BUILDING WEB APPLICATIONS WITH REACT

EXERCISE MANUAL

Table of Contents

This page intentionally left blank.

Exercise 0.1: Inspecting the Complete Solution

In this exercise, you will explore a completed version of the application you will build this week. In the process, you will ensure that you have everything configured correctly to proceed with the remaining exercises.

1. Begin by downloading the exercises, solutions, and other materials

You can either download the zip file and extract it to your hard drive, or, if you prefer, use git to clone the course load.

2. OPTIONAL STEP: Navigate to `http://code.visualstudio.com/` and download and install Visual Studio Code, a free cross-platform code editor with IntelliSense support and terminal/command line integration.

From this point on, the course notes assume the use of VS Code. You can, however, use any code editor you prefer, or even a simple text editor combined with terminal/command windows. Simply adjust the instructions accordingly as you go.

3. Use VS Code to open the `React/Materials/Express` folder. The folder contains a (very simple) `server.js` web server providing a RESTful service at `localhost:3030`.

Depending on your installation/platform, you can open a folder in VS Code either by right-clicking the folder and selecting **Open with Code**, or by opening VS Code directly and select **File | Open Folder...** and then browsing to the folder.

4. Open a terminal/command window in VS Code (via the **View** menu item or holding down the control key and clicking back-tick, CTRL+`), and run the command `npm install`

WARNING! If you receive an error saying that npm is not recognized, you must install node. You can download node from https://nodejs.org/en/download/. Once node has been successfully installed, run the command `npm install` again and proceed with the next step. You may need to reboot your machine before npm is available.

5. The necessary packages will be installed. This can take some time, depending on your internet connection. Wait for the installation to complete before moving on to the next step.

You can safely ignore any warnings during the installation.

6. Wait for the install to complete, and then enter and run the command:

```
npm start
```

7. Once the Express server is running, you should see a console message telling you: `Book Reactions Express server listening on port 3030 in development mode`

8. The server will need to be running for the rest of this exercise. You can stop it at any time using CTRL+C (on some platforms you may also need to confirm with `Y`), and restart it using `npm start`. Any time you restart the server, the data will return to its original state. Feel free to examine the code. The web service has been configured to allow Cross Origin Resource Sharing, so that you can call it from the React application, which will be served from `localhost:3000`.

Now that the RESTful service is running, it's time to install the necessary npm packages for the React application.

9. Open the folder `React/Exercises/exercise-0.1` in a separate instance of Visual Studio Code and use a terminal/console window to run `npm install`. Wait for the packages to install before proceeding to the next step.

You can safely ignore any warnings during the installation.

Every exercise after Exercise 1.1 begins with a start point that is the completed version of the previous exercise, with all the bonus exercises completed. You can refer to the next exercise code at any point if you need assistance with completing the exercises.

10. Once the npm packages have successfully installed, run the application by entering `npm start` in the terminal. You should see various messages in the terminal, including that the application compiled successfully and is running in development mode.

11. Your default browser should open automatically at `http://localhost:3000`. If not, open Google Chrome and type in the address. You should see a page with a list of books.

The instructions will assume Google Chrome from now on because of its excellent tooling for debugging React. If you choose to use a different browser, you will need to adjust some of the instructions as you go.

12. Explore the site and then close Google Chrome. Then terminate the application by clicking in the terminal window and pressing CTRL+C to terminate the batch job.

You can also use CTRL+C to terminate the Express server. You won't be needing it again until the chapter on REST.

Bonus Exercise (to be attempted if time permits):

13. Applications built with `create-react-app` already contain the necessary configuration for deployment. Run the command `npm run build` in the terminal and examine the output.

14. Once the `build` folder has been created by the build process, examine the contents of the folder, and the compiled versions of the JavaScript files. They are in the `build/static/js` folder.

All that's required to deploy your application is to move the contents of the build folder to your production server!

Congratulations! You have completed the exercise.

Exercise 1.1: Getting Started with React

In this exercise, you will create a React application using `create-react-app`. This is the only exercise that does not have a start point.

1. Open a terminal/command window in the folder `React/Exercises`.

In windows, you can open a command window via the context menu when you hold down the SHIFT key and right-click a folder.

2. Enter the following command to create a simple React application in the folder `exercise-1.1`:

```
npx create-react-app exercise-1.1
```

The `create-react-app` tool is installed via `npx` to ensure you are always using the latest version. If this command fails, you may need to install the latest version of npm. If it still fails, your system may require you to install `create-react-app` globally using `npm` before running the `npx` command, and potentially to uninstall it prior to doing so.

