Professional ReactJS

Completed source code for all labs (for checking your work) can be found at:

`https://github.com/chrisminnick/professional-reactjs`

Version 18.2.8, October 2023
by Chris Minnick

www.watzthis.com

Table of Contents

Disclaimers and Copyright Statement

Disclaimer

WatzThis? takes care to ensure the accuracy and quality of this courseware and related courseware files. We cannot guarantee the accuracy of these materials. The courseware and related files are provided without any warranty whatsoever, including but not limited to implied warranties of merchantability or fitness for a particular purpose. Use of screenshots, product names and icons in the courseware are for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the courseware, nor any affiliation of such entity with WatzThis?.

Third-Party Information

This courseware contains links to third-party web sites that are not under our control and we are not responsible for the content of any linked sites. If you access a third-party web site mentioned in this courseware, then you do so at your own risk. We provide these links only as a convenience, and the inclusion of the link does not imply that we endorse or accept responsibility for the content on those third-party web sites. Information in this courseware may change without notice and does not represent a commitment on the part of the authors and publishers.

Copyright

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, 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, Skillshare, O'Reilly Media, and Pluralsight.

Minnick has authored and co-authored books and articles on a wide range of Internet-related topics including JavaScript, ReactJS, HTML, CSS, mobile apps, e-commerce, Web design, SEO, and security. His published books include JavaScript All-In-One For Dummies, Coding All-In-One For Dummies, ReactJS Foundations, 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.

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).

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/chrisminnick/professional-reactjs`

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

Yarn or npm?

Yarn and npm are both package managers for Node. Yarn was developed by Facebook, and npm is included with Node. You can use whichever one you prefer for all the labs in this course, but the instructions in this course use npm to keep the number of required pre-requisite installations to a minimum.

If you want to try Yarn, or if you prefer it to npm, you must have it installed globally on your computer. You can find instructions for installing Yarn here: https://yarnpkg.com/getting-started/install

Lab 01: Get Started with Create React App

☐ 1. Open your terminal application (**Terminal** on MacOS or **git-bash** on Windows) or the Terminal window in VSCode.

☐ 2. Change to your home directory or your **professional-reactjs** project directory.

```
cd professional-reactjs
```

☐ 3. Use Create React App to make a new React project. This will be the project we'll be working on for most of the labs in this course.

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

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

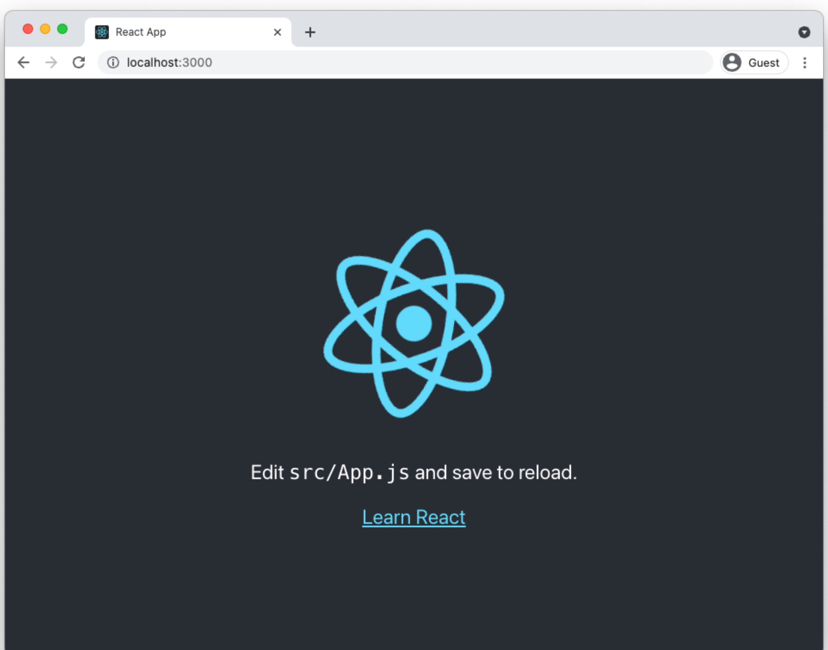
☐ 4. Go into the new directory.

```
cd react-bookstore
```

☐ 5. Test that everything was installed and works.

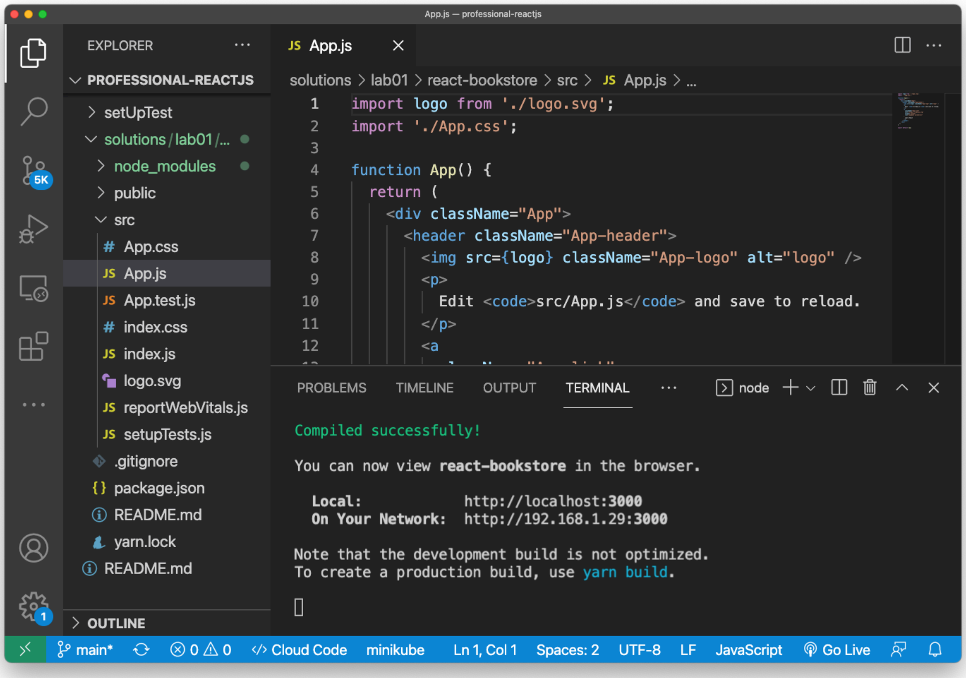
```
npm start
```

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



☐ 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.



☐ 8. Delete the `<img />` element and replace it with an `<h1>` element containing `Welcome to React Bookstore`.

