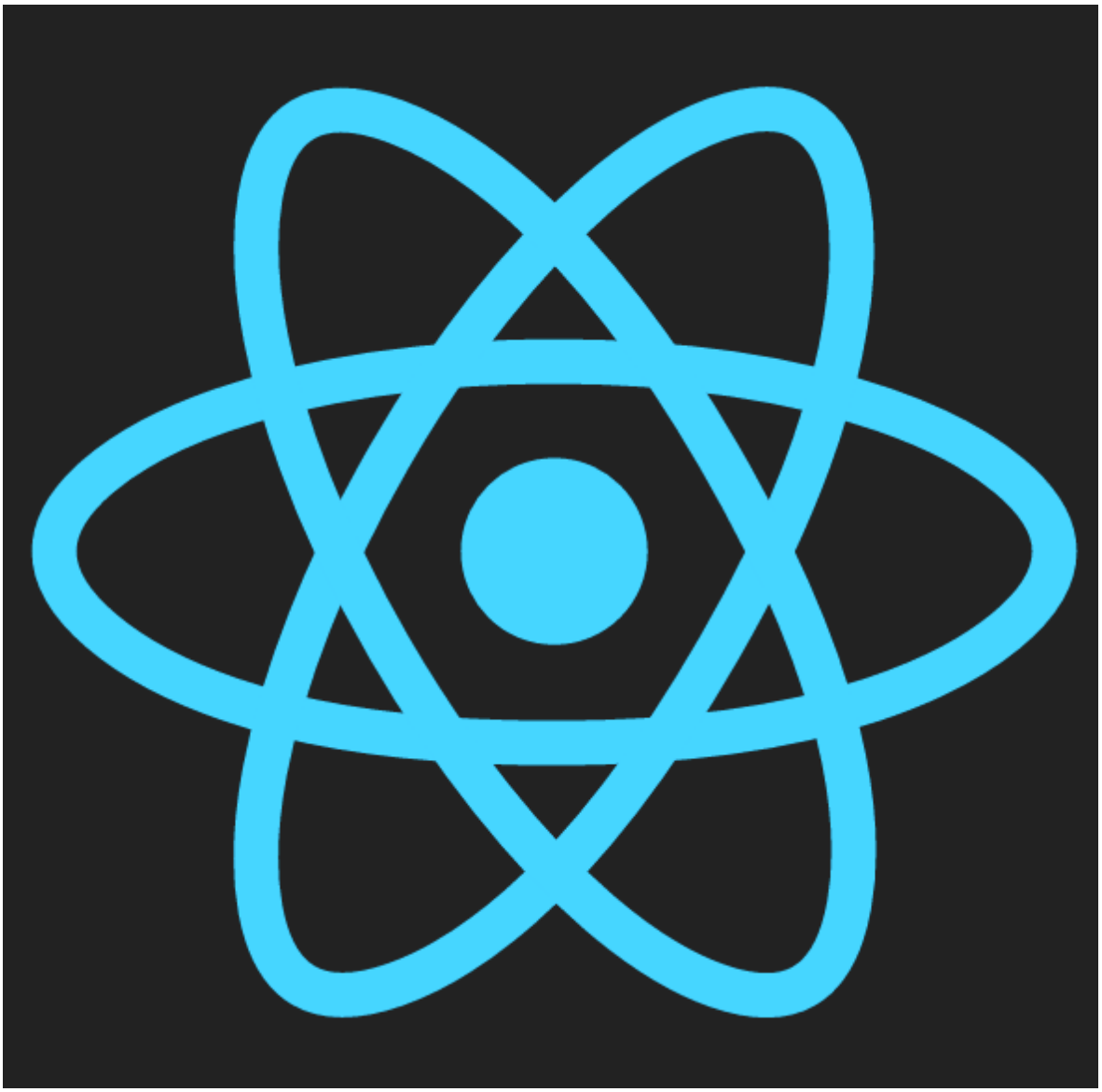
Professional ReactJS



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

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

Version 18.2.8, October 2023
by Chris Minnick

www.watzthis.com

Table of Contents

Disclaimers and Copyright Statement

Disclaimer

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

Third-Party Information

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

Copyright

Help us improve our courseware

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

Credits

About the Author

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

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

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

Course Requirements

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

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

Introduction and Git Repo Info

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

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

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

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

Yarn or npm?

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

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

Lab 01: Get Started with Vite

☐ 1. With the professional-reactjs folder open in VS Code, open a new terminal window in VS Code.

☐ 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
```

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. Set the name of the project to react-bookstore.

☐ 4. Select React as the framework.

☐ 5. Select React with Typescript as the variant.

☐ 6. Go into the new directory.

```
cd react-bookstore
```

☐ 7. Install the new project.

```
npm install
```

☐ 8. Test that everything was installed and works.

```
npm run dev
```

If the app was successfully created, you'll see a message in the terminal with a URL. Command-click that link to open it in your default browser, or open a browser and go to http://localhost:5173.

☐ 9. Find **App.tsx** inside the **src** directory and open it for editing.

☐ 10. Delete everything inside the function body (including the call to useState() and the return statement, and replace it with just a return statement with a div containing an <h1> element with the text "Welcome to React Bookstore". It should look like this:

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

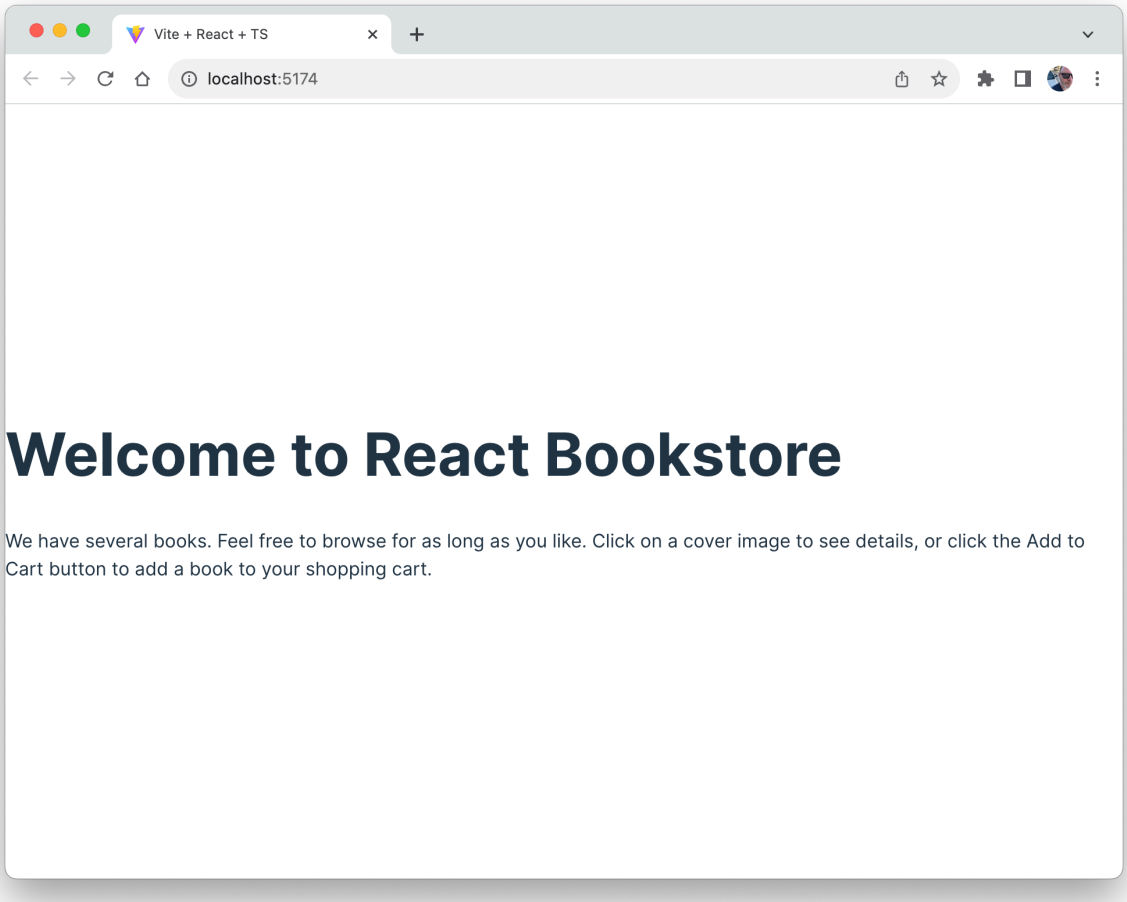
☐ 11. Delete all of the import statements since we're not using them anymore.

☐ 12. Create a new paragraph below the <h1>, containing the following text:

```
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.
```

☐ 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`).


Vite + React + TS
localhost:5174
Welcome to React Bookstore
We have several books. Feel free to browse for as long as you like. Click on a cover image to see details, or click the Add to Cart button to add a book to your shopping cart.

Lab 02: Your First Component

React components let you divide your user interface into independent and reusable pieces. The simplest components simply output some piece of HTML, given some input. 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.tsx** in the **src** directory

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

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

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

export default Footer;
```

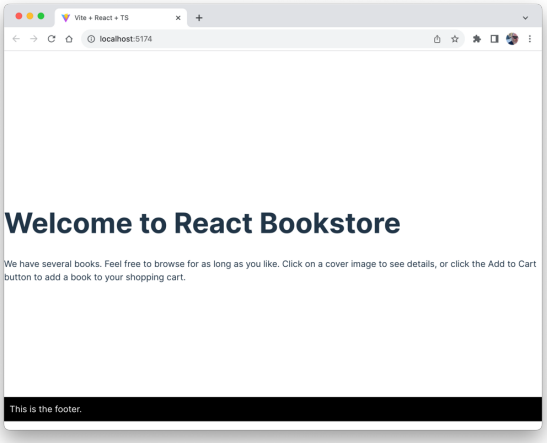
☐ 3. Add the following to the beginning of **App.tsx**

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

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

```
<Footer />
```

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



Lab 03: Create More Components

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

Part 1: Making new components

☐ 1. Using what you learned from creating **Footer.tsx**, make **Header.tsx** and **MainContainer.tsx** to replace code in **App.tsx**.