3. This command will take some time. Wait for it to complete and then open the new `exercise-1.1` folder using Visual Studio Code.

4. Open a terminal/command window in VS Code (via the **View** menu item or holding down the control key and clicking back-tick, CTRL+`), and run the command `npm start`

Your browser should open, and a simple React app should be displayed.

5. Leaving the application running, open the file `src/App.js` inside VS Code. Delete everything inside `return();`.

6. Add a `<div>` element inside the `return()` and nest `<h1>` and `<p>` elements inside it. Set `Welcome to Book Reactions` as the content of the `<h1>` and `Where you react to books` as the content of the `<p>`.

The complete method is below, with your changes in bold.

```
return (
  <div>
    <h1>Welcome to Book Reactions</h1>
    <p>Where you react to books</p>
  </div>
);
```

7. Save the file and examine the results in your browser.

You should see your new content displayed in the browser, as `react-create-app` is configured to use hot reloading. You can work on your project without having to stop and recompile your react components.

8. Press CTRL+C, (and then `Y` if your platform requires it) to stop the React application.

Bonus Exercise (to be attempted if time permits):

9. Your application is going to use the CSS library `bootstrap`. Open the file `public/index.html` inside VS Code and add a link to a CDN copy of `bootstrap 4.0` to the head of the page.

10. While you're inside `index.html`, change the page `title` to `Book Reactions`.

11. The application title is also set inside `package.json` in the root of the application. Open the file and change `exercise-1.1` to `book-reactions`.

12. If the application is running, press CTRL+C to stop the React application.

Congratulations! You have completed the exercise.

This page intentionally left blank.

Exercise 1.2: Creating a React Component

In this exercise, you will create a new React component and use it from an existing component.

1. Open the folder `React/Exercises/exercise-1.2` in VS Code. Then open a terminal/command window inside VS Code.

The start point is a completed version of Exercise 1.1 with all the bonus exercise steps. Every start point from now on is simply the completed version of the previous exercise including any bonus exercise steps.

2. Run `npm install` to add all the necessary packages. Once the `node_modules` are installed, use `npm start` to run the solution in the browser.

3. Add a new folder `components` under the `src` folder, and then a second folder `common` underneath `components`.

You can create a folder in VS Code by right-clicking the parent folder and selecting **New Folder**. The `common` folder will be used for components that are used throughout the application. Later, you will add additional folders directly underneath `src` for other aspects of your application such as the API to talk to the REST server.

4. Next, add a new file `Navigation.js` inside the `common` folder.

You are going to create a simple functional component that will eventually hold the navigation for the Book Reactions application. For now, you will add place-holder HTML and a branding element.

5. Inside the new file, create a `const Navigation`, and assign an empty arrow function as the value. Then add an E6 export statement, exporting `Navigation` as `default`.

```
const Navigation = () => ( );
export default Navigation;
```

The body of the function is going to provide the render method for the React component.

6. Add a well-formed HTML `nav` directly inside the body of the method, without enclosing it inside quotation marks, and save the file.

```
const Navigation = () => (<nav></nav>);
```

7. Open `src/App.js` in VS Code and add an `import` statement for your new component to the top of the file.

```
import Navigation from './components/common/Navigation';
```

8. Add the `<Navigation />` element for your component as the first child of the outer `<div>` element and save the file. You should see your new navigation element displayed in the browser.

9. Return to `Navigation.js` and add the following CSS classes to the `div`, and again save your file: `navbar navbar-expand-sm bg-light`

```
<nav className="navbar navbar-expand-sm bg-light">
```

Remember that JSX uses the alias `className` for the HTML `class` attribute, in order to avoid a naming conflict with the JavaScript `class` keyword.

10. Next, add an HTML anchor element `a` inside the `nav`, with the following properties:

    a. text: **`Book Reactions`**

    b. **`href: /`**

    c. CSS class: **`navbar-brand`**.

```
<a className="navbar-brand" href="/">Book Reactions</a>
```

The first version of the component is now complete. Later, you will revise this content, as well as adding links to additional logical pages.

11. Next, make the following changes to `App.js`:

    a. Delete the `h1` element.

