Professional React Development - Labs

Complete Lab Instructions for Professional React Development Course

📚 **Course Repository:**

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

Version: 19.2.2

Date: October 2025

Author: Chris Minnick

Website: https://www.watzthis.com

Table of Contents

Lab 01: Get Started with Vite / React

1. Open a new Terminal in VSCode.
2. Use Vite to make a new React project. This will be the project we'll be working on for most of the labs in this course.

   ```
   npm create vite@latest react-bookstore -- --template react
   ```

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

3. Go into the new directory.

   ```
   cd react-bookstore
   ```

4. Install the dependencies.

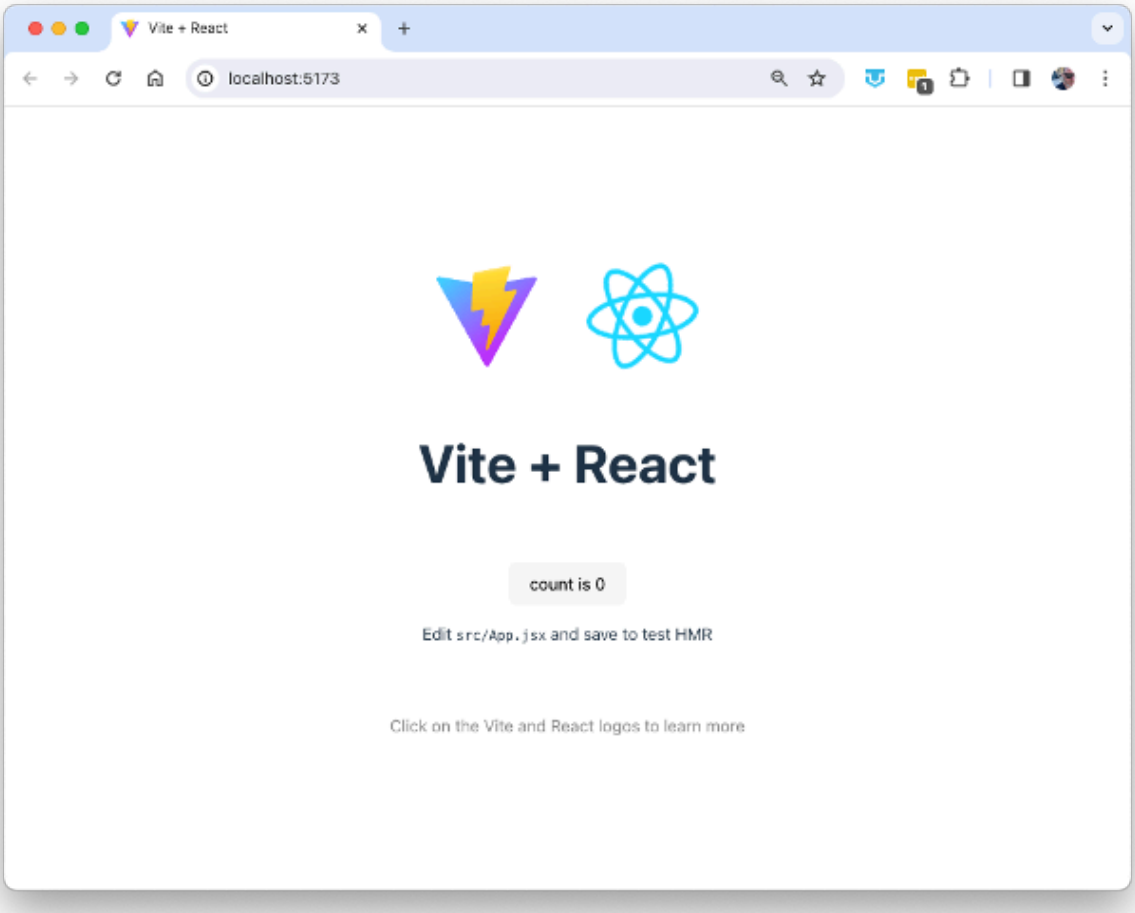
   ```
   npm install
   ```

5. Test that everything was installed and works.

   ```
   npm run dev
   ```

   After a moment, you'll see a URL in the terminal. Ctrl-Click the URL to open it in your default browser, or open a browser and go to the URL manually.

   If the app was successfully created, you should see the following page:



6. Find **App.jsx** inside the **src** directory and open it for editing.
7. Inside the return statement, delete the `<div>` containing the Vite and React logos.
8. Delete the logo import statements from the beginning of the file.
9. Change the text inside the `<h1>` element to `Welcome to React Bookstore`.
10. Create a new `<p>` element below the `<h1>` you added 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 imag
```

11. Open App.css and delete everything except for the first CSS rule:

```
#root {max-width: 1280px;
}
```

12. Open index.css and delete the following rule:

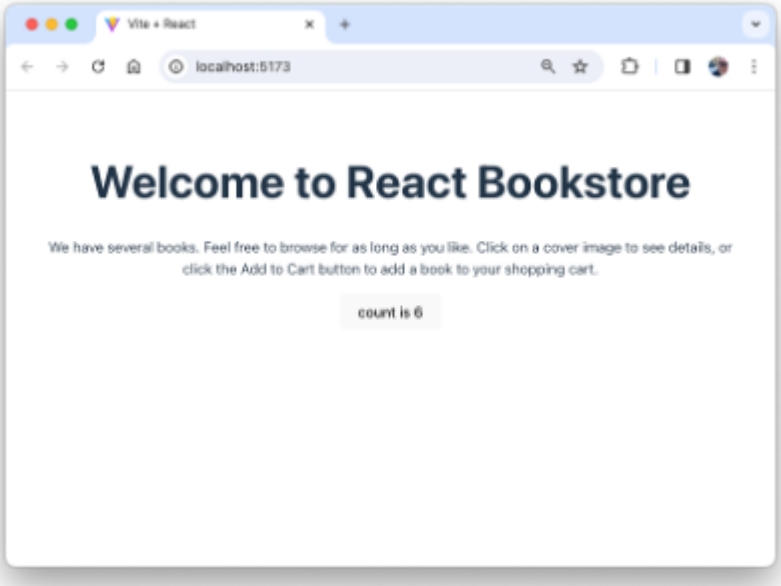
```
body {
  display: flex;
  place-items: center;
  min-width: 320px;
  min-height: 100vh;
}
```

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

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

14. Click the button to see the counter increase. Exciting!
15. Delete the text below the counter.

    If you get an error, make sure that all your opening tags have matching tags.



Lab 02: Your First Component

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.jsx** in the **src** directory.
2. Type the code below into **Footer.jsx**:

