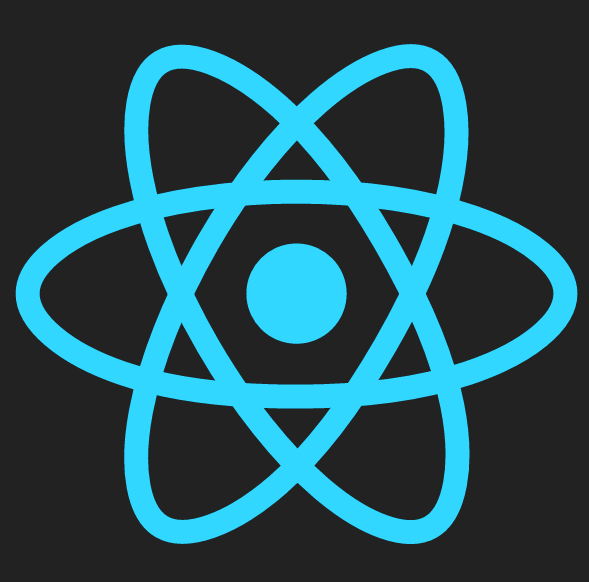
Professional ReactJS



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

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



Version 17.9, April 2022

by Chris Minnick

Copyright 2022, WatzThis?

www.watzthis.com

# Table of Contents

Table of Contents 2

Disclaimers and Copyright Statement 4

Disclaimer 4

Third-Party Information 4

Copyright 4

Help us improve our courseware 4

Credits 5

About the Author 5

Setup Instructions 6

Course Requirements 6

Classroom Setup 6

Testing the Setup 6

Introduction and Git Repo Info 8

Yarn or npm? 8

Lab 0: Using the UMD Build 9

Lab 01: Get Started with Create React App 10

Lab 02: Your First Component 13

Lab 03: Create More Components 14

Lab 04: Testing React 15

Lab 05: Static Version 16

Lab 06: Styling React 18

Lab 07: Props and Containers 22

Lab 08: Adding State 24

Lab 09: Interactions, Events, and Callbacks 27

React's Synthetic Events 27

Unidirectional Data Binding 27

State in a Class Component 27

Lab 10: Component Lifecycle and AJAX 33

Lab 11: Converting App to a Function Component 35

Lab 12: PropTypes and defaultProps 36

Lab 13: Converting to TypeScript 38

Lab 14: Testing with Jest and Enzyme 40

Lab 15: Implementing Redux 41

Step 1: Create a store 41

Step 2: Write the reducers 41

Step 3: Write the Actions and Action Creators 43

Step 4: Modify App.js to use the Redux store. 44

Lab 16: Redux Thunk 46

Lab 17: Redux Saga 49

Lab 18: Persisting data in localStorage using Redux 51

Lab 19: Refactoring and Organizing 53

Lab 20: React Router 55

Lab 21: Creating Micro Frontends 56

Part 1: Create the Content Server 56

Part 2: Create the container 56

Part 3: Making a Micro Frontend 59

Lab 22: Getting the Bookstore Working as a Micro Frontend 62

Bonus Lab: Sharing State Between Micro Frontends 63

Bonus Lab: Authentication with JWT 64

Bonus Lab - SwimCalc 65

The Story 65

Getting Started 65

# 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

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior expressed permission of the owners, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the owners, at info@watzthis.com.

## 

## 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 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.

# 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 and then install it.

1. Install a code editor.

We recommend the free Visual Studio Code editor, which can be downloaded from [https://code.visualstudio.com](https://code.visualstudio.com/)

1. Make sure Google Chrome is installed.
2. 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/chrisminnick/professional-reactjs

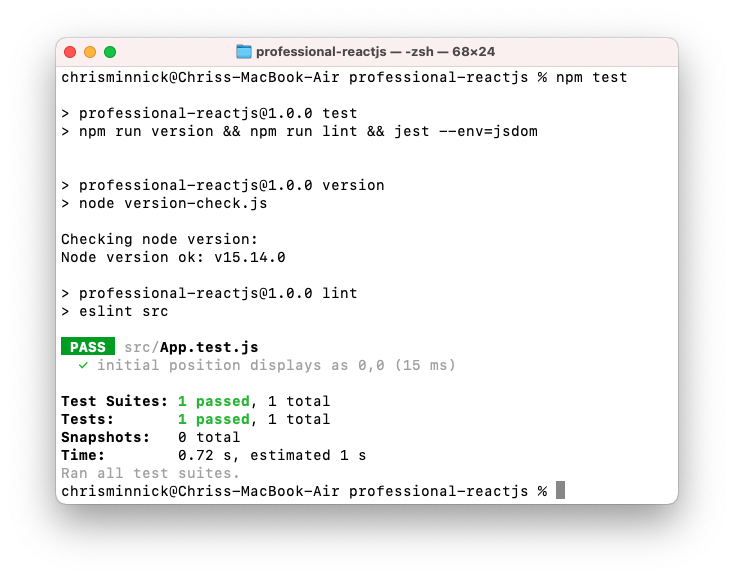
The lab solution files for the course will download into a new directory called professional-reactjs

1. Enter cd professional-reactjs to switch to the new directory.
2. Enter cd setUpTest
3. 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

1. When everything is done, enter npm run test
2. 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.
3. A series of things will happen and then a message will appear and tell you that the test passed.



# 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 0: Using the UMD Build

If you want to add React to an existing web page or application, you can do so by just including the UMD (universal module definition) versions of React and ReactDOm in an HTML page. Follow these steps to get started.

1. Get the CDN links from https://reactjs.org/docs/cdn-links.html
2. Paste the React and ReactDOM script tags before </body>
3. Create a container element.

<div id="root"></div>

1. Make a new <script> element underneath the imports of React and ReactDOM, with the following code.

class HelloWorld extends React.Component {

constructor(props) {

super(props);

this.state = { personName: 'World' };

}

render() {

return React.createElement('h1', null, 'Hello, ' + this.state.personName);

}

}

1. After the HelloWorld class, use ReactDOM.render to render the class to the browser, inside the container element:

ReactDOM.render(React.createElement(HelloWorld),

document.getElementById('root'));

1. Open the HTML page in a browser.
2. Try changing the value of this.state.personName and then see the change reflected in the browser.

# 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

1. 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).

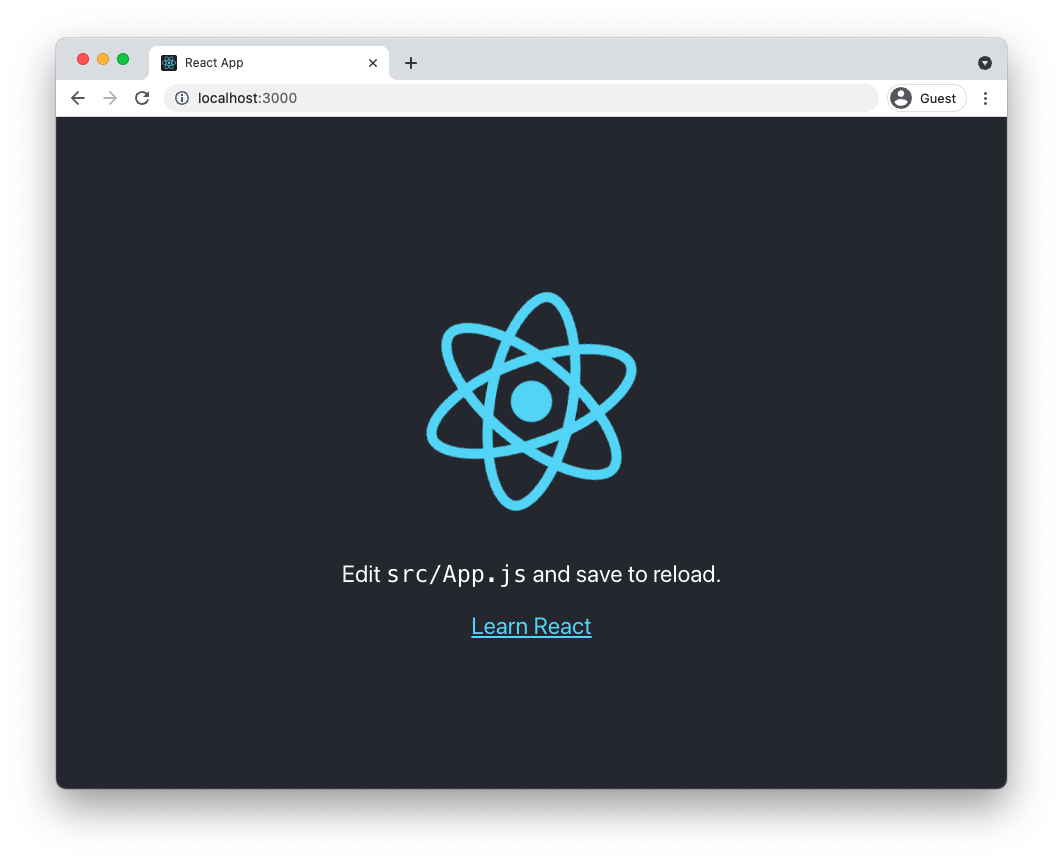
1. Go into the new directory.