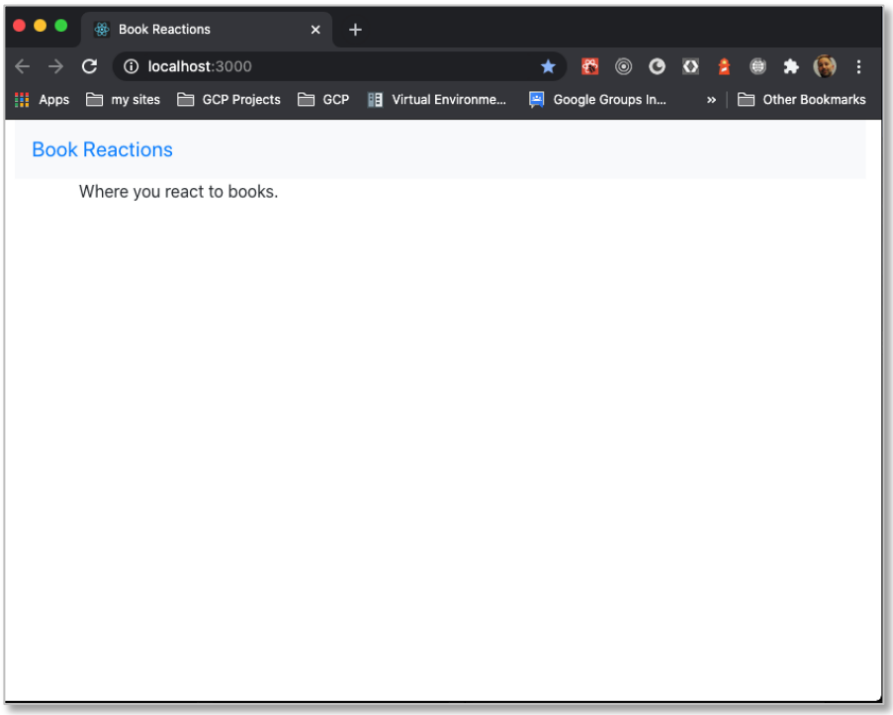
    b. Add the CSS class `container-fluid` to the outer `div`.

    c. Add the CSS class `container` to the `p`.

The code for this step is as follows.

```
 <div className="container-fluid">
     <Navigation />
     <p className="container">
      Where you react to books.
     </p>
</div>
```

12. Save the file and examine the revised application in the browser. It should look something like this screenshot:



13. Press CTRL+C to stop the React application.

Congratulations! You have completed the exercise.

Exercise 1.3: Working with JSX

In this exercise, you will create both class and functional components and combine JSX with JavaScript to create an HTML table.

1. Open `React/Exercises/exercise-1.3` with VS Code, and either run `npm install` to add all the necessary packages or move the `node_modules` folder from the `exercise-1.2` folder into the `exercise-1.3` folder. Once the `node_modules` are moved/installed, use `npm start` to run the solution in the browser.

You will need to do this on every start point, so from now on these detailed instructions will be assumed. It is much faster to move the `node_modules` than to copy or reinstall them, so this is the most efficient way to proceed. In the real world, you would work inside a single project and this would not be necessary.

2. Add a new folder `books` under the `components` folder and create a new file `BookList.js` inside the new folder.

For now, this component will be a table that displays a single book (via another component). Later, you will pass book details into this component and build the table.

3. Inside the new file, `import React` and then create a `class BookList` that `extends React.Component` and has a single method `render()`.

4. Inside the `render()` method, return an empty `div` element. Then set `BookList` as the `default export` from the file and save the file.

The code for these two steps is on the next page.

```
import React from 'react';

class BookList extends React.Component {
    render() {
        return (<div></div>)
    }
}

export default BookList;
```

5. Next, add an HTML `table` element inside the `div`. It should have a single row containing two `th` cells, one with the content: `Book`, the other with `Author`.

```
<table>
     <tr>
          <th>Book</th>
          <th>Author</th>
     </tr>
</table>
```

6. Open `App.js`, `import` the new component, and add the `BookList` element immediately after the text in the `p` element. Also, change the `p` element to a `div` to improve the HTML.

```
<div className="container">
   Where you react to books.
   <BookList />
</div>
```

7. Save your work and check that the web page is displaying correctly.

It doesn't look very impressive at the moment, as the table has no content or CSS formatting, but you should see the contents of the header cells displayed beneath the existing text.