```
function Footer() {
  const footerStyle = {
    backgroundColor: 'black',
    color: 'white',
    padding: '1rem',
    textAlign: 'center'
  };
  return <p style={footerStyle}>This is the footer.</p>;
}
export default Footer;
```

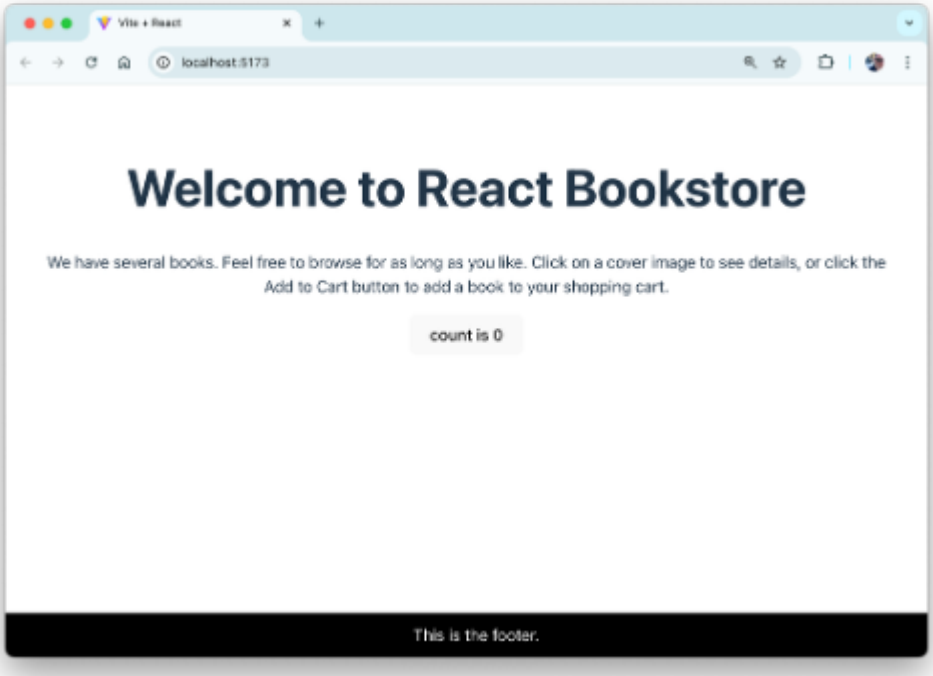
3. Add the following to the beginning of **App.jsx**:

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

4. Add the following inside **App.jsx** (before `</>`):

```
<Footer />
```

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


Vite + React
localhost:5173
Welcome to React Bookstore
We have several books. Feel free to browse for as long as you like. Click on a cover image to see details, or click the Add to Cart button to add a book to your shopping cart.
count is 0
This is the footer.

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.jsx**, make **Header.jsx** and **ClickCounter.jsx** to replace code in **App.jsx**.

   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.jsx** should match this:

   ```
   return (<div className="App">
   );
   ```

   Note: For the ClickCounter component to continue working, you'll need to move the `useState` import and the initialization of the `count` and `setCount` variables to **ClickCounter.jsx**.

Part 2: Writing tests

To get started with testing, you'll need to install several dependencies, add a testing configuration to the Vite config file, and create a new test script in `package.json`. Follow these steps:

1. Install vitest and several other dependencies by running the following npm install commands in the terminal:

   ```
   npm install --save-dev vitest @testing-library/react jsdom
   ```

2. Add the testing config to `vite.config.js`:

   ```
   import { defineConfig } from 'vite'
   export default defineConfig({
     test: { globals: true, environment: 'jsdom' },
   })
   ```

3. Add a test script to `package.json`:

```
"scripts": {
  "test": "vitest --environment jsdom"
}
```

4. Create a new file named `App.test.jsx` inside `src`.
5. Enter the following code into `App.test.jsx`:

```
import { describe, it, expect } from 'vitest';
import { render } from '@testing-library/react';
import App from './App';

describe('App', () => {
  it('renders without crashing', () => {
    render(<App />);
  });
});
```

6. Run your test script and confirm that your test passes:

```
npm run test
```

7. Make copies of **App.test.jsx** for testing **Footer.jsx**, **Header.jsx**, and **ClickCounter.jsx**.
8. Modify the contents of the new files to test that the new components render.
9. Run your tests by entering the following in the command line:

```
npm run test
```

10. 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.png** 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.

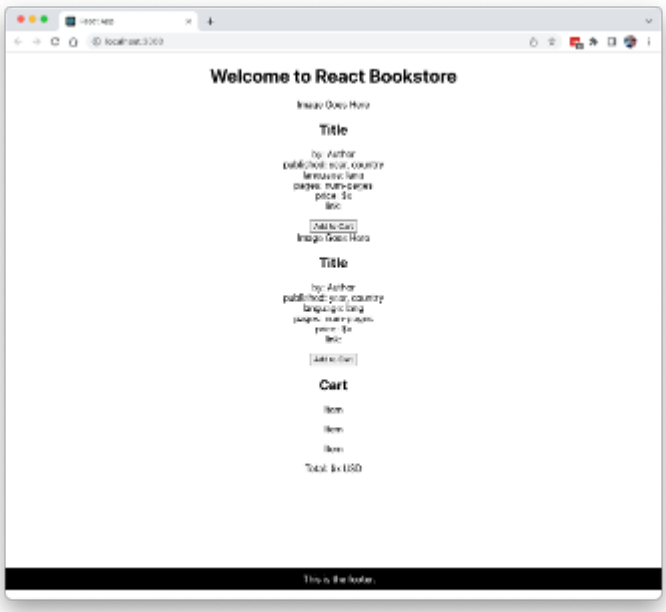
   Hint 2: Look for repeating elements that can be made into components.

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 **App.jsx** 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 Header from './Header';
import ProductList from './ProductList.jsx';
import Cart from './Cart.jsx';
function App() {
  return (
    <div>
      <Header />
      <ProductList />
      <Cart />
    </div>
  );
}
export default App;
```

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. Create basic tests for each of the new components, using the same structure you used in the previous lab.
10. Run `npm run dev` to verify that your code builds. Your UI should now look something like this:



Lab 05: Node.js Command Line Program for Data Management

In this lab, you'll create a Node.js command-line program that demonstrates modern JavaScript techniques while managing book data for your React bookstore project. This program will allow you to add new books to the `products.json` file using ES6 classes, async/await, destructuring, template literals, and other modern JavaScript features.

Prerequisites

Before starting this lab, ensure you've completed Lab 04 and have the `data` folder with `products.json` in your `react-bookstore/src` directory.