cd react-bookstore

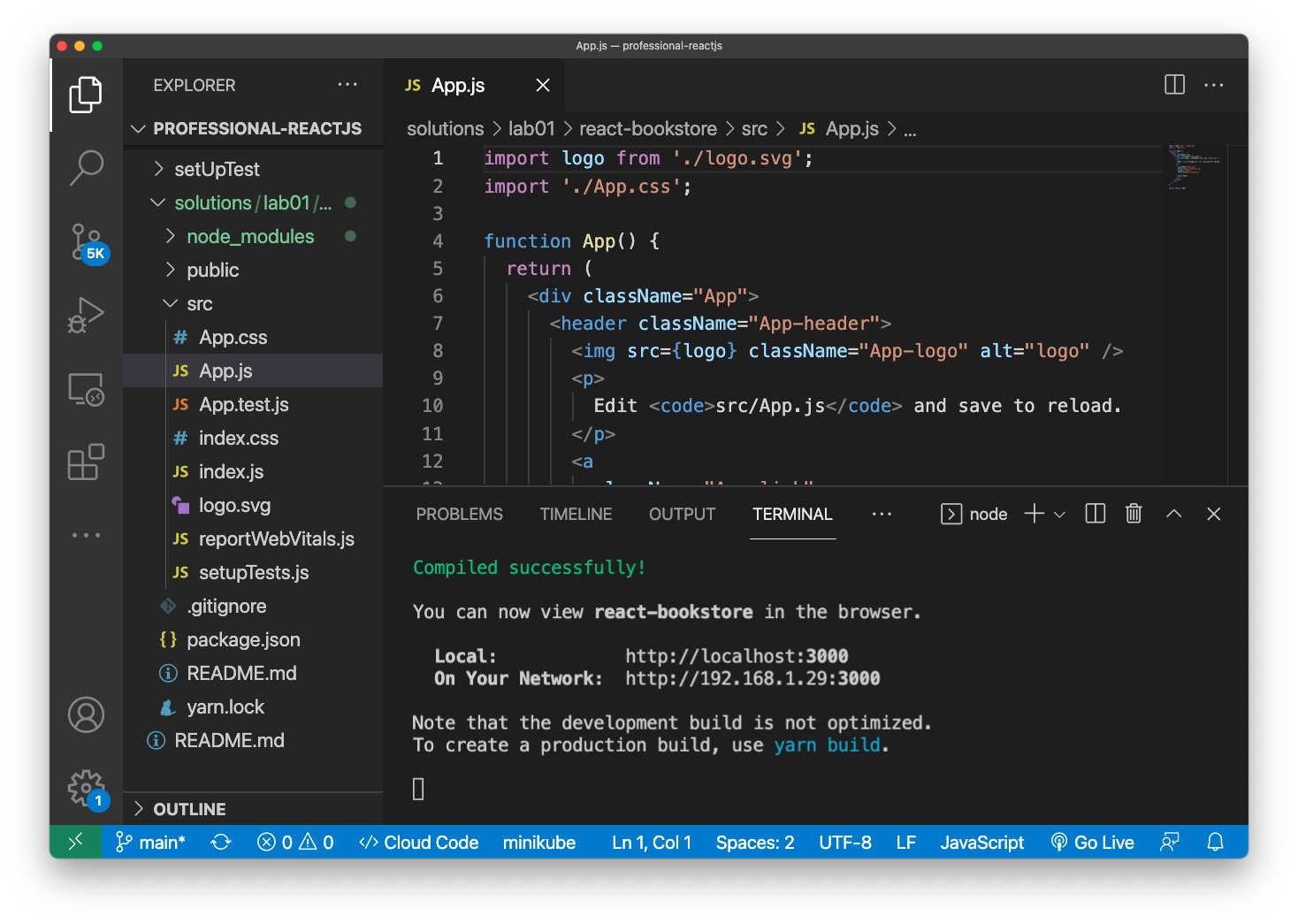
1. Test that everything was installed and works.

npm start

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



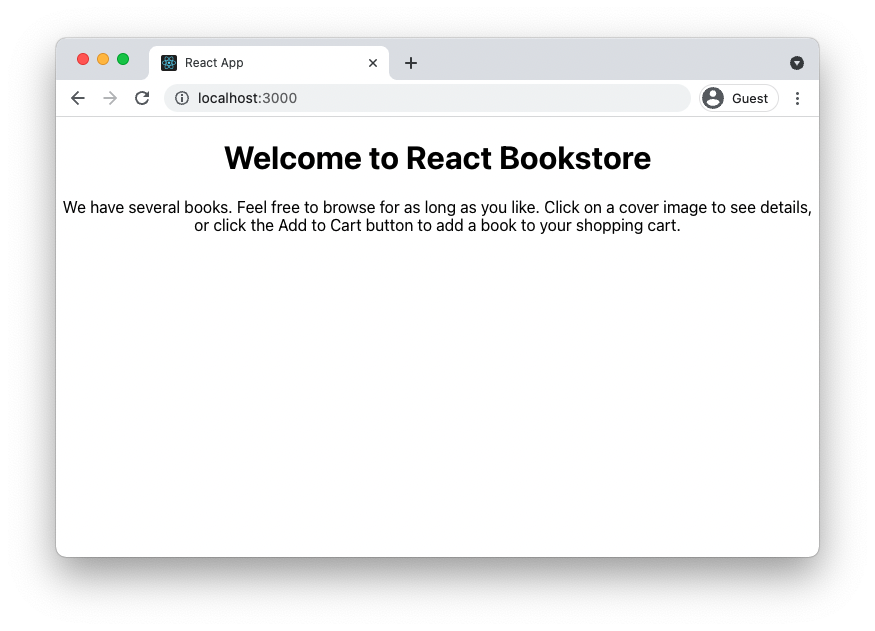
1. Open your new project in the code editor of your choice.
2. Find **App.js** inside the **src** directory and open it for editing.



1. Delete the <img /> element and replace it with an <h1> element containing Welcome to React Bookstore.
2. Delete the import of **logo.svg** from the beginning of **App.js** since it's no longer being used.
3. 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.

1. Delete the link under the <p> element.
2. Delete the className="App-header" attribute from the <header> element.
3. 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).

# 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** in the **src** directory
2. Type the code below into **Footer.js**

function Footer(){

return (

<footer>

<p>This is the footer.</p>

</footer>

);

}

export default Footer;

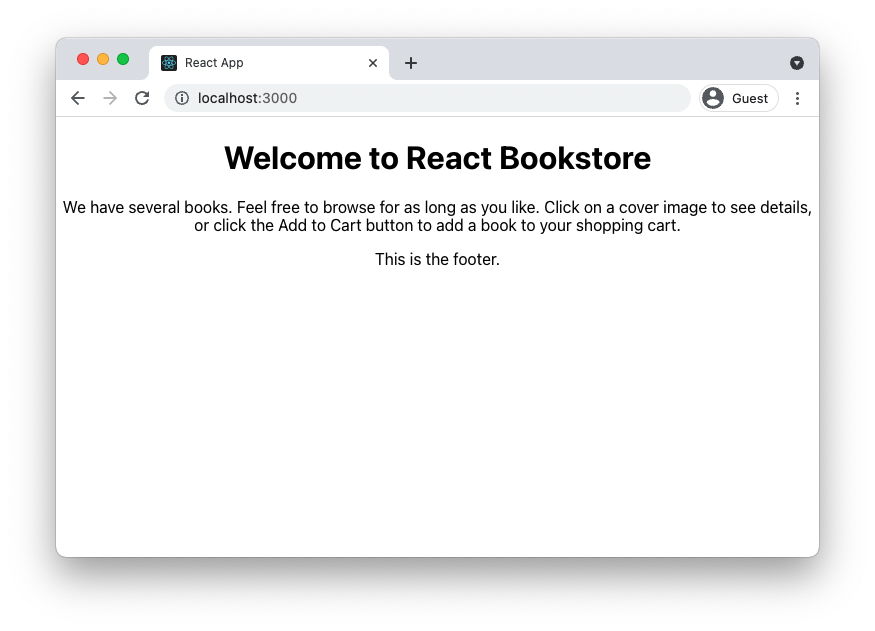
1. Add the following to the beginning of **App.js**

import Footer from './Footer.js';

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

<Footer />

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



# 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. 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>

);