8. Add another new component `Book` in the `books` folder. This time create a functional component with an explicit `return`. Declare two `const` variables, `title` and `author`. Set them to any title and author you like.

```
function Book() {
    const title = "The Lord Of The Rings";
    const author = "J R R Tolkien";

    return ()
}
```

9. Complete this version of the component by returning a table row from the `render()` method, writing the values of the `title` and `author` variables into the JSX, and making the `Book` function the `default export`

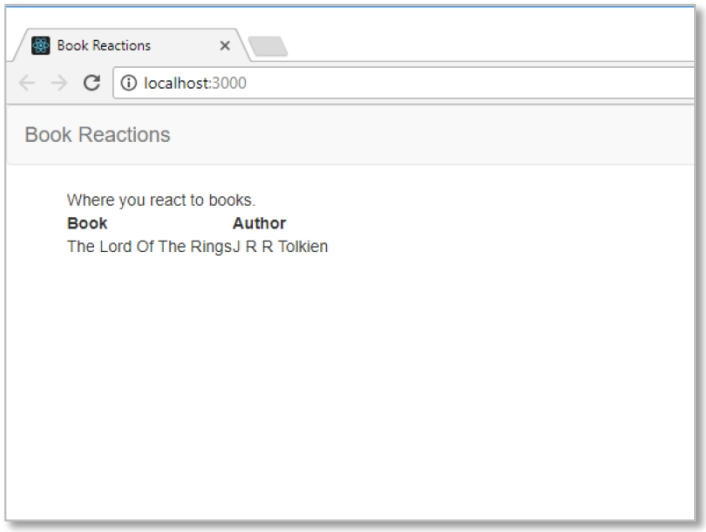
The complete code for `Book.js` follows.

```
function Book() {
    const title = "The Lord Of The Rings";
    const author = "J R R Tolkien";

    return (
        <tr>
            <td>
                {title}
            </td>
            <td>{author}</td>
        </tr>
    )
}

export default Book;
```

10. Finally, return to the `BookList` component, `import Book` and add the new `<Book />` element after the existing table row in the render method.

```
<div>
   <table>
      <tr>
            <th>Book</th>
            <th>Author</th>
       </tr>
          <Book />
   </table>
</div>
```

11. Save your work and view the page in Google Chrome. It should look something like the screenshot below.



12. Press CTRL+C to stop the React application.

You have created both class and functional components and combined them via composition, but the HTML is not currently valid, and the table lacks even basic styling. Both issues are addressed in the bonus exercise.

Bonus Exercise (to be attempted if time permits):

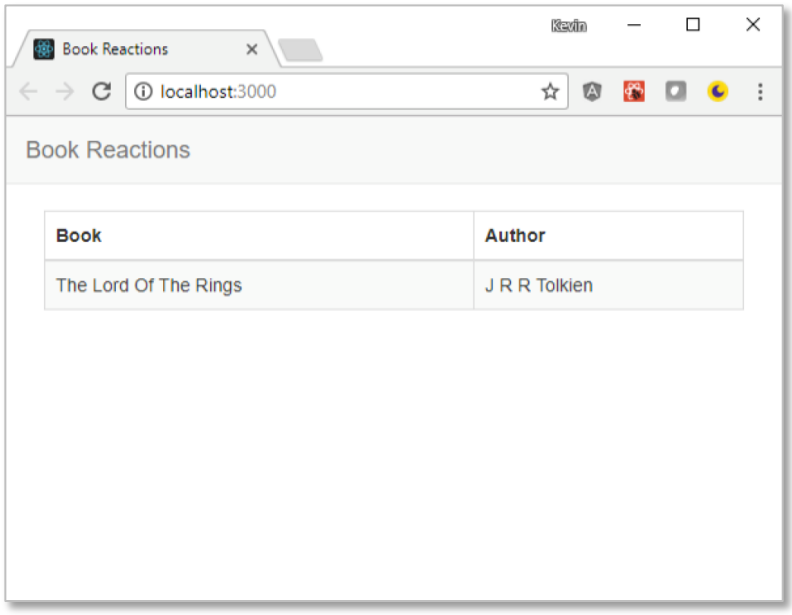
The table is not very attractive or user-friendly yet. You will improve the presentation by adding bootstrap classes and modifying the HTML.

13. First, create valid HTML for the table by adding `thead` and `tbody` elements around the table header and body, respectively.

14. Next, add the bootstrap CSS class `table-responsive` to the outer `<div>` in the `BookList` component. Add the classes `table table-bordered table-striped` to the table itself.

15. Finally, remove the text `Where you react to books` from `App.js`.

16. Run and test your application. It should look something like the screenshot below.



17. Press CTRL+C to stop the React application.

Congratulations! You have completed the exercise.

