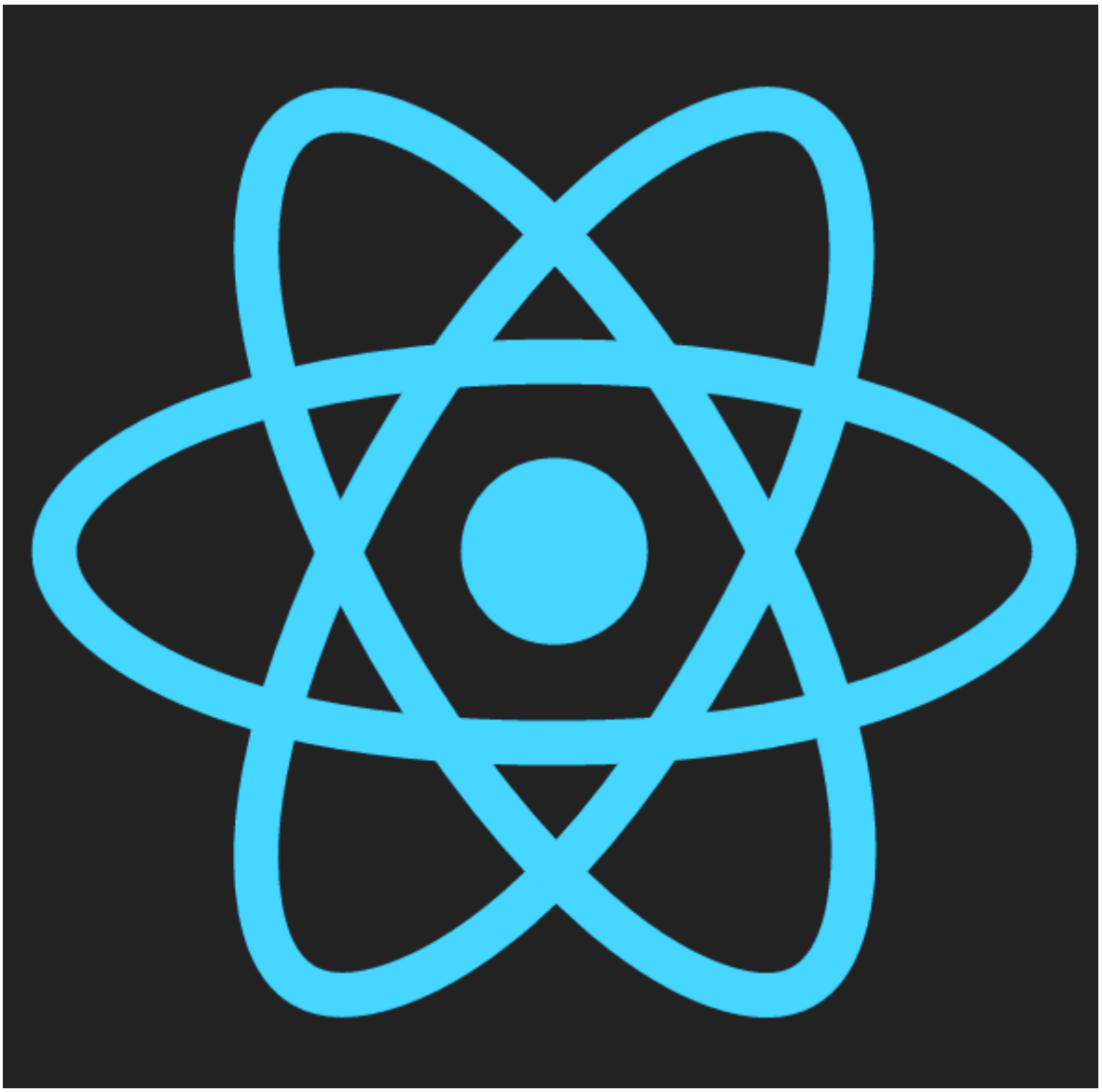
Professional ReactJS



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

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

Version 19.3.0, October 2025
by Chris Minnick

www.watzthis.com

Table of Contents

Disclaimers and Copyright Statement

Disclaimer

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

Third-Party Information

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

Copyright

Help us improve our courseware

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

Credits

About the Author

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

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

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

Course Requirements

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

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

Introduction and Git Repo Info

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

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

The url for the course repository is:
`https://github.com/chrisminnick/professional-reactjs`

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

Yarn or npm?

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

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

Lab 01: Get Started with 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.

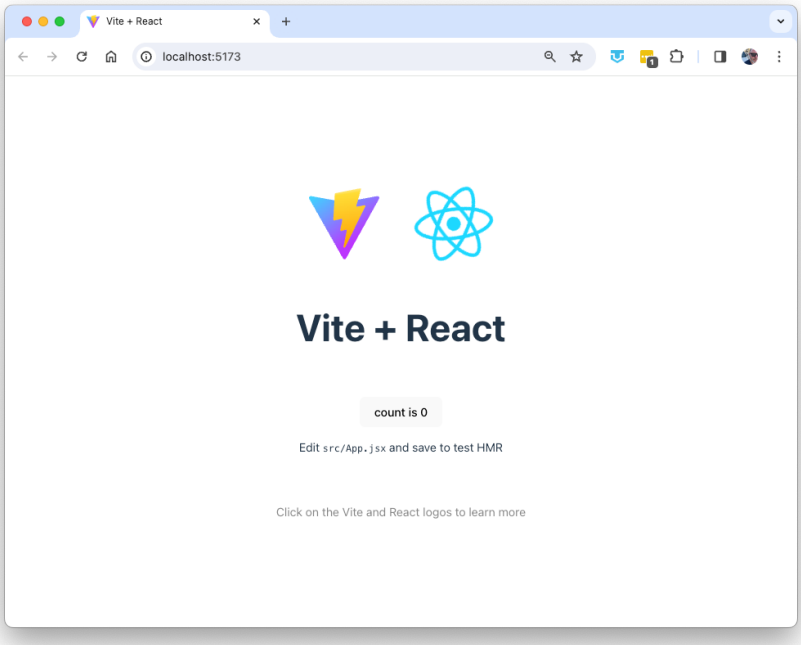
```
npm install
```

☐ 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 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 image to see details, or click
the Add to Cart button to add a book to your shopping cart.
```

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