Instructions

1. Navigate to your `react-bookstore` project directory and create a new folder called `tools`:

```
cd react-bookstore
mkdir tools
```

2. Create a new file called `book-manager.js` in the `tools` directory:

```
touch book-manager.js
```

3. Open `book-manager.js` and start by importing the required Node.js modules using ES6 import syntax. Add the following at the top of your file:

```
import { readFile, writeFile } from 'fs/promises';
import path from 'path';
import { fileURLToPath } from 'url';
```

4. Set up the file path configuration using modern JavaScript techniques:

```
// Get current directory path (ES6 modules)
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
const DATA_FILE = path.join(__dirname, '..', 'src', 'data', 'products.json');
```

5. Create a `Book` class that represents a book entity with modern class syntax and methods:

```
class Book {
  }
  }
  }
  }
} }
```

6. Create a `BookManager` class that handles file operations using async/await:

```
class BookManager {
  }
  }
  }
  }
  }
} } }
```

7. Create a user interface class using readline for interactive input:

```
class BookCLI {
  }
  }
  }
  }
} } }
```

8. Add the main execution code using modern JavaScript patterns:

```
// Main execution with error handling
}
```

9. Test your command-line program by running it:

```
node tools/book-manager.js
```

The program should display a menu and allow you to add books or list existing ones.

```
🚀 Welcome to the Book Manager CLI!

📚 Book Manager CLI
==================
1. Add a new book
2. List all books
3. Exit

Enter your choice (1-3): 
```

Modern JavaScript Features Demonstrated

This lab demonstrates several modern JavaScript features:

- **ES6 Classes**: `Book`, `BookManager`, and `BookCLI` classes with constructor and methods
- **Async/Await**: File operations and user input handling
- **Destructuring Assignment**: In the `Book` constructor
- **Template Literals**: String interpolation throughout the code
- **Arrow Functions**: Used in array methods and promises
- **Default Parameters**: In the `Book` constructor
- **ES6 Modules**: Import/export syntax
- **Promises**: Promisifying readline and file operations
- **Error Handling**: Try/catch blocks with async functions
- **Array Methods**: `filter()`, `forEach()`, `join()`

Challenge Exercises

1. Add a search feature to find books by title or author
2. Implement a delete book functionality
3. Add data validation for duplicate books
4. Export the book list to different formats (CSV, XML)
5. Add book categories and filtering options

Next Steps

The books you add using this command-line tool will be available in your React application when you continue with the remaining labs!

Lab 06: Styling React

In this lab, you'll learn how to style your React components using CSS and inline styles.

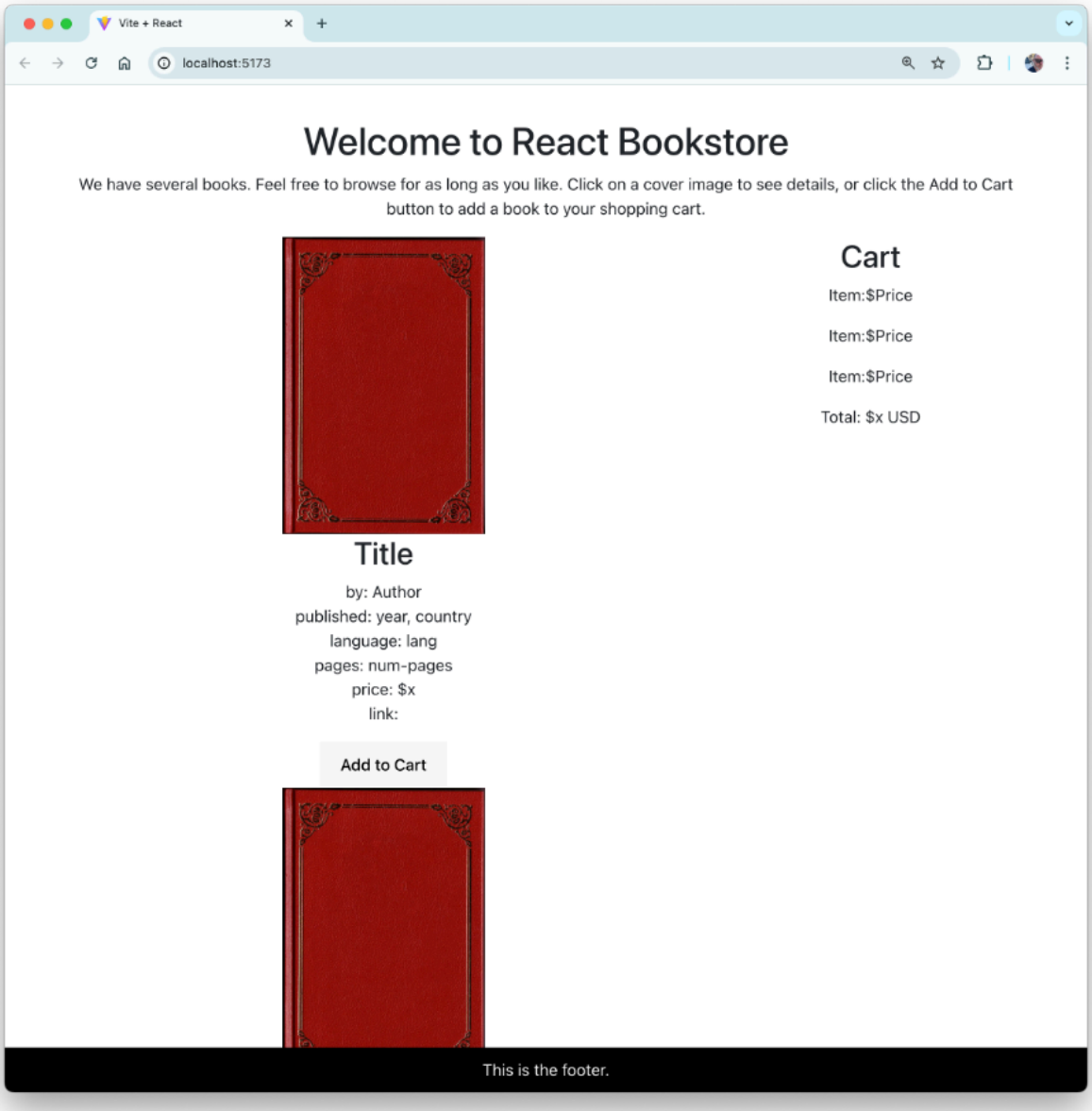
1. Open your React project in your code editor.
2. Create a new CSS file named `styles.css` in the `src` directory.
3. Add the following styles to `styles.css`:

```
body {
  font-family: Arial, sans-serif;
}

h1 {
  color: #333;
}

button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  border: none;
  cursor: pointer;
}

button:hover {
  background-color: #0056b3;
}
```

4. Import the `styles.css` file into your `App.jsx` file:

```
import './styles.css';
```

Your application should now show the updated styling:

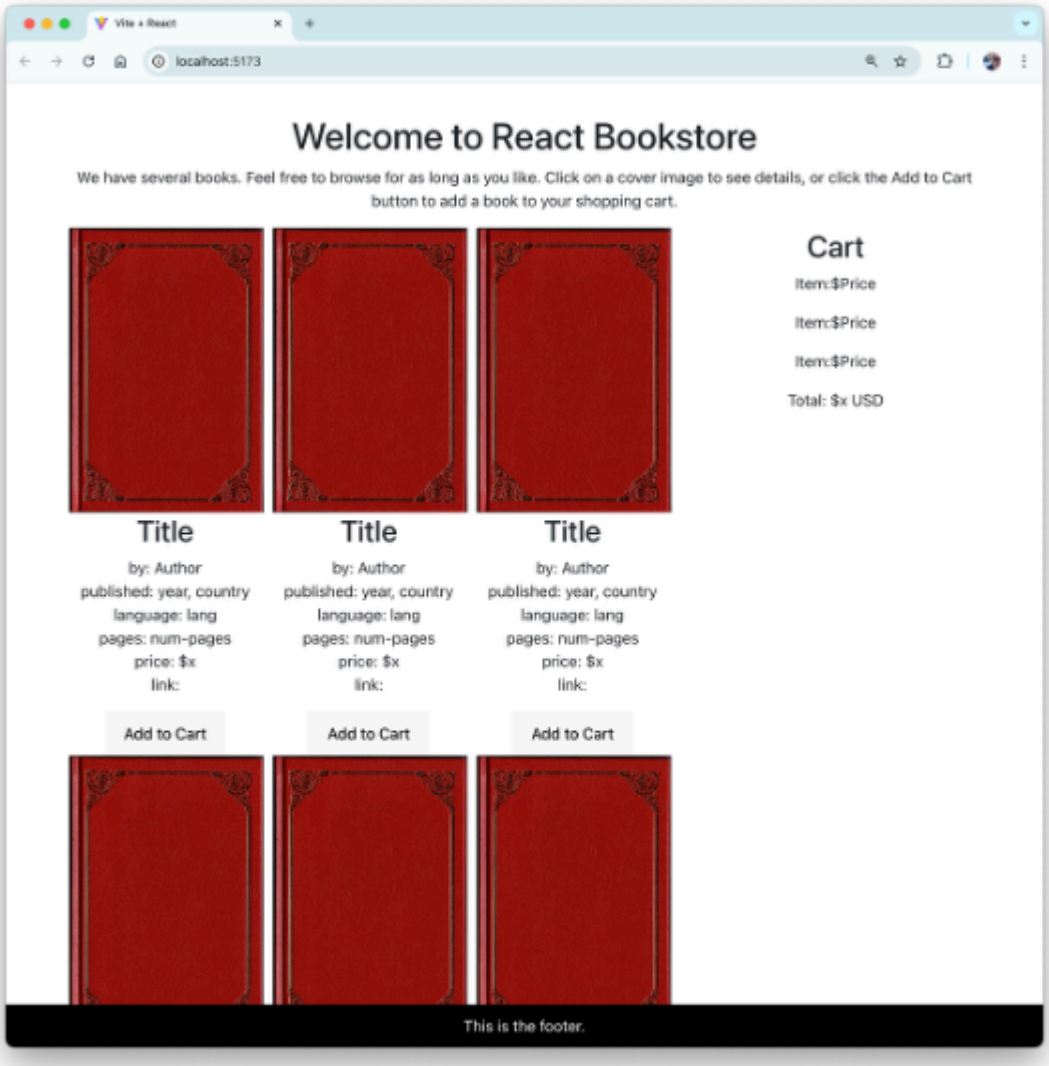


5. Add inline styles to one of your components. For example, modify the `Footer.jsx` component to include inline styles:

```
function Footer() {
  const footerStyle = {
    backgroundColor: 'black',
    color: 'white',
    textAlign: 'center',
    padding: '1rem',
    position: 'fixed',
    left: '0',
    bottom: '0',
    width: '100%',
  };

  return <footer style={footerStyle}>This is the footer.</footer>;
}

export default Footer;
```

6. Run your application and verify that the styles are applied correctly.

   Your footer should now appear styled with the inline styles:



7. Experiment with adding more styles to your components and CSS file.

Lab 07: Props and Containers

In this lab, you'll learn how to use props to pass data between components and how to create container components.

1. Open your React project in your code editor.
2. Create a new component named `Book.jsx` in the `src` directory:

```
function Book({ title, author }) {
  return (
    <div>
      <h2>{title}</h2>
      <p>{author}</p>
    </div>
  );
}

export default Book;
```

3. Create a new component named `BookList.jsx` in the `src` directory:

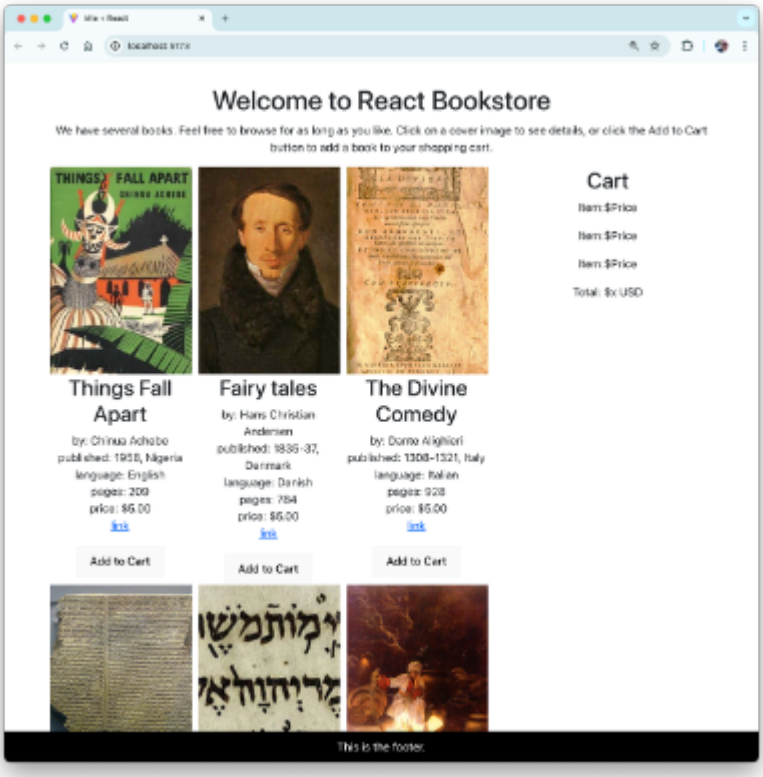
```
import Book from './Book';

function BookList() {
  const books = [
    { id: 1, title: 'The Great Gatsby', author: 'F. Scott Fitzgerald' },
    { id: 2, title: 'To Kill a Mockingbird', author: 'Harper Lee' },
    { id: 3, title: '1984', author: 'George Orwell' },
    { id: 4, title: 'The Catcher in the Rye', author: 'J.D. Salinger' },
    { id: 5, title: 'The Hobbit', author: 'J.R.R. Tolkien' },
    { id: 6, title: 'Fahrenheit 451', author: 'Ray Bradbury' },
    { id: 7, title: 'Jane Eyre', author: 'Charlotte Brontë' },
    { id: 8, title: 'Wuthering Heights', author: 'Emily Brontë' },
    { id: 9, title: 'Pride and Prejudice', author: 'Jane Austen' },
    { id: 10, title: 'Moby-Dick', author: 'Herman Melville' },
  ];

  return (
    <div>
      {books.map((book) => (
        <Book key={book.id} title={book.title} author={book.author} />
      ))}
    </div>
  );
}

export default BookList;
```

4. Import and use the `BookList` component in your `App.jsx` file:

```
import BookList from './BookList';

function App() {
  return (
    <div>
      <h1>Welcome to React Bookstore</h1>
      <BookList />
    </div>
  );
}

export default App;
```

5. Run your application and verify that the list of books is displayed correctly.

   Your application should now display the book list with props being passed correctly:



6. Experiment with adding more books to the `books` array in `BookList.jsx`.

Lab 08: Adding State

In this lab, you'll learn how to use React's `useState` hook to add state to your components.

1. Open your React project in your code editor.
2. Create a new component named `Counter.jsx` in the `src` directory:

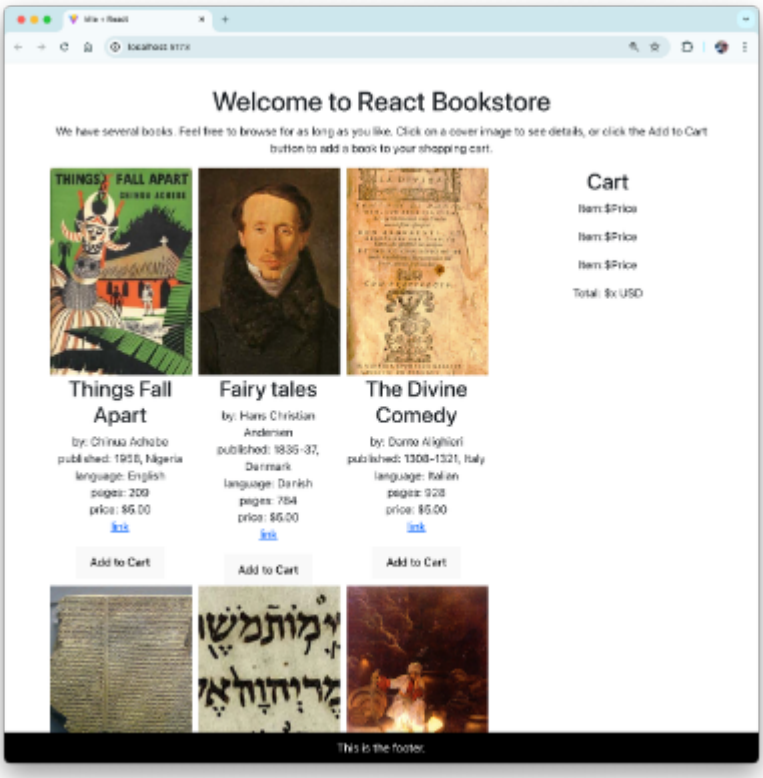
```
import {useState} from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}
export default Counter;
```

3. Import and use the `Counter` component in your `App.jsx` file:

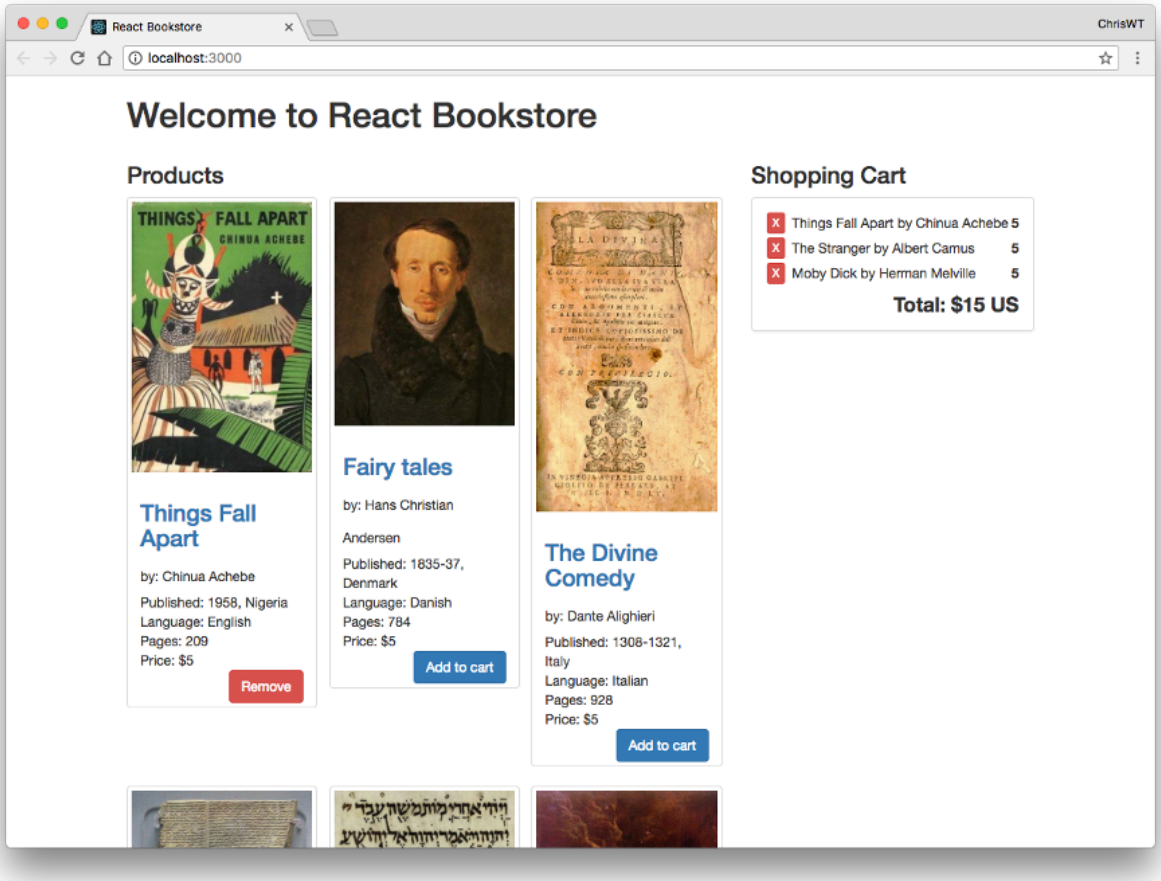
```
import Counter from './Counter';
function App() {
  return (
    <div>
      <h1>Welcome to React Bookstore</h1>
      <Counter />
    </div>
  );
}
export default App;
```

4. Run your application and verify that the counter works correctly.

   You should see the counter component with functioning buttons:



After clicking the increment button a few times, you should see the state updating:



5. Experiment with adding more state variables to the `Counter` component, such as a `step` variable to control the increment/decrement amount.

Lab 09: Interactions, Events, and Callbacks

In this lab, you'll learn how to handle user interactions and events in React components.

1. Open your React project in your code editor.
2. Create a new component named `Button.jsx` in the `src` directory:

```
function Button({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}
export default Button;
```

3. Create a new component named `ButtonGroup.jsx` in the `src` directory:

```
import Button from './Button';
function ButtonGroup() {
  return (
    <>
      <Button label="Button 1" onClick={() => handleClick('Button 1')} />
      <Button label="Button 2" onClick={() => handleClick('Button 2')} />
      <Button label="Button 3" onClick={() => handleClick('Button 3')} />
    </>
  );
}
export default ButtonGroup;
```

4. Import and use the `ButtonGroup` component in your `App.jsx` file:

```
import ButtonGroup from './ButtonGroup';
function App() {
  return (
    <div>
      <h1>React Bookstore</h1>
      <ButtonGroup />
    </div>
  );
}
export default App;
```

5. Run your application and verify that clicking each button displays the correct alert message.
6. Experiment with adding more buttons to the `ButtonGroup` component and handling different types of events, such as mouse hover or double-click.

Lab 10: Component Lifecycle and AJAX

In this lab, you'll learn about React's component lifecycle methods and how to make AJAX requests in a React application.

1. Open your React project in your code editor.
2. Create a new component named `UserList.jsx` in the `src` directory:

```
import { useEffect, useState } from 'react';
function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(data => setUsers(data));
  }, []);
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
export default UserList;
```

3. Import and use the `UserList` component in your `App.jsx` file:

```
import UserList from './UserList';
function App() {
  return (
    <div>
      <h1>Welcome to React Bookstore</h1>
      <UserList />
    </div>
  );
}
export default App;
```

4. Run your application and verify that the list of users is displayed correctly.
5. Experiment with adding more lifecycle-related functionality, such as cleaning up resources or handling errors in the `useEffect` hook.

Lab 11: Converting App to a Function Component

In this lab, you'll convert a class-based React component into a function component using hooks.

1. Open your React project in your code editor.
2. Create a new file named `ClassApp.jsx` in the `src` directory and add the following class-based component:

```
import React, { Component } from 'react';
class ClassApp extends Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Hello, Class Component!' };
  }
  updateMessage = () => {
    this.setState({ message: 'Hello, Function Component!' });
  }
  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <button onClick={this.updateMessage}>Update Message</button>
      </div>
    );
  }
}
export default ClassApp;
```

3. Create a new file named `FunctionApp.jsx` in the `src` directory and convert the class-based component into a function component:

```
import { useState } from 'react';
function FunctionApp() {
  const [message, setMessage] = useState('Hello, Function Component!');
  const updateMessage = () => {
    setMessage('Hello, Function Component!');
  };
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={updateMessage}>Update Message</button>
    </div>
  );
}
export default FunctionApp;
```

4. Import and use the `FunctionApp` component in your `App.jsx` file:

```
import FunctionApp from './FunctionApp';
function App() {
  return (
    <div>
      <h1>React Bookstore</h1>
      <FunctionApp />
    </div>
  );
}
export default App;
```

5. Run your application and verify that the function component works correctly.
6. Experiment with adding more hooks to the `FunctionApp` component, such as `useEffect` or `useReducer`.

Lab 12: Creating and using a color theme context

React's Context API gives you a way to bypass passing data via props. Context is useful for data that's needed in many different components or that are deeply nested in the component hierarchy. In this lab, you'll create a color theme switcher.

Instructions

1. Create a new folder named context inside src.
2. Create a new file named ThemeContext.jsx in your src/context directory.
3. Import createContext and useState into ThemeContext.jsx.

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

4. Use the createContext function to create a new context.

```
const ThemeContext = createContext();
```

5. Create a provider component that will hold the state for the theme and provide it to the rest of the app.

```
const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  const toggleTheme = () => {
    setTheme(theme === 'light' ? 'dark' : 'light');
  };
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export { ThemeProvider, ThemeContext };
```

6. In your main.jsx file, wrap your app with the ThemeProvider to make the theme context available throughout the app.

```
import { ThemeProvider } from './context/ThemeContext';
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <ThemeProvider>
      <App />
    </ThemeProvider>
  </React.StrictMode>
);
```

   Now you can use the ThemeContext in any component by importing the useContext hook and the ThemeContext. Let's try it with the footer.

7. Import useContext and ThemeContext into Footer.jsx.

```
import React, { useContext } from 'react';
import { ThemeContext } from '../context/ThemeContext';
```

8. Inside Footer(), pass the ThemeContext to useContext and destructure it to get the theme and the toggleTheme function.

```
const { theme, toggleTheme } = useContext(ThemeContext);
```

9. Modify the footerStyle object to switch the backgroundColor and the color properties between white and black based on the value of theme.

```
const footerStyle = {
  [illegible] theme === 'light' ? 'white' : 'black',
  [illegible]
};
```

10. Add a button to the footer that will toggle the theme:

```
return (
  [illegible]
  <button onClick={toggleTheme}>Toggle Theme</button>
  [illegible]
);
```

11. Run `npm run dev` and test it out!
12. See if you can figure out how to modify the theme of other components when you toggle the theme using the button in the Footer.
13. Run your tests and notice that the tests for App and Footer fail. To fix them, you need to import the ThemeProvider into their test files and wrap the element you're rendering with it, for example:

```
describe('Footer Component', () => {
  [illegible]</ThemeProvider>);
  [illegible]
});
```

Lab 13: Custom Hooks

Custom hooks aren't a built-in feature of React. Instead, they're a technique. A custom hook is a function that uses other hooks (such as useState and useEffect) to make reusable functionality available to components.

Follow these steps to create a custom hook to simplify the process of using the ThemeContext.