This page intentionally left blank.

Exercise 2.1: Passing Properties to Components

In this exercise, you will set up your `Book` component to receive `props` and pass the values in from the `BookList` parent component.

1. Open `React/Exercises/exercise-2.1` with VS Code, add the `node_modules`, and use `npm start` to run the application.

2. Modify the JSX inside the `Book` component so that `title` and `author` are no longer hard-coded inside the component, but are provided by `props` passed in to the function as an argument. The code for this step is below.

```
function Book(props) {

const { title, author } = props;
```

The table body no longer has any data. You will fix this by passing in `props` from the parent component `BookList`. Later, you will retrieve the data from a Redux store. For now, you will simply create the data inside `BookList`.

3. Modify the `Book` JSX element inside the `BookList render()` method to pass in a `title` and `author` prop of your choice.

```
<Book title="The Lord Of The Rings" author="J R R Tolkien" />
```

The page works, but this is hardly a satisfactory solution. The individual `Book` components should be built dynamically from array data, which is what you will do next.

4. Return to the `BookList` component, and add a `const books` inside the `render()` method, before the `return` statement. Create an array of two object literals, each with `title` and `author` properties, and assign the array to the `books` variable.

```
const books = [{
            title: "The Lord Of The Rings",
            author: "J R R Tolkien"
        },{
            title: "The Hobbit",
            author: "J R R Tolkien"
        }];
```

5. Next, use `books.map()` to pass the `title` and `author` props into instances of the `Book` component. The code for this step follows.

```
{
   books.map(function (item, i) {
      return <Book author={item.author} title={item.title} />;
   })
}
```

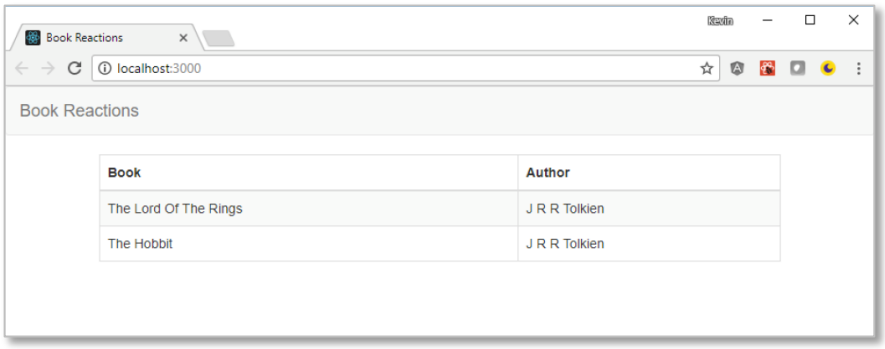
6. Save your work and examine the results in Google Chrome. You should see two rows inside the table, and this code appears to work correctly. However, there is a problem with the code as written. Open the developer tools and examine the warning. What is missing from the `Book` component tag?

In React, iterated child components should have a unique key to allow React to keep track of elements as they are sorted, reordered, etc.

7. Add a `key` property to the `Book` tag and set it to the **`i`** iterator variable. Then, refresh Google Chrome and examine the developer tools. The warning message should no longer be present.

```
return <Book author={item.author} title={item.title} key={i}  />;
```

Your page should look something like this.



Your Book component works, but it does not currently define what `props` it expects, or whether they are required. You will use `PropTypes` to correct this.

8. Stop the application and run the following in the command window to install the `PropTypes` package:

```
npm install prop-types
```

9. When the command completes, examine the file `package.json` in the root and note that a dependency has been added.

Note that you can add npm packages to the base configuration supplied by `create-react-app`.

10. Open `Book.js` and `import PropTypes from 'prop-types'`. Then add code just before the `export` statement to set `Book.propTypes` equal to an object literal.

11. Inside the object literal, define two properties, `title` and `author`. Each one should be set equal to `PropTypes.string`. Both strings are required.

    The code for both these steps is as follows:

```
Book.propTypes = {
   title: PropTypes.string.isRequired,
   author: PropTypes.string.isRequired
};
```

12. Run the application again. It should work as before, but now you are defining the expected `props`.

13. Experiment with removing either the `title` or `author` attribute from the `<Book />` element in `BookList.js`. You will need to have the Google Chrome development tools open to see the warning re. missing `props`.

14. Press CTRL+C to stop the React application.

Bonus Exercise (to be attempted if time permits):

You will add default values to replace any missing properties passed in to the `Book` component.