```
#root {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}
```

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

```
body {
  margin: 0;
  display: flex;
  place-items: center;
  min-width: 320px;
  min-height: 100vh;
}
```

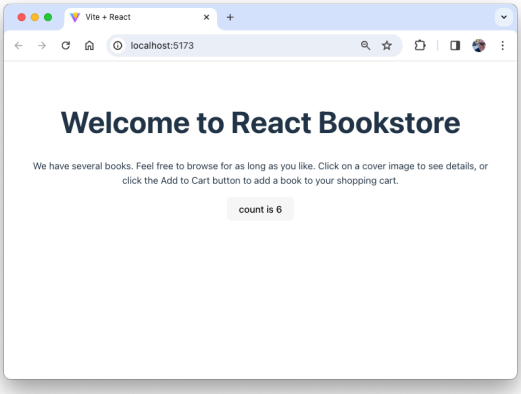
☐ 13. Return to your web browser and notice that the text has been automatically refreshed (if your app is still running).

If it's 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

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.jsx** in the **src** directory

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

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

export default Footer;
```

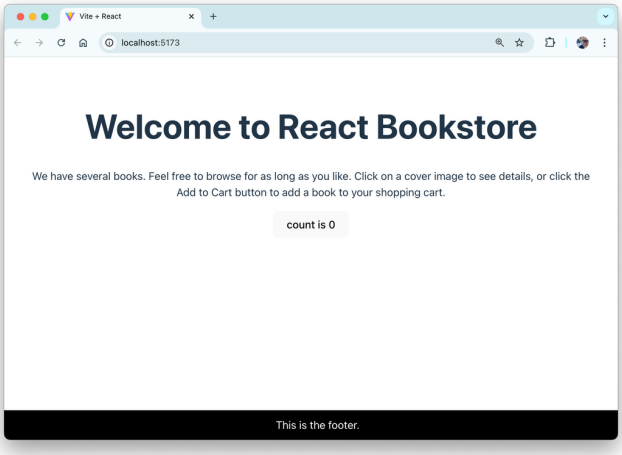
☐ 3. Add the following to the beginning of **App.jsx**

```
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.



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">
    <Header />
    <ClickCounter />
    <Footer />
  </div>
);
```

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
npm install --save-dev jsdom
npm install --save-dev @testing-library/jest-dom
npm install --save-dev @testing-library/react
npm install --save-dev @vitest/coverage-v8
```

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

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
export default defineConfig({
  plugins: [react()],
  test: { globals: true, environment: 'jsdom', },
});
```

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

```
"scripts": {
  "dev": "vite",
  "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, expect, it } from 'vitest';
import { render, screen } from '@testing-library/react';
import '@testing-library/jest-dom/vitest';
import App from './App.jsx';

describe('App Component', () => {
  it('Renders', () => {
    render(
        <App />
    );
    let element = screen.getByText(/Bookstore/i);
    expect(element).toBeInTheDocument();
  });
});
```

☐ 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.

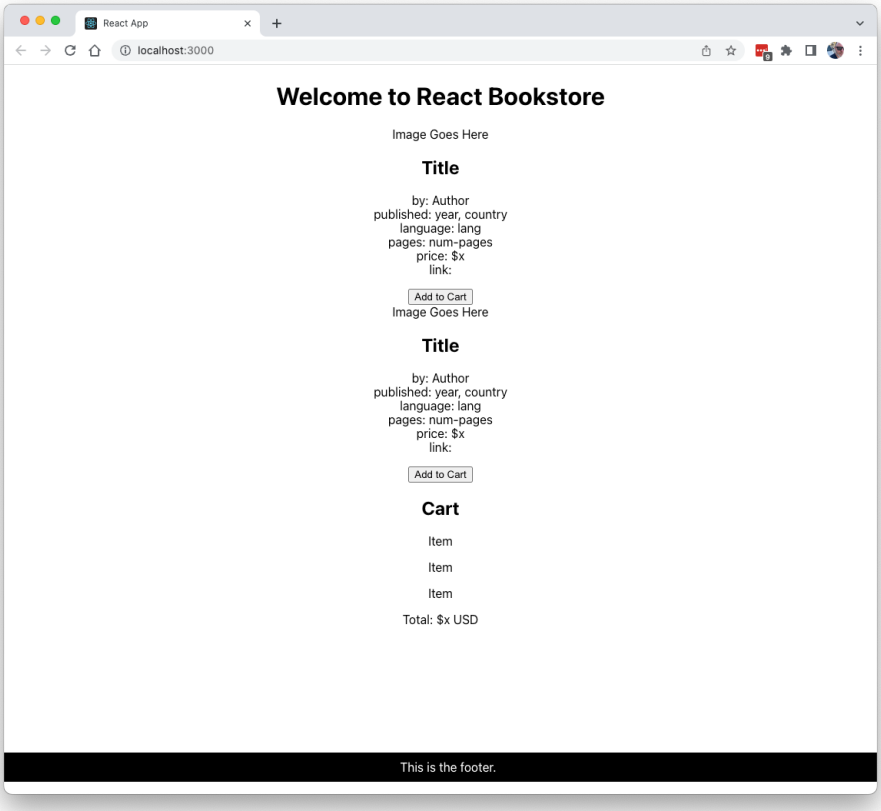
☐ 7. Open the **App.jsx** component in your project and modify it to the following.