☐ 9. Delete the import of **logo.svg** from the beginning of **App.js** since it's no longer being used.

☐ 10. Move the paragraph inside the `<header>` to a new `<main>` element below the header and change its content to a welcome message, such as the following:

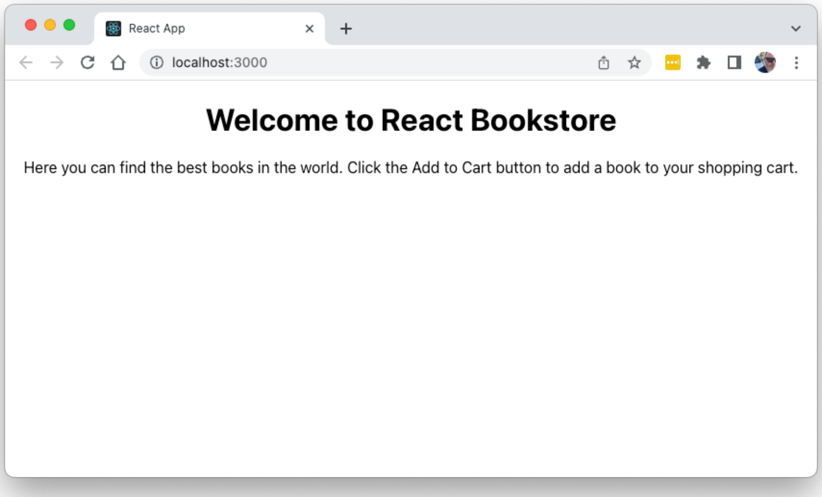
```
We have several books. Feel free to browse for as long as
you like. Click on a cover image to see details, or click
the Add to Cart button to add a book to your shopping cart.
```

☐ 11. Delete the link under the `<p>` element.

☐ 12. Delete the `className="App-header"` attribute from the `<header>` element.

☐ 13. Return to your web browser and notice that the text has been automatically refreshed (if your app is still running).

If it's doesn't refresh, click the browser refresh button, or return to your Terminal emulator and restart the development server (using `npm start`).


React App
localhost:3000
Welcome to React Bookstore
Here you can find the best books in the world. 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. 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 (or Footer.jsx)** in the **src** directory

☐ 2. Type the code below into **Footer.js**

```
function Footer() {
  const footerStyle = {
    backgroundColor: 'black',
    color: 'white',
    padding: '10px',
    position: 'fixed',
    bottom: '0',
    width: '100%',
  };
  return <p style={footerStyle}>This is the footer.</p>;
}

export default Footer;
```

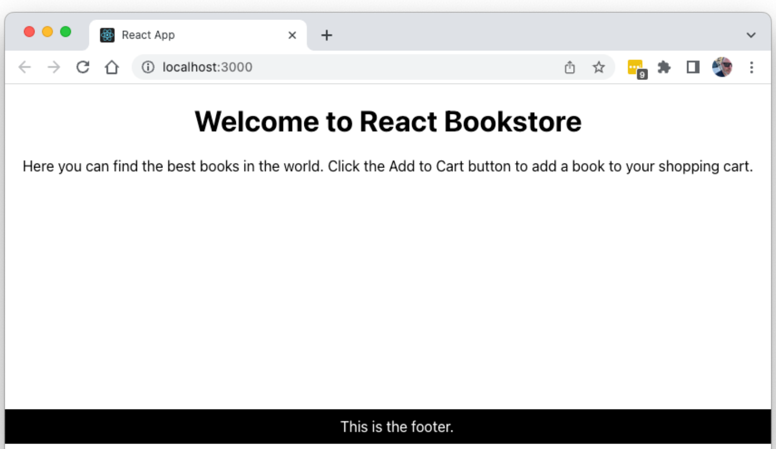
☐ 3. Add the following to the beginning of **App.js**

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

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

```
<Footer />
```

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



Lab 03: Create More Components and Writing Tests

In this lab, you'll make your React application more modular by turning the main parts of the view into components, then you'll create simple tests for your new components.

Part 1: Making new components

☐ 1. Using what you learned from creating **Footer.js**, make **Header.js** and **Main.js** 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 />
    <Main />
    <Footer />
  </div>
);
```

Part 2: Writing tests

Create React App generates a simple test for whether the sample component (**App.js**) renders the text "Learn React". In this lab, you'll make the test for the App component pass and then use the sample test file to create tests for the new components you created in part 1.

☐ 1. Open **src/App.test.js** and modify it so that its test will pass. Visit the React Testing Library documentation to learn more (https://testing-library.com/docs/react-testing-library/intro/). Hint: the test in App.test.js will look at what's rendered by App.js as well as by any components that are used in the return statement of App.js.

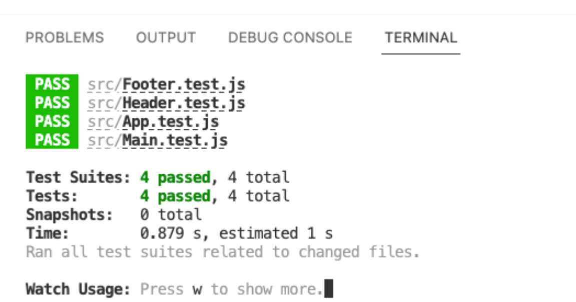
☐ 2. Make copies of **App.test.js** for testing **Footer.js**, **Header.js**, and **Main.js**

☐ 3. Modify the contents of the new files to test the new components.

☐ 4. Run your tests by entering the following in the command line