15. Remove `required` from the `book.propTypes`.

You will be using standard JavaScript default values for arguments if no value is submitted, so the value can never be empty. Required is no longer necessary.

16. Modify the destructuring code in the function to supply the string literal `unknown` as the default value for both `author` and `title`.

```
const { title = "unknown", author  = "unknown" } = props;
```

17. Add two new objects to the `books` array in `BookList`. For one, assign a `title` but no `author`. For the other, assign an `author`, but no `title`.

18. Run the application again. It should work as before but substituting default `props` for missing values.

19. Press CTRL+C to stop the React application.

Congratulations! You have completed the exercise.

This page intentionally left blank.

Exercise 2.2: Working with Forms, State, and Events

In this exercise, you will create a React form and update `this.state` using events in a class-based component.

1. Open `React/Exercises/exercise-2.2` with VS Code, add the `node_modules`, and use `npm start` to run the application.

2. Add a new folder `forms` underneath the `components` folder and add a file `BookForm.js` inside the new folder.

You are going to create a form with controlled inputs and use it to add a new book to the array passed in from a parent component. Later, you will replace the form with a more fully featured implementation using `formik`

3. The new file should contain a class `BookForm` that `extends React.Component`. It should have `constructor()` and `render()` methods. For now, the constructor should simply call `super()`, and the render should return an empty `<form>`. `BookForm` should be the `default export`.

   The complete code so far follows:

```
import React from 'react';

class BookForm extends React.Component {
    constructor() {
        super();
    }
    render() {
        return (<form>
        </form>);
    }
}

export default BookForm;
```

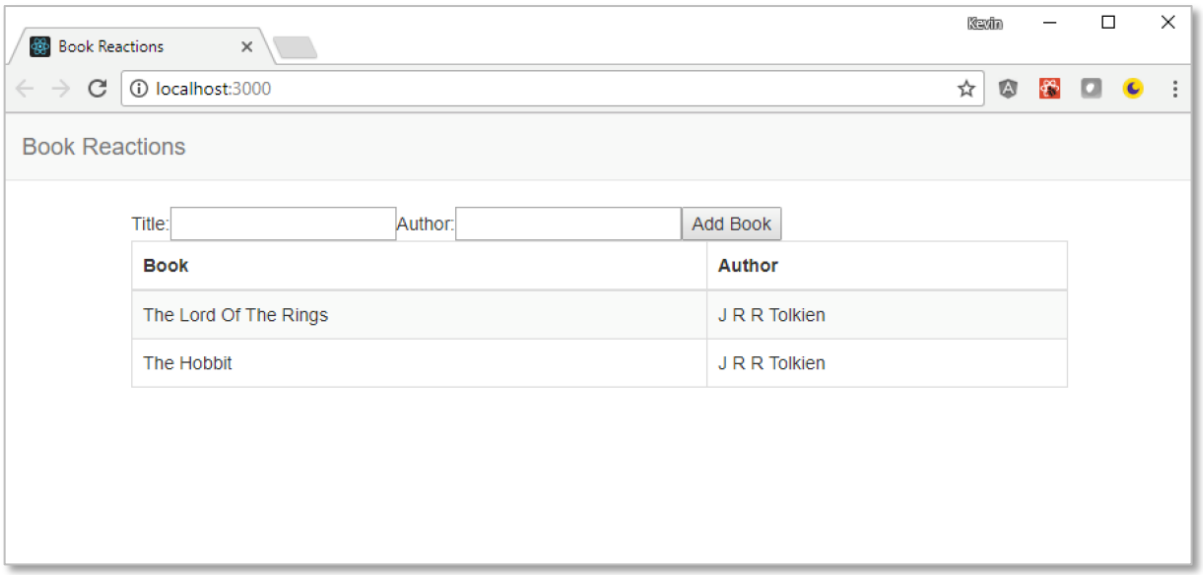
4. Inside the `return` statement, use JSX to create a form with two `text input` controls. one for `Title` and one for `Author`, and a `button` with the type `submit`. At this stage, the code for the form is as follows:

```
return (<form >
        Title:<input type="text" />
        Author:<input type="text" />
        <button type="submit" >Add Book</button>
  </form>);
```

Don't worry about trying to make the form pretty or user-friendly at this stage. Focus on making it work.

5. Save `BookForm.js`. Then open `BookList.js` and add an `import` statement for `'../forms/BookForm'`. Add the `<BookForm />` tag immediately below the opening `<div>` element in the `render` method.