Instructions

1. Open contexts/ThemeContext.jsx
2. Import useContext from react.
3. After the export statement for the ThemeProvider create a new module for the theme hook:

```
export const useTheme = () => {
  const context = useContext(ThemeContext);
  if (context === undefined) throw new Error('useTheme must be used within a ThemeProvider');
  return context;
};
```

4. Remove the imports of useContext and ThemeContext from Footer.jsx and replace it with the import for useTheme():

```
import {useTheme} from '../contexts/ThemeContext';
```

5. Remove the call to useContext and replace it with a call to useTheme():

```
const { theme, toggleTheme } = useTheme();
```

6. Run `npm run dev` and test it out.

   Hooks that aren't connected to Context are usually kept in a folder in src named hooks. For example, you could create a custom hook from the fetchData() function in App.jsx or from the shuffleData function. Doing so will simplify App and potentially make this functionality available to other components that may need it.

7. Create the src/hooks folder.
8. Create a new file in /hooks named useBooks.js

   Notice that hooks aren't typically named with .jsx. This is because they don't return JSX.

9. Try to extract the fetchData functionality from App.jsx into a function named useBooks() that returns an array containing the products array and the isLoading status.

   Check the solution file if you need help!

10. Remove the useState() and useEffect function calls from App.
11. Import useBooks() into App.jsx and use it to get the values of products and isLoading.

12. **Challenge:** Create a useCart() hook that exports the itemsInCart array, the addToCart function, and the removeFromCart function.

Lab 14: Converting to TypeScript

In this lab, you'll convert the React bookstore to use TypeScript.

Instructions

1. Install Typescript

```
npm install typescript --save-dev
```

2. Install types for React and ReactDom

```
npm install @types/react @types/react-dom --save-dev
```

3. In package.json, modify the dev and build scripts to compile the Typescript before running the app:

```
"dev": "tsc && vite",
"build": "tsc && vite build",
```

4. Create a file named tsconfig.json at the root of your project, with the following content:

```
{
  "compilerOptions": {
    "target": "ESNext",
    "useDefineForClassFields": true,
    "lib": ["DOM", "DOM.Iterable", "ESNext"],
    "allowJs": false,
    "skipLibCheck": true,
    "esModuleInterop": false,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "ESNext",
    "moduleResolution": "Node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": ["src"]
}
```

5. Rename App.jsx to App.tsx and update the link to App from main.js.

   You should now see a lot of red underlining in VS Code.

6. Create a new folder named types inside src.
7. Make a file in types named book.tsx and create an interface for Book:

```
export interface Book {
  id: number;
  title: string;
  author: string;
  price: string;
}
```

8. Import the Book type from types/book.tsx into App.tsx.
9. Assign types to any state variables that you haven't extracted into custom hooks yet:

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

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

```
function addToCart(product: Book) {
```

11. Test out the app to make sure it still works.
12. Rename ProductList.jsx to ProductList.tsx
13. Install the CSS modules typescript plugin

```
npm install typescript-plugin-css-modules --save-dev
```

14. Rename Footer.jsx to Footer.tsx
15. Add the following import to Footer.tsx:

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

16. Annotate footerStyle using the CSSProperties interface:

```
const footerStyle: CSSProperties =
```

17. Add the css modules typescript plugin to the compilerOptions object in tsconfig.json:

```
"plugins": [{ "name": "typescript-plugin-css-modules"}]
```

18. Create a new file named Globals.d.ts inside src, with the following content:

```
declare module '*.module.css';
```

19. Create a new file in /types named product.ts and make an interface called ProductProps that contains the props received by ProductList.

```
import {Book} from '../book';
export interface ProductProps {
  addToCart: (book: Book) => void;
```

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;
```

20. Import ProductProps into ProductList and specify the type of the props object received by ProductList:

```
function ProductList(props: ProductProps) {
```

21. Convert the rest of the components to TypeScript.

Lab 15: Testing with React Testing Library

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

Instructions

1. Create a new directory outside of src and name it tests. Create a new file named setup.ts in tests with the following content:

```
import '@testing-library/react/vitest';
```

2. Create a file named vitest.config.ts at the root of your project, with the following content:

```
import { defineConfig } from 'vitest/config';
import tsconfigPaths from 'vite-tsconfig-paths';

export default defineConfig({
  plugins: [tsconfigPaths()],
});
```

3. Install some dependencies.

```
npm install --save-dev @vitejs/plugin-react
```

4. Modify your test script in package.json:

```
"test": "vitest --ui",
```

5. Run `npm test` in the root of your project. The results will appear in the terminal and the vitest ui will open in a browser window.

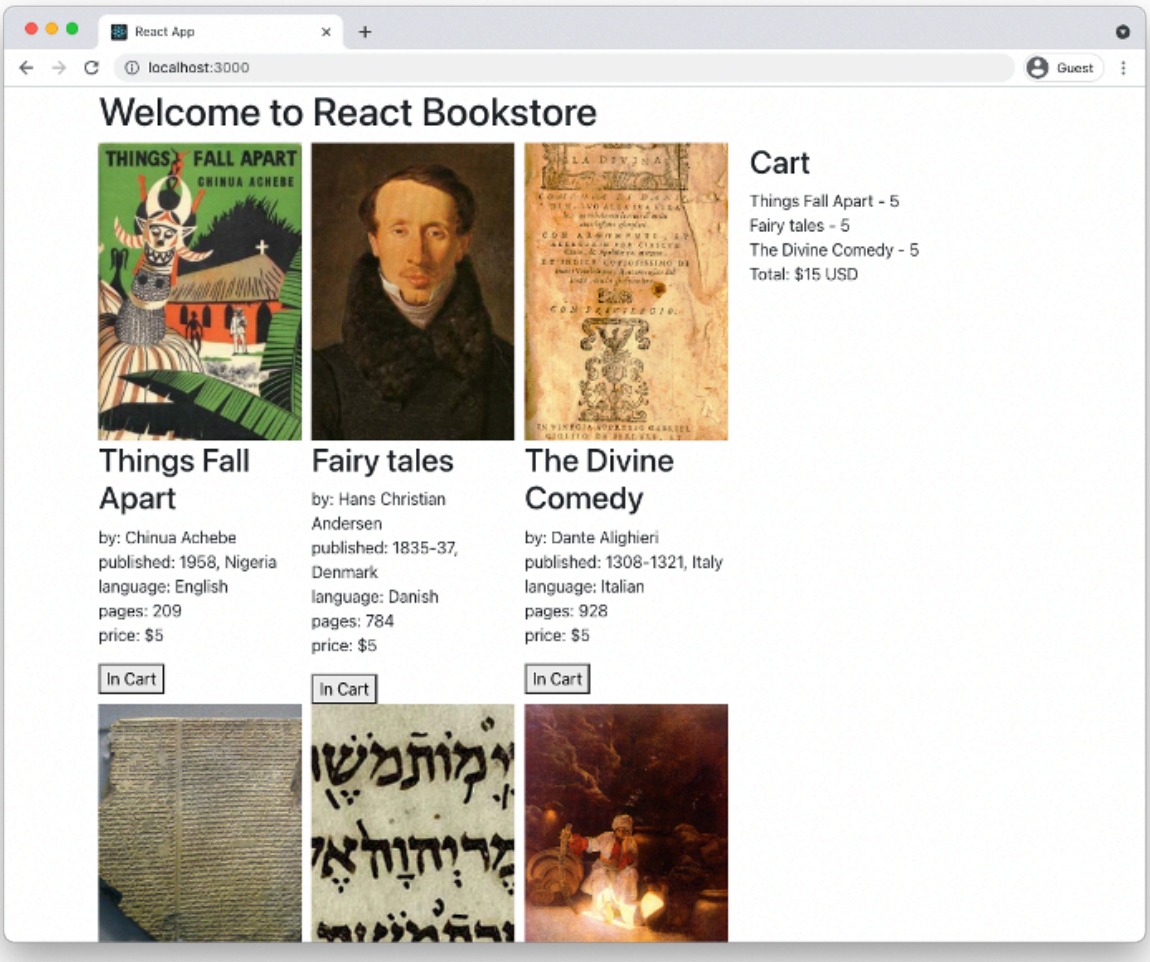
   You probably already have some basic tests that you created in previous labs. However, because of changes to the components you're testing, your tests may not all pass.

6. 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

7. When all your tests pass, you'll see a coverage report in the terminal, and you can access a coverage report in Vitest UI.

   The Vitest UI should display your test results and coverage information:



8. Write more tests, refine your existing tests, and increase your test coverage %. See the solution for more ideas of tests to write and how to write them.

Lab 16: 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.

Instructions

1. Install Redux, the react-redux library, and Redux toolkit.