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.tsx** should match this:

```
return (
  <div>
    <Header />
    <MainContainer />
    <Footer />
  </div>
);
```

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.js** in your code editor.

This is a file containing 100 great books. We'll be building a store using this data.

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

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

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

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

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

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

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

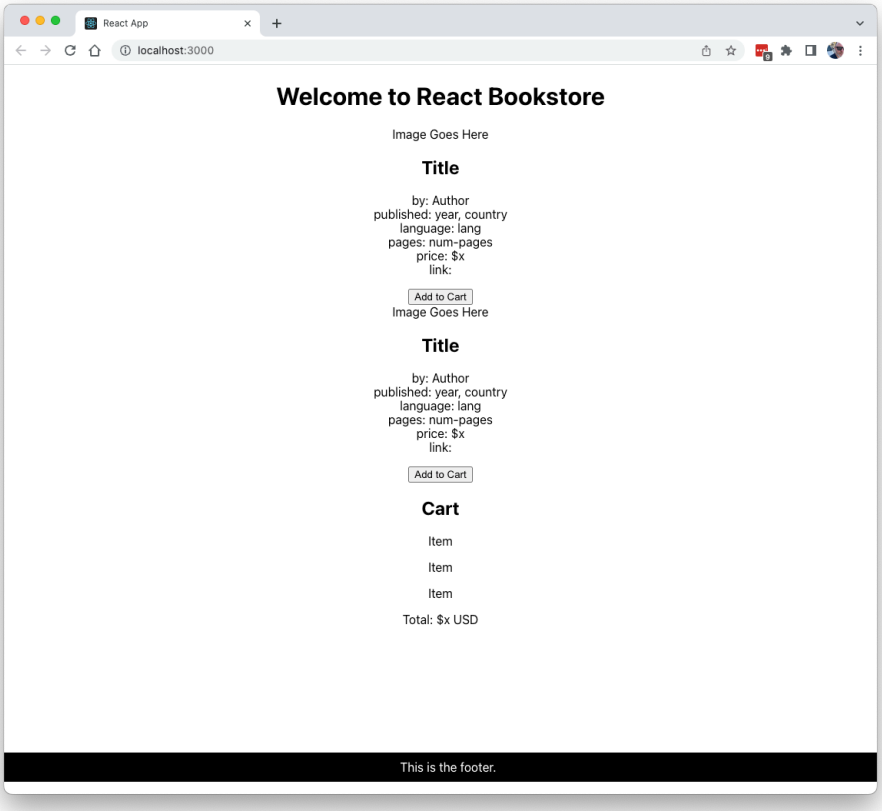
☐ 7. Open the **MainContainer.tsx** component in your project and modify it to the following.

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

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

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

export default MainContainer;
```

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

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



Lab 05: DOM Manipulation and Modern JavaScript

PART I. Working with an API

In this first part, you'll write a JavaScript application to make a call to NASA's API to get data on the near-Earth asteroids based on their closest approach date to Earth. All the data is from the NASA JPL Asteroid team (http://neo.jpl.nasa.gov/). We are looking for data regarding the asteroids' distance from the Earth, speed, size, and whether each asteroid is potentially hazardous.

☐ 1. In your code editor, create a new file and name it asteroids.js.

☐ 2. Create a variable called `today` and set it to a string containing the current date.

```
const today = new Date().toISOString().slice(0,10);
```

We will need an *API key* to access NASA's API. Without the key we would be limited as to how many calls we could make to the API per hour. With our key, we are allowed to call the API thousands of times per hour.

☐ 3. Navigate to https://api.nasa.gov/index.html#apply-for-an-api-key.

☐ 4. Enter your information and check your email for the API key. Congratulations, you have a developer's API key from NASA!

☐ 5. Create a variable called key and set it equal to the API key from NASA.

```
const key = "your-very-own-and-very-long-key";
```

Next, if we are calling an API, how do we call it? We call the URL that NASA provides and add parameters to specify the data we want. Here is an example:

```
https://api.nasa.gov/neo/rest/v1/feed?start_date=START_DATE&api_key=API_KEY
```

Notice the parameters `start_date` and `api_key`. In the example, each parameter is set equal to a variable, either `START_DATE` or `API_KEY`. Here is where we will use our variables by using a template literal string.

☐ 6. Create the variable called url.

```
const url
```

☐ 7. Set the variable url equal to the string below.

Notice that the string begins and ends with a backtick and contains two variables.

```
const url =
`https://api.nasa.gov/neo/rest/v1/feed?start_date=${today}&api_key=${key}
`;
```

We are now ready to build the Asteroid class and a static method that that calls the API and that will make asteroid objects.

☐ 8. Create a new class named `Asteroid`, and give it a constructor.

```
class Asteroid {
  constructor() {}
}
```

☐ 9. Each asteroid will have four properties: hazardous, distance, speed, and size. Add these as parameters to the constructor and initialize their corresponding properties:

```
constructor(isHazardous, distance, speed, size) {
  this.isHazardous = isHazardous;
  this.distance = distance;
  this.speed = speed;
  this.size = size;
}
```

☐ 10. Create a new static method named `getAsteroids()` in the Asteroid class. A static method is one that belongs to the class, rather than to instances of the class. Since we'll be running asynchronous code in this method, make it also be an async function:

```
static async getAsteroids() {
}
```

We'll use the `fetch()` method inside `getAsteroids()` to get the data from the NASA API.

The fetch method provides a way for JavaScript to make HTTP asynchronously.

☐ 11. Call `fetch()` asynchronously, passing it the `url`, and assign the result to a constant named `response`.

```
const reponse = await fetch(url);
```

The response of a fetch request is a Stream object. Eventually, we want our fetch method to return JSON when it is successfully finished, as the promise of this finished stream.

Async functions wrap function calls made with the await keyword with a promise. What is a promise? A promise is a first-class representation of a value that may be made asynchronously and is available in the future. A promise represents the eventual result of an asynchronous operation. We don't know if the call will be successful or not until the call is made, but we still have write our code moving forward as though it could be successful. If it is successful, we can do any number of things and continue with our coding. If it is not successful, we can return an error message.

☐ 12. To catch potential errors with the fetch(), wrap the async statement with a try/catch block:

```
static async getAsteroids() {
  try {
    const response = await fetch(url);
  } catch (error) {
    console.log(error);
}
```

It's also possible that the `fetch()` will be successful, but the response will be something other than what we want. For example, if you exceed your limit for API calls. To address that, we can check the value of `response.ok`.

☐ 13. Add a check for whether `response.ok` is true (inside the `try` block, after the fetch) and throw an error if it's not.

```
if (!response.ok) {
  throw new Error('Network response was not ok');
}
```

☐ 14. Call the json() method on response to parse the stream as JSON and return a JavaScript object.

```
const data = await response.json();
```

☐ 15. Add `console.log(data)` so you can examine the data.

```
console.log(data);
```

Type `about:blank` into your browser's address bar to open a blank window. Copy your code and paste it into your browser's JavaScript console to run it. Type `Asteroid.getAsteroids()` in the console to call the static `getAsteroids()` method. After a moment, data should return from NASA and you'll see all the data for the asteroids based on the approach date that we chose.

To test your try/catch block, try refreshing your browser window, then paste the code into the console again and run `getAsteroids()`, but change the url to make it invalid (such as by changing nasa.gov to nasa.com).

PART II JSON

Now we'd like to use the data we got from the NASA API. JSON is a data-interchange format that stands for "JavaScript Object Notation." It is built on two structures:

a. Name/Value Pairs: JSON acts similarly to JavaScript objects. We know that an object is a collection of properties, and a property is made up of a name/value pairs, i.e `const object = { 'name' : 'value' }`
   i. We can access a value by name in two different ways., dot notation and bracket notation.
      `object.name  => 'value'`
      `object['name'] => 'value'`

b. Arrays: In JSON, we also find ordered list of values, which we know are arrays, i.e., `[ 'Bob', 'Jim', 'Lara']`, which could be nested inside an object, i.e., `const object = { 'names' : [ 'Bob', 'Jim', 'Lara'] }`.

i. To access an array inside JSON we need to first know what name is associated with the array, then access it by order (remember, arrays start at index 0).