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.jsx';
import ProductList from './ProductList.jsx';
import Cart from './Cart.jsx';
import Footer from './Footer.jsx';
import './App.css';

function App() {
    return (
      <>
            <Header />
            <ProductList />
            <Cart />
            <Footer />
      </>
    );
}
```

```
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 the react-bookstore project directory and create a new folder called tools :

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

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

☐ 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 fs from 'fs/promises';
import path from 'path';
import readline from 'readline';
import { fileURLToPath } from 'url';
```

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

```
// Get current directory path for ES modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
// Path to the products.json file
const PRODUCTS_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 {
  constructor({
    title,
    author,
```

```
    country,
    language,
    pages,
    published,
    price,
    image = '',
    url = '',
  }) {
    this.id = this.generateId();
    this.title = title;
    this.author = author;
    this.country = country || 'Unknown';
    this.language = language || 'English';
    this.pages = pages;
    this.published = published;
    this.price = price;
    this.image = image;
    this.url = url;
  }

  generateId() {
    return
`book_${Date.now()}_${Math.random().toString(36).substr(2,
9)}`;
  }

  // Method to validate book data
  validate() {
    const required = ['title', 'author', 'pages',
'published', 'price'];
    const missing = required.filter((field) =>
!this[field]);

    if (missing.length > 0) {
      throw new Error(`Missing required fields:
${missing.join(', ')}`);
    }

    if (isNaN(this.pages) || this.pages <= 0) {
      throw new Error('Pages must be a positive number');
    }

    if (isNaN(this.price) || this.price <= 0) {
      throw new Error('Price must be a positive number');
    }
  }

  // Method to format book for display
  toString() {
    return `"${this.title}" by ${this.author}
(${this.published}) - $${this.price}`;
  }
```

```
}
```

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

```
class BookManager {
  constructor(filePath) {
    this.filePath = filePath;
  }
  // Read existing books from JSON file
  async loadBooks() {
    try {
      const data = await fs.readFile(this.filePath, 'utf8');
      return JSON.parse(data);
    } catch (error) {
      if (error.code === 'ENOENT') {
        console.log('Products file not found. Creating new
file...');
        return [];
      }
      throw new Error(`Error reading books file:
${error.message}`);
    }
  }
  // Save books to JSON file
  async saveBooks(books) {
    try {
      const jsonData = JSON.stringify(books, null, 2);
      await fs.writeFile(this.filePath, jsonData, 'utf8');
      console.log(
        `Successfully saved ${books.length} books to
${this.filePath}`
      );
    } catch (error) {
      throw new Error(`Error saving books: ${error.message}`);
    }
  }
  // Add a new book to the collection
  async addBook(bookData) {
    try {
      const book = new Book(bookData);
      book.validate();
      const books = await this.loadBooks();
      books.push(book);
      await this.saveBooks(books);
      console.log(`Added new book: ${book.toString()}`);
      return book;
    } catch (error) {
      throw new Error(`Failed to add book: ${error.message}`);
    }
  }
  // Display all books
  async listBooks() {
    try {
```

```
      const books = await this.loadBooks();
      if (books.length === 0) {
        console.log('No books found in the collection.');
        return;
      }
      console.log('\nCurrent Book Collection:');
      console.log('='.repeat(50));
      books.forEach((book, index) => {
        console.log(`${index + 1}. ${book.title}`);
        console.log(`   Author: ${book.author}`);
        console.log(
          `   Published: ${book.published} | Pages:
${book.pages} | Price: $${book.price}`);
        console.log('');
      });
      console.log(`Total books: ${books.length}`);
    } catch (error) {
      console.error(`Error listing books: ${error.message}`);
    }
  }
}
```

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

```
class BookCLI {
  constructor() {
    this.rl = readline.createInterface({
      input: process.stdin,
      output: process.stdout,
    });
    this.manager = new BookManager(PRODUCTS_FILE);
  }
// Promisify readline question
  question(prompt) {
    return new Promise((resolve) => {
      this.rl.question(prompt, resolve);
    });
  }
  // Main menu
  async showMenu() {
    console.log('\nBook Manager CLI');
    console.log('==================');
    console.log('1. Add a new book');
    console.log('2. List all books');
    console.log('3. Exit');
    const choice = await this.question('\nEnter your choice
(1-3): ');
    return choice.trim();
  }
  // Get book information from user
  async getBookInfo() {
    console.log('\nEnter book information:');
    const title = await this.question('Title: ');
```

```
    const author = await this.question('Author: ');
    const country = await this.question('Country
(optional): ');
    const language = await this.question('Language
(default: English): ');
    const pages = await this.question('Number of pages: ');
    const published = await this.question('Publication
year: ');
    const price = await this.question('Price ($): ');
    const imageUrl = await this.question('Image URL
(optional): ');
    const bookUrl = await this.question('Book URL
(optional): ');
    return {
      title: title.trim(),
      author: author.trim(),
      country: country.trim(),
      language: language.trim() || 'English',
      pages: parseInt(pages),
      published: parseInt(published),
      price: parseFloat(price),
      image: imageUrl.trim(),
      url: bookUrl.trim(),
    };
  }
  // Main application loop
  async run() {
    console.log('Welcome to the Book Manager CLI!');
    try {
      while (true) {
        const choice = await this.showMenu();
        switch (choice) {
          case '1':
            try {
              const bookData = await this.getBookInfo();
              await this.manager.addBook(bookData);
            } catch (error) {
              console.error(`Error: ${error.message}`);
            }
            break;
          case '2':
            await this.manager.listBooks();
            break;
          case '3':
            console.log('Goodbye!');
            this.rl.close();
            return;
          default:
            console.log('Invalid choice. Please try
again.');
        }
      }
```

```
    } catch (error) {
      console.error(`Application error: ${error.message}`);
    } finally {
      this.rl.close();
    }
  }
}
```

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

```
// Main execution with error handling
async function main() {
  try {
    const cli = new BookCLI();
    await cli.run();
  } catch (error) {
    console.error(`Fatal error: ${error.message}`);
    process.exit(1);
  }
}
// Execute only if this file is run directly
if (import.meta.url ===
pathToFileURL(process.argv[1]).href) {
  main();
}

// Export classes for potential reuse
export { Book, BookManager, BookCLI };
```

☐ 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.

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

Lab 06: Styling React

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

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

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

☐ 2. Link to **bootstrap.css** inside of **src/main.jsx.**

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

☐ 3. Change the <> and </> to and in **App.jsx**, and add the container class.

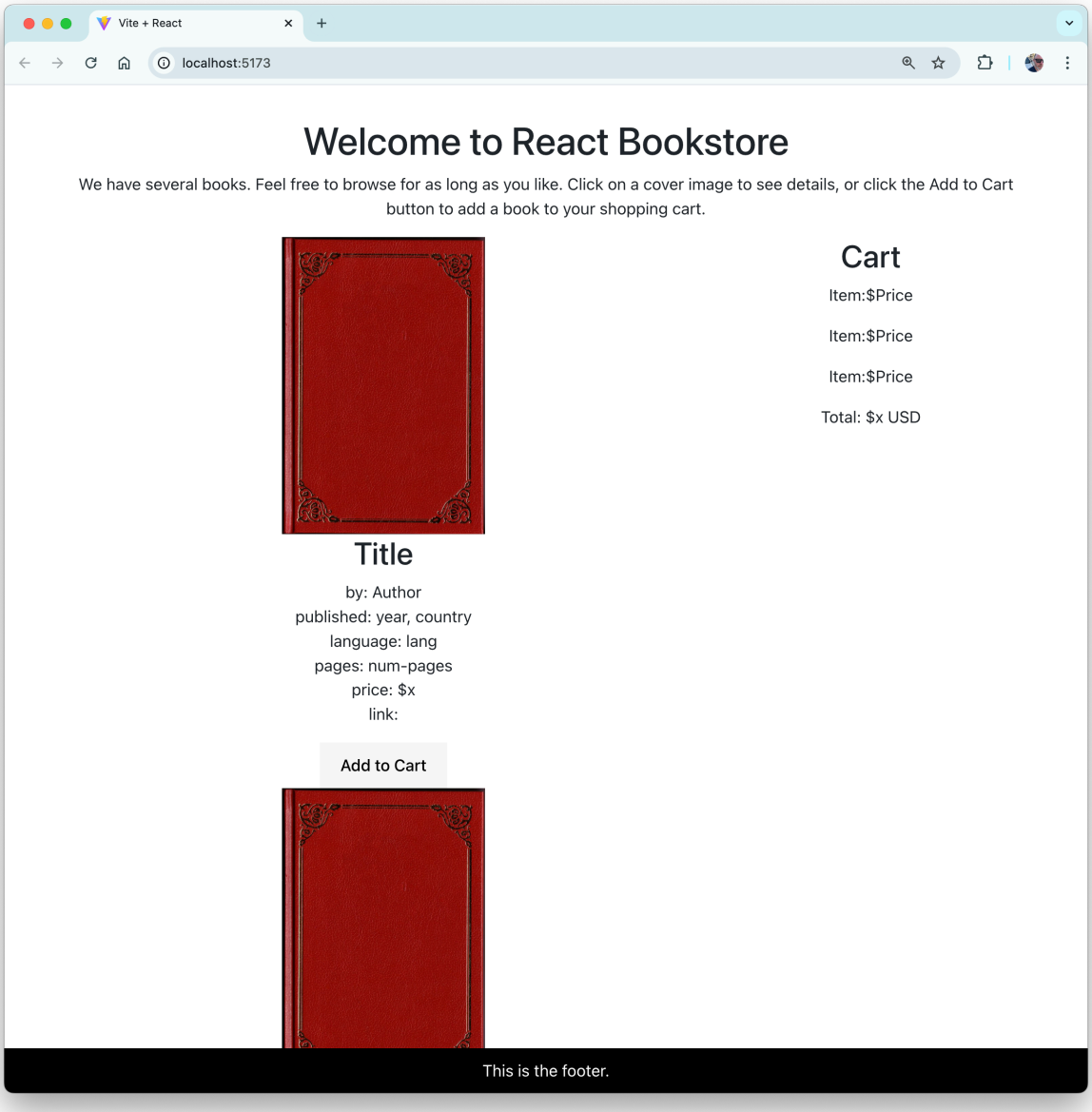
```
import Header from './Header.jsx';
import ProductList from './ProductList.jsx';
import Cart from './Cart.jsx';
import Footer from './Footer.jsx';
import './App.css';

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

export default App;
```

☐ 4. Surround the ProductList and Cart components with a with the class of "row". Create two columns in App.jsx using Bootstrap's responsive grid classes.

```
  return (
    <div className="container">
      <Header />
      <div className="row">
      <div className="col-md-8">
        <ProductList />
      </div>
      <div className="col-md-4">
        <Cart />
      </div>
      </div>
      <Footer />
    </div>
  );
```

☐ 5. Run `npm test` and make sure that your tests pass.

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



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

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

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

This will be our first style module.

☐ 9. 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.

☐ 10. Import the style module into `ProductList.jsx` and give the module the name `styles`.

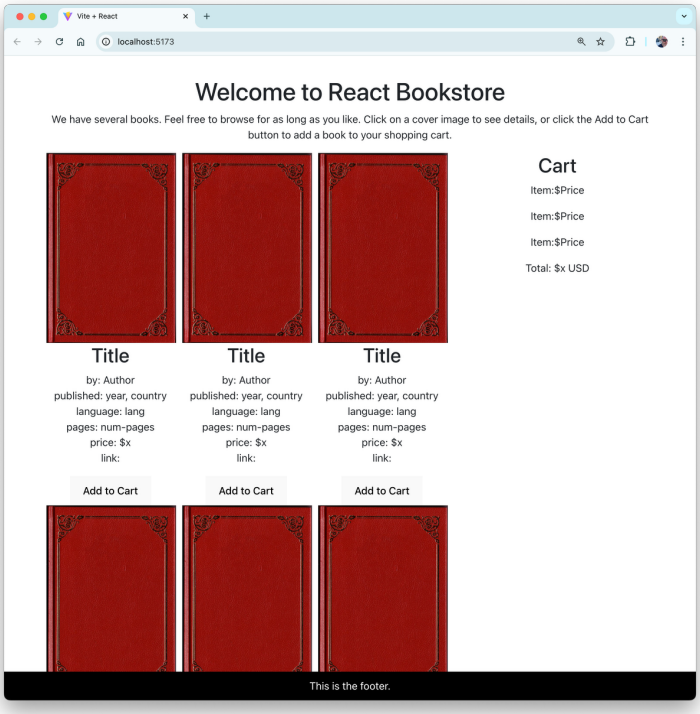
```
import styles from './ProductList.module.css';
```

☐ 11. Attach the styles to the appropriate elements.

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

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

☐ 13. Preview the styled list 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.
Cart
Item:$Price
Item:$Price
Item:$Price
Total: $x USD
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
link:
Add to Cart
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
link:
Add to Cart
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
link:
Add to Cart
This is the footer.

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
ProductList
Product
Cart
CartItem
```

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

☐ 1. Create a folder inside **src** named **components**

☐ 2. Move **Header.jsx**, **Footer.jsx**, **ProductList.jsx**, **Product.jsx**, **Cart.jsx**, and **CartItem.jsx**, along with their test suites and css files, into the **components** directory.

Going forward, you'll put any components that receive props from a page and render a part of a page in this folder.

☐ 3. Create another folder named **pages** and put **App.jsx** (and its tests and css) into it.

Going forward, you'll put components that have state and that correspond to 'pages' in this folder.

☐ 4. Import the data file into **App.jsx.**

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

☐ 5. Test that the JSON file was correctly imported by printing out one of its values. Inside the return statement of App.jsx, use the following code to write out the title of the first book in the data file:

```
{productsData[0].title}
```

☐ 6. If the data file was imported correctly, delete the testing code you wrote in the previous step.

☐ 7. Modify each of the subcomponents of App to accept a props object. For example, here's what the function header of the Product component should look like:

```
function Product(props) {
```

☐ 8. Pass `productsData` from the `App` component to the `ProductList` component as a prop called `products`.

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

☐ 9. 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>
        );
```

☐ 10. 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;
```

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

☐ 12. Create a file named Product.module.css and import it into Product.jsx.

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

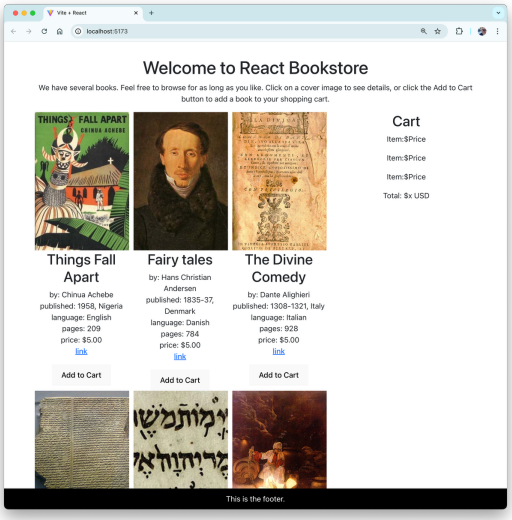
☐ 13. Create a new style rule in **Product.module.css** called thumbnail and set properties to format the book thumbnail images.

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

☐ 14. 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} />
```

☐ 15. Run `npm run dev`. It should look like this:



☐ 16. Run `npm run test` and notice that your tests for the ProductList and Product components are failing. Can you figure out how to make them pass? Think about it for a moment before moving on to the next step.

... thinking

..... still thinking

.....thinking some more

.....and some more

☐ 17. The testing framework renders components just like a browser does. If you don't provide the data that a component requires to render, the test will fail. To solve the problem, your tests need to pass the correct props to the component being tested.

☐ 18. In Product.test.js, create a mock product object inside the describe function (but above your test):

```
const book = {
  id: '1',
  title: 'Fake Title',
  price: '10',
  author: 'Author',
  published: 'Published Date',
  country: 'Country',
  lang: 'Language',
  pages: '100',
  image: 'image.jpg',
```

```
    url: 'Link URL',
    inCart: false,
    avgRating: '5',
}
```

☐ 19. Inside your test, use the spread operator to pass the all the properties of this book object to the Product component as props.

```
it('Renders', () => {
  render(<Product {...book} />);
  let element = screen.getByText(/Fake Title/i);
  expect(element).toBeInTheDocument();
});
```

☐ 20. Use this same technique to fix the ProductList test.

☐ 21. Run `npm run test` to make sure your tests pass.

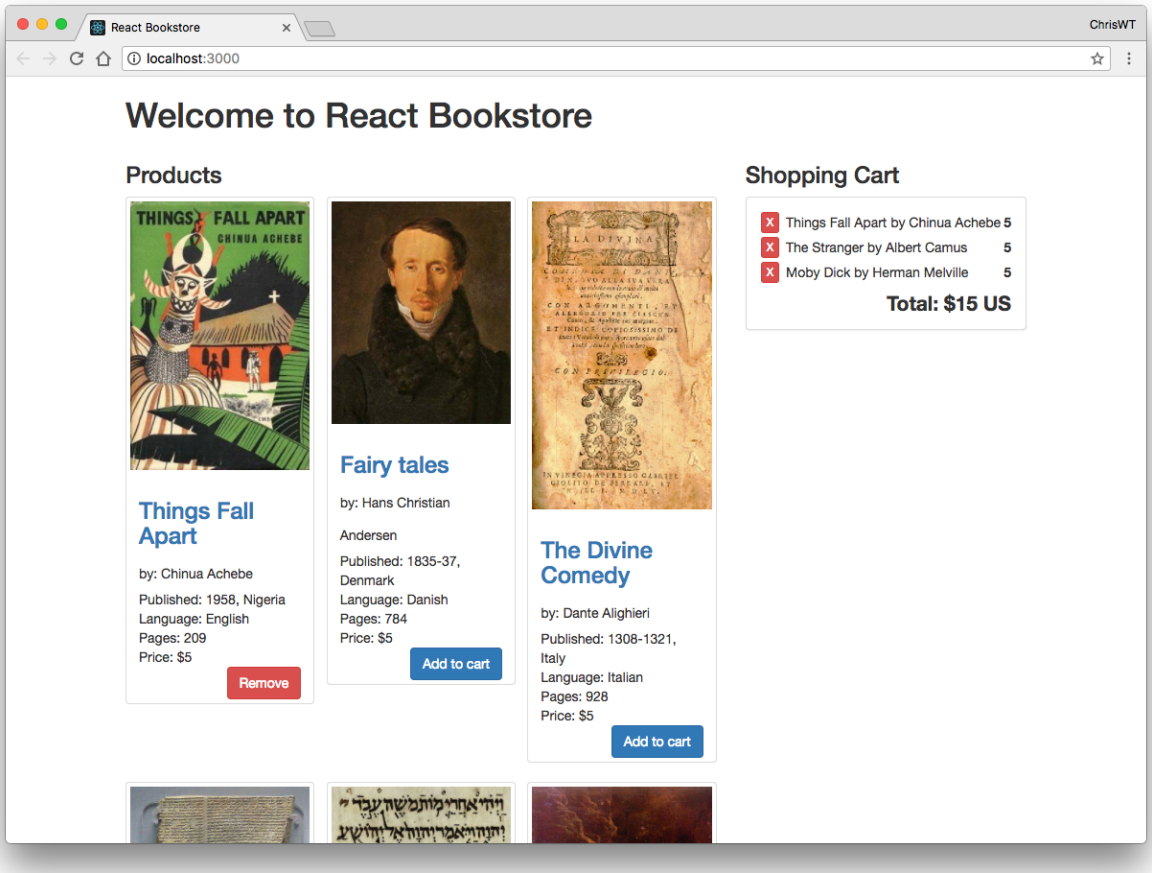
Lab 08: Adding State

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 only have one component that fits this description, `App`. So, we'll put the state into `App`.

Follow these steps to add state to the application.

☐ 1. Open **src/pages/App.jsx** in your code editor.
☐ 2. Import `React.useState`

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

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

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

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