```
npm install --save redux react-redux @reduxjs/toolkit
```

Step 1: Create a store

1. In main.tsx, import configureStore from Redux toolkit.

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

2. In main.tsx, configure the store:

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

3. Import combineReducers from redux.

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

4. Import Provider from react-redux.

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

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

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

6. Create the root reducer.

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

7. 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.

3. 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 cartItems(state = { items: [] }, action = {}){
}
```

4. Inside the CART_ADD case, use the functionality from the addToCart function (in **App.tsx**) to add the productId passed by the action to the items array.

```
case 'CART_ADD':
  return [...state.items, action.payload],
```

5. 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.filter(
```

6. 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.

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

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(product) {
  return {type: 'ADD_TO_CART', product};
}

export function removeFromCart(productId) {
  return {type: 'REMOVE_FROM_CART', 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.

3. Import the action creator functions, the connect method of react-redux, and the bindActionCreator method into **pages/App.tsx**.

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

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

```
function mapStateToProps(state: any, props: any) {
  return {
    cart: state.cart
  };
}

function mapDispatchToProps(dispatch: any, props: any) {
  return bindActionCreators(actionCreators, dispatch);
}
```

5. 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.tsx to use the Redux store.

1. In **App.tsx**, remove the following:

- The useState function calls for itemsInCart and for products.
- The addToCart method
- The removeFromCart method

1. Make sure the props object is a parameter of App

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

2. Update the useEffect method to call the loadProducts method and pass it the products array from the custom hook. Make sure to put products into the dependency array so the effect with run when the custom hook returns the data.

```
useEffect(() => {
  loadProducts(products);
}, [products]);
```

3. Modify the props passed to the ProductList and Cart components to use the props that are passed in from the connection to the Redux store.

```
<div className="col-md-8">
  <ProductList products={products} addToCart={addToCart} />
</div>
<Cart itemsInCart={props.itemsInCart} />
```

4. Fix any typescript errors that you have.
5. Try converting reducers/index.js and actions/index.js to TypeScript.

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

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

Step 5: Looking at Redux Toolkit

1. Redux Toolkit has a standard Vite template that demonstrates some best practices and patterns for using Redux. Run the following command outside of your react-bookstore folder to create a project with the Redux Toolkit template.

   ```
   npx degit reduxjs/redux-templates/packages/vite-template-redux my-app
   ```

2. CD into the new project and run `npm install`.
3. Run `npm run dev` to see it running, then look through the code at how it works. Can you apply some of the other Redux Toolkit methods and conventions to the React Bookstore?

Lab 17: 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.

Instructions

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.js:

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

4. In **actions/index.js**, 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. Pass the submitCart action creator from App to the Cart.

```
<Cart removeFromCart={props.removeFromCart} submitCart = {props.submitCart} itemsInCart={
```

6. 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.itemsInCart)}>Check Out</button></div>
```

7. 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.
8. Open a new terminal window and change to the **starter/lab17/server** directory.
9. Run `npm install` in the server directory
10. Run the server by entering `npm start`.

11. 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.

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

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

13. 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: [],
```

14. 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 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.

Instructions

1. Add a new action creator to actions/index.js for reading the cart from localStorage:

```
export function readCartFromLocalStorage() {
  return { type: 'READ_CART', payload: getItem('cart') };
}
```

2. Create a new reducer case for "READ_CART".

```
case 'READ_CART':
  return action.payload || [],
```

3. Import readCartFromLocalStorage into useBooks.
4. Import useDispatch from react-redux into useBooks.
5. Call useDispatch inside useBooks() to return the dispatch function.

```
const dispatch = useDispatch();
```

6. After the products are loaded, call dispatch(readCartFromLocalStorage);
7. 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.

8. After the line in main.tsx that creates the store, subscribe localStorage to the store, so that whenever the store changes the cart will be written to localstorage:

```
store.subscribe(() => {
  setItem('cart', store.getState().cart.items)
});
```

9. Run and test. Can you now add books and remove books and have them persist each time you re-visit the bookstore? If so, you've got it right!

Lab 19: React Router

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

Instructions

1. Install react-router-dom

```
npm install react-router-dom
```

2. Import BrowserRouter as Router into **main.tsx** and wrap the Router component around the `<Provider>` element in createRooter.render()
3. In **App.tsx**, import Routes and Route from react-router-dom.
4. In App'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>
</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. Import Link from react-router-dom into Header and create a nav bar that links to the home page and to the cart.
8. **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 20: Microfrontends with Single SPA

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

Instructions

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": {
        "react": "https://cdnjs.cloudflare.com/ajax/libs/react/17.0.2/umd/react.production.min.j
        "react-dom": "https://cdnjs.cloudflare.com/ajax/libs/react-dom/17.0.2/umd/react-dom.produ
      }
    }</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.