6. Save your work and view the page in Google Chrome. You should see your new form above the books table, as in the screenshot:



7. Next, use the `constructor` in `BookForm.js` to initialize `this.state` to an object with two empty string properties, `title` and `author`.

```
constructor() {
   super();
  this.state = {
      title: "",
      author: ""
  };
}
```

8. Move to the `render()` method and set the values of the inputs to `this.state.author` and `this.state.title`, respectively. The code for the author input follows.

```
Author:<input type="text" value={this.state.author} />
```

9. Next, create a `setTitle(e)` method, and use it to call `setState()` on the `title` property of the `state` object.

```
setTitle(e) {
   this.setState({title: e.target.value});
}
```

10. Add the matching `onChange()` event handler to the title `input` inside the `render()` method.

```
<input type="text" onChange={e => this.setTitle(e)}
                            value={this.state.title}  />
```

11. Create the equivalent method and event handler for the `author` input, and then test your page in Google Chrome to ensure that you can successfully type in the textboxes and set the values in state.

Remember, you can check state and props using the Google Chrome React Developer Tools extension.

We now have working controlled inputs, but the form still doesn't do anything. Before we can complete it, we need to make changes to the parent component, `BookList`.

12. Add a `constructor` to the `BookList` class, and use it to initialize `this.state` to an object with the single property `books`. Assign the value of the `const books` from the `render()` method as the value of `books`. Then delete the **`const`**, as it is no longer needed. The code for the constructor is as follows:

```
constructor() {
 super();
 this.state = {books: [{
                        title: "The Lord Of The Rings",
                        author: "J R R Tolkien"
                },{
                         title: "The Hobbit",
                         author: "J R R Tolkien"
                }]};
}
```

13. Next, modify the call to `books.map()` in the `render()` method so that you are iterating through your new `this.state.books` variable.

```
{this.state.books.map(function (item, i) {
  return <Book author={item.author} title={item.title} key={i} />;
  }
 )}
```

You now have books available as a state variable. The next step is to create a method which can add new books to the `books` array.

14. Create an `addBook` arrow function with the arguments `(title, author)` inside `BookList`. In the function, `push()` a new object into `this.state.books`. Then call `setState()` and assign `this.state.books` to the `books` variable.

```
addBook = (title, author) => {
     this.state.books.push({
         title: title,
         author: author
     });
     this.setState({ books: this.state.books });
}
```

15. Next, pass the `addBook` function into `BookForm` as a property.

```
<BookForm addBook={this.addBook}  />
```

`BookList` is now complete. It is time to return to `BookForm` and prepare it to receive and call the `addBook` function.

16. Add an `addBook()` arrow function to `BookForm`. This method should:
    a. Have a single argument `e`.
    b. Use `e.preventDefault()` to prevent submission of the form.
    c. Call `this.props.addBook`, passing in `this.state.title` and `this.state.author`.
    d. Reset the `title` and `author` state properties to empty strings to clear the form.

```
addBook = (e) => {
  e.preventDefault();
  this.props.addBook(this.state.title,
                     this.state.author);
  this.setState({title: "", author: ""});
}
```

This code is more complicated than it will later become and gives an early hint of the kind of complexity that leads React to use the Flux pattern for simpler, cleaner unidirectional data flow.

17. Finally, add an `onSubmit` event handler to the `form`, pointing at `this.addBook` and binding it to the current object.

```
<form onSubmit={this.addBook}>
```

18. Test your page. You should now be able to add new books and see them displayed inside the table.

19. Press CTRL+C to stop the React application.

Bonus Exercise (to be attempted if time permits):

It would be better if the `BookForm` informed the parent component developer that the `AddBook` function is required. Modify your code to accomplish this.

20. Import `PropTypes` to `BookForm`, and specify the necessary function is required. Hint: the type for functions is `func`.

Your form works (at a very basic level), but it is not well-formed or styled.

21. Modify your form to use labels instead of plain text and use bootstrap classes to manage the look and feel.

You can always look at the solution if you need hints on how to do this.

Congratulations! You have completed the exercise.

Exercise 2.3: Using State in Functional Components

In this exercise, you will add state to functional components with the `useState` hook.

This start point for this exercise is the solution to the previous exercise combined with some new content. A `reviews` folder has been created, along with two functional components: `ReviewList` and `Review`. These components contain no new learning points, so have been pre-created. `App.js` has been temporarily modified to display `ReviewList` rather than `BookList`.