```
object.names[0] => 'Bob'
object.names[2] => 'Lara'
```

Take a look in the console and browse the data printed to the console. Now that we have our data, how do we make sense of it? Remember, JSON is built on two structures, name/value pairs and arrays.

Let's start to access the JSON to get the data we want. We want near Earth objects and we want them by a certain date. First, find the name `near_earth_objects` and expand it. You'll see a name, `yyyy-mm-dd`, similar to the format of the date string we created. If you expand one of those dates you'll see that it contains an array. This array is our list of asteroids!

Let's access this collection of asteroids.

☐ 16. Inside the `getAsteroids()` function, comment out `console.log(data)` and create a variable called `asteroids` and set it equal to `data`.

```
//console.log(data);
const asteroids = data
```

☐ 17. Use dot notation to access near_earth_objects from data.

```
const asteroids = data.near_earth_objects
```

☐ 18. Use bracket notation, [today], after json.near_earth_objects.

We do this because we cannot access the collection by using dot notation with the `yyyy-mm-dd` format. We do know that our `today` variable is equal to the value or property of the object we wish to access. Therefore, we used the variable `today` in bracket notation.

```
const asteroids = data.near_earth_objects[today];
```

☐ 19. Print the collection of asteroids to the console by writing `console.log(asteroids);`

Reload your browser, paste your code into the console, then run the `getAsteroids()` method and search for names or properties that relate to distance, speed, size, and hazard info.

We see we have an ordered list of objects representing each asteroid. We can iterate or loop through this list and access the properties of distance, speed, size and whether or not it is hazardous for each object.

How do we loop through an array? There are a few ways to do this. We will use the `.map()` function. The `map()` function takes a callback function as an argument, performs that function on each element of the array, and creates a new array with the results of that callback. We need to write a function that, as we loop through each element of `asteroids`, accesses those values we want (distance, speed, size, hazard).

☐ 20. Chain `.map()` to our variable `asteroids`.

At this point, we can also remove our console.log statements.

```
const asteroids = data.near_earth_objects[today];
asteroids.map();
```

☐ 21. Pass the argument `function(element) {}` to the `map()` function.

We pass `element` to our callback function as an argument. This represents each element in our collection of asteroid objects. As we loop through the collection, we can execute code on each element.

```
const asteroids = json.near_earth_objects[today];
asteroids.map(function(element) {
});
```

☐ 22. In the block of code for the callback function of the map method, print element to the console. This will allow us to see each element in our collection. What names and values do you notice?

```
const asteroids = json.near_earth_objects[today];
asteroids.map(function(element) {
    console.log(element);
});
```

If you run `getAsteroids()` in your console now, you'll see that each element that's logged by the `map()` function is an object and we can look for the properties of each asteroid. If you look closely, you'll see that these names correspond to the data we want (distance, speed, size, hazard info). We will start by looking into the name or property `close_approach_data`.

☐ 23. Create a variable called `approach` and set it equal to the `element`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element) {
    console.log(element);
    let approach = element
});
```

☐ 24. Print `approach` to the console and comment out `console.log(element)`.

We can now inspect our `approach` element to see what property we need to chain next.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element) {
    //console.log(element);
    let approach = element;
    console.log(approach);
});
```

☐ 25. Use dot notation to chain `close_approach_data` onto our `element`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element) {
    //console.log(element);
    let approach = element.close_approach_data;
    console.log(approach);
});
```

When you print the variable `approach` to the console, you get an array with one element. Access that element, which is at the 0-index.

☐ 26. Use bracket notation, and write `[0]` after `close_approach_data` to access the array with one element.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0];
    console.log(approach);
});
```

Further into this element, you'll see another object with an interesting property: `miss_distance`. We can now focus on distance, as we get closer to specific data.

☐ 27. Create a variable called `distance` and set it equal to `approach`. In the `console.log` command, replace `approach` with `distance`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0];
    let distance = approach;
    console.log(distance);
});
```

☐ 28. Using dot notation to chain `miss_distance` onto `approach`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0]
    let distance = approach.miss_distance;
    console.log(distance);
```

```
});
```

Printing `distance` to the console at this point reveals an object with properties that describe units of length. The values that are decimals in string format. For example:

```
[object Object] {
    astronomical: "0.3577465984",
    kilometers: "53518132",
    lunar: "139.1634216309",
    miles: "33254624"
}
```

☐ 29. Chain the property `.kilometers` to the property `miss_distance`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0]
    let distance = approach.miss_distance.kilometers;
    console.log(distance);
});
```

When you print `distance` to the console, you'll see it's a string that represents the distance of each asteroid from Earth in kilometers.

Let's move on to accessing the speed of the asteroid. Print `approach` to the console. One name/property of this object is `relative_velocity`.

☐ 30. Create a variable called `speed` and set it equal to `approach`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0]
    let distance = approach.miss_distance.kilometers;
    let speed = approach;
});
```

☐ 31. Chain `.relative_velocity` to `approach` in the `speed` variable.

Print `speed` to the console.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0]
    let distance = approach.miss_distance.kilometers;
    let speed = approach.relative_velocity;
    console.log(speed);
});
```

If you view `speed` by printing it to the console, you'll see another object with more name/value pairs that measure the speed in different units. Let's look at the name `kilometers_per_hour`.

☐ 32. Chain .kilometers_per_hour onto relative_velocity.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    console.log(speed);
});
```

When you print speed to the console, you'll get back a float, or a number that has a decimal point. This is the speed of each asteroid. You now have two points of data: distance and speed. Size is next.

Look back at the `element`, the original argument in our `.map()` function. If you print `element` to the console and look beyond and below the property `close_approach_data`, you'll eventually see the name `estimated_diameter`. Use that property of our element to access the size of each asteroid.

☐ 33. Create a variable called `size`. Set `size` equal to `element` and chain `estimated_diameter` to it.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    let size = element.estimated_diameter;
});
```

If you print size to the console, you'll get an object with properties of length. Look at the 'meters' property.

☐ 34. Chain .meters onto estimated_diameter.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    let size = element.estimated_diameter.meters;
```

```
});
```

When you print `size` to the console, you'll see an object that has two names relating to the estimated maximum and minimum size of the object. Since you never want to underestimate asteroids, go with the property `estimated_diameter_max`.

☐ 35. Chain .estimated_diameter_max after meters.

We now have the size data for each asteroid measured in meters.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    let size =
element.estimated_diameter.meters.estimated_diameter_max;
});
```

Now you just need one more piece of data: Is the asteroid a hazard? Print `element` to the console and look beyond `estimated_diameter`. You'll see a property named `is_potentially_hazardous_asteroid`. This is it! The value of this property is a Boolean, simply true or false. It is a potentially hazardous asteroid or it is not. Good info to know, right?

☐ 36. Create a variable called `hazard` and set this equal to `element`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    let size =
element.estimated_diameter.meters.estimated_diameter_max;
    let hazard = element;
});
```

☐ 37. Chain `.is_potentially_hazardous_asteroid` to `element`.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
    //console.log(element);
    let approach = element.close_approach_data[0]
    let distance = approach.miss_distance.kilometers;
    let speed =
approach.relative_velocity.kilometers_per_hour;
    let size =
element.estimated_diameter.meters.estimated_diameter_max;
    let hazard = element.is_potentially_hazardous_asteroid;
```

```
    });
```

☐ 38. After the `hazard` variable, insert a console.log statement to log each piece of data about each asteroid to the console.

```
let asteroids = json.near_earth_objects[today];
asteroids.map(function(element){
  let approach = element.close_approach_data[0];
  let distance = approach.miss_distance.kilometers;
  let speed =
approach.relative_velocity.kilometers_per_hour;
  let size =
element.estimated_diameter.meters.estimated_diameter_max;
  let hazard = element.is_potentially_hazardous_asteroid;
  console.log(`
  Hazard: ${hazardous}
  Distance: ${distance} km
  Speed: ${speed} km/h
  Size: ${size} m
  `);
});
```

☐ 39. Add a call to `Asteroid.getAsteroids()` to your asteroids.js file, after the class definition.