```
<Cart itemsInCart={itemsInCart} />
```

☐ 5. Add a product to the `itemsInCart` array, for testing. We'll only be using the id, title and price in the Cart (to start, anyway), so you can just pass those three.

```
const [itemsInCart,setItemsInCart] = useState([
  {
    "id": "1",
    "title": "Things Fall Apart",
    "price": "5"
  }
]);
```

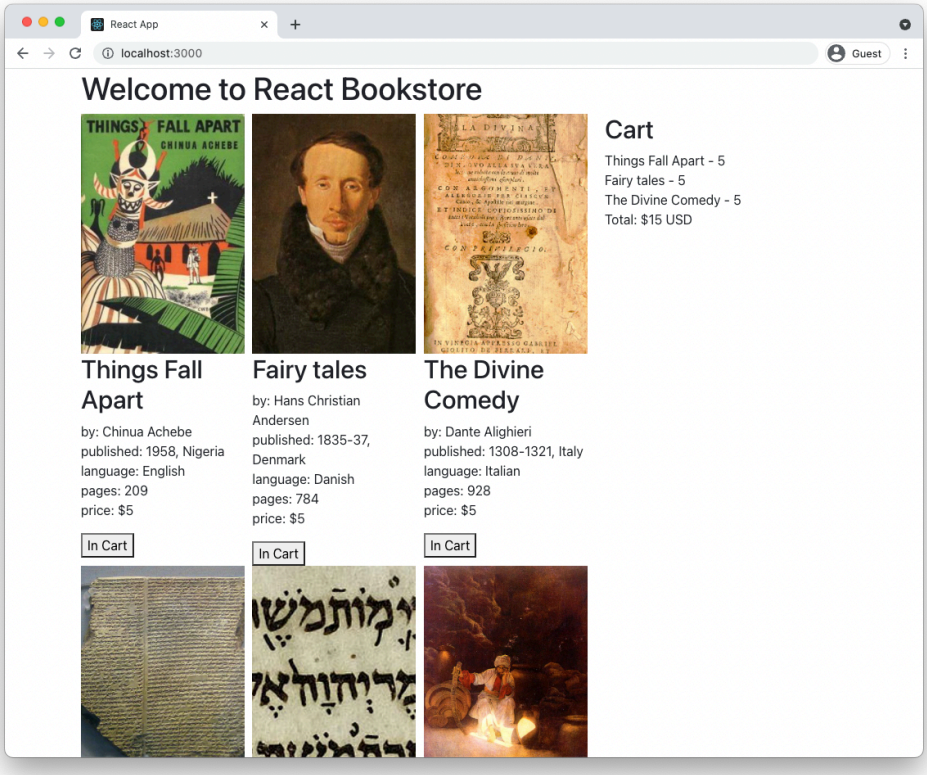
☐ 6. Pass `itemsInCart` to `ProductList`. We'll need that data in ProductList as well so we can show which products are already in the cart.
☐ 7. Inside the `ProductList` component, figure out how to pass a prop down to each product that is currently in the shopping cart and change the message on its button from **Add to Cart** to **In Cart**.

Hint: One way to do it is to create an array of just the ids from itemsInCart, then use the `Array.includes()` method to test whether each product's id is in the cart.

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

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

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



☐ 11. Run your tests. Notice that ProductList and Cart are failing. Can you figure out how to fix them?

☐ 12. Can you improve any of your other tests, such as CartItem?

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 called **Synthetic Events**.

Except for the fact that it works the same in every browser and some React-specific properties, Synthetic Events work the same as the native browser events.

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. 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 products. In the previous lab, we set the initial state of the application to an array containing a single product. 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 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.jsx** into a class component, following these steps:

- Import `React`
- 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`

Your App component should now match the following:

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

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

```
    super(props);
    this.state = {
      itemsInCart: [
        {
          id: '1',
          title: 'Things Fall Apart',
          price: '5',
        },
      ]
    }
  }

  render(){
    return (
      <div className="container">
        <Header />
        <div className="row">
          <div className="col-md-8">
            <ProductList
              itemsInCart={this.state.itemsInCart}
              products={productsData}
            />
          </div>
          <div className="col-md-4">
            <Cart itemsInCart={this.state.itemsInCart} />
          </div>
        </div>
        <Footer />
      </div>
    );
  }
}

export default App;
```

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

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

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

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

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

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

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

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

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

☐ 6. Inside the `Product` component, create a new function, called `handleClick`. The job of this function will be to call the `addToCart` function, passing it an object containing the id, title, and price of the current `Product`.