```
npm test
```

☐ 5. Make sure that all the tests pass.



Lab 04: 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 react-bookstore application and you'll create components to make a mockup of the catalog.

☐ 1. Open **professional-reactjs/starter/lab04/**.

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

☐ 2. Open **data/products.json** in your code editor.

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

☐ 3. Open **starter/lab04/mockup** and look at the **mockup.jpg** image.

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

☐ 4. 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.

☐ 5. Move the **data** directory from the **/starter/lab04** directory into the **src** directory inside your **react-bookstore** project.

☐ 6. Move the **images** directory into the **public** directory.

☐ 7. Open the **Main.js** component in your project and modify it to the following.

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

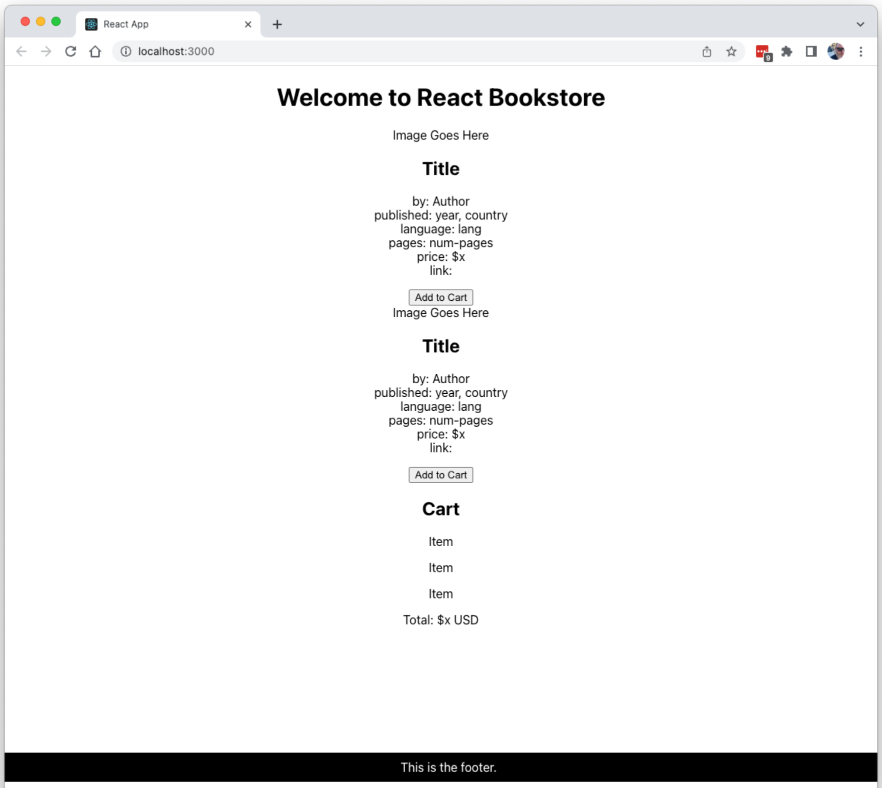
```
import ProductList from './ProductList';
import Cart from './Cart';

function Main() {
    return (
      <main>
            <ProductList />
            <Cart />
      </main>
    );
}

export default Main;
```

☐ 8. Create basic components for `ProductList` and `Cart` and their sub-components. Don't worry about styling them, but try to make each one contain the basic information (without images at this point) as in the mockup.

☐ 9. Run `npm start` to verify that your code builds. Your UI should now look something like this:



Lab 05: Convert to TypeScript and Vite

☐ 1. Open a terminal window and create a new project using Vite:

```
npm create vite@latest
```

☐ 2. When vite asks for a name for the project, use react-bookstore-typescript
☐ 3. Select react as the framework
☐ 4. Select typescript
☐ 5. Install the dependencies

```
cd react-bookstore-typescript
npm install
```

☐ 6. Copy the App, Header, Main, ProductList, Product, Cart, CartItem, and Footer components, plus your tests, from react-bookstore/src to react-bookstore-typescript/src
☐ 7. Delete the App.tsx file that was created by Vite
☐ 8. Rename Main.jsx to MainContainer.tsx and fix the import statement in App.js
☐ 9. Rename your all of your custom components to have the .tsx extensions.
☐ 10. Enter **npm run dev** to start the development server.
☐ 11. Look for files in VS Code that are now red (there should be only one if the above steps have been done correctly). These have Typescript errors. See if you can fix them (by searching Google or looking at the solutions/lab04-typescript folder).

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 **src/main.js.**

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

☐ 3. Add the Bootstrap container class to **App.tsx**.

```
import Header from './Header'
import Main from './Main'
import Footer from './Footer';
import './App.css';

function App() {
  return (
    <div className="container">
        <Header />
        <Main />
        <Footer />
    </div>
  );
}

export default App;
```

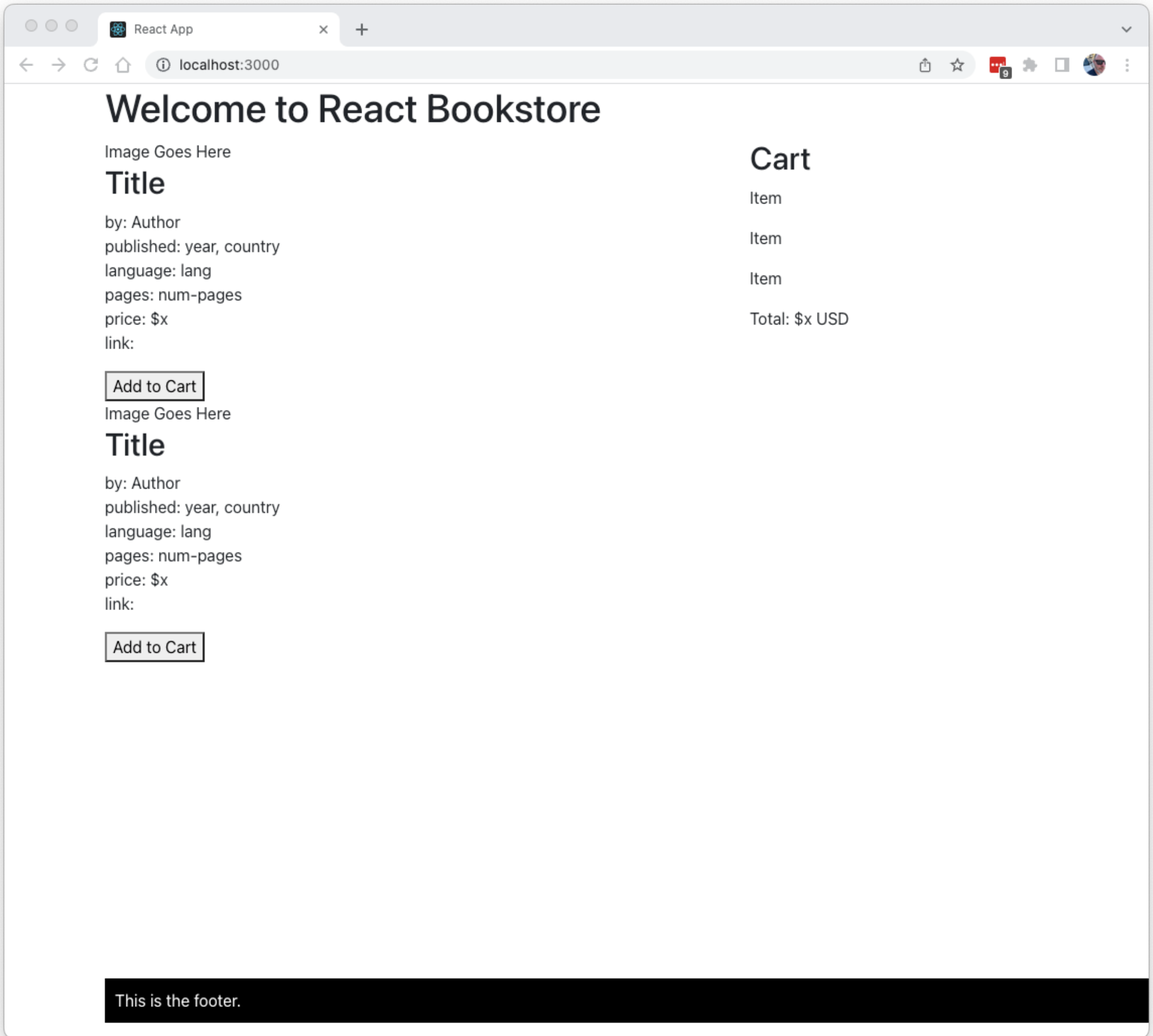
☐ 4. Open **MainContainer.tsx** and create two columns.

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

function Main() {
    return (
        <main className="row">
            <div className="col-md-8">
                <ProductList />
            </div>
            <div className="col-md-4">
                <Cart />
            </div>
        </main>
    );
}

export default Main;
```