Here's what your `getAsteroids()` method should look like so far:

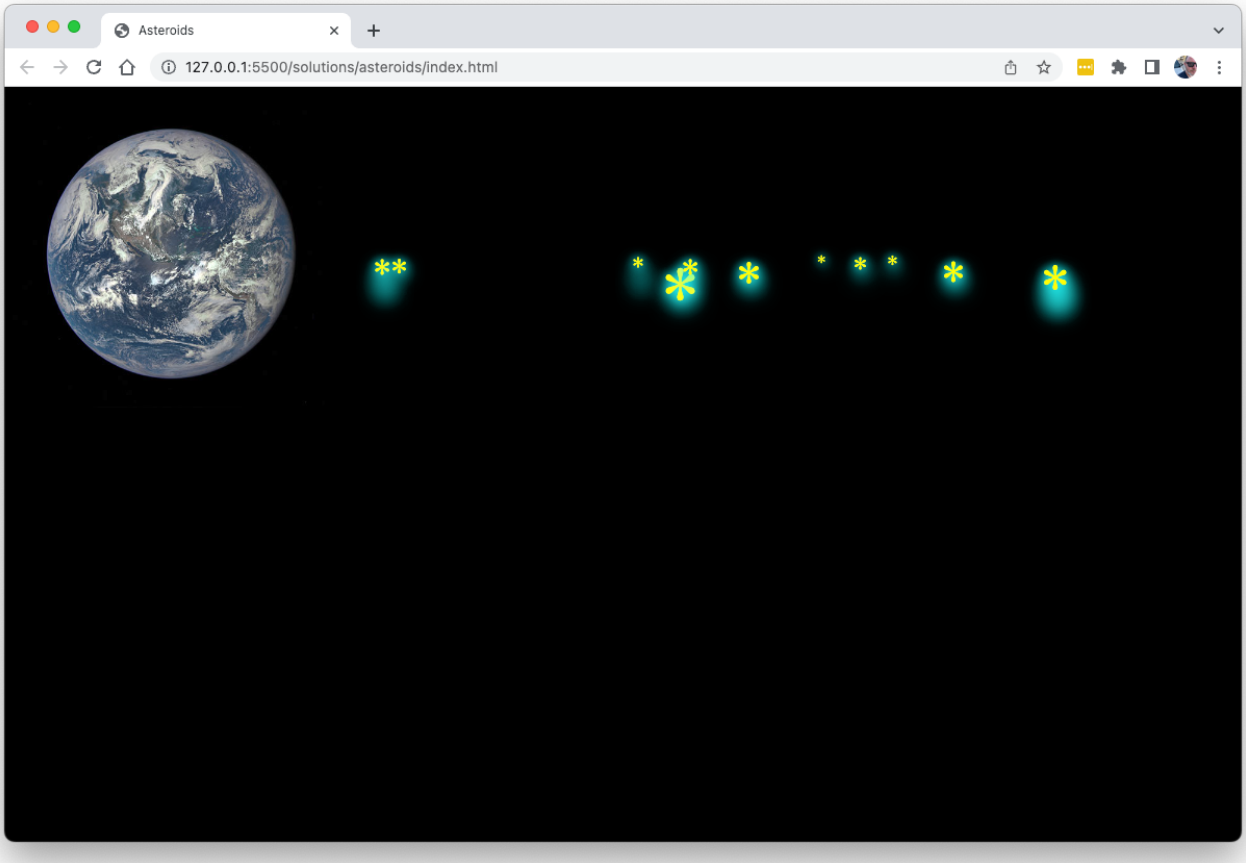
```
static async getAsteroids() {
    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error('Network response was not ok');
      }

      const data = await response.json();
      console.log(data);
      const asteroids = data.near_earth_objects[today];
      console.log(asteroids);
      asteroids.map(function (element) {
        let approach = element.close_approach_data[0];
        let distance = approach.miss_distance.kilometers;
        let speed = approach.relative_velocity.kilometers_per_hour;
        let size = element.estimated_diameter.meters.estimated_diameter_max;
        let hazardous = element.is_potentially_hazardous_asteroid;
        console.log(`
        Hazard: ${hazardous}
        Distance: ${distance} km
        Speed: ${speed} km/h
        Size: ${size} m
        `);
      });
    } catch (error) {
      console.log(error);
```

```
    }
}
```

When you run this code, you'll see all the properties printed to the console for each asteroid in a nice, readable format. Compare asteroids. Which is the biggest? Which is the fastest? Which is the closest to Earth? Which are potentially hazardous? All of this is important, because very soon you'll begin to create a visual display of these asteroids!

PART III. DOM Manipulation/Writing Functions

Our next goal is to display our collection of asteroids in a web browser. To create a visualization of the asteroids we will use JavaScript, HTML and CSS. In doing this, we will manipulate the document object model (DOM). Here's what the finished UI will look like:



The DOM is an application programming interface or API for HTML documents. It represents elements as a tree structure. Imagine a tree turned upside down. The trunk of the tree (the parent node) has branches (child nodes of the parent). These branches may have more branches and so on and so forth. The objects or elements in the tree

can be addressed and manipulated using methods. We will begin to write functions in JavaScript. These functions will use our variables, create asteroid elements, and append them HTML nodes in the DOM.

The HTML, CSS, and image you'll need have all been provided in `starter/lab05` directory in this course's repository. Copy them into your project and take a look them to get familiar with them.

Once you've copied them over, open index.html in your browser and you'll see the planet Earth over the black background of space.

We will manipulate the DOM by writing functions.

☐ 40. Return to the JavaScript file and create an instance of Asteroid inside the `map()` function:

```
let asteroid = new Asteroid(hazardous, distance, speed,
size);
```

☐ 41. Define a method in the Asteroid class called `placeAsteroid()`.

This function will place the asteroids in the DOM.

```
placeAsteroid() {}
```

To create an asteroid, you need to create the element that the program will append to the DOM.

☐ 42. In the method body, write the following statement:

```
 let asteroidElement = document.createElement('a').
```

The `document` object provides properties and methods to access all node objects. Calling `createElement` on `document` and pass the string `'a'` as an argument creates an `<a></a>` (anchor) tag.

☐ 43. On the next line, call the `.textContent()` method on the `asteroidElement` variable and set this equal to `'*'`.

This changes the text content of the <a> element to *. Now, you have an asteroid in the shape of an asterisk. Here's what the placeAsteroid() method should look like now:

```
placeAsteroid(hazardous, distance, speed, size) {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
}
```

Next, you'll add a class, `asteroid`, to the asteroid element to style it. This class has already been defined inside the style.css file. Return to JavaScript and on a

new line, write the variable `asteroid` and call the method `className` on it. Set this equal to the string `'asteroid'`.

```
function placeAsteroid(hazardous, distance, speed, size) {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className = 'asteroid';
}
```

The next step is to access an element of the DOM to which the new element will be appended. When an element is appended, it is appended to a parent element as a child of that element. The HTML document has a with the id “solar-system” that will be the parent of the asteroid elements.

The document object model has a method called `getElementById()` that can be used to get the with an id of “solar-system”.

☐ 44. Create a variable called `solarSystem` and set it equal to `document.getElementbyId('solar-system').`

This will give you the parent that you will append the asteroid elements to. Here's what placeAsteroid() should look like now:

```
placeAsteroid() {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className = 'asteroid';

    let solarSystem = document.getElementById('solar-system');
}
```

To append the `asteroidElement` to the `solarSystem`, define a private method named `append` with two parameters: `parent` and `el`.

The `parent` represents the element in the DOM that already exists on our page. The `el` represents the element which we will append to the parent. When we append `el` to the `parent`, our element is added as the last child of that parent node.

Private methods can be created in JavaScript by prefixing the name of the method with #. A private function can only be called from within the object.

☐ 45. Define a method named `#append` with parameters of `parent` and `el`. The `appendChild()` method is called on `parent` and takes `el` as an argument.

```
#append(parent, el) {
    parent.appendChild(el);
}
```

☐ 46. Call the `#append()` method from inside the `placeAsteroid()` method.

```
placeAsteroid(hazardous, distance, speed, size) {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className = 'asteroid';
```

```
        let solarSystem = document.getElementById('solar-system');
        this.#append();
    }
```

☐ 47. Pass the arguments `solarSystem` and `asteroidElement` to `append()`.

The `solarSystem` variable represents the `parent`, so it is written first. The `asteroidElement` variable is the child or `el`, so it is written second.

```
placeAsteroid(hazardous, distance, speed, size) {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className = 'asteroid';

    let solarSystem = document.getElementById('solar-system');
    this.#append(solarSystem, asteroidElement);
}
```

Refresh index.html in your browser. Nothing will happen at this point, because placeAsteroid() hasn't been called yet.

☐ 48. Add a call to `asteroid.placeAsteroid()` at the end of the `try` block in the getAsteroids() method :

```
asteroid.placeAsteroid();
```

Right now, the program can create many asteroids, but they are all be the same distance from the earth and the visualization looks as though there is only one asteroid.

In the next steps, you'll add functions to visualize the speed, distance, size, and hazard data.

☐ 49. Define a method called `#setDistance()`. This function takes two arguments: `asteroid` and `distance`.

```
#setDistance(asteroidElement) {
}
```

This method will use the float value of the `distance` property to change the location of an `asteroid` element on the page by adjusting the style of the element. More specifically, it will adjust the margin-left property. It is currently set at 0px, which places the asteroid next to planet Earth. If you increase the value of the margin-left property it will move the element away from Earth.

☐ 50. Call .style on asteroidElement.

```
#setDistance(asteroidElement) {
    asteroidElement.style
}
```

☐ 51. Chain `.marginLeft` after `.style` on the `asteroidElement` element.

```
#setDistance(asteroidElement) {
    asteroidElement.style.marginLeft
}
```

The variables for `distance` range in millions of km so it will need to be scaled down for the visualization. Try dividing distance by 100000 and set the pixel size of the margin-left to this value.

☐ 52. Set asteroid.style.marginLeft equal to distance/100 000.

```
#setDistance(asteroidElement) {
    asteroidElement.style.marginLeft = this.distance/100000;
}
```

☐ 53. Concatenate ‘px’ onto distance/100000.

When `'px'` is concatenated onto `distance/100000,` this entire value is coerced into a string. This string becomes the value for the `margin-left` property in the CSS DOM.

```
#setDistance(asteroidElement) {
    asteroidElement.style.marginLeft = this.distance/100000 + 'px';
}
```

☐ 54. Call `this.#setDistance(asteroidElement)` in the `placeAsteroid()` function.

Now, each asteroid in the collection should have a property, `margin-left,` that has a value based on its distance from Earth.

You're not finished with styling the asteroid elements just yet, though. You still have to set the hazard, speed and size of the asteroids.

☐ 55. Start a new method named `#setSize()`. This method will style the CSS `font-size` property of each asteroidElement so that we can see their relative size.

```
#setSize(asteroidElement) {
}
```

☐ 56. Similar to setting the distance, call `.style` on `asteroidElement`, then chain `.fontSize` onto that. Set this all equal to `size`. Concatenate `'px'` onto size.

```
#setSize(asteroidElement) {
    asteroidElement.style.fontSize = size + 'px';
}
```

The `size` values range from single to triple digits(in meters). This means the `font-size` will be huge for some asteroids. This may be distracting, so we can add a

conditional statement to this function. If the value of `size` is greater than a certain number, scale it down. Otherwise, let `size` remain the same.

☐ 57. In the `#setSize` method, before the `font-size` of `asteroid` is set, start a conditional statement:

```
#setSize(asteroidElement) {
    if

    asteroidElement.style.fontSize = this.size + 'px';
}
```

☐ 58. In parentheses, add the condition `( this.size > 100 )`.

```
#setSize(asteroidElement) {
    if ( this.size > 100 )

    asteroid.style.fontSize = this.size + 'px';
}
```

☐ 59. After the condition and in curly brackets, write `{ this.size = this.size/10 }`.

This reassigns the value of the `size` property to be a tenth of its original value.

```
#setSize(asteroidElement) {
    if ( this.size > 100 ) {
        this.size = this.size / 10
    }
    asteroidElement.style.fontSize = this.size + 'px';
}
```

☐ 60. Add this method to the `#placeAsteroid()` method and the asteroid elements will vary in size and distance.

```
#placeAsteroid() {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className += 'asteroid';

    this.#setSize(asteroidElement);
    this.#setDistance(asteroidElement);

    let solarSystem = document.getElementById('solar-system');
    this.#append(solarSystem, asteroidElement);
  }
```

Next up, speed! To visualize speed, we will add a motion blur that looks like tail for each asteroid. The faster ones should look like they're traveling faster, and while asteroids generally don't have tails, we will give them a longer motion blur which will appear like a longer tail.

To achieve this visualization, we will add a class written in CSS to each asteroid. If you look at the css file, you'll see that it contains three classes for adjusting the speed: `.speed-low`, `.speed-medium`, and `.speed-high`. The values of the `text-shadow` properties will differ in pixel length. By creating multiple shadows you can give the impression that faster asteroids show a longer tail (motion blur) than slower asteroids.

☐ 61. Define the method, `#setSpeed()` that will add these classes to the asteroid.

```
#setSpeed(asteroidElement) {
}
```

Speeds can be 10000 to over 100000km/hr. Let's choose some integers within this range to add each class. For asteroids with speeds greater than 50000km/hr we add 'speed-high', greater than 25000 we add speed-medium, and all other speeds below 25000 we add speed-low. Therefore we will need more than one conditional statement.

To write multiple conditional statements that executes different code for different conditions you can use the `if, else if,` and `else` commands.

☐ 62. Write `if ( speed > 50000 ) {}` for the first condition in the `#setSpeed` method.

```
#setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
    }
}
```

☐ 63. After the curly brackets of the first condition, write the second condition, `else if ( this.speed > 25000 ) {}`.

```
function setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
    } else if ( this.speed > 25000 ) {
    }
}
```

☐ 64. Write the keyword `else` and another set of curly brackets, `{}` after the second set of curly brackets. The code for this condition will be executed if the first two conditions are false.

```
function setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
    } else if ( this.speed > 25000 ) {
    } else {
    }
}
```

Each condition will execute code that adds a class to the asteroid element. You can use the same method you used to add the `'asteroid'` class to style the `asteroidElement` element , `.className`. Because the asteroid elements already have a class, you need to concatenate by using the '+=' operator and putting an extra space before the speed class string. If you used the = operator, you would overwrite the `asteroid` class.

☐ 65. Write `asteroid.className += ' speed-high'` for the code block of the first condition.

```
#setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
        asteroidElement.className += ' speed-high';
    } else if ( this.speed > 25000 ) {

    } else {

    }
}
```

☐ 66. Write `asteroidElement.className += ' speed-medium'` for the code block of the second condition.

```
#setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
        asteroidElement.className += ' speed-high';
    } else if ( this.speed > 25000 ) {
        asteroidElement.className += ' speed-medium';
    } else {

    }
}
```

☐ 67. Write `asteroidElement.className += ' speed-low'` for the code block of the third condition, executed only if the first two conditions are found to be false.

```
#setSpeed(asteroidElement) {
    if ( this.speed > 50000 ) {
        asteroidElement.className += ' speed-high';
    } else if ( this.speed > 25000 ) {
        asteroidElement.className += ' speed-medium';
    } else {
        asteroidElement.className += ' speed-low';
    }
}
```

☐ 68. Add the method call `#setSpeed(asteroidElement)` to the `placeAsteroid()` method. We can display our asteroid motion blurs!

```
#placeAsteroid() {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className += 'asteroid';

    this.#setSpeed(asteroidElement);
    this.#setSize(asteroidElement);
    this.#setDistance(asteroidElement);

    let solarSystem = document.getElementById('solar-system');
```

```
    this.#append(solarSystem, asteroidElement);
  }
```

Next, we want to display whether or not the asteroid is a hazard. What is a good color to represent a hazardous object? Why not red? This function will add the hazardous class to our element depending on the whether the value of the `isHazardous` property is true or false.

☐ 69. In JavaScript, define a method named `#setHazard(asteroidElement).`

```
function setHazard(asteroidElement) {
}
```

☐ 70. Write a conditional statement and a block of code to run if `isHazardous` is true, `if ( isHazardous === true ) { //execute code to add class }.`

```
function setHazard(asteroidElement) {
    if ( this.isHazardous === true ) {
        //execute code to add class
    }
}
```

☐ 71. Write `asteroidElement.className += ' hazardous'` in the conditional statement block of code. Remember the `+=` operator and the space before the word hazardous in the string that represents the class.

```
#setHazard(asteroidElement) {
    if ( this.isHazardous === true ) {
        asteroidElement.className += ' hazardous';
    }
}
```

☐ 72. Add a call to setHazard(asteroidElement) to the placeAsteroid() method.

```
#placeAsteroid() {
    let asteroidElement = document.createElement('a');
    asteroidElement.textContent = '*';
    asteroidElement.className += 'asteroid';

    this.#setSpeed(asteroidElement);
    this.#setSize(asteroidElement);
    this.#setDistance(asteroidElement);
    this.#setHazard(asteroidElement);

    let solarSystem = document.getElementById('solar-system');
    this.#append(solarSystem, asteroidElement);
}
```

Now, potentially hazardous asteroids are red! One may feel a bit of anxiety knowing the distance, speed, size and hazard info of all these asteroids. What can we do about that? Let's blow them up with JavaScript!

PART IV - Events

We have received asteroid data from NASA and displayed those asteroids and their distance from Earth, speed, size and whether or not they are potentially hazardous. The next step to consider is how to interact with those asteroids and blow them up! To do this, we will add an event listener that will blow up the asteroid when the user clicks it.

☐ 73. Define a method named `#targetAsteroid` that takes an argument of `asteroidElement`,

```
#targetAsteroid(asteroidElement) {
}
```

☐ 74. Define another method named `#boom()`, outside the scope of the `#targetAsteroid()` method.

This is the function that will run when an asteroid element is clicked. Notice that it does not take an argument. Why do you think this is?

```
#boom() {
}
```

We will return to writing the code for `#boom()` at a later step.

☐ 75. Return to `#targetAsteroid()` and inside the method, write the asteroid element.

```
#targetAsteroid(asteroidElement) {
    asteroidElement
}
```

☐ 76. Chain an event listener, `.addEventListener()` to the element.

```
#targetAsteroid(asteroidElement) {
    asteroidElement.addEventListener();
}
```

The event listener method will take two arguments, an event that it "listens" for and a callback function to run when the event occurs. There are many types of events that can occur within a page. Each event is represented by an object. This object can have custom fields or functions to get additional information about that event. Events can represent anything from basic user interactions (such as clicking a mouse or a 'click' event) to things that occur upon rendering or loading the DOM.

☐ 77. Add the event `'click'` as the first argument of the `addEventListener` function, `addEventListener('click')`.

```
#targetAsteroid(asteroidElement) {
    asteroidElement.addEventListener("click");
}
```

☐ 78. Add the function `boom` as the second argument to the addEventListener function, `addEventListener('click', boom)`.

Notice that when we write this function as an argument, we leave off the parentheses `()`. The `boom` function is a callback function, run only when the event occurs. When the parentheses are added to the end of a function, this executes the functions. We want to reference our function, only to be executed when the event occurs, and therefore, we leave off the parentheses.

```
#targetAsteroid(asteroidElement) {
    asteroidElement.addEventListener("click", this.#boom);
}
```

☐ 79. Return to defining the boom method. Add `this` to the function.

```
#boom() {
    this
}
```

Inside boom(), `this` refers to the `asteroidElement` element that triggered it to run.

☐ 80. Chain `.innerHTML` on `this`.

This will allow us to access the inner HTML of our asteroid element. Our asteroid element is currently an `<a>` tag and inside that tag is an asterisk that represents the shape of an asteroid. We intend to change that.

```
#boom() {
    this.innerHTML
}
```

☐ 81. Set `this.innerHTML` equal to a span containing the word `'BOOM!!!'`. Use the `=` operator.

This will reassign the inner HTML of our asteroid ‘a’ tag from the string `'*'` to the explosion.

```
#boom() {
  this.innerHTML = '<span class="boom">BOOM!!!</span>';
}
```

☐ 82. Add a call to targetAsteroid() to the placeAsteroid() function to set the event listener for each asteroid. For targetAsteroid to be able to set an event listener, the call to it must appear after the element has been inserted into the DOM (using the #append() method).

```
this.#targetAsteroid(asteroidElement);
```

Click on the asteroids and blow them up! BOOM!!! What an event! If you have time, think about other improvements you could make to this program. For example, how would you make the app animate asteroids or show a series of days? Can you make a spaceship that starts out at the middle of the screen that the user can control and use to destroy asteroids by shooting at them?

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. Delete all the styles from index.css.

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

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

☐ 3. Link to **bootstrap.css** inside of **src/main.tsx.**

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

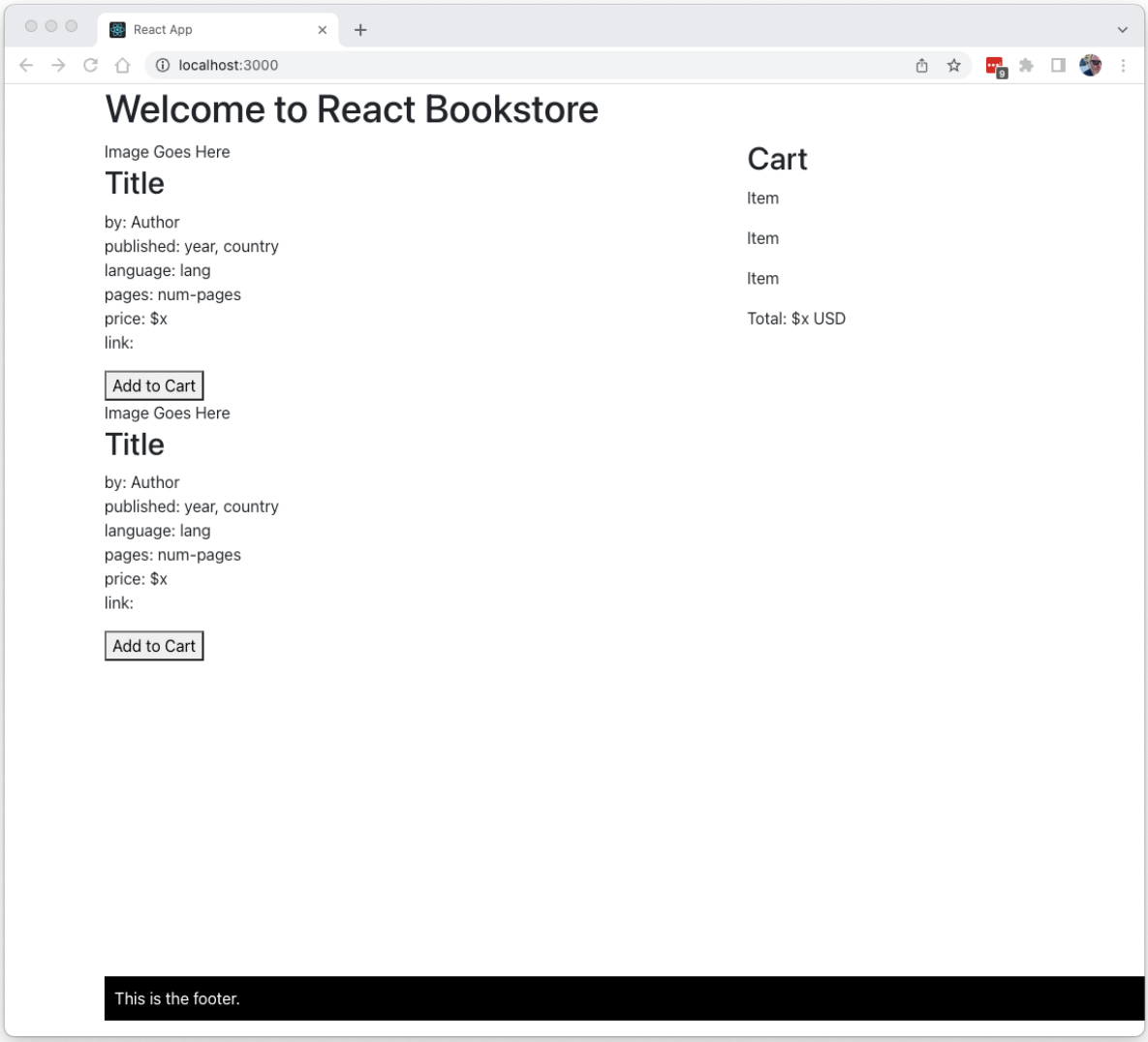
☐ 4. Add the Bootstrap container class to **App.tsx**.

```
import Header from './Header.tsx'
import MainContainer from './MainContainer.tsx'
import Footer from './Footer.tsx';
import './App.css';

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

export default App;
```

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

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

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

export default MainContainer;
```

☐ 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.tsx` and give the module the name `styles`.

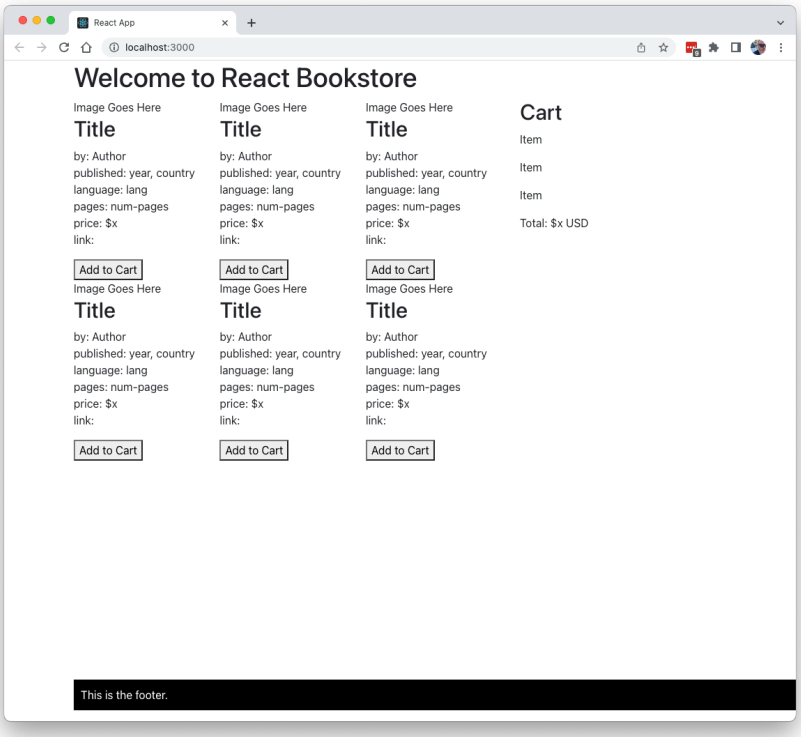
```
import styles from './ProductList.module.css';
```

☐ 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.



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

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

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

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

Lab 07: Props and Containers

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

```
App
Header
Footer
MainContainer
ProductList
Product
Cart
CartItem
```

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

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

☐ 2. Move **App.tsx**, **Header.tsx**, **Footer.tsx**, **MainContainer.tsx**, **ProductList.tsx**, **Product.tsx**, **Cart.tsx**, and **CartItem.tsx**, along with their css modules, into the **components** directory.

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

☐ 3. Import the data from data/products.js into **App.tsx.**

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

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

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