```
function handleClick() {
    props.addToCart({id:id,title:title,price:price});
  }
```

☐ 7. 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>
```

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

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

☐ 10. Run your tests to make sure they still work, and fix them if not.

Lab 10: Component Lifecycle and AJAX

Right now, the bookstore retrieves product data from a local file and displays all the books in the order in which they're in the array. But, what if you want to retrieve the data from the web and display in a random order (or, better, according to some algorithm, such as which books the user is most likely to buy or a user-chosen filter) each time a visitor comes to the store?

You could change the order of the items inside the `ProductList` component, but this has unintended consequences. Try the following to find out what happens.

☐ 1. Add the following function inside the `ProductList` component:

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

☐ 2. Create a new array by sorting the one passed as a prop.

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

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

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

☐ 4. Run `npm run dev`, and try adding some products to the cart.

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

One way to fix this problem is to use React's `componentDidMount()` lifecycle method to only sort the products once, after the component has mounted. To do this, follow these steps:

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

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

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

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

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

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

What's happening is that the `productsData` is loading before the `App` component is mounted. So, the initial render of the component uses the default sorting of the array, but the

`componentDidMount` method is shuffling the array after the component finishes mounting. The result is that the next re-render of the component will causes it to display the products in their shuffled order.

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

- ☐ 10. Add a property named `products` with a default value of an empty array (`[]`) to the `state` object.
- ☐ 11. Add a property named `loading` with a default value of `false` to the `state` object (in the `constructor`).
- ☐ 12. Remove the import of the product data.
- ☐ 13. Pass `this.state.products` to the `ProductList` component instead of `productsData`.
- ☐ 14. Move `src/data/products.json` to `public/data/products.json`. Tip: Make sure to remove the data folder from src, or it will cause you problems later on.
- ☐ 15. Confirm that the json file is in the right place by visiting the following url in your browser: **http://localhost:5173/data/products.json**
- ☐ 16. 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:5173/data/products.json')
            .then(response => response.json())
            .then(data => this.shuffleArray(data))
            .then(data => {
                this.setState(
                 {products:data,loading:false})
            })
};
```

- ☐ 17. Make a loading message display on the page when loading === true. One way to do this is by short-circuiting the return statement. Here's how that works:

```
if(this.state.loading){
  return <p>Loading...</p>
} else {

// insert normal return statement here

}
```

- ☐ 18. Run `npm run dev`
- ☐ 19. Run your tests. The test for App will fail. Do you see why? Check the debug messages in the terminal where you're running Vitest.

What's happening is that when the testing framework is rendering the test, it doesn't have access to the URL where the data file is. This is fine, because it's a best practice to make a stand-in for any network requests or external dependencies needed for tests (a mock). However, fixing this is a bit more complicated than fixing previous testing errors because retrieving the data is an asynchronous process that the test runner (Vitest) will need to know about.

You learn about Vitest in an upcoming lab. For now, copy the App.test.jsx file from the solutions folder into your /pages folder and see if you can figure out how it works.

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

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

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

```
useEffect(() => {
    async function fetchData() {
      try {
        setIsLoading(true);
        const response = await fetch(
          'http://localhost:5173/data/products.json'
        );
        const json = await response.json();
        const shuffledArray = shuffleArray(json);
        setProducts(shuffledArray);
        setIsLoading(false);
      } catch (e) {
        console.error(e);
      }
    }
    fetchData();
  }, []);
```

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

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.

☐ 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((prevTheme) => (prevTheme === '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 React from 'react';
import ReactDOM from 'react-dom/client';
import { ThemeProvider } from './context/ThemeContext';
import App from './components/App.jsx';
import './index.css';
import 'bootstrap/dist/css/bootstrap.css';

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 './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 = {
  backgroundColor: theme === 'light' ? 'white' : 'black',
  color: theme === 'light' ? 'black' : 'white',
  padding: '10px',
  position: 'fixed',
  left: '0',
  bottom: '0',
  margin: '0',
  width: '100%',
};
```

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

```
return (
  <p style={footerStyle}>
    This is the footer.
    <button onClick={toggleTheme}>Toggle Theme</button>
  </p>
);
```

☐ 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', () => {
  it('Renders', () => {
    render(<ThemeProvider><Footer /></ThemeProvider>);
    let element = screen.getByText(/footer/i);
    expect(element).toBeInTheDocument();
  });
});
```

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's 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.

☐ 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) {
    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.

☐ 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": true,
    "skipLibCheck": true,
    "esModuleInterop": false,
    "allowSyntheticDefaultImports": true,
    "allowImportingTsExtensions": 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: string;
  title: string;
  author?: string;
  published?: string;
  country?: string;
  lang?: string;
  pages?: string;
```

```
  image?: string;
  url?: 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 type ProductProps = Book & {
  inCart: boolean;
  addToCart: (product: Book) => void;
  removeFromCart: (id: string) => 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.

☐ 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/jest-dom/vitest';

import { cleanup } from '@testing-library/react';
import { expect, afterEach } from 'vitest';

afterEach(() => {
  cleanup();
});
```

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

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

export default defineConfig({
  plugins: [react(), tsconfigPaths()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
    coverage: {
      enabled: true,
      include: ['src/**/*.{js,jsx,ts,tsx}'],
      exclude: ['src/generated/**/*.ts'],
      reporter: ['text', 'html'],
    },
  },
});
```

☐ 3. Install some dependencies.