☐ 5. Run `npm run dev` to verify that your code builds. Your UI should now look like this:



☐ 6. Modify your `ProductList` component to make each product an item in an unordered list.

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

☐ 7. Create a new file in the **src** directory named **ProductList.module.css**.

This will be our first style module.

☐ 8. Inside **ProductList.module.css** create two styles, `productList` and `productListItem`.

```
.productList {
    padding: 0;
    display: flex;
    flex-wrap: wrap;
```

```
    justify-content: space-between;
    align-items: stretch;
}

.productListItem {
    list-style: none;
    width: 32%;
}
```

These two styles will control the layout of the products inside the product list.

☐ 9. Import the style module into `ProductList.tsx` and give the module the name `styles`.

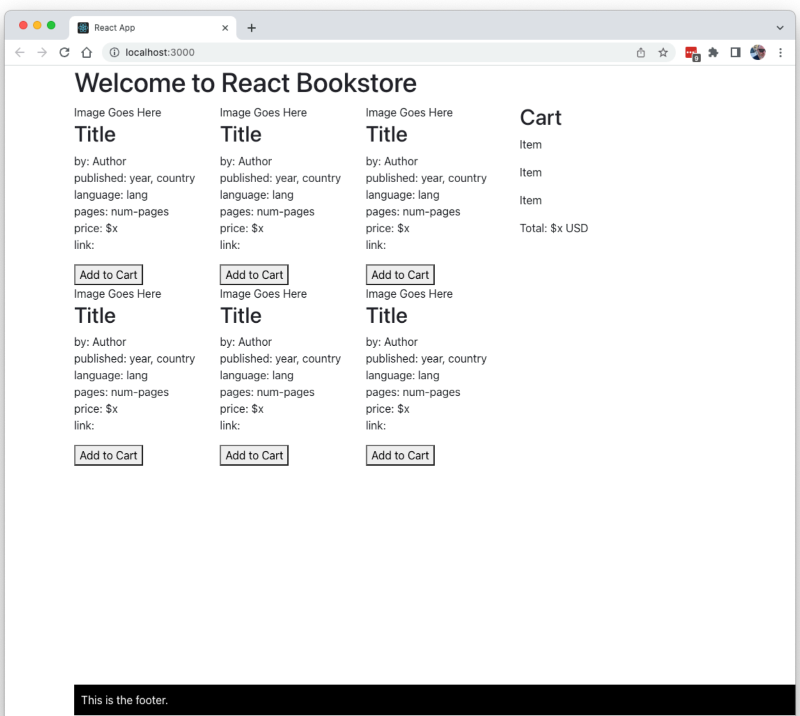
```
import styles from './ProductList.module.css';
```

☐ 10. Attach the styles to the appropriate elements.

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

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

☐ 12. Preview the styled list in your browser.



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

☐ 14. 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 className={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
Header
Footer
Main
ProductList
Product
Cart
CartItem
```

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

- ☐ 1. Create a directory inside **src** named **components**
- ☐ 2. Move **App.tsx**, **Header.tsx**, **Footer.tsx**, **MainContainer.tsx**, **ProductList.tsx**, **Product.tsx**, **Cart.tsx**, and **CartItem.tsx**, along with their test suites and css modules, into the **components** directory.

Note: Make sure to update the import of `App` in **main.ts**

- ☐ 3. Import the data file into **App.tsx.**

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';
```

- ☐ 4. Modify each of your components to accept a props object. For example, here's what the function header of the Main component should look like:

```
function Main(props) {
```

- ☐ 5. Pass `productsData` from the `App` component to the `Main` component as a prop called `products`.

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

- ☐ 6. Pass `products` from `Main` to the `ProductList` component.

```
<ProductList products={props.products} />
```

- ☐ 7. 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 className={styles.productList}>
        {props.products.map(product => (
            <li key={product.id}
                className={styles.productListItem}>
                <Product {...product} />
```

```
                    </li>
                ))}
            </ul>
        );
```

☐ 8. Inside `Product` (outside of the return statement), deconstruct the `props` object into individual constants (to save yourself from having to type '`props.`' repeatedly in the return statement.

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

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

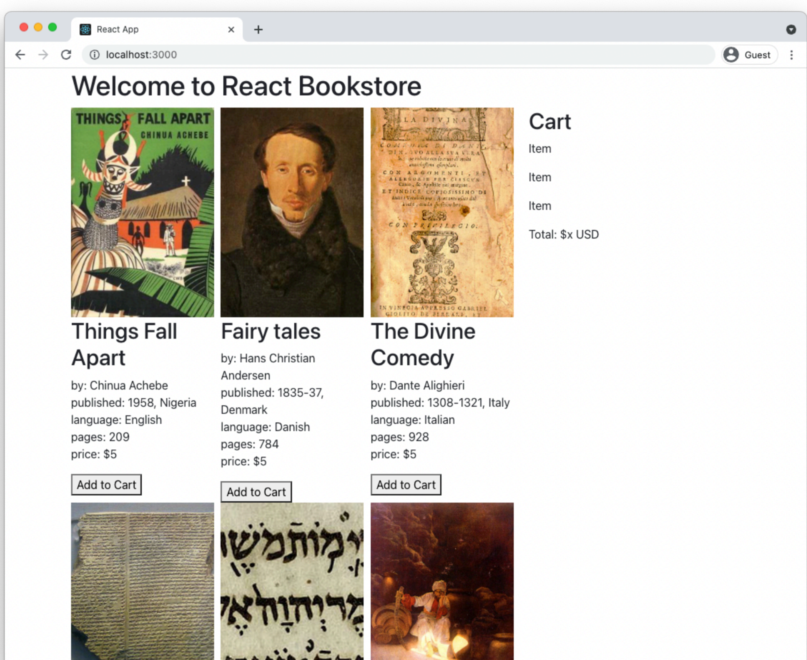
☐ 10. Create a new style rule in **Product.module.css** called thumbnail and set properties to format the book thumbnail images.