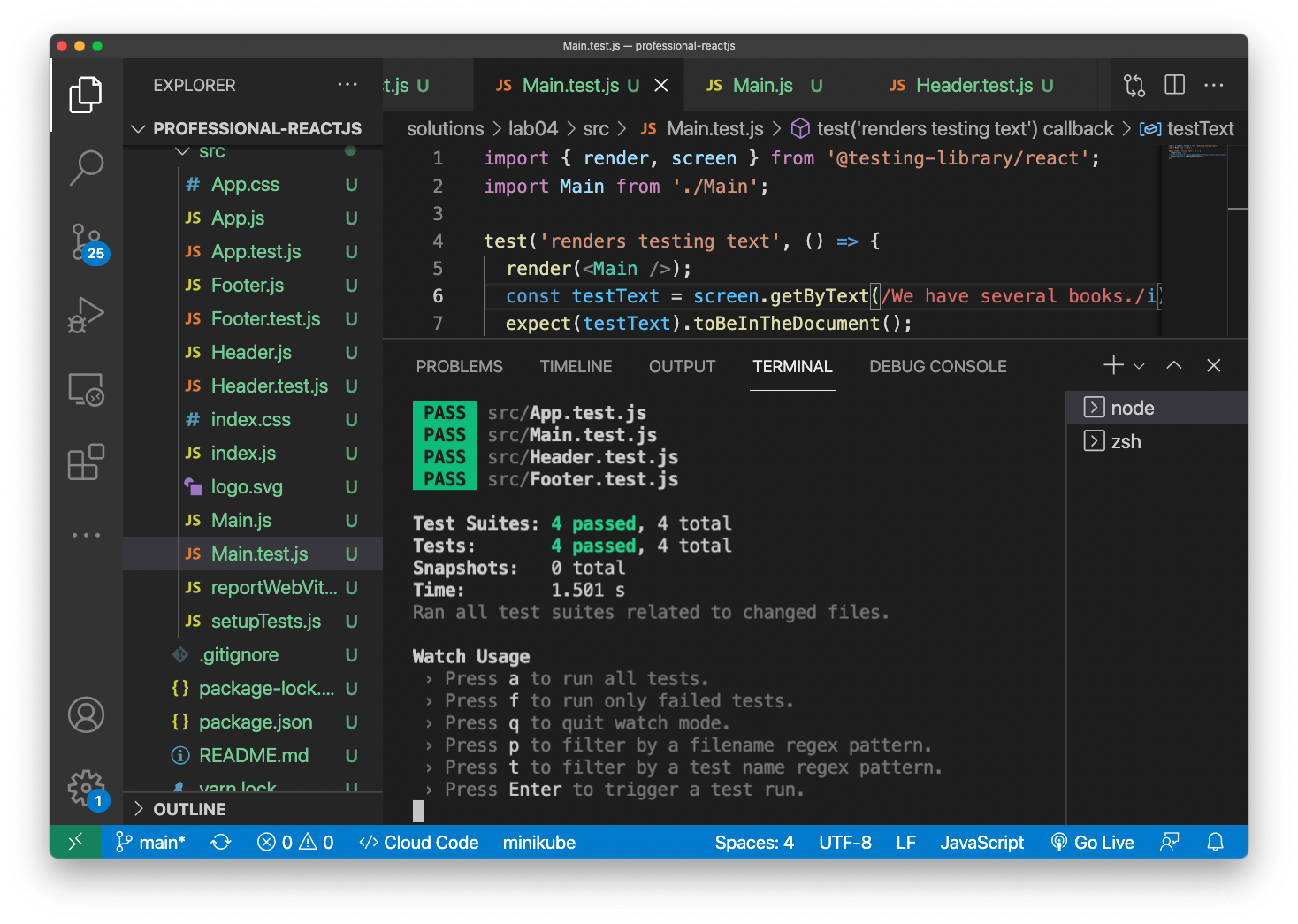
# Lab 04: Testing React

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. 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**).
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

1. Make sure that all the tests pass.



# 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 react-bookstore 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/chrisminnick/professional-reactjs

1. Open **professional-reactjs/labs/lab05/**.

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

1. 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 using this data.

1. Open **labs/lab05/mockup** and look at the **mockup.jpg** image.

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

1. 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.

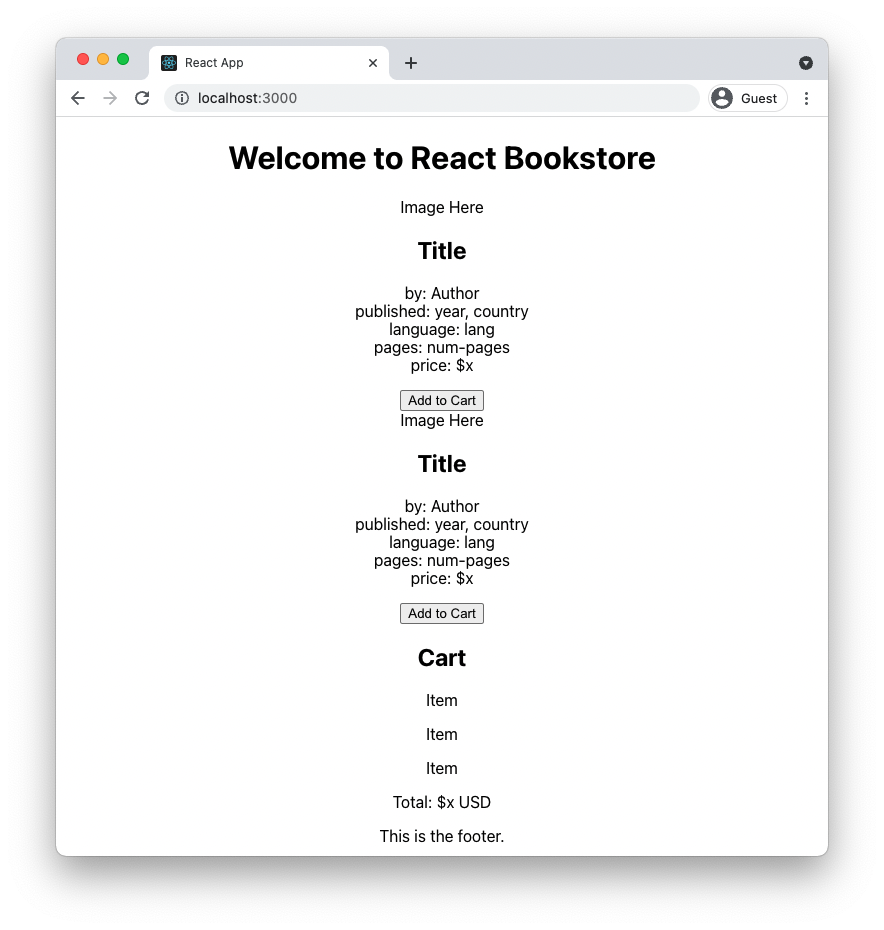
**Hint 2:** Look for repeating elements that can be made into components.

1. Move the **data** directory from the **/labs/lab05** directory into the **src** directory inside your **react-bookstore** project.
2. Move the **images** directory into the **public** directory.
3. 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;

1. 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.
2. Run npm start to verify that your code builds. Your UI should now look something like this:



# 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

1. Link to **bootstrap.css** inside of **index.js.**

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

1. Add the Bootstrap container class to **App.js**.

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;

1. Open **Main.js** 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>

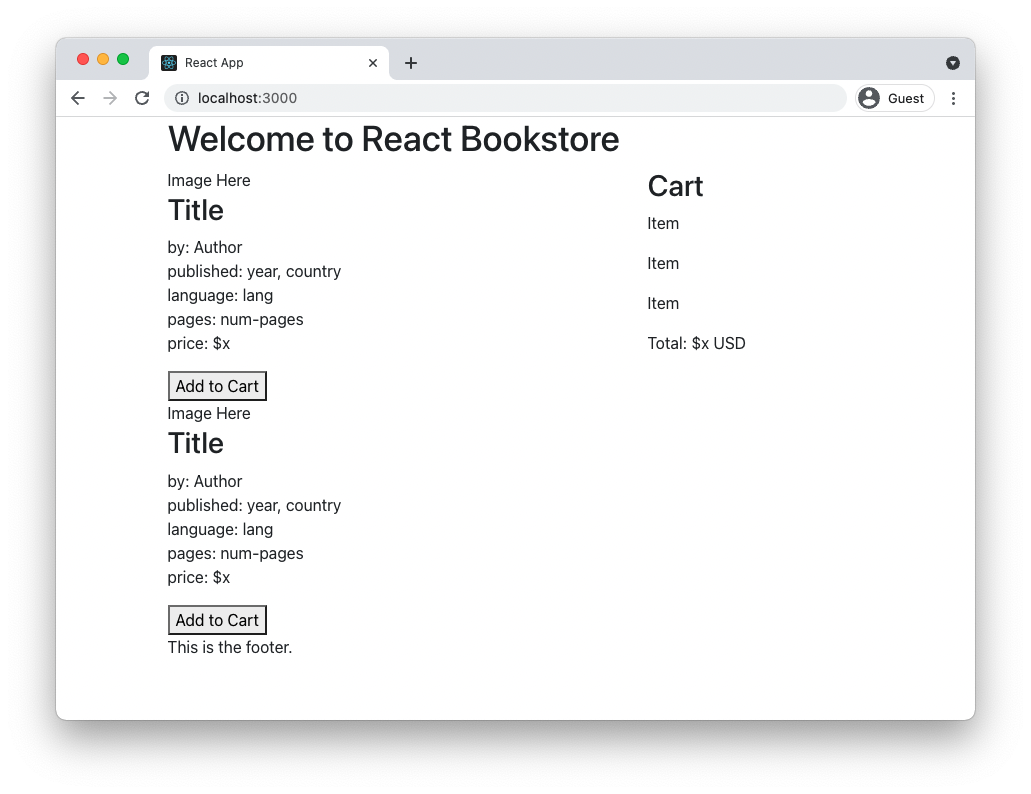
</main>

);