```
npm install --save-dev @vitejs/plugin-react
npm install --save-dev @vitest/ui
npm install --save-dev vite-tsconfig-paths
```

☐ 4. Modify your test script in package.json:

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

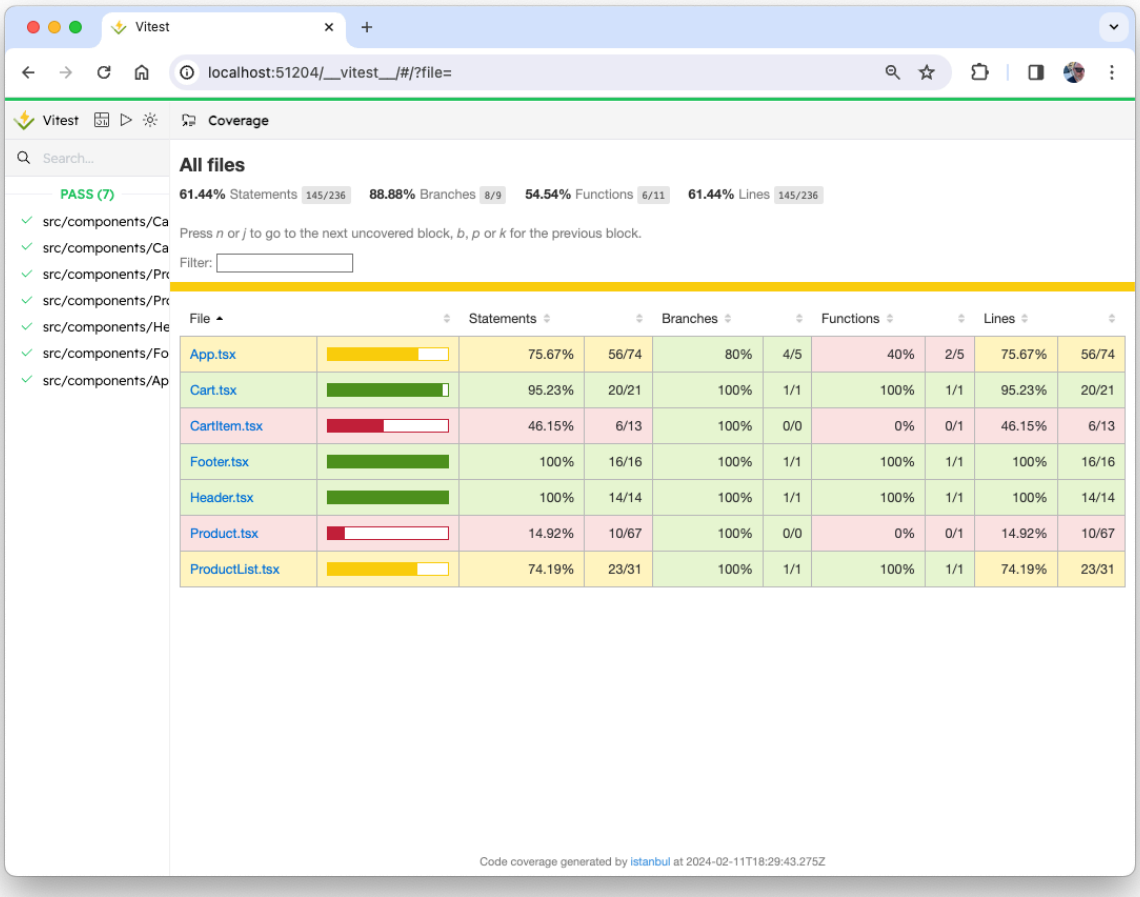
☐ 5. Run `npm test` in the root of your project. The results will appear in the terminal and the vitest ui will open in a browser window.

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.



☐ 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.

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

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

Step 1: Create a store

☐ 2. In `main.tsx`, import `configureStore` from Redux toolkit.

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

☐ 3. In main.tsx, configure the store:

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

☐ 4. Import `combineReducers` from redux.

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

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

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

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

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

☐ 7. Create the root reducer.

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

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

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

Step 2: Write the reducers

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

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

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

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

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

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

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

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

☐ 14. 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 products(state = { products: [] }, action =
{}) {
    switch (action.type) {
        case 'LOAD_PRODUCTS':
            return {
                ...state,
                products: action.products
            };
        default:
            return state; //no relevant action type
    }
}
```

Step 3: Write the Actions and Action Creators

☐ 15. Create a new directory in **src**, named **actions**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

☐ 22. Update the `useEffect` method to call the `loadProducts` method 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(() => {
    props.loadProducts(products);
    }, [products]);
```

☐ 23. 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
    addToCart={props.addToCart}
    removeFromCart={props.removeFromCart}
    itemsInCart={props.itemsInCart}
    products={props.products}
  />
</div>
<div className="col-md-4">
  <Cart itemsInCart={props.itemsInCart} />
</div>
```

☐ 24. Fix any typescript errors that you have.
☐ 25. Try converting reducers/index.js and actions/index.js to TypeScript.

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

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

Step 5: Looking at Redux Toolkit

☐ 27. 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
```

☐ 28. CD into the new project and run npm install.
☐ 29. 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.

☐ 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={props.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.

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

```
export function readCartFromLocalStorage() {
  let cart = localStorage.getItem('itemsInCart');
  cart = JSON.parse(cart);
  console.log(cart);
  return {
    type: 'READ_CART',
    payload: cart || [],
  };
}
```

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

```
case 'READ_CART':
  return {
    ...state,
    items: 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(() => {
  localStorage.setItem(
    'itemsInCart',
    JSON.stringify(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.

- ☐ 1. 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>
  <Route
    path="/"
    element={
      <ProductList
        products={products}
        itemsInCart={props.itemsInCart}
        addToCart={props.addToCart}
        removeFromCart={props.removeFromCart}
      />
    }
   />
   <Route
     path="/cart"
     element={
       <Cart
         itemsInCart={props.itemsInCart}
         removeFromCart={props.removeFromCart}
         submitCart={props.submitCart}
       />
     }
    />
</Routes>
```

- ☐ 6. Test it out. When you first start up the app (and the route is '/') it should display the `ProductList`, and if you change the url in the address bar to `'/cart'` it should display the cart. Everything should still work.
- ☐ 7. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bonus Lab: Authentication with JWT

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

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

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

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

Bonus Lab: React Asteroids

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