```
.thumbnail{
    width: 200px;
    height: 293px;
    object-fit: cover;
}
```

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

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

☐ 12. Run `npm run dev`



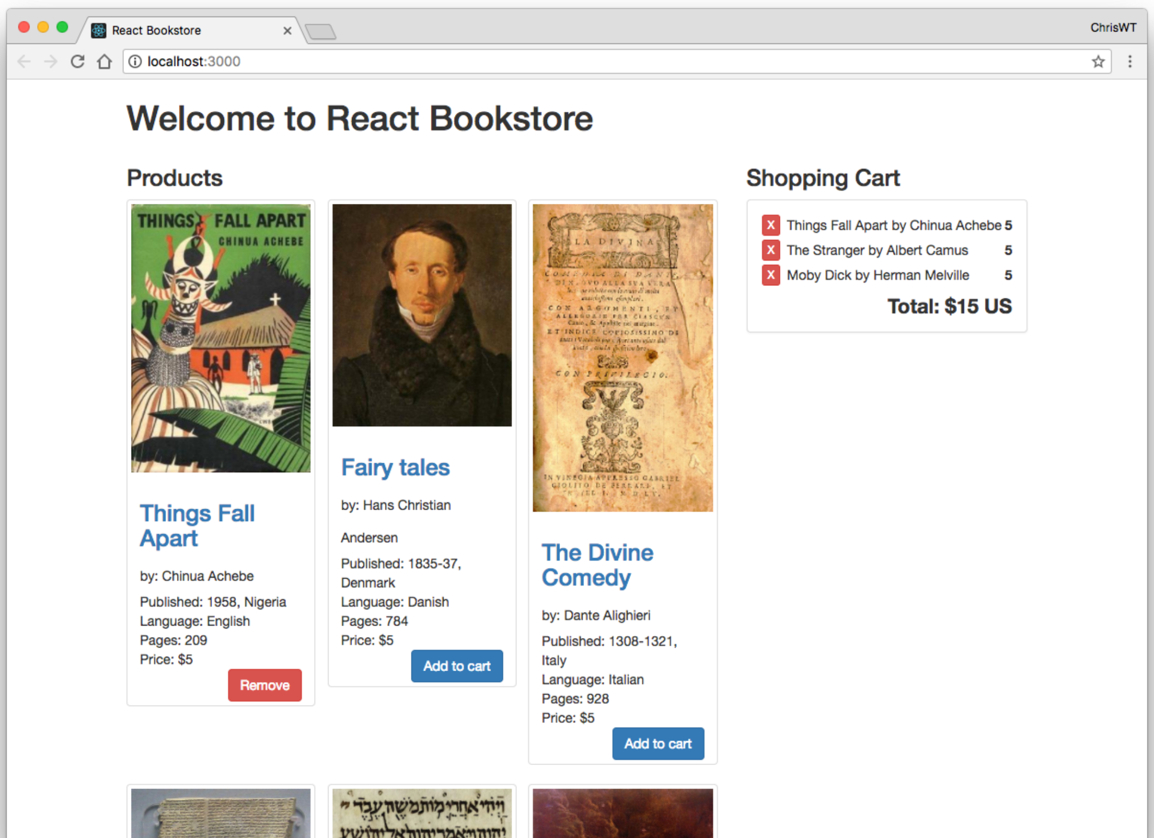
Lab 08: Adding State

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?



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

In this React Bookstore application, the only thing 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`. We have two components that fit this description, `App` and `Main`.

If you expect that the `Header` and `Footer` components may need access to the list of items in the cart at some point, it might be wise to keep this state in App. Also, it's a good practice in React to keep the number of stateful components to a minimum. It's mostly a judgement call at this point, but we'll put the state in `App`.

Follow these steps to add state to the application.

☐ 1. Open **src/components/App.tsx** in your code editor.

☐ 2. Import `React.useState`

```
import {useState} from 'react';
```

☐ 3. Create a state variable and setter function for `itemsInCart` and initialize it as an empty array.

```
const [itemsInCart,setItemsInCart] = useState([]);
```

☐ 4. Pass `itemsInCart` into the `MainContainer` component as props.

```
<MainContainer products={productsData}
     itemsInCart={itemsInCart} />
```

☐ 5. Add some default product ids to the `itemsInCart` array, for testing.

```
const [itemsInCart,setItemsInCart] =
useState(["1","2","3"]};
```

☐ 6. Pass `itemsInCart` from MainContainer to `ProductList`.

☐ 7. 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.

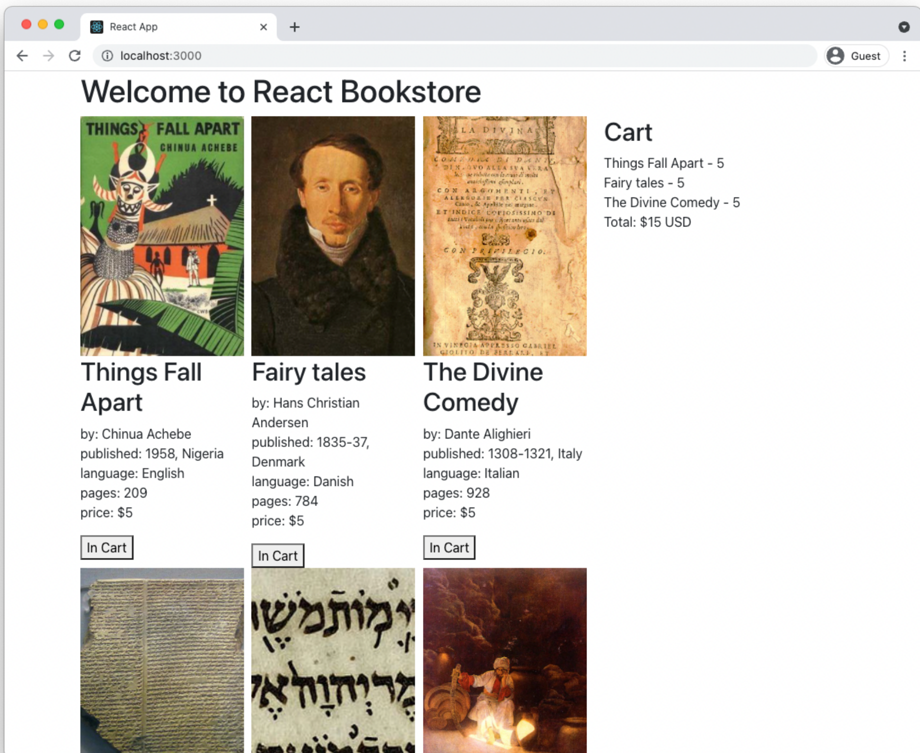
If you get stuck, look at the solution inside **intro-to-react/solutions/lab08**.

☐ 8. Inside the `MainContainer` component, figure out how to use the `itemsInCart` array to generate a list of products called `cartItems` to pass to the `Cart` component.

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 `itemsInCart`. 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);
}
```

- ☐ 9. Render the list of `cartItems` inside the `Cart` component.
- ☐ 10. Modify the `CartItem` component to display the name and price of the item.
- ☐ 11. 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.

React's Synthetic Events

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.

Unidirectional Data Binding

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 passing functions from parent components to child components and you'll learn how to call functions to update the state of a React view.

State in a Class Component

Class components are components created by extending the `React.Component` class. Although almost everything in React can be done using function components, understanding class components will give you a deeper understanding of how React works, and it will also enable you to use the features and techniques in React that aren't accessible using function components.

To create a class component, import React into your module and then extend React's `Component` class.

```
import React from 'react';

class MyComponent extends React.Component {

...

}
```

A JavaScript class can have a constructor method, which will only run once during the lifecycle of the component. The constructor is used to initialize the state object and to bind functions to the class. The constructor method is optional, but if you do use it, you must call the `super()` method as the first thing inside constructor. The `super()` method calls the constructor of the parent class. You should also pass the props object to `super()`.

```
import React from 'react';

class MyComponent extends React.Component {

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

...

}

export default MyComponent;
```

The rest of a component may contain any number of methods, but one method, `render()` must be present. The `render()` method of a class component is essentially the same as a function component (except that it can't access hooks). The render method has a return statement that uses JSX to define the part of the user interface the component is responsible for.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      ...
    }
  }

  render(){
    return (<h1>Welcome to my component.</h1>);
  }
}

export default MyComponent;
```

The state object, which can be initialized in the constructor of a class component, holds the stateful properties of a component. When these properties change, React re-renders the component. The reason React knows to re-render the component when the state object changes, is because the developer only changes the state object using React's `setState()` method.

So, 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 wante. 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 class components, 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. Convert **App.tsx** into a class component, following these steps:
   - Import `Component` from the React library
   - Change the function header to a class header.
   - Create a `constructor`
   - Call the `super()` method
   - Initialize the `this.state` object in the constructor, with one property, `itemsInCart`
   - Delete the call to `useState()`, along with the import of `useState()`
   - Create a `render` method and copy the existing `return` statement from the function component.
   - In the `return` statement, reference `itemsInCart` using `this.state.itemsInCart`
   - Remove the `setItemsInCart` attribute from the `<Main />` element.

   Your App component should now match the following:

```
import {Component} from 'react';
import Header from './Header'
import MainContainer from './MainContainer'
import Footer from './Footer';
import './App.css';
import productsData from '../data/products';

class App extends Component {
  constructor(props){
```

```
    super(props);
    this.state = {
      itemsInCart: ["1","2","3"]
    }
  }

  render(){
    return (
      <div className="container">
        <Header />
        <MainContainer products = {productsData}
              itemsInCart = {this.state.itemsInCart}
              />
        <Footer />
      </div>
    );
  }
}

export default App;
```

☐ 2. Create the following method inside the App component.

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

☐ 3. 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);
```

☐ 4. Pass the bound `addToCart` function down to the `Main` component as a prop.

```
<Main  products = {productsData}
       itemsInCart = {this.state.itemsInCart}
       addToCart = {this.addToCart}
/>
```

☐ 5. Open the Main component and pass the `addToCart` function to the `ProductList` component as a prop.

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

☐ 7. 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`.

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

☐ 8. Call the `handleClick` function as the event handler for the click event on the button.

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

☐ 9. Run **`npm run dev`** and test out your application.

☐ 10. 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.itemsInCart.filter(
                        id => id !== idToRemove);
      this.setState({itemsInCart: newItems});
}
```

Lab 10: Component Lifecycle and AJAX

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 (or, better, according to some algorithm, such as which books the user is most likely to buy or a user-chosen filter) each time a visitor comes to the store?

You could change the order of the items 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:

```
function 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. Create a new array by sorting the one passed as a prop.

```
let sortedProducts = shuffleArray(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 use React's `componentDidMount()` lifecycle method to only sort the products once, after the component has mounted. To do this, follow these steps:

☐ 5. Create a method named `componentDidMount()` in the `App` component.

☐ 6. Copy the `shuffleArray` method from `ProductList` and paste it into the `App` component (modifying it to use method notation if your App component is a class component).

☐ 7. Revert the `ProductList` component to how it was before you made the previous changes.

☐ 8. In `App`'s `componentDidMount` method, call `this.shuffleArray()` and pass in the `productsData`.

```
this.shuffleArray(productsData);
```

☐ 9. Run `npm start` and notice what happens when you click one of the `AddToCart` buttons. Can you explain why this happens?

What's happening is that the `productsData` is loading before the `App` component is mounted. So, the initial render of the component uses the default sorting of the array, but the `componentDidMount` method is shuffling the array after the component finishes mounting. The result is that the next re-render of the component will causes it to display the products in their shuffled order.

To fix this, we can load the `productsData` in the `componentDidMount` lifecycle method. We'll do so using the `fetch` method.

☐ 10. Add a property named `products` with a default value of an empty array (`[]`) to the `state` object.

☐ 11. Add a property named `loading` with a default value of `false` to the `state` object (in the `constructor`).

☐ 12. Remove the import of the product data.

☐ 13. Pass `this.state.products` to the `Main` component instead of `productsData`.

☐ 14. Copy `products.json` from `starter/lab10/data` to `public/data/` in your project

☐ 15. Delete the `src/data` folder in your project.

☐ 16. Confirm that the json file is in the right place by visiting the following url in your browser: **http://localhost:3000/data/products.json**

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

```
componentDidMount() {
    this.setState({loading:true});

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

☐ 18. Run `npm run dev`

Lab 11: Converting App to a Function Component

In this lab, we'll use the `useEffect` and `useState` hooks to convert `App` back to a function component.

- ☐ 1. Open `App.tsx` in your code editor and modify the import from 'react' to import `useState` and `useEffect` (instead of `Component`).
- ☐ 2. Change the class header to a function header. Since this is the root component, the function doesn't need to take `props` as a parameter.
- ☐ 3. Delete the `constructor` method.
- ☐ 4. Use `useState` to initialize the three state variables.

```
const [itemsInCart, setItemsInCart] = useState([]);
const [products, setProducts] = useState([]);
const [isLoading, setLoading] = useState(false);
```

- ☐ 5. Rewrite the `componentDidMount` method using `useEffect` and `async` functions.

```
useEffect(() => {
    async function fetchData() {
        try {
            const response = await
fetch('http://localhost:3000/data/products.json');
            const json = await response.json();
            setProducts(json);
        } catch (e) {
            console.error(e);
        }
    };
    fetchData();
  }, [setProducts]);
```

- ☐ 6. Shuffle the product data array before the products state array is updated.
- ☐ 7. Convert the functions written using method syntax to use the function keyword or arrow functions.
- ☐ 8. Change `setState` in the `addToCart` and `removeFromCart` functions to use `setItemsInCart`, and change the references to `this.state` to refer to the stateful `itemsInCart` variable.
- ☐ 9. Take the `return` statement out of the `render` method and delete the `render` method.
- ☐ 10. Update references to `this` in the `return` statement.
- ☐ 11. Use the `isLoading` state variable to conditionally display a loading message or loader animation when data is loading.
- ☐ 12. Start the app and fix any errors that occur.

Lab 14: Testing with React Testing Library

In this lab, you'll use Jest and React Testing Library to write more tests for your components.

To use Jest and React Testing Library, you'll first need to install them. Follow the instructions here:

`https://dev.to/hannahadora/jest-testing-with-vite-and-react-typescript-4bap`

☐ 1. Add a new script to the script object in package.json:

```
"test": "jest --coverage --watchAll --no-cache",
```

☐ 2. Run `npm test` in the root of your project. You probably already have some basic tests that you created in previous labs. However, because of changes to the components you're testing, your tests may not all pass.

☐ 3. Update your existing tests so they all pass. You may need to use different methods from React Testing Library. Reference the testing library cheatsheet at the following URL to find additional queries to use:

https://testing-library.com/docs/react-testing-library/cheatsheet

☐ 4. Write more tests, refine your existing tests, and increase your test coverage %.

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

Lab 15: 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.

Redux Toolkit simplifies many of the steps involved in implementing and using Redux, but in this lab, we're going to do it the hard way first.

☐ 1. Install Redux and the react-redux library.

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

Step 1: Create a store

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

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

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

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

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

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

☐ 5. Create the root reducer.

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

☐ 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.

```
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Step 2: Write the reducers

☐ 9. Create a directory in **src** named **reducers**.

☐ 10. Create **index.ts** 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 `productId` 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.tsx**) 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 (also in **reducers/index.ts**). 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.tsx**.

```
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.ts** 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 **components/App.tsx**.

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

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

```
const mapStateToProps = (state, props) => {
    return {
        itemsInCart: 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.tsx**.

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

Step 4: Modify App.js to use the Redux store.

☐ 20. In **App.tsx**, remove the following:
- ☐ The `useState` function calls for `itemsInCart` and for `products`.
- ☐ The `addToCart` method
- ☐ The `removeFromCart` method

☐ 21. Make sure the `props` object is a parameter of `App`

```
function App(props) {
```

☐ 22. Update the `useEffect` method to call the `loadProducts` method instead of setting the state directly.

```
useEffect(() => {
    async function fetchData() {
        try {
            setIsLoading(true);
            const response = await
fetch('http://localhost:3000/data/products.json');
            const json = await response.json();
            props.loadProducts(json)
            setIsLoading(false);
        } catch (e) {
            console.error(e);
        }
    };
    fetchData();
  }, [props.loadProducts]);
```

☐ 23. Update the `useEffect` callback that shuffles the products so that it accepts the list of products and returns a shuffled list, rather than shuffling the products array directly. OR, call `shuffleArray` from within the same `useEffect` callback that loads the products.

☐ 24. Modify the props passed to the `Main` component to use the action creators and change the `inCart` prop to use the prop that was passed in from **index.js**.

```
<Main products = {props.products}
            itemsInCart = {props.itemsInCart}
            addToCart = {props.addToCart}
            removeFromCart = {props.removeFromCart}
            />
```

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

- ☐ 25. Look at the terminal. You'll see a warning message about a missing dependency. Our app works fine as it is, but can you figure out how to make that warning message go away?
- ☐ 26. Add a 'Remove' button to the `CartItem` component that causes the item to be removed from the cart.

Part 2: Redux Toolkit

- ☐ 1. Install Redux Toolkit

```
npm install @reduxjs/toolkit
```

- ☐ 2. Remove the import of createStore() from 'redux' in main.ts
- ☐ 3. Import configureStore from Redux Toolkit

```
import { configureStore } from '@reduxjs/toolkit';
```

- ☐ 4. Replace the creation of the store variable in index.js with the Redux Toolkit configureStore() function call:

```
const store = configureStore({ reducer: rootReducer });
```

- ☐ 5. Delete the initialState variable from index.js
- ☐ 6. Define the initial state in reducers/index.js using default parameters, like this:

```
export function cart(state = { items: [] }, action = {}) {
  ...
}
```

and

```
export function products(state = { products: [] }, action =
{}) {
  ...
}
```

Check whether everything is still working correctly. Now that you're using Redux Toolkit, take a look at its other capabilities for making working with Redux easier by going to: `https://redux-toolkit.js.org/introduction/getting-started`

Lab 16: Redux Thunk

Redux Thunk middleware allows you to write action creators that return functions rather than actions. 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.

We're going to write an action creator containing a function that will perform an HTTP post. We'll be using the built-in fetch() method to do the API request, and we'll use Redux Thunk with Redux Toolkit's createAsyncThunk function to make the request prior to running a reducer.

☐ 1. 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
```

☐ 2. In **actions/index.ts**, import `axios` at the beginning of the file.