}

export default Main;

1. Run npm test and make sure that your tests pass.
2. Run npm start to verify that your code builds. Your UI should now look like this:



1. Modify your ProductList component to make each product an item in an unordered list.

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

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

This will be our first style module.

1. 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.

1. Import the style module into ProductList.js and give the module the name styles.

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

1. Attach the styles to the appropriate elements.

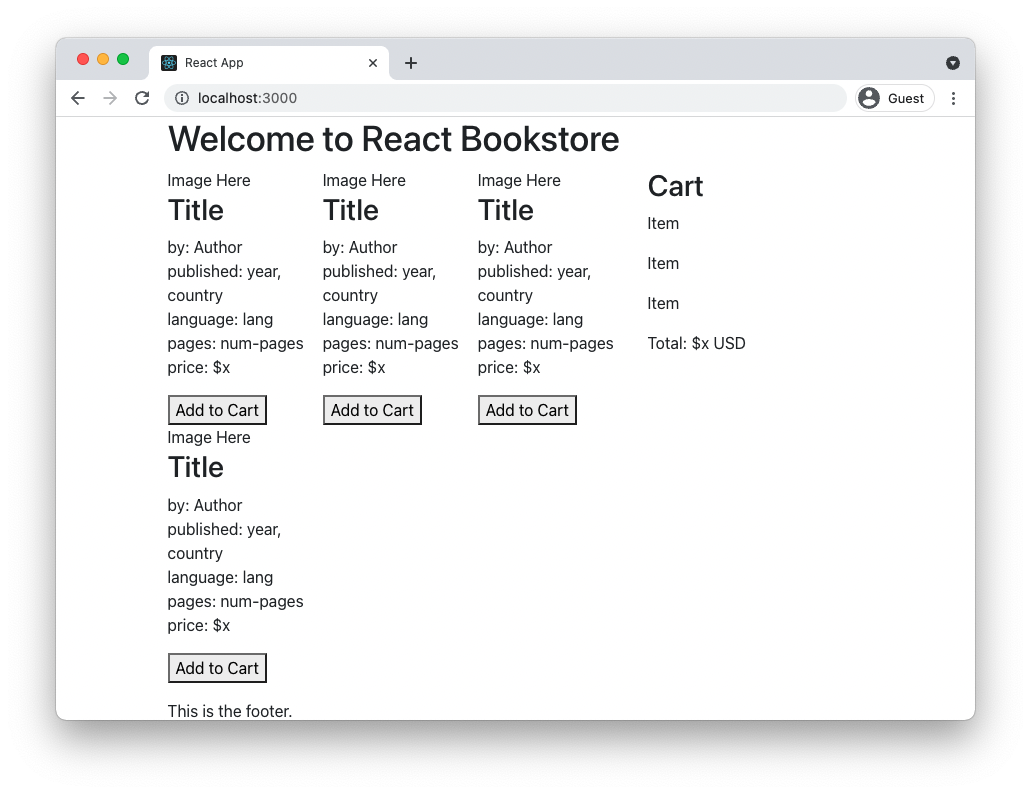
<ul className={styles.productList}>

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

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

</ul>

1. Add as many additional <Product /> elements to the list as you want by copying and pasting additional lines in the return statement.
2. Preview the styled list in your browser.



1. Create empty style modules for Product, Cart, and CartItem and import them into each module using the same pattern shown above.
2. 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.js**, **Header.js**, **Footer.js**, **Main.js**, **ProductList.js**, **Product.js**, **Cart.js**, and **CartItem.js**, along with their test suites and css modules, into the **components** directory.

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

1. 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';

1. 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) {

1. Pass productsData from the App component to the Main component as a prop called products.

<Main products={productsData} />

1. Pass products from Main to the ProductList component.

<ProductList products={props.products} />

1. 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>

);

1. 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;

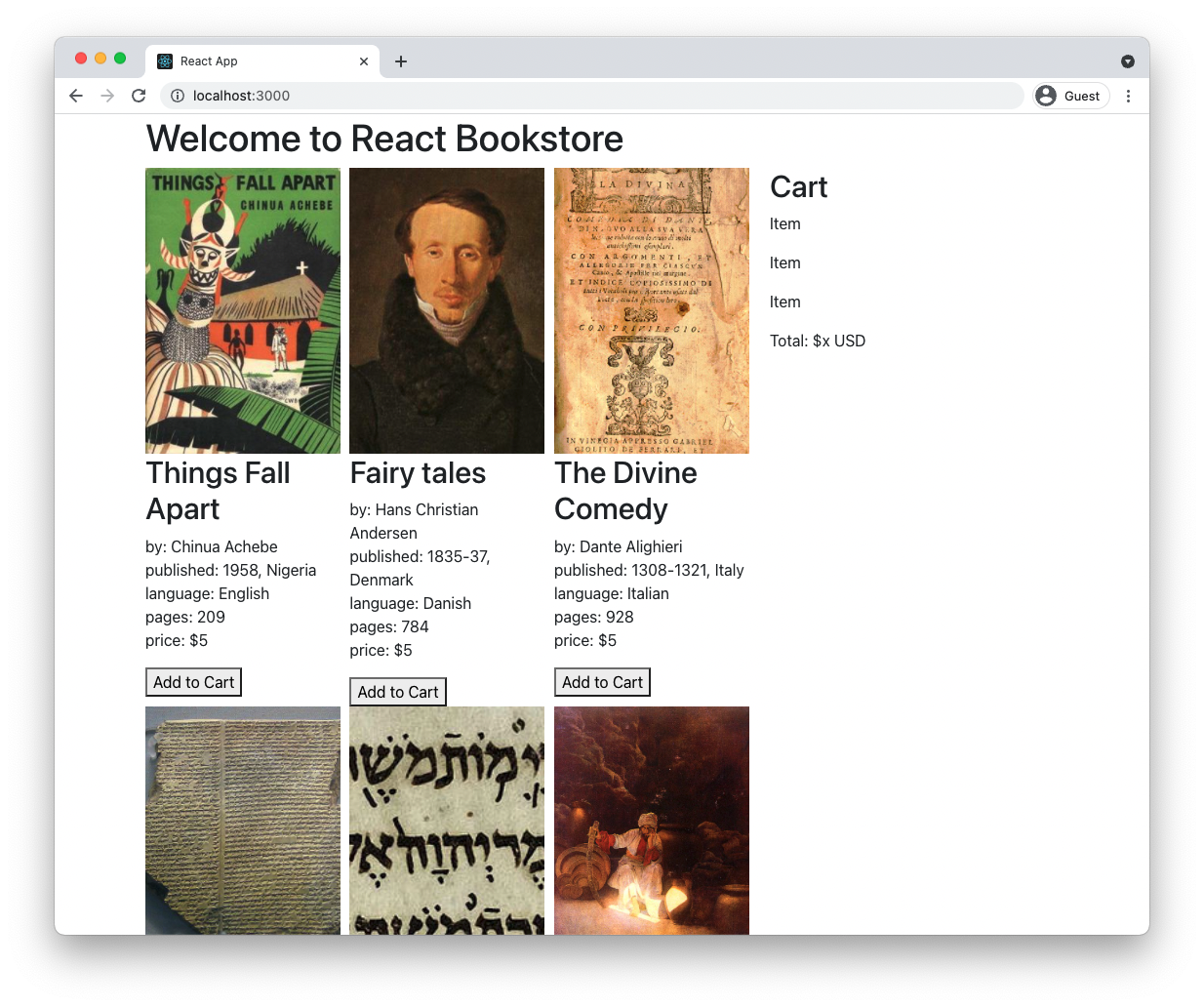
1. Update Product to make use of the props passed to it to display data about each product.
2. Create a new style rule in **Product.module.css** called thumbnail and set properties to format the book thumbnail images.