```
function MainContainer (props) {
```

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

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

☐ 6. Pass `products` from `MainContainer` to the `ProductList` component.

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

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

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

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

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

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

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

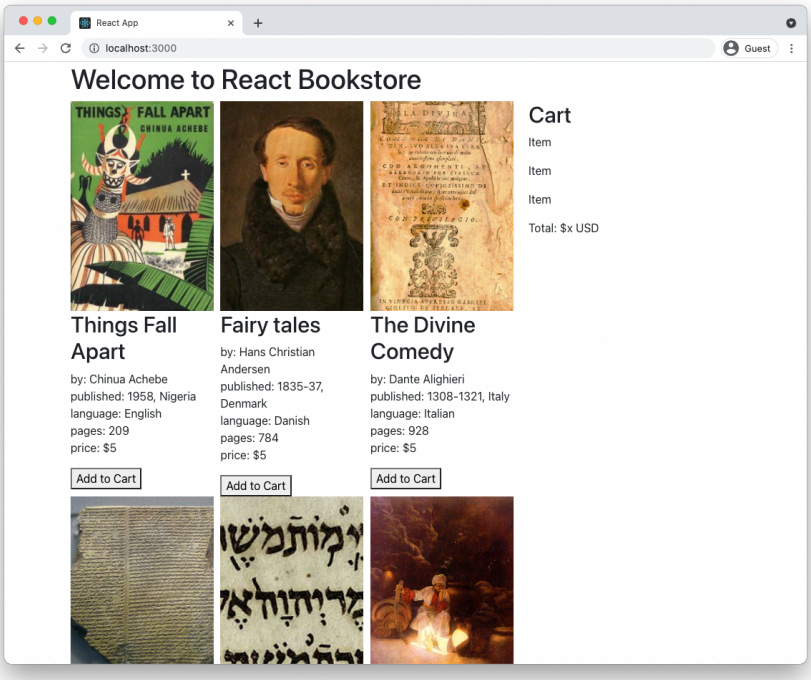
☐ 10. Create a new style rule in **Product.module.css** called thumbnail and set properties to format the book thumbnail images.

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

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

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

☐ 12. Run `npm run dev`



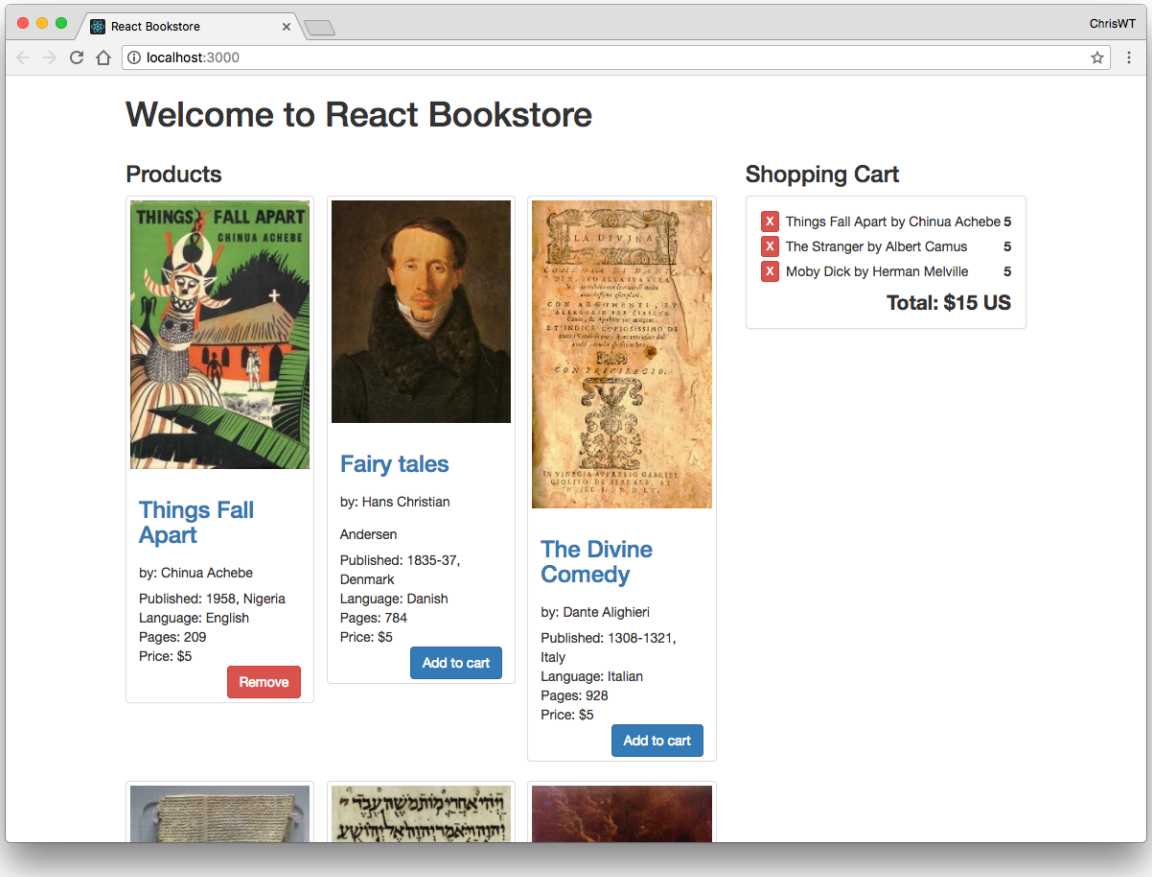
Lab 08: Adding State

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

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

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

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



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

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

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

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

To determine where the state should live, look for a component that is a common parent (or ancestor) to both `Cart` and `Product`. We have two components that fit this description, `App` and `MainContainer`.

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

Follow these steps to add state to the application.

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

☐ 2. Import `React.useState`

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

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

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

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

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

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

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

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

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

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

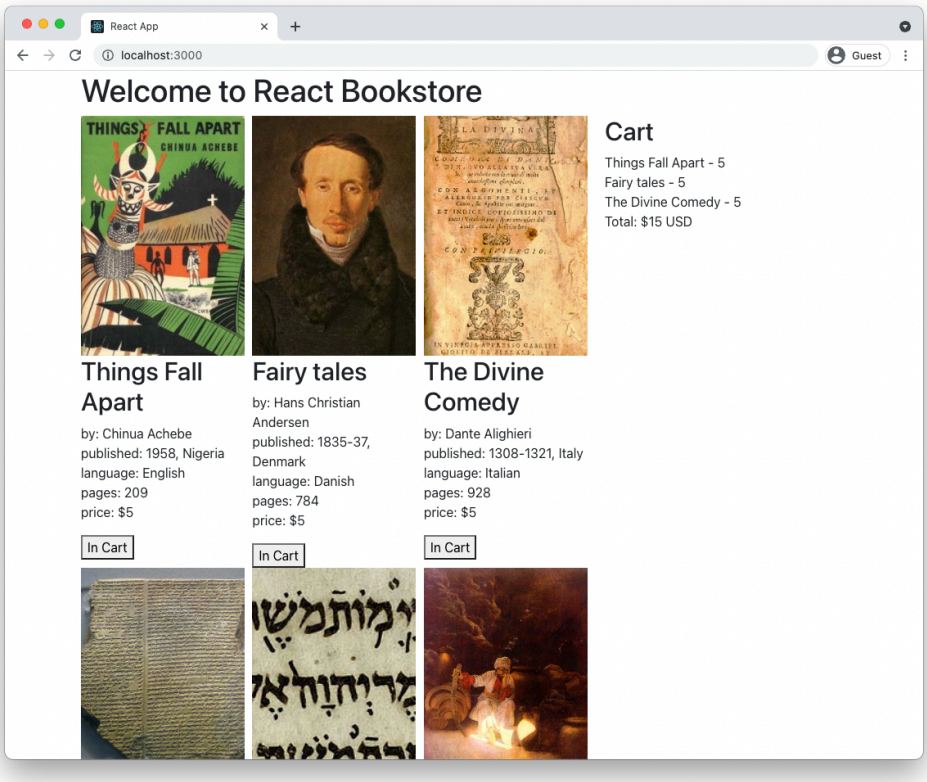
If you get stuck, look at the solution inside **intro-to-react/solutions/lab08**.

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

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

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

- ☐ 9. Render the list of `cartItems` inside the `Cart` component.
- ☐ 10. Modify the `CartItem` component to display the name and price of the item.
- ☐ 11. Calculate the total price of all the items in the shopping cart and display it in the `Cart` component.
- ☐ 12. Add type annotations where needed.



Lab 09: Interactions, Events, and Callbacks

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

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

React's Synthetic Events

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

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

Unidirectional Data Binding

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

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

State in a Class Component

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

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

```
import React from 'react';

class MyComponent extends React.Component {

...

}
```

A JavaScript class can have a constructor method, which will only run once during the lifecycle of the component. The constructor is used to initialize the state object and to bind functions to

the class. The constructor method is optional, but if you do use it, you must call the `super()` method as the first thing inside constructor. The `super()` method calls the constructor of the parent class. You should also pass the props object to `super()`.