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

☐ 3. Import createAsyncThunk into actions/index.ts:

```
import { createAsyncThunk } from '@reduxjs/toolkit';
```

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

```
export const submitCart = createAsyncThunk('CHECKOUT', async (data) => {
  const res = await axios.post('http://localhost:8080/checkout', data);
  return res.data;
});
```

☐ 5. 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}}
}
```

☐ 6. Pass the `submitCart` action creator from `App` to `Main`, and from `Main` to the `Cart`.

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

☐ 7. Add a button to the `Cart` that calls the `submitCart` method when clicked and passes `props.cartItems` into it. Wrap it in a `div` element so that it will appear below the cart items and the total.

```
<div><button
onClick={()=>props.submitCart(props.cartItems)}>
Check Out
</button></div>
```

☐ 8. 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 `CHECKOUT/rejected` action is dispatched.

☐ 9. Open a new terminal window and change to the **starter/lab16/server** directory.

☐ 10. Run `npm install` in the server directory

☐ 11. Run the server by entering `npm run dev`.

☐ 12. Click the **Check Out** button in the React app.

You should see that the `CHECKOUT/fulfilled` 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.

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

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

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

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

☐ 15. 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.

Lab 17: Persisting data in localStorage using Redux

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. Create an action creator named `readCart` that creates the `READ_CART` action.

☐ 5. After the products are loaded (in the fetchData() function in App.js), call readCart() to dispatch the READ_CART action.

☐ 6. 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.

☐ 7. 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
        };
```

☐ 8. 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.

☐ 9. Once you can add books and have them saved in localStorage, do the same thing in the `CART_REMOVE` case.

☐ 10. 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!

Lab 18: React Router

In this lab, you'll use React Router to create a separate route for the shopping cart.

☐ 1. Install `react-router-dom`

☐ 2. Import `BrowserRouter` as `Router` into **main.ts** and wrap the `Router` component around the `<Provider>` element in `ReactDOM.render`

☐ 3. In **MainContainer.tsx**, import `Routes` and `Route` from `react-router-dom`.

☐ 4. In the Main's return statement, change the page layout to a 1-column layout by removing the `</div>` and `<div>` from between `Cart` and `ProductList` and changing the `className` passed to the outside `div` to `col-md-12`.

☐ 5. Replace `ProductList` and `Cart` with a `Routes` component containing two Routes. The first should render `ProductList` when the path is exactly `'/'` and the second should render `Cart` when the path is `'/cart'`.

```
<Routes>
  <Route
    path="/"
    element={
      <ProductList
        products={props.products}
        itemsInCart={props.itemsInCart}
        addToCart={props.addToCart}
        removeFromCart={props.removeFromCart}/>
    }/>
  <Route
    path="/cart"
    element={
      <Cart
        cartItems={cartItems}
        removeFromCart={props.removeFromCart}
        submitCart={props.submitCart}
      />
    }/>
</Routes>
```

☐ 6. Test it out. When you first start up the app (and the route is '/') it should display the `ProductList`, and if you change the url in the address bar to `'/cart'` it should display the cart. Everything should still work

☐ 7. Challenge: Make a Shopping Cart button component that displays the number of items in the cart in the header and that links to the shopping cart (using `react-router-dom`'s `Link` component). You can use the fontawesome React component to render the icon: **`https://fontawesome.com/v5.15/how-to-use/on-the-web/using-with/react`**

Lab 19: Microfrontends with Single SPA

In this lab, you'll use the Single SPA framework to create a microfrontend.

☐ 1. In an empty directory that's not inside of any other Node project (no **package.json** at a higher level), open a new terminal window and invoke `create-single-spa`.

```
npx create-single-spa --moduleType root-config
```

☐ 2. Answer all the questions that `create-single-spa` asks, choosing the defaults whenever possible.

☐ 3. Run **npm start** in your new project and open a browser to http://localhost:9000.

You now have a root config and an example application. You'll see some instructions for what to do next in the sample application that's running at port 9000. Read through those instructions. We're going to use Single SPA to run two React applications and share dependencies between them.

☐ 4. Open a new terminal window and generate a single-spa application by running:

```
npx create-single-spa --moduleType app-parcel
```

☐ 5. When you're asked for a directory and a name for the application, name the directory something creative like 'app1' and the app 'my-first-app.'

☐ 6. Once it finishes, cd to your new directory and run **npm start**.

☐ 7. Open a browser and go to the localhost post that it gives you when it starts up. (probably localhost:8001).

☐ 8. Read through this page, but don't follow these instructions just yet.

☐ 9. Open **src/index.ejs** in your root config (not in your app1 subdirectory) and find the script element with `type="systemjs-importmap"`. Since all of our microfrontends will use React, we need to add React and ReactDOM to this import map.

☐ 10. Go to https://cdnjs.com/libraries/react and get the latest link for the React library (it should have **umd** in the URL) and add it to the **importmap**, then do the same for the ReactDOM library (you can just copy the same url and change "react" to "react-dom" in the URL.

☐ 11. When you're finished, your importmap should look like this:

```
    <script type="systemjs-importmap">
      {
        "imports": {
          "single-spa": "https://cdn.jsdelivr.net/npm/single-
spa@5.9.0/lib/system/single-spa.min.js",
          "react":
"https://cdnjs.cloudflare.com/ajax/libs/react/18.2.0/umd/react.production.min.js",
          "react-dom": "https://cdnjs.cloudflare.com/ajax/libs/react-
dom/18.2.0/umd/react-dom.production.min.js"
        }
      }
    </script>
```

☐ 12. Open **src/microfrontend-layout.html** and find the `<route>` element. Add your new application as a 2nd application. You can get the value for the name property from the **package.json** file in your **app1** directory. For example:

```
<application name="app1/@minnick/my-first-
app"></application>
```

☐ 13. In your root config's **src** directory, open **index.ejs** and add your application to the `importmap`. Note that there are two importmaps in the file, and you should add your application to both. Here's an example of what you should add:

```
"@minnick/my-first-project": "//localhost:8081/minnick-my-first-app.js",
```

☐ 14. Stop both your root config and you application and restart them. In your browser, you should now see a message saying that your application is mounted. It will look like this:

```
@minnick/my-first-app is mounted!
```

☐ 15. Remove the sample application from your **src/index.ejs** so your new application is the only one being rendered.

☐ 16. Create a second application and render that one in addition to the first.

Bonus Lab: Authentication with JWT

In this lab, you'll learn how to implement authentication in the container component and then pass an authentication token to micro frontends.

☐ 1. Read the following article to learn about implementing JWT in React

```
https://www.alibabacloud.com/blog/how-to-implement-authentication-in-reactjs-using-jwt_595820
```

☐ 2. Use this technique, or another of your choosing, to implement authentication and create a protected "Account Info" area in the bookstore app.

Bonus Lab: React Asteroids

Convert the JavaScript application you built in Lab 4 to a React application.