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

1. 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}" />

1. Run npm start



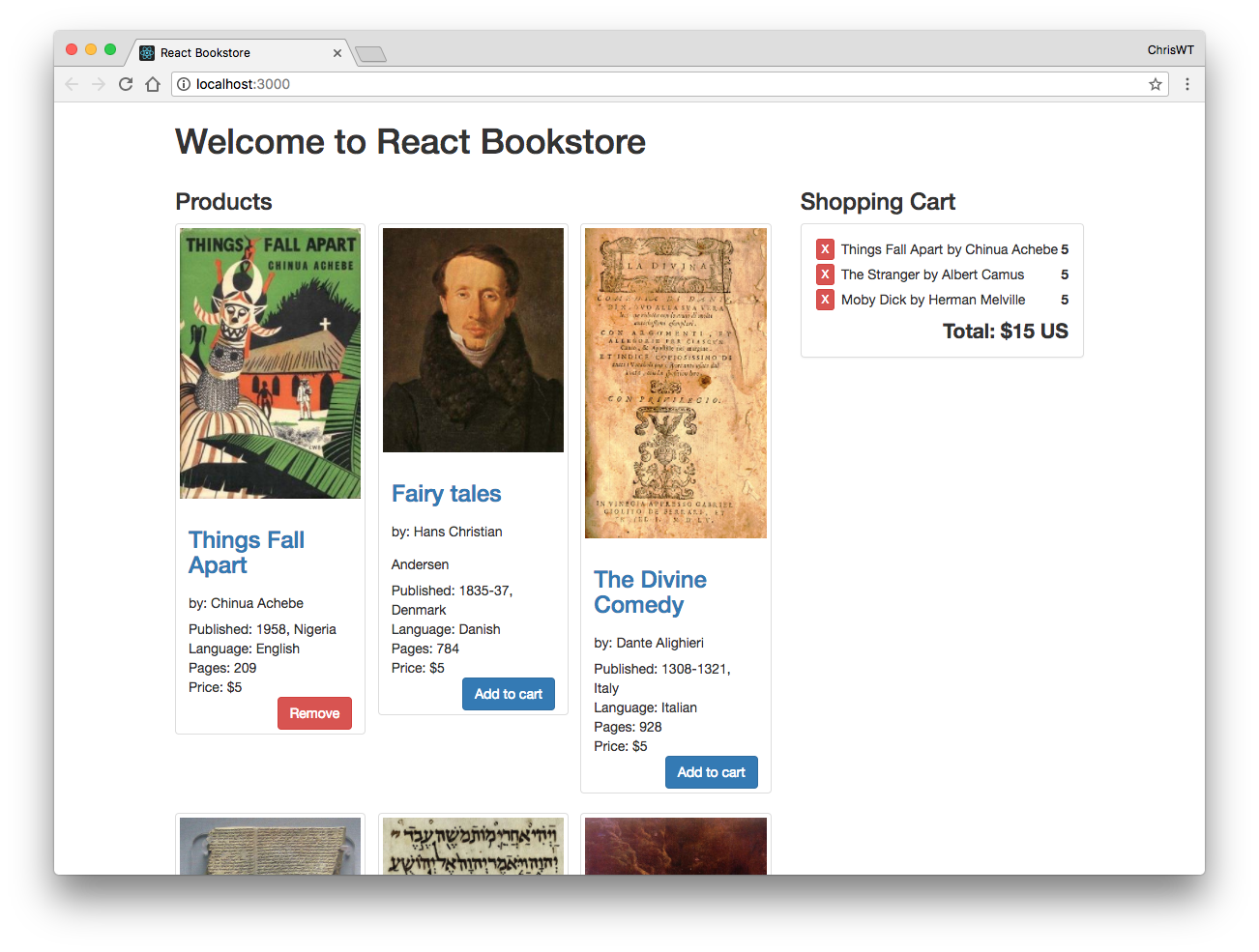
# Lab 08: Adding State

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.js** in your code editor.
2. Import React.useState

import {useState} from 'react';

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

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

1. Pass itemsInCart into the Main component as props.

<Main products={productsData}

itemsInCart={itemsInCart} />

1. Add some default product ids to the itemsInCart array, for testing.

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

1. Pass itemsInCart from Main to ProductList.
2. 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.

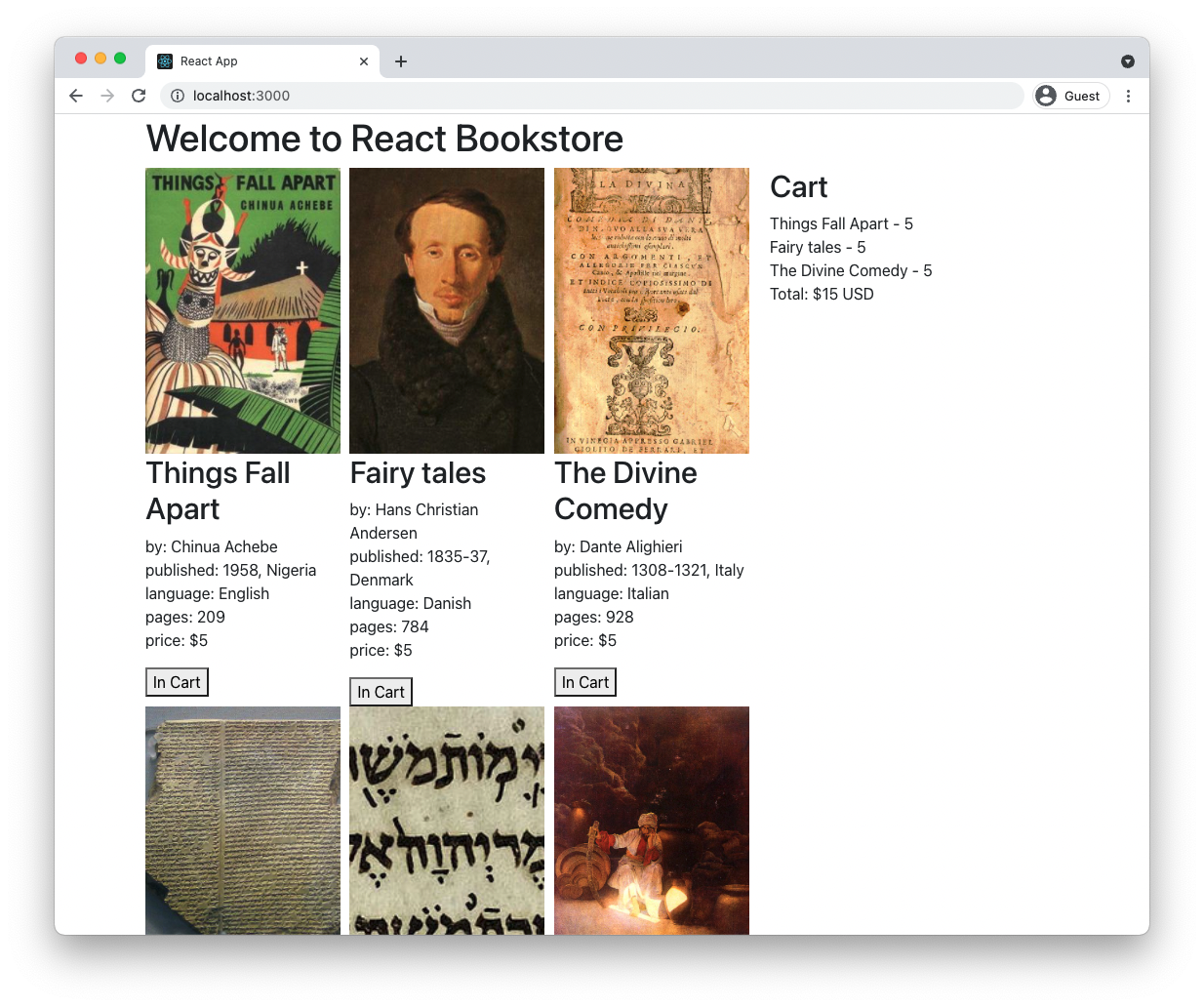
If you get stuck, look at the solution inside **intro-to-react/solutions/lab08**.

1. Inside the Main 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 ids 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);  
}

1. Render the list of cartItems inside the Cart component.
2. Modify the CartItem component to display the name and price of the item.
3. 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 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 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.js** 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 Main from './Main'

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 />

<Main products = {productsData}

itemsInCart = {this.state.itemsInCart}

/>

<Footer />

</div>

);

}

}

export default App;

1. 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.

1. 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);

1. Pass the bound addToCart function down to the Main component as a prop.

<Main products = {productsData}  
 itemsInCart = {this.state.itemsInCart}

**addToCart = {this.addToCart}**  
/>

1. Open the Main component and pass the addToCart function to the ProductList component as a prop.
2. Open the ProductList component and pass the addToCart function to the Product components as a prop.
3. 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);  
}

1. 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>

1. Run **npm start** and test out your application.
2. 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);

1. Replace the array used for displaying the products with the new randomly sorted array.

{sortedProducts.map(product => (

1. 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:

1. Create a method named componentDidMount() in the App component.
2. 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).
3. Revert the ProductList component to how it was before you made the previous changes.
4. In App's componentDidMount method, call this.shuffleArray() and pass in the productsData.

this.shuffleArray(productsData);

1. 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.

1. Add a property named products with a default value of an empty array ([]) to the state object.
2. Add a property named loading with a default value of false to the state object (in the constructor).
3. Remove the import of the product data.
4. Pass this.state.products to the Main component instead of productsData.
5. Make a copy of products.js named products.json and save it in a subdirectory of your **public** directory named **data**.
6. Open public/data/products.json and remove the variable assignment and the semicolon and export statement so that the file starts with [ and ends with ]
7. Confirm that the json file is in the right place by visiting the following url in your browser: **http://localhost:3000/data/products.json**
8. 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})  
 })  
};

1. Run npm start

# 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.js 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 [loading, setLoading] = useState(false);

1. 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]);

1. Use useEffect to shuffle the products array when the list of products is updated.
2. Convert the functions written using method syntax to use the function keyword or arrow functions.
3. 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.
4. Take the return statement out of the render method and delete the render method.
5. Update references to this in the return statement.
6. Use the isLoading state variable to conditionally display a loading message or loader animation when data is loading.
7. Start the app and fix any errors that occur.

# Lab 12: PropTypes and defaultProps

PropTypes are a built-in way to typecheck your React components' props. Default props set default values for props. In this lab, you'll use PropTypes and defaultProps to your components.

1. In ProductList, import prop-types.

import PropTypes from 'prop-types';

1. In **ProductList.js**, but outside of the function definition, add a propTypes object to ProjectList.

ProductList.propTypes = {

};

1. 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,

};

Hint: PropList receives a products prop that's an array of objects. You'll need to use PropTypes.arrayOf and PropTypes.shape

1. 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.
2. Comment out one or more of the props passed from Main to ProductList.

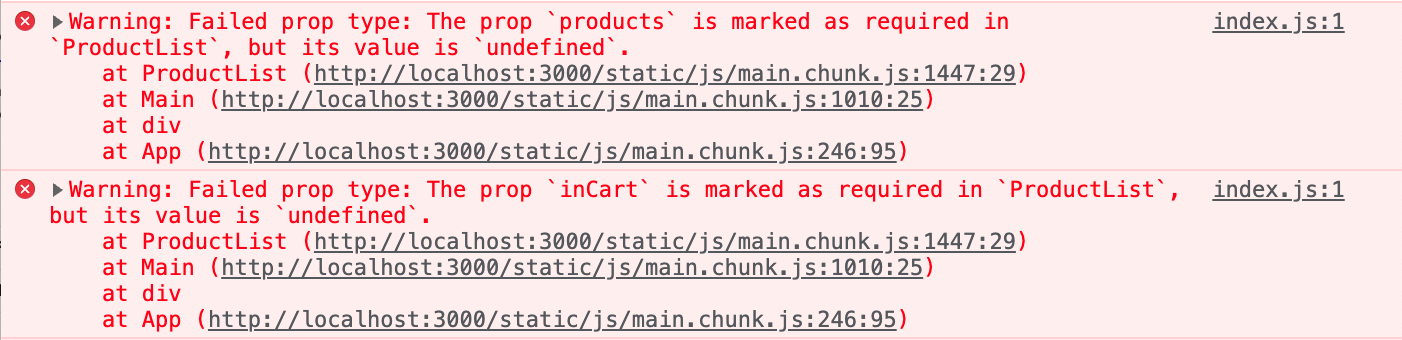
<ProductList /\* products = {props.products} \*/

itemsInCart = {props.itemsInCart}

addToCart = {props.addToCart}

removeFromCart = {props.removeFromCart} />

1. Check the JavaScript console to see the PropType warnings.



1. Add a defaultProps object to ProductList:

ProductList.defaultProps = {

}

1. Set default props for each of the arrays that ProductList receives. Set the defaults to empty arrays.
2. Restore the prop passing so that there are no errors.
3. Add a propType object and a defaultProps object to every other component that receives props.

# Lab 13: Converting to TypeScript

In this lab, you'll make a TypeScript version of the BookStore app.

1. Make a new Create React App project, using the typescript template:

npx create-react-app my-app --template typescript

1. Copy /src and /public from the previous lab and replace src and public in the new typescript project.
2. Install Bootstrap in the new project:

npm install --save bootstrap

1. Run the new project to make sure it still works.
2. Rename App.js to App.tsx. You should now see a lot of errors in the terminal and in the browser.
3. Create a Book type or interface in App.tsx. Something like this should work:

interface Book {

id: string;

title: string;

author: string;

published?: string;

country?: string;

lang?: string;

pages?: string;

image?: string;

url?: string;

price?: string;

}

1. Assign types to the three state variables:

const [itemsInCart, setItemsInCart] = useState<Array<string>>([]);

const [products, setProducts] = useState<Array<Book>>([]);

const [isLoading, setIsLoading] = useState<Boolean>(false);

1. Assign types to the parameters of addToCart and removeFromCart. For example:

function addToCart(id: string) {

1. Specify that the parameter to shuffleArray should be an array of Books.

function shuffleArray(array: Book[]) {

1. Remove the PropTypes and defaultProps from Main.js
2. Test out the app to make sure it still works.
3. Rename Main.js to Main.tsx
4. Create an interface for Book (it could also be put into a separate file so it can be imported into both App and Main).
5. Create an interface called Props that contains the props received by Main. Function types in the interface need to have their parameters and return values. If the function doesn't return anything, use void. For example:

addToCart: (id: string) => void;

1. Specify the type of the props object received by Main:

function Main(props: Props) {

1. Convert the rest of Main.tsx to TypeScript
2. Remove the PropTypes and default Props from ProductList
3. Convert the rest of the components to TypeScript.

# 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.

1. Modify your test script in package.json to the following:

"test": "react-scripts test --coverage --watchAll --no-cache",

1. 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.
2. 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

1. Have Jest include a coverage report by running npm test using the following command:

npm test

1. Write more tests, refine your existing tests, and increase your test coverage %.
2. 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.

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 and the React bindings.

npm install --save redux react-redux

## Step 1: Create a store

1. In index.js, import createstore and combineReducers from redux.

import {createStore, combineReducers} from 'redux';

1. Import Provider from react-redux.

import {Provider} from 'react-redux';

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

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

1. Create the root reducer.

const rootReducer = combineReducers({

cart: cart,

products: products

});

1. Define the store's initial state.

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

1. Create the store by passing the root reducer and the initial state into the createStore method.

let store = createStore(

rootReducer,

initialState

);

1. Wrap the App component in a Provider and pass the store to Provider as a prop.

ReactDOM.render(<Provider store={store}><App /></Provider>,

document.getElementById('root'));

## Step 2: Write the reducers

1. Create a directory in **src** named **reducers**.
2. 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.

1. 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

}

}

1. 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]  
 };

1. 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)  
};

1. Write and export the products reducer function (also in **reducers/index.js**). 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

1. Create a new directory in **src**, named **actions**.
2. 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.

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

import \* as actionCreators from '../actions';

import {bindActionCreators} from 'redux';

import {connect} from 'react-redux';

1. 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);  
};

1. 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 App.js to use the Redux store.

1. In **App.js**, remove the following:
   * The useState function calls for itemsInCart and for products.
   * The addToCart method
   * The removeFromCart method
2. Make sure the props object is a parameter of App

function App(props) {

1. 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]);

1. 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.
2. 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.

1. 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?
2. Add a 'Remove' button to the CartItem component that causes the item to be removed from the cart.
3. Install Redux Dev Tools in your browser and project by following the instructions here: **https://github.com/reduxjs/redux-devtools/tree/main/extension**

# 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.

1. Install Redux Thunk.

npm install --save redux-thunk

1. Include ReduxThunk in **src/index.js.**

import ReduxThunk from 'redux-thunk';

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

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

1. 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\_\_()  
);

1. 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.
2. 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

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

import axios from 'axios';

1. 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  
 })  
 );  
 };  
}

1. 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}}  
}

1. Pass the submitCart action creator from App to Main, and from Main to the Cart**.**

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

1. 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>

1. 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.
2. Open a new terminal window and change to the **labs/lab16/server** directory.
3. Run npm install in the server directory
4. Run the server by entering npm start.
5. 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.