You will add and implement a `ReviewForm` functional component and utilize the `useState` hook in both `ReviewForm` and `ReviewList` to manage state.

1. Open `React/Exercises/exercise-2.3` with VS Code, add the `node_modules`, and use `npm start` to run the application.

2. Add a new file `ReviewForm.js` inside the `forms` folder.

You are going to create a form with a single controlled input and use it to create a new review. This review will be passed up to the parent, where it will be added to an existing review array. As with the book form, you will later replace this form with a more sophisticated implementation using `formik`.

3. The new file should contain a functional component `ReviewForm`. For now, `import` the non-default export `useState` from `react`, pass `props` to the function, and return `<form></form>` from the function. Set `ReviewForm` as the `default export`.

The complete code so far is on the next page.

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

function ReviewForm(props) {

    return (<form></form>)
}

export default ReviewForm;
```

4. Inside the `return` statement, use JSX to create a form with a single `text input` control. and a `button` with the type `submit`. At this stage, the code for the form is as follows:

```
return (<form>
         Review: <input type="text" />
         <button type="submit">Add Review</button>
        </form>);
```

Don't worry about trying to make the form pretty or user-friendly at this stage. Focus on making it work.

5. Save `ReviewForm.js`. Then open `ReviewList.js` and add an `import` statement for `'../forms/ReviewForm'`. Add the `<ReviewForm />` tag immediately above the `<table>` element in the JSX. Then run your application and ensure that everything is working so far.

You are going to take advantage of the `useState` hook in both components. First, you will set a simple string inside the form. Later, you will set the state for an array of objects inside the list component.

6. Return to `ReviewForm.js` and add a `const` array inside the function. The array should have two values, `content` and `setContent`, and should be assigned `useState`, passing in an empty string as the sole argument.

```
const [content, setContent] = useState("");
```

7. Inside the form, set the value of `input` to `content`, and assign an arrow function to the `onChange` event. Use the arrow function to pass `e.target.value` to `setContent`.

```
<input type="text" value={content}
    onChange={(e) => setContent(e.target.value)} />
```

8. Save your work, return to the browser, and type in the textbox. At this point, it should work as expected.

9. Return to `ReviewList.js` and add an `import` for `useState` from React. Then copy the value of the `reviews const` before deleting it entirely.

10. Next, create a `const` array with the two values `reviews` and `setReviews`, assigning `useState` and passing in the array you copied earlier. The code for the `const` follows:

```
const [reviews, setReviews] = useState(
      [{
          content: "A towering masterpiece."
      },
      {
          content: "I hated it."
      }]
);
```

You have now created all the necessary state. The next step is to create a method to update the array by adding a new review, and to pass this function to the form component as a `prop`.

11. Still inside `ReviewList.js`, create a `const` arrow function `addReview` accepting a single argument `review`.

12. Inside the `addReview()` function, create an arrow function that calls `setReviews()`. The function should accept the single argument `oldReviews` and use the array spread operator to create a new array combining `oldReviews` and `reviews`.

The code for the last two steps follows.

```
const addReview = (review) => {
   setReviews(oldReviews => [...oldReviews, review]);
}
```

13. Finally, inside the JSX pass `addReview` to `ReviewForm` as the value of an `addReview` property.

```
<ReviewForm addReview={addReview} />
```

14. Return to `ReviewForm.js` and destructure the `props` argument into an `addReview const`. Then define an arrow function `const createReview` that accepts a single argument `e`.

15. Inside the body of the function, call `e.preventDefault()` and then call the `addReview` method passing in a single object literal argument where `content` is assigned to `content`.

The code for the last two steps follows.

```
const {  addReview  } = props;
const createReview = (e) => {
     e.preventDefault();
     addReview({content: content});
}
```

16. Finally, complete the functionality by assigning `createReview` to the form's `onSubmit` handler in the JSX.

```
<form onSubmit={createReview}>
```

17. Test your work in the browser. You should now be able to add reviews to the array using your form.

Bonus Exercise (to be attempted if time permits):

18. Your `ReviewForm` relies upon the `addReview` prop. Import `propTypes` and set the function as required.

Your form works (at a very basic level), but it is not well-formed or styled.

19. Modify your form to use labels instead of plain text and use bootstrap classes to manage the look and feel.

You can always look at the solution if you need hints on how to do this.

Congratulations! You have completed the exercise.

This page intentionally left blank.

Exercise 2.4:
Improving Performance with `shouldComponentUpdