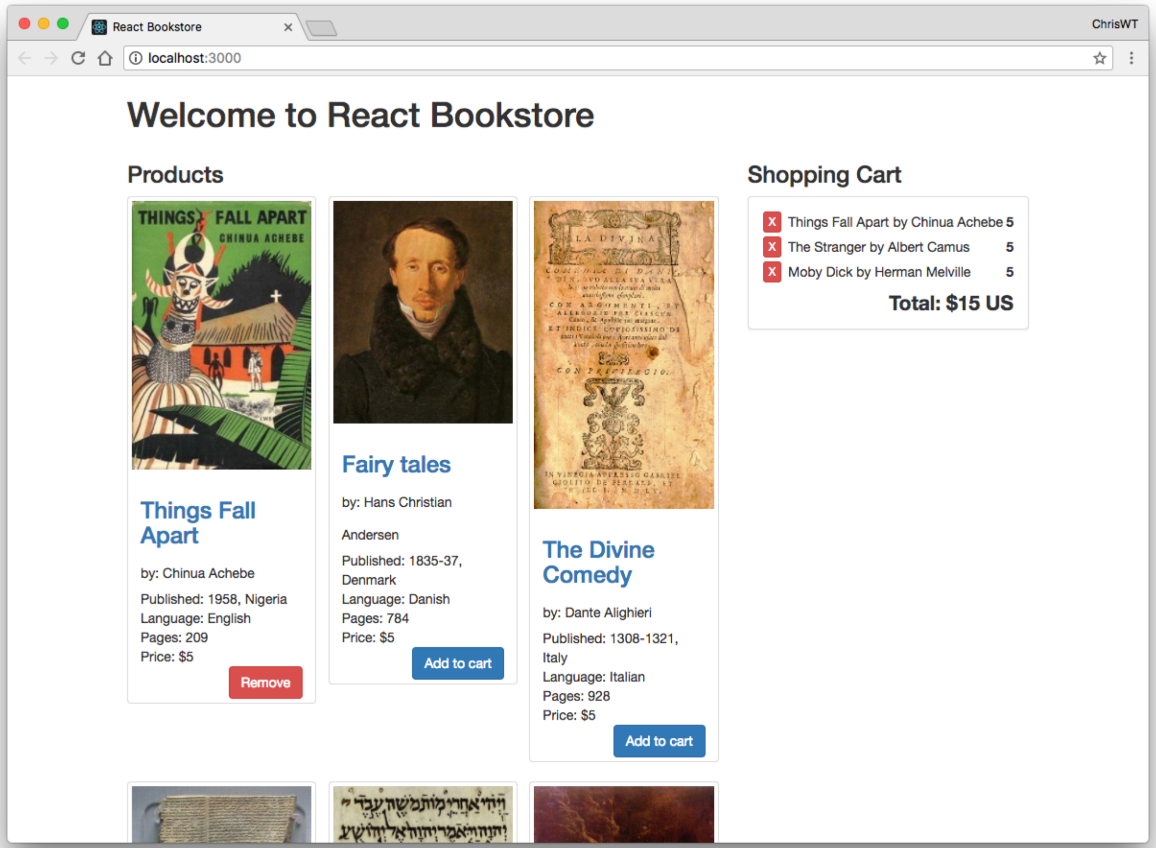
Lab 08: Adding State

So far, we have a static version of the React Bookstore, built using components that pass data down using props. At this point, there's no way for the data to change or for users of the bookstore to add products to their cart.

State is the data in your application that makes your application interactive. The first step in adding state to a React application is to figure out what data needs to be part of the state object, and then to set this initial state and pass it down to the components that need it.

To determine what is state, think about what data changes in response to user input, isn't passed down via props, and can't be computed based on props.

Looking at the following screenshot of the finished store and shopping cart, what information fits this description?



Welcome to React Bookstore

Products

Things Fall Apart
by: Chinua Achebe
Published: 1958, Nigeria
Language: English
Pages: 209
Price: $5

Fairy tales
by: Hans Christian Andersen
Published: 1835-37, Denmark
Language: Danish
Pages: 784
Price: $5

The Divine Comedy
by: Dante Alighieri
Published: 1308-1321, Italy
Language: Italian
Pages: 928
Price: $5

Shopping Cart

X	Things Fall Apart by Chinua Achebe	5
X	The Stranger by Albert Camus	5
X	Moby Dick by Herman Melville	5

Total: $15 US

When you think that you know, turn the page to see the answer.

In this React Bookstore application, the only thing that fits that needs to be part of the state is the list of items that are currently in the shopping cart.

The next step in adding state to our application is to figure out where the state should live. Look again at the screenshot on the previous page. Which components need to know what's in the shopping cart?

If you said `Cart` and `Product`, you're correct.

To determine where the state should live, look for a component that is a common parent (or ancestor) to both `Cart` and `Product`. The only component that fits this description is the container component, `App`. Therefore, `App` is where the state should live.

Follow these steps to add state to the application.

☐ 1. Open **src/containers/App.js** in your code editor.

☐ 2. Convert `App` from a functional (stateless) component to a class.

```
class App extends React.Component {
    render() {
        return (
            <div className="container">
                <header className="row">
                    <div className="col-md-12">
                        <h1>Welcome to React Bookstore</h1>
                    </div>
                </header>
                <div className="row">
                    <div className="col-md-8">
                        <ProductList
                           products={productsData}/>
                    </div>
                    <div className="col-md-4">
                        <Cart/>
                    </div>
                </div>
                <footer>
                </footer>
            </div>
        );
    }
}
```

☐ 3. Add a constructor to `App`, above the `render` method.

```
constructor(){

}
```

☐ 4. Call `super()` inside the constructor.

```
constructor(){
      super();
}
```

☐ 5. Create an empty object and assign it to the initial state.

```
constructor(){
      super();
      this.state = {};
}
```

☐ 6. Create a property of the state object named items and initialize it as an array.

```
constructor(){
      super();
      this.state = {
            items: []
      }
}
```

The `items` array will hold an array of the id numbers for each product currently in the cart.

☐ 7. Pass the items array into both the `ProductList` and `Cart` components as props.

```
<div className="col-md-8">
      <ProductList products={productsData}
                   inCart={this.state.items}/>
</div>
<div className="col-md-4">
      <Cart inCart={this.state.items} />
</div>
```

☐ 8. Add some product ids to the items array, for testing.

```
this.state = {
      items: ['1','3','5']
}
```

☐ 9. Inside the `ProductList` component, figure out how to pass a prop down to each product that is currently in the shopping cart and change the message on its button from **Add to Cart** to **In Cart**.

Hint: ES2016 contains an `Array.includes()` method which returns true if the value passed into it is the value of an element in the array.

If you get stuck, look at the solution inside **intro-to-react/solutions/lab08**.

☐ 10. Inside the `Cart` component, figure out how to use the items array to generate a list of `CartItem` components.

Hint: One way to do this would be to create an array of just the product objects with `id`s that match the values in `this.state.items`. You can then pass that array to the `Cart` component. You could use this method to find each matching product.

```
getProduct(products, item) {
    return products.find(product => item === product.id);
}
```

☐ 11. Modify the `CartItem` component to display the name and price of the item.

☐ 12. Calculate the total price of all the items in the shopping cart and display it in the `Cart` component.

Lab 09: Interactions, Events, and Callbacks

User interactions happen when a user clicks a button, moves their mouse, enters text into a form, interacts with a touch screen, and so forth. These interactions trigger events in the web browser (or another user agent), which can be listened for and responded to using JavaScript.

In addition to user interactions, many other things trigger events that can be listened for and responded to.

Over the years, web browsers have developed slightly different ways of handling events. To eliminate these differences, it's common for JavaScript libraries and frameworks to wrap the browser's native events in a cross-browser abstraction layer. React's cross-browser event handling system is call **Synthetic Events**.

Except for the fact that it works the same in every browser, Synthetic Events works the same as the native browser event handling.

Unlike many other JavaScript frameworks and libraries, React doesn't feature 2-way data binding. What this means is that changes to the model in a React application (i.e. the state object) trigger updates to the view, but changes to the view don't automatically update the model. This one-way data flow makes it easier to test and reason about React applications, but it is also the cause of one of the trickiest parts of React to understand.

In this lab, you'll learn about using `bind()` to create functions that can be passed from parent components to child components, you'll learn how to pass those functions, and you'll learn how to call child functions to update the state of a React view.

The first key to understanding how to create dynamic user interfaces with React is to understand React's `setState()` method.

The `setState()` method takes as its argument an object representing a change to the state object. Calling `setState()` also triggers the `render()` method, which causes the component and its children to be updated in the browser to reflect the new data.

In our application, the state consists of an array of item numbers. In the previous lab, we set the initial state of the application to an array containing three items. If another item were added to the cart, you might consider using an array method to update the state and then use `setState()` to trigger the re-rendering, like this:

```
this.state.items.push(newItem);        // <=== don't do this
this.setState({items: this.state.items});
```

However, in React, state should be treated as immutable. What this means is that you should never perform operations on the state object directly, except in the component's constructor.

Directly manipulating the state object can cause problems with the rendering and lifecycle methods in React.

Instead, you should use the `setState()` method, which accepts as its argument an object to be merged into the state. For example, if you set the initial state in the constructor, like this:

```
constructor(props){
  super(props);
  this.state = {items: [], isVisible:false}
}
```

You can mutate the state outside of the constructor by creating an object containing the property or properties that you want to change and passing it into the `setState` method.

```
this.setState({
     items: [...this.state.items, newItem]
     });
```

This example uses the ES6 spread operator to split the items array into separate values. You can then add the new item to the end of the array and update `state.items` without mutating the state object directly.

If you want to remove an item from an array in the state object, one way to do it is by knowing the position of the element you want to remove. You can then create a new array without the item in question, using the following code:

```
 let newData = this.state.data.slice(); //copy array
 newData.splice(index, 1); //remove element
 this.setState({data: newData}); //update state
```

Another way to remove an item from an array is by using the `Array.filter` method, like this:

```
let newData = this.state.data.filter(
     id => id !== idToRemove); //filter out a value
     this.setState({data: newData}); //update state
```

Now that you understand how to update the state in React, the next thing to understand is how child components can call functions that affect the state of the parent component.

The key is in the `bind()` function. The job of `bind()` is to create a new function that has the `this` keyword set to a specific value, and with a list of arguments passed to the new function when it's called.

In React, we use `bind()` to create a function in one component that can be called in response to an event in another component but that will affect the original component.

To see how this works in practice, follow these steps to add interactivity to the React Bookstore user interface.

☐ 1. Create the following method inside the App component (**containers/App.js**).

```
addToCart(id) {
    let newItems = [...this.state.items, id];
    this.setState({
        items: newItems
    })
}
```

It's possible now to call the `addToCart` function from within the `App` component by using `this.addToCart()`. However, what we want to do is to call `addToCart()` in response to a click on the button in the `Product` component.

To make it possible to call the function with the context of the `App` component, we need to bind it.

☐ 2. Add the following inside of the constructor for the `App` component to create a new function that's explicitly bound to `App`.

```
this.addToCart = this.addToCart.bind(this);
```

☐ 3. Pass the bound `addToCart` function down to the `ProductList` component as a prop.

```
<ProductList addToCart = {this.addToCart}
             products = {productsData}
             inCart = {this.state.items}
/>
```

☐ 4. Open the `ProductList` component and pass the `addToCart` function to the `Product` components as a prop.

```
<Product {...product}
         inCart = {inCart.includes(product.id)?'1':''}
         addToCart = {this.props.addToCart}
         />
```

☐ 5. Inside the Product component, create a new function, called `handleClick`. The job of this function will be to call the `addToCart` function, passing it the `id` of the current Product.

```
handleClick(){
    this.props.addToCart(this.props.id);
}
```

☐ 6. Call the `handleClick` function as the event handler for the click event on the button. Bind the `handleClick` function, this time binding it in the render method.

```
<button
      onClick={this.handleClick.bind(this)}>
      {this.props.inCart?"In Cart":"Add to Cart"}
</button>
```

Note: You could also bind `handleClick()` inside the `Product` component's constructor.

Note: Another way to call the addToCart function and pass in the id is to use an arrow function as the event handler, like this:

```
<button onClick={()=>{this.props.addToCart(this.props.id)}}>
```

☐ 7. Run **`npm start`** and test out your application.

☐ 8. Challenge: Make clicking on the button when it displays the "In Cart" message remove the product from the cart.

Here's a function you can use to do the removal of items:

```
removeFromCart(idToRemove) {
      let newItems = this.state.items.filter(
                        id => id !== idToRemove);
      this.setState({items: newItems});

}
```

Lab 10: Component Lifecycle

Right now, the bookstore retrieves product data from an array and displays books in the order in which they're in the array. But, what if you want to retrieve the data from the web and display in a random order each time a visitor comes to the store?

You could randomize the array inside the `ProductList` component, but this has unintended consequences. Try the following to find out what happens.

- ☐ 1. Add the following function inside the `ProductList` component (outside of the render method):

```
shuffleArray(array) {
    for (let i = array.length - 1; i > 0; i--) {
        let j = Math.floor(Math.random() * (i + 1));
        let temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    return array;
}
```

- ☐ 2. Inside `render()`, create a new array by sorting the one passed as a prop.

```
let sortedProducts =
this.shuffleArray(this.props.products);
```

- ☐ 3. Replace the array used for displaying the products with the new randomly sorted array.

```
{sortedProducts.map(product => (
```

- ☐ 4. Run **npm start**, and try adding some products to the cart.

  Notice that the order of the cart changes every time you click a button. Clearly this is not what we want.

One way to fix this problem is to add the products array to the application state. To do this, follow these steps:

- ☐ 5. Copy the `shuffleArray` method from `ProductList` and paste it into the `App` component.
- ☐ 6. Revert the `ProductList` component to how it was before you made the previous changes.
- ☐ 7. In `App`'s constructor, add `products` to the `state`, sorting it using the `shuffleArray` function:

```
this.state = {
    items: [],
    products: this.shuffleArray(productsData)
};
```

- ☐ 8. Change the value for the `products` prop that's passed to the `ProductList` component to `this.state.products`.

```
<ProductList
      addToCart={this.addToCart}
      removeFromCart={this.removeFromCart}
      products={this.state.products}
      inCart={this.state.items}
/>
```

☐ 9. Run **npm start** and notice that the array no longer changes after the page loads.

Now that we have the products added to the state, it's easy to load them from a web service using AJAX before sorting them.

This is where component lifecycle methods come in handy. We can use the `componentDidMount` method to load and sort the array of product data once the component is fully loaded and ready to receive data.

☐ 10. Set product equal to an empty array in the constructor.

```
this.state = {
    items: [],
    products: []
};
```

☐ 11. Inside the constructor, create an empty array for the sorted products.

```
this.sortedProducts = [];
```

☐ 12. Override componentDidMount inside App.js.

```
componentDidMount(){

}
```

☐ 13. Remove the import of the product data.

☐ 14. Make a copy of products.js named `products.json` and save it in your public directory.

☐ 15. Open `products.json` and remove the variable assignment and the semicolon and export statement so that the file starts with [and ends with];

☐ 16. Inside `componentDidMount`, use the native browser fetch method to load the product data, sort it, and then update the state:

```
componentDidMount() {
    fetch('http://localhost:3000/data/products.json')
            .then(response => response.json()
            .then(products => this.shuffleArray(products))
            .then(products => {
                this.setState({products:products})
            }))
};
```

☐ 17. Start the server and preview the app in your browser -- **notice that it produces an error.**

The problem is that the UI is loading before the data, so functions that rely on having elements in the products array will fail.

☐ 18. Fix the ProductList component by checking whether there are products before attempting to render it.

```
render(){
    const inCart = this.props.inCart;
    const products = this.props.products;

if(products.length > 0) {

    return (
        <ul style={styles.productList}>
            {products.map(product => (
                <li key={product.id} style={styles.productListItem}>
                    <Product {...product}
                      inCart={inCart.includes(product.id) ? '1' : ''}
                      addToCart={this.props.addToCart}
                      removeFromCart={this.props.removeFromCart}
                    />
                </li>
            ))}
        </ul>
    );

} else {
    return null;
}
    }
}
```

Note: There are more concise (although perhaps not as clear) ways to write the above conditional rendering code. Visit https://www.robinwieruch.de/conditional-rendering-react/ to see and try out some other methods.

Lab 11: PropTypes

PropTypes are a built-in way to typecheck your React components' props. In this lab, you'll use PropTypes to make sure that your components receive the correct props.

- ☐ 1. In `ProductList`, import `prop-types`.

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

- ☐ 2. In **ProductList.js**, but outside of the class definition, add a `propTypes` object to `ProjectList`.

  ```
  ProductList.propTypes = {
  };
  ```

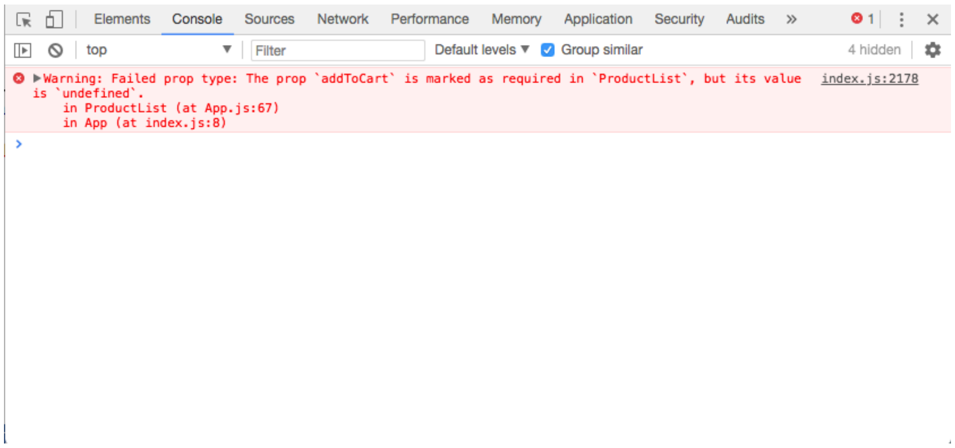
- ☐ 3. In `ProductList.PropTypes`, create a property for each prop that `ProductList` receives, and use a PropType validator to make sure that it's the correct type and that required props are received. I've done the first one for you:

  ```
  ProductList.propTypes = {
      addToCart: PropTypes.func.isRequired,


  };
  ```

- ☐ 4. Run your application and open the JavaScript console (**CTRL-SHIFT-J** in Chrome for Windows or **CMD-Option-J** on Mac) to verify that there are no errors.
- ☐ 5. Comment out one or more of the props passed from `App` to `ProductList`.

  ```
  <ProductList
      /* addToCart={this.addToCart}*/
      removeFromCart={this.removeFromCart}
      products={this.sortedProducts}
      inCart={this.state.items}
  />
  ```

- ☐ 6. Check the JavaScript console to see the PropType warning.



```
▶Warning: Failed prop type: The prop `addToCart` is marked as required in `ProductList`, but its value   index.js:2178
is `undefined`.
    in ProductList (at App.js:67)
    in App (at index.js:8)
```

- ☐ 7. Restore the prop passing so that there are no errors.

☐ 8. Add a `propType` object to every other component that receives props.

Lab 12: Testing with Jest and Enzyme

In this lab, you'll configure Jest and Enzyme and write tests for your components.

☐ 1. Install Enzyme and the Enzyme adapter.

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

☐ 2. Create a new file called **setupTests.js** inside **src** and configure the Enzyme adapter.

```
import { configure } from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

☐ 3. Inside **CartItem.test.js**, import `shallow` from `enzyme`.

```
import {shallow} from 'enzyme';
```

☐ 4. Modify your smoke test in **CartItem.test.js** to use enzyme and to check that it renders what you expect it to render.

```
it('renders without crashing', () => {
     const component =
          shallow(<CartItem title="test book" price="6"/>);
     expect(component.text())
          .toEqual('test book - 6');


});
```

☐ 5. Run `npm test` to verify that this works and that your test passes.

☐ 6. Define the component variable outside of the `it()` function.

```
let component;
```

☐ 7. Write a `beforeEach` method to render the component before each test runs.

```
beforeEach(() => {

  component =
     shallow(<CartItem
                title="test book"
                price="6" />);
}
```

☐ 8. Remove the rendering of the component from the `it()` function.