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

case 'CHECKOUT':

return {

};

1. 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: []**  
 };

1. 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: Redux Saga

In this lab, you'll convert your Redux application to use Redux Saga instead of Thunk.

1. Install the redux-saga package.

npm install redux-saga --save

1. Replace the import of ReduxThunk with an import of createSagaMiddleware from redux-saga in src/index.js

import createSagaMiddleware from 'redux-saga';

1. Make a new directory in /src named sagas
2. Make a new file in the src/sagas directory named index.js
3. In sagas/index.js, import call, put, and takeEvery from redux-saga/effects

import {call, put, takeEvery} from 'redux-saga/effects';

1. Make a file named submitCart in a new directory named apis in src/ and write a function named submitCart. This function will be based on the submitCart function you wrote in actions/ for thunk, but without the dispatch:

import axios from 'axios';

function submitCart(data) {

return axios.post('http://localhost:8080/checkout', {

data

});

};

export default submitCart;

1. Import submitCart into sagas/index.js

import submitCart from '../apis/submitCart';

1. Write a worker saga in sagas/index.js that we'll fire on SUBMIT\_CART actions and that will call the submitCart function:

function\* checkout(action) {

try {

const response = yield call(submitCart, action.payload);

yield put({ type: 'CHECKOUT\_SUCCEEDED', response: response });

console.log(response);

} catch (e) {

yield put({ type: 'CHECKOUT\_FAILED', message: e.message });

}

}

1. Write a watcher function to listen for SUBMIT\_CART events and export it:

function\* mySaga() {

yield takeEvery('SUBMIT\_CART', checkout);

}

export default mySaga;

1. Go back to src/index.js and import mySaga:

import mySaga from './sagas';

1. Create the saga middleware:

const sagaMiddleware = createSagaMiddleware();

1. In createStoreWithMiddleware, replace ReduxThunk with sagaMiddleware:

const createStoreWithMiddleware =

compose( applyMiddleware(sagaMiddleware) )(createStore);

1. After the statement that creates the store (let store = ...) but before the ReactDOM.createRoot statement, start the watcher function:

sagaMiddleware.run(mySaga);

1. Remove the checkOut function from actions/index.js, as well as the import of axios.
2. Make the submitCart action creator in actions/index.js return a normal (not thunk) action:

export function submitCart(data){

return {type: 'SUBMIT\_CART', payload: {data}}

}

1. Rename the CHECKOUT reducer case (in reducers/index.js) to CHECKOUT\_SUCCEEDED so that it will clear the cart when the server returns an approval.
2. Write a reducer case for CHECKOUT\_FAILED that just returns the same state for now.
3. Make sure the server is running, then start up the bookstore app and see if the checkout works!

# Lab 18: 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");

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

cart = JSON.parse(cart);

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

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

1. Create an action creator named readCart that creates the READ\_CART action.
2. After the products are loaded, dispatch READ\_CART.

dispatch('READ\_CART');

1. 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.

1. 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  
};

1. 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.
2. Once you can add books and have them saved in localStorage, do the same thing in the CART\_REMOVE case.
3. 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 19: Refactoring and Organizing

We've come a long way and now have a book catalog and a cart that allow a user to add and remove products to a cart and to submit the cart to a server. But, there are still a lot of missing pieces to finish. Before we move on, let's take a step back and finish some tasks that will make it easier for us to continue the development of this app.

The first thing we'll do is to move the product loading into Redux. You'll need to create an action creator (to dispatch a new action named GET\_PRODUCTS), a new function in api/index.js and a saga and then invoke the action creator from the effect in App.

1. Write a new watcher saga named watchGetProducts. This saga should take the latest GET\_PRODUCTS action and call the getProducts worker saga (which we'll create in a moment).
2. Write the getProducts worker saga, which will call the getProducts api and then dispatch the LOAD\_PRODUCTS action, passing in the data from the api.
3. Extract the fetch statement from the useEffect hook into a new module in the **apis** directory. Export the new api from **apis/index.js**.

Next we need to create a root saga that will start both watcher sagas in parallel.

1. Import all into sagas/index.js from **redux-saga/effects**
2. Use all to combine and export rootSaga:

export default function\* rootSaga() {

yield all([

watchGetProducts(),

watchSubmitCart()

])

}

1. Update the saga import and run statements in **index.js**
2. Write the getProducts action creator and call it from within the useEffect hook in **App.js**
3. Create a directory named **src/utils** and **src/utils/index.js** and copy the shuffleArray method into it and export it as a named export.
4. If you want to continue using shuffleArray, import it and call it from an appropriate place....probably the getProducts saga.
5. Split the reducers/index.js file into cart.js and products.js and then move the combineReducers constant from index.js into reducers/index.js.
6. Inside **src/actions**, create a file named **ActionTypes.js**. Use this file to create and export a constant for each action type in the application. For example:

export const CART\_ADD = 'CART\_ADD';

1. Import **ActionTypes.js** into **actions/index.js**

import \* as ActionTypes from './ActionTypes';

1. Change the string action types in the action creators (in **action/index.js**) to use constants from **ActionTypes.js**. For example:

export function addToCart(productId) {

return {

type: ActionTypes.CART\_ADD,

payload: {

productId

}

}

}

1. Optionally, separate the action creator functions into separate files, **cart.js** and **products.js**
2. Make sure everything still works.

# Lab 20: React Router

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

1. Install react-router-dom@5.3.0
2. Import BrowserRouter as Router into **index.js** and wrap the Router component around the <Provider> element in ReactDOM.render
3. In **Main.js**, import Switch and Route from react-router-dom.
4. In the render method of Main, 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 Switch 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'.

<Switch>

<Route exact path='/'>

<ProductList products = {props.products}

itemsInCart = {props.itemsInCart}

addToCart = {props.addToCart}

removeFromCart = {props.removeFromCart} />

</Route>

<Route exact path='/cart'>

<Cart cartItems = {cartItems}

removeFromCart = {props.removeFromCart}

submitCart = {props.submitCart} />

</Route>

</Switch>

1. 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
2. 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**
3. **Challenge: Change the version number for react-router-dom in package.json to the latest version (which you can find at https://github.com/remix-run/react-router) and run npm install. Use the React Router docs and the error messages to upgrade your code for the new version.**

# Lab 21: Creating Micro Frontends

## Part 1: Create the Content Server

All of our micro frontends will need access to certain common resources, including the react and react-dom libraries, global CSS, and images. In this first part, we'll create a server for these common resources.

1. Create a new directory named **content**.
2. Run npm init -y inside content to initialize a node project and create the **package.json** file.
3. Install the serve package @ version 6.5.2 inside **content**

npm install --save-dev serve@6.5.2

1. Write a start script in content's **package.json** to start up the server and serve the contents of the **public** directory.

"start": "serve --port 5000 --cors public"

1. Make a folder named **public** inside **content** and copy the **data** and **images** directories from the **bookstore** project into it.
2. Download the latest UMD builds of react and react-dom from the following link, and put them in the content/public directory:

**https://reactjs.org/docs/cdn-links.html**

1. Create a file named **style.css** inside **content/public**. We'll use this CSS file for common styles in all our micro frontends. For now, it will only contain a single style rule:

.App {

text-align: center;

}

1. Run npm start in **content**, then open up your browser and go to **http://localhost:5000/data/products.json** to confirm that you can access the books data. If so, then you're ready to move on to the next step! Leave the server running and open a new Terminal window.