```
import React from 'react';

class MyComponent extends React.Component {

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

...

}

export default MyComponent;
```

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

```
import React from 'react';

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

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

export default MyComponent;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
this.setState({data: newData}); //update state
```

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

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

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

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

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

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

☐ 1. Convert **App.tsx** into a class component, following these steps:

- Import `Component` from the React library
- Change the function header to a class header.
- Create a `constructor`
- Call the `super()` method
- Initialize the `this.state` object in the constructor, with one property, `itemsInCart`
- Delete the call to `useState()`, along with the import of `useState()`
- Create a `render` method and copy the existing `return` statement from the function component.
- In the `return` statement, reference `itemsInCart` using `this.state.itemsInCart`
- Remove the `setItemsInCart` attribute from the `<MainContainer />` element.

Your App component should now match the following:

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

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

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

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

export default App;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

☐ 11. Add type annotations where needed.

Lab 10: Component Lifecycle and AJAX

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

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

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

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

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

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

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

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

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

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

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

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

- ☐ 10. Add a property named `products` with a default value of an empty array (`[]`) to the `state` object.
- ☐ 11. Add a property named `loading` with a default value of `false` to the `state` object (in the `constructor`).
- ☐ 12. Remove the import of the product data.
- ☐ 13. Pass `this.state.products` to the `MainContainer` component instead of `productsData`.
- ☐ 14. Copy `products.json` from `starter/lab10/data` to `public/data/` in your project
- ☐ 15. Delete the `src/data` folder in your project.
- ☐ 16. Confirm that the json file is in the right place by visiting the following url in your browser: **http://[your-localhost-url/data/products.json**
- ☐ 17. Inside `componentDidMount`, use the `fetch` method to load the product data, sort it, and then update the state:

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

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

- ☐ 18. Run `npm run dev`

Lab 11: Converting App to a Function Component

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

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

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

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

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

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

Lab 12: PropTypes and defaultProps

PropTypes are a way of adding types to your components' props if you're not using TypeScript. The propTypes part of this lab should be skipped (or done using the non-typescript solution file from the previous lab) if you are using TypeScript. Default props can still be used with TypeScript.

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

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

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

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

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

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

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

};
```

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

☐ 4. Run your application and open the JavaScript console (**CTRL-SHIFT-J** in Chrome for Windows or **CMD-Option-J** on Mac) to verify that there are no errors.

☐ 5. Comment out one or more of the props passed from `MainContainer` to `ProductList`.

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

☐ 6. Check the JavaScript console to see the PropType warnings.

☐ 7. Add a `defaultProps` object to `ProductList`:

```
ProductList.defaultProps = {
}
```

☐ 8. Set default props for each of the arrays that `ProductList` receives. Set the defaults to empty arrays.

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

☐ 10. Add a `propType` object and a `defaultProps` object to every other component that receives props.

Lab 14: Testing with React Testing Library

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

Before you can use Jest in a project created with Vite, you'll need install some things and do some configuration. Follow these steps:

☐ 1. Change the extends array in .eslintrc.cjs to add jest support.

```
extends: [
    'eslint:recommended',
    'react-app/jest',
    'plugin:@typescript-eslint/recommended',
    'plugin:react-hooks/recommended',
  ],
```

☐ 2. Install Jest

```
npm install jest
```

☐ 3. Install babel presets to enable Jest to understand React and Typescript.

```
npm install @babel/preset-env @babel/preset-react @babel/preset-
typescript
```

☐ 4. Create .babelrc at the root of your project, with the following content:

```
{
  "presets": [
    "@babel/preset-env",
    ["@babel/preset-react", { "runtime": "automatic" }],
    "@babel/preset-typescript"
  ]
}
```

☐ 5. Install some additional dependencies for Jest and testing:

```
npm install @testing-library/react @testing-library/jest-dom jest-svg-
transformer identity-obj-proxy
```

☐ 6. Add the jest config to package.json (at the end, just before the closing }.

```
,
  "jest": {
```

```
    "testEnvironment": "jsdom",
    "moduleNameMapper": {
      "^.+\\.svg$": "jest-svg-transformer",
      "^.+\\.(css|less|scss)$": "identity-obj-proxy"
    },
    "setupFilesAfterEnv": [
      "<rootDir>/setupTests.js"
    ]
  }
```

☐ 7. Install a jest environment for jsdom.

```
npm install jest-environment-jsdom
```

☐ 8. Create a file named setupTests.js in the root of your project, containing the following:

```
import "@testing-library/jest-dom";
```

☐ 9. Install support for snapshot testing

```
npm install react-test-renderer
```

☐ 10. Add a test script to package.json

```
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "lint": "eslint . --ext ts,tsx --report-unused-disable-directives
--max-warnings 0",
    "preview": "vite preview",
    "test": "jest --watchAll --coverage --no-cache"
  },
```

☐ 11. Add a file named App.test.tsx to your project with the following content:

```
import { render } from '@testing-library/react';
import App from '../components/App';

test('demo', () => {
  expect(true).toBe(true);
});

test('Renders the main page', () => {
  render(<App />);
  expect(true).toBeTruthy();
```

```
});

test('renders as expected', () => {
  const { container } = render(<App />);
  expect(container).toMatchSnapshot();
});
```

☐ 12. Enter npm test to try it out!

☐ 13. Write simple tests for all your components. 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

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

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

Lab 15: Implementing Redux

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

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

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

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

Step 1: Create a store

☐ 2. In `main.tsx`, import `createStore` and `combineReducers` from redux.

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

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

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

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

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

☐ 5. Create the root reducer.

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

☐ 6. Define the store's initial state.

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

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

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

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

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

Step 2: Write the reducers

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

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

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

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

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

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

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

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

☐ 14. Write and export the `products` reducer function (also in **reducers/index.ts**). It should have one case, named `LOAD_PRODUCTS` which will update the state with the list of products fetched by the `componentDidMount` method of **App.tsx**.

```
export function products(state = {}, action = {}) {
    switch (action.type) {
        case 'LOAD_PRODUCTS':
            return {
                ...state,
                products: action.products
            };
```

```
            default:
                return state; //no relevant action type
        }
    }
```

Step 3: Write the Actions and Action Creators

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

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

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

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

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

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

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

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

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

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

    }
```

```
};

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

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

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

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

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

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

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

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

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

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

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

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

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

☐ 25. Look at the terminal. You'll see a warning message about a missing dependency. Our app works fine as it is, but can you figure out how to make that warning message go away?

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

Part 2: Redux Toolkit

☐ 1. Install Redux Toolkit

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

☐ 2. Remove the import of createStore() from 'redux' in main.ts

☐ 3. Import configureStore from Redux Toolkit

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

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

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

☐ 5. Delete the initialState variable from index.js

☐ 6. Define the initial state in reducers/index.js using default parameters, like this:

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

and

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

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

Lab 16: Redux Thunk

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

☐ 8. Run your app, add some items to the cart, and then open the Redux DevTools and click the Check Out button. You should see that the `CHECKOUT/rejected` action is dispatched.
☐ 9. Open a new terminal window and change to the **starter/lab16/server** directory.
☐ 10. Run `npm install` in the server directory
☐ 11. Run the server by entering `npm run dev`.
☐ 12. Click the **Check Out** button in the React app.

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

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

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

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

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

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

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

Lab 17: Persisting data in localStorage using Redux

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

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

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

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

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

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

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

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

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

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

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

Why? Because there is nothing in localStorage yet.

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

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

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

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

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

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

```
        return {
            ...state,
            items: newCart
        };
```

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

Lab 18: React Router

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

- ☐ 1. Install `react-router-dom`
- ☐ 2. Import `BrowserRouter` as `Router` into **main.tsx** and wrap the `Router` component around the `<Provider>` element in `ReactDOM.render`
- ☐ 3. In **MainContainer.tsx**, import `Routes` and `Route` from `react-router-dom`.
- ☐ 4. In the MainContainer's return statement, change the page layout to a 1-column layout by removing the `</div>` and `<div>` from between `Cart` and `ProductList` and changing the `className` passed to the outside `div` to `col-md-12`.
- ☐ 5. Replace `ProductList` and `Cart` with a `Routes` component containing two Routes. The first should render `ProductList` when the path is exactly `'/'` and the second should render `Cart` when the path is `'/cart'`.

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

- ☐ 6. Test it out. When you first start up the app (and the route is '/') it should display the `ProductList`, and if you change the url in the address bar to `'/cart'` it should display the cart. Everything should still work
- ☐ 7. Challenge: Make a Shopping Cart button component that displays the number of items in the cart in the header and that links to the shopping cart (using `react-router-dom`'s `Link` component). You can use the fontawesome React component to render the icon: **`https://fontawesome.com/v5.15/how-to-use/on-the-web/using-with/react`**

Lab 19: Microfrontends with Single SPA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bonus Lab: Authentication with JWT

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

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

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

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

Bonus Lab: React Asteroids

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