```
it('renders without crashing', () => {

     expect(component.text())
          .toEqual('test book - 6');


});
```

☐ 9. Convert the other test files (for `App`, `ProductList`, `Product`, and `Cart`) to use Enzyme and get all your tests to pass.

☐ 10. Challenge: Create a mock for the `handleClick` function in `CartItem` and test that clicking the button causes the function to be called.

Lab 13: Implementing Redux

As your application grows, you may find it useful to transition to Redux. It's unlikely that our existing app would benefit at this point from Redux, but the process that we'll go through to convert it to Redux will show you the steps involved in a simplified example.

The initial setup of Redux involves many steps and can be confusing at first. But once we have it in place, you'll see how easy it is to add more actions and reducers as the application grows.

☐ 1. Install redux, the React bindings, and the developer tools.

```
npm install --save redux react-redux redux-devtools
```

Step 1: Create a store

☐ 2. In `index.js`, import `createstore` and `combineReducers`, and from redux.

```
import {createStore, combineReducers} from 'redux';
```

☐ 3. Import `Provider` from react-redux.

```
import {Provider} from 'react-redux';
```

☐ 4. Create the root reducer.

```
const rootReducer = combineReducers({
      cart: cart,
      products: products
});
```

☐ 5. Import the reducers (which we'll create in a moment).

```
import {cart, products} from './reducers';
```

☐ 6. Define the store's initial state.

```
const initialState = {
    cart: {items:[]},
    products: {products:[]}
};
```

☐ 7. Create the store by passing the root reducer and the initial state into the `createStore` method.

```
let store = createStore(
      rootReducer,
      initialState
);
```

☐ 8. Wrap the **App** component in a **Provider** and pass the store to **Provider** as a prop.

```
ReactDOM.render(<App /></Provider>,
                              document.getElementById('root'));
```

Step 2: Write the reducers

- ☐ 9. Create a directory in **src** named **reducers**.
- ☐ 10. Create **index.js** inside **reducers**.

The next step is to define the ways in which the state of the cart can change. Changes in redux happen in response to actions. So, our reducer needs to listen for certain actions that correspond to different changes in the state of the cart and then make those changes.

- ☐ 11. Write and export the cart reducer function as a module. The cart reducer contains a switch statement with a case for each possible action that can happen in the cart.

```
export function cart(state = {}, action = {}) {
        switch(action.type) {
                case 'CART_ADD':
                        return; //todo: finish this
                case 'CART_REMOVE':
                        return; //todo: finish this
                default:
                        return state; //no relevant action type
        }
}
```

- ☐ 12. Inside the `CART_ADD` case, use the functionality from the addToCart function (in **App.js**) to add the product id passed by the action to the items array.

```
case 'CART_ADD':
        return {
                ...state,
                items: [...state.items, action.payload.productId]
                };
```

- ☐ 13. Inside the `CART_REMOVE` case, use the functionality from the removeFromCart function (in App.js) to remove the **productId** passed to it from the items array.

```
case 'CART_REMOVE':
return {
     ...state,
     items: state.items.filter(id => id !==
action.payload.productId)
};
```

- ☐ 14. Write and export the **products** reducer function. It should have one case, named LOAD_PRODUCTS which will update the state with the list of products fetched by the componentDidMount method of App.js.

```
export function products(state = {}, action = {}) {
    switch (action.type) {
        case 'LOAD_PRODUCTS':
            return {
                ...state,
                products: action.products
            };
        default:
            return state; //no relevant action type
    }
}
```

Step 3: Write the Actions and Action Creators

- ☐ 15. Create a new directory in **src**, named **actions**.
- ☐ 16. Create a file named **index.js** inside **actions**, then write (and export) the functions inside it that will create the actions that trigger changes to the state inside the reducers we just wrote.

```
export function addToCart(productId) {
    return {
        type: 'CART_ADD',
        payload: {
            productId
        }
    }
}

export function removeFromCart(productId) {
    return {
        type: 'CART_REMOVE',
        payload: {
            productId
        }
    }
}


export function loadProducts(products) {
    return {type: 'LOAD_PRODUCTS', products}
}
```

Now that we have the action creators that will be dispatched when the user interacts with the application, and we have the reducers that will mutate the state in response to those actions, the last step is to hook up the user interactions (button clicks) to the dispatch of the actions.

- ☐ 17. Import the action creator functions, the connect method of react-redux and the bindActionCreator method into **containers/App.js**.