## Part 2: Create the container

1. Make a new app named container using the following command:

npx create-react-app container

1. Install react-app-rewired in the container app. This will allow us to override settings in create-react-app that we need to change to make a micro frontend.

npm install react-app-rewired --save-dev

1. Change the scripts in **package.json** to use react-app-rewired, as follows (on Windows):

"start": "set PORT=3000 && react-app-rewired start",

"build": "react-app-rewired build",

"test": "react-app-rewired test"

1. On MacOS, change the scripts in **package.json** to use react-app-rewired, as follows:

"start": "PORT=3000 react-app-rewired start",

"build": "react-app-rewired build",

"test": "react-app-rewired test"

1. Put the config path for react-app-rewired in **package.json**

"config-overrides-path": "node\_modules/react-app-rewire-micro-frontends",

1. Install the react-app-rewired script for micro frontends.

npm install --save-dev react-app-rewire-micro-frontends

1. Create a file named **.env** in the container directory and specify the urls for the bookstore and content micro frontends, as follows:

REACT\_APP\_BOOKSTORE\_HOST=http://localhost:3001

REACT\_APP\_CONTENT\_HOST=http://localhost:5000

1. Open **container/public/index.html** and put a link to the common css file in the <head>.

<link rel="stylesheet" href="%REACT\_APP\_CONTENT\_HOST%/style.css"></link>

1. Add the following script tags right above </body> in **container/public/index.html** (customizing them to match the versions you downloaded to the content micro frontend if necessary):

<script src="%REACT\_APP\_CONTENT\_HOST%/react.development.js">

</script>

<script src="%REACT\_APP\_CONTENT\_HOST%/react-dom.development.js"></script>

1. Make a component named **MicroFrontend.js** in **container/src** with the following code:

import React from 'react';

class MicroFrontend extends React.Component {

componentDidMount() {

const { name, host, document } = this.props;

const scriptId = `micro-frontend-script-${name}`;

if (document.getElementById(scriptId)) {

this.renderMicroFrontend();

return;

}

fetch(`${host}/asset-manifest.json`)

.then(res => res.json())

.then(manifest => {

const script = document.createElement('script');

script.id = scriptId;

script.crossOrigin = '';

script.src = `${host}${manifest.files['main.js']}`;

script.onload = this.renderMicroFrontend;

document.head.appendChild(script);

});

}

componentWillUnmount() {

const { name, window } = this.props;

window[`unmount${name}`](`${name}-container`);

}

renderMicroFrontend = () => {

const { name, window, history } = this.props;

window[`render${name}`](`${name}-container`, history);

};

render() {

return <main id={`${this.props.name}-container`} />;

}

}

MicroFrontend.defaultProps = {

document,

window,

};

export default MicroFrontend;

This is the component that will load the micro frontend components into the container. It does this by getting the **main.js** file from the micro frontends (which it finds in the **asset-manifest.json** file in each micro frontend and injecting a script tag and root element into the DOM.

1. Install react-router-dom in the **container** directory

npm install --save react-router-dom@5.3.0

1. Import BrowserRouter, Switch, and Route from react-router-dom into App in the container app.

import { BrowserRouter, Switch, Route } from 'react-router-dom';

1. Import MicroFrontend into App

import MicroFrontend from './MicroFrontend';

1. Remove the imports of **App.css** and **logo.svg** from **App.js**
2. Before the App function, make a constant to hold the bookstore host in **App.js**

const {

REACT\_APP\_BOOKSTORE\_HOST: bookstoreHost,

} = process.env;

1. Make a const to hold the MicroFrontend for Bookstore

const Bookstore = ({ history }) => (

<MicroFrontend history={history} host={bookstoreHost} name="Bookstore" />

);

1. Wrap the contents of the return statement in App with <BrowserRouter>.
2. Use a Switch with a Route in it to render the Bookstore

<Switch>

<Route exact path="/" component={Bookstore} />

</Switch>

1. The return statement of **containers/src/App.js** should now look like this:

return (

<BrowserRouter>

<Switch>

<Route exact path="/" component={Bookstore} />

</Switch>

</BrowserRouter>

);

1. Make a new component named Header (it can return anything at this point), import it into App, and render it before the BrowserRouter component, wrapping everything in a React.Fragment component make sure only one element is returned.
2. Start up the container app. You'll get an error. This is because it's trying to fetch **asset-manifest.json** from the bookstore micro frontend, which doesn't exist yet. Let's start that now. Leave the **container** app running and open a new terminal.

## Part 3: Making a Micro Frontend

1. Use create-react-app to make a new app named bookstore.
2. Install react-app-rewired in the bookstore app. This will allow us to override settings in create-react-app that we need to change to make a micro frontend.

npm install react-app-rewired --save-dev

1. Change the scripts in **package.json** to use react-app-rewired, as follows (on Windows):

"start": "set PORT=3001 && react-app-rewired start",

"build": "react-app-rewired build",

"test": "react-app-rewired test"

1. On MacOS, change the scripts in **package.json** to use react-app-rewired, as follows:

"start": "PORT=3001 react-app-rewired start",

"build": "react-app-rewired build",

"test": "react-app-rewired test"

1. Put the config path for react-app-rewired in **package.json**

"config-overrides-path": "node\_modules/react-app-rewire-micro-frontends",

1. Install the react-app-rewired script for micro frontends.

npm install --save-dev react-app-rewire-micro-frontends

1. In **bookstore/public/index.html**, use the same imports for the global CSS, React, and React DOM that you used in the container project.
2. Change the id of the root node in bookstore/public/index.html to 'container'. This will make it possible for you to view the bookstore app on its own port as well as rendered inside the container.
3. Delete the root <div> and replace it with a <main> element with an id of container.

<main id="container"></main>

1. Add the following script element under the imports of React and ReactDOM in index.html:

<script type="text/javascript">

window.onload = () => {

window.renderBookstore('container');

};

</script>

1. Make a **.env** file in bookstore with a link to the content server:

REACT\_APP\_CONTENT\_HOST=http://localhost:5000

1. Replace ReactDOM.render in **index.js** with the following:

window.renderBookstore = (containerId, history) => {

ReactDOM.render(

<App />,

document.getElementById(containerId),

);

};

window.unmountBookstore = containerId => {

ReactDOM.unmountComponentAtNode(

document.getElementById(containerId)

);

};

1. Make a file named **setupProxy.js** in **src** with the following content. This will allow the container to access the Micro Frontend.

module.exports = app => {

app.use((req, res, next) => {

res.header('Access-Control-Allow-Origin', '\*');

next();

});

};

1. Start up the bookstore app. Make sure that your content server and your container are both running too and then go to **http://localhost:3000** in your browser.

If you did all the steps correctly, you should see the default create-react-app app (although the logo image will be broken) and your Header component (from the container app) will be rendered above it.

Congratulations! You've created a Micro Frontend User Interface!

1. Modify App.js to display some generic text, such as "Bookstore Micro Frontend".
2. Make a copy of the bookstore app, name it cart, change every instance of bookstore in it to cart, modify **package.json** so that it starts on port 3002, and add it to the container so that it displays when the url is **localhost:3000/cart**. You'll need to stop and restart the container app so that it gets the new env variable for the cart application.

# Lab 22: Getting the Bookstore Working as a Micro Frontend

In this lab, you'll move the bookstore app into your micro frontend architecture. To start with, we'll keep the cart and the bookstore in a single application. We can separate them after we get the application working as it is.

1. Copy all of dependencies and devDependencies from the working react/redux bookstore app's package.json to the bookstore micro frontend. Make sure that the bookstore micro frontend still has react-app-rewired and react-app-rewire-micro-frontends
2. Run npm install in the bookstore micro frontend.
3. Copy the components, actions, apis, reducers, sagas, and utils directories from the src directory of the bookstore app to the src directory of the bookstore micro frontend.
4. Update src/index.js in the bookstore micro frontend to include all the code from the bookstore app's src/index.js, plus the two micro frontend functions.
5. Make sure that the micro frontend index.js uses only the window.renderBookstore function to call ReactDOM.render, and that you copy the elements to be rendered from the working bookstore app.
6. Start the content and container micro frontends, then start the bookstore micro frontend. If you go to http://localhost:3000 now, you should see the header of the bookstore, but no books. The reason is that we need to update the fetching of the books.
7. In apis/index.js, get the content host and use it instead of the hard coded url.

const {

REACT\_APP\_CONTENT\_HOST: contentHost,

} = process.env;

...

1. Refresh the container app. You should now see the book data, but the images are all broken.
2. Update the app as necessary (without copying the book images to the bookstore micro frontend!) to make the images display.

# Bonus Lab: Sharing State Between Micro Frontends

1. Working in teams, figure out how to move the Cart out of the Bookstore Micro frontend and into its own.
2. Implement the necessary shared state to make the application work correctly.

The following articles provide some useful information:

https://martinfowler.com/articles/micro-frontends.html

https://dev.to/luistak/cross-micro-frontends-communication-30m3

# 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

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

# Bonus Lab - SwimCalc

In this lab, you'll build a React application from scratch. You may choose to use Redux or not for this project. Or, start out not using Redux, and then convert it to use Redux.

## The Story

Linda is a distance swimmer. Each month, she buys a lap swim pass from the city Department of Parks and Recreation that gets her 20 entries to the pool and is only good for one month.

The current cost of the pass is $50.

The first time she swims each month, she swims 1 kilometer (1000 meters). She increases her distance by 100 meters each time she swims during the month

Build an app that will tell Linda:

* How far she will have swum if she swims 20 times
* What is her price per kilometer swum
* What do the numbers look like if any of the variables in the equation change: -- Price for the lap swim pass -- Number of times she uses the pass in a month -- Starting distance -- Daily increase in distance

## Getting Started

The finished project might look something like this:

Graphical user interface, text

Description automatically generated

**Starter Project:** https://github.com/watzthisco/tdd-react-labs-v3.x/tree/lab30

**Example solutions:**

https://github.com/watzthisco/tdd-react-labs-v3.x/tree/lab30solutions