```
import * as actionCreators from '../actions';
import {bindActionCreators} from 'redux';
import {connect} from 'react-redux';
```

- ☐ 18. In App.js (below the class declaration, but above the export statement) map the state to props and bind the action creators to the dispatcher.

```
const mapStateToProps = (state, props) => {
    return {
        items: state.cart.items,
        products: state.products.products

    }
};

const mapDispatchToProps = (dispatch) => {
    return bindActionCreators(actionCreators, dispatch);
};
```

- ☐ 19. Use the connect method to merge mapStateToProps and mapDispatchToProps into App in the export statement at the bottom of App.js.

```
export default connect(mapStateToProps,
mapDispatchToProps)(App);
```

Step 4: Modify the Container to use the Redux store.

- ☐ 20. In **App.js**, remove the following:
  - The `this.state` assignment statement in the constructor.
  - The binding of `addToCart` and `removeFromCart` in the constructor.
  - The `addToCart` method
  - The `removeFromCart` method
- ☐ 21. Update the componentDidUpdate method to call the loadProducts method instead of setting the state directly.

```
componentDidMount() {
    fetch('http://localhost:3000/data/products.json')
        .then(response => response.json()
            .then(products => this.shuffleArray(products))
            .then(products => {
                this.props.loadProducts(products);
            }))
};
```

- ☐ 22. Modify the `items` assignment statement inside the `render` method to use data from the redux store.

```
let items = this.props.items.map(id =>
this.getProduct(this.props.products, id));
```

- ☐ 23. Modify the props passed to the **ProductList** component to use the action creators and change the products and `inCart` props to use the props that were passed in from **index.js**.

```
<ProductList

     addToCart={this.props.addToCart}
```

```
                removeFromCart={this.props.removeFromCart}
                products={this.sortedProducts}
                inCart={this.props.items}/>
```

Test it out! Everything should now work with no additional changes.

☐ 24. Search Google for the Redux DevTools Chrome browser extension and install it.

☐ 25. Add a third parameter to the createStore function in src/index.js to use the Redux DevTools.

```
let store = createStore(
    rootReducer,
    initialState,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

☐ 26. Close and reopen your Chrome browser and open the app (at localhost:3000). In the Chrome Developer Tools, click on the Redux tab. Add and remove items from the cart to see how the Redux DevTools display the different mutations and the store.

☐ 27. Add a 'Remove' button to the CartItem component that causes the item to be removed from the cart.

Lab 14: Redux Thunk

Redux Thunk middleware allows you to write action creators that return a function rather than an action. This function can be used to delay the dispatch of an action, to cause the action to only be dispatched if a condition is met, or to fetch data asynchronously, for example.

In this lab, you'll use Redux Thunk to post a message to a server and receive a response when a Checkout button is clicked in the Cart component.

☐ 1. Install Redux Thunk.

```
npm install --save redux-thunk
```

☐ 2. Include redux-thunk in **src/index.js.**

```
import ReduxThunk from 'redux-thunk';
```

☐ 3. Add `applyMiddleware` and `compose` to the list of imports from redux in **src/index.js**.

```
import {createStore, combineReducers, applyMiddleware,
compose} from 'redux';
```

☐ 4. Use Redux's compose to create a new version of `createStore` that will apply the `ReduxThunk` middleware to the created store.

```
const createStoreWithMiddleware =
    compose( applyMiddleware(ReduxThunk) )(createStore);

let store = createStoreWithMiddleware(
    rootReducer,
    initialState,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
    window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

☐ 5. Run **npm start**. Your app should run the same as at the end of Lab 13, since you don't have any action creators that return functions yet.

☐ 6. We're going to write an action creator containing a function that will perform an HTTP post using the **axios** library. So, we'll need to install **axios** first.

```
npm install --save axios
```

☐ 7. In **actions/index.js**, import **axios** at the beginning of the file.

```
import axios from 'axios';
```

☐ 8. In **actions/index.js**, add a new action creator for submitting the cart.

```
export function submitCart(data) {
    return dispatch => {
            axios.post('http://localhost:8080/checkout', {
                data
            })
            .then(response => {
                console.log(response.data);
                dispatch(checkOut(response.data));
            })
            .catch(error => dispatch({
                    type: 'FETCH_FAILED', error
                })
            );
    };
}
```

☐ 9. Write the `checkOut` action creator, which will be dispatched when the HTTP post in the Thunked function resolves successfully.

```
export function checkOut(data){
    return {type: 'CHECKOUT', payload: {data}}
}
```

☐ 10. Pass the `submitCart` action creator from **App** to the **Cart.**

```
<Cart removeFromCart={this.props.removeFromCart} submitCart
= {this.props.submitCart} inCart={items}/>
```

☐ 11. Write a method named `handleSubmitCart` in the **Cart** that calls `this.props.submitCart` and passes the value of `this.props.inCart` as an argument.

```
handleSubmitCart(){
    this.props.submitCart(this.props.inCart);
}
```

☐ 12. Add a button to the **Cart** that calls the `handleSubmitCart` method when clicked. Wrap it in a `div` element so that it will appear below the cart items and the total.

```
<div><button onClick={this.handleSubmitCart.bind(this)}>

Check Out

</button></div>
```

☐ 13. Run your app, add some items to the cart, and then open the Redux DevTools and click the Check Out button. You should see that the **FETCH_FAILED** action is dispatched.

☐ 14. Open a new terminal window and change to the **labs/server** directory.

☐ 15. Run the server by entering **npm start**.

☐ 16. Click the Check Out button in the React app.

You should see that the `CHECKOUT` action was dispatched. In the browser console, you should see the return data from the server.

Right now, the React Bookstore doesn't do anything in response to the action, because we don't have a reducer that's listening for it. Let's fix that.

☐ 17. In **reducers/index.js**, write a new case in the cart reducer for the `CHECKOUT` action.

```
case 'CHECKOUT':
        return {
        };
```

☐ 18. Inside the `CHECKOUT` case, we'll return the state, with the items array emptied, which will just empty the cart.

```
case 'CHECKOUT':
    return {
        ...state,
        items: []
    };
```

☐ 19. Make sure that the server is running, then run **npm start** to build your React app and test it out by adding and removing items from the cart and then checking out.

☐ 20. Convert the application to use Redux Saga instead of Thunk.

Lab 15: Persisting data in localStorage using Redux and displaying it in React

Our application is now using React and Redux together. We've implemented an Ajax call to fetch the initial data for our store. But we have an opportunity for improvement. Note that every time you refresh the page, it forgets what was in the cart. What if our user wants to close the browser and then come back at a different time?

In this lab, we'll fix that by writing our cart to localStorage every time it changes. And we'll read the stored cart whenever the client starts up our application.

☐ 1. Create a new reducer case for "READ_CART". It should pull a value from localStorage with a key of "cart". Do something like this:

```
let cart = localStorage.getItem("cart");
```

☐ 2. Since only strings are in localStorage and we need an array, you should JSON.parse() the value.

```
cart = JSON.parse(cart);
```

☐ 3. Then we want to load that array in a state object and return it. Something like this should work:

```
return {
     ...state,
     items: cart || []
};
```

☐ 4. Inside index.js, dispatch READ_CART after the store is created.

```
store.dispatch({ type: "READ_CART" });
```

☐ 5. Run and test. You should have no errors, but you should still see an empty cart.

Why? Because there is nothing in localStorage yet.

Let's write to localStorage now. We'll do it after every change to the cart.

We should write something to local storage after every change to the cart. Since we're using Redux we know that there is only one place that cart can change; in the reducer.

When you edit the reducer, you'll find both cases where cart can change (ADD_TO_CART and REMOVE_FROM_CART).

In the next few steps we will be writing to localStorage.

☐ 6. Change the CART_ADD case. Just before you return the new state, write the cart to localStorage using "setItem()". Of course, the cart array must be JSON.stringified before it can be written. It may look something like this:

```
const newCart = [...state.items, action.payload.productId];
localStorage.setItem("cart", JSON.stringify(newCart));
console.log(newCart);
return {
    ...state,
    items: newCart
};
```

- ☐ 7. Run and test. You'll know you've got it right when you can add one or more books to the cart, then refresh the page and see those same books in your initial cart.
- ☐ 8. Once you can add books and have them saved in localStorage, do the same thing in the CART_REMOVE case.
- ☐ 9. Run and test. Can you now add books and remove books and have them persist each time you re-visit the bookstore? If so, you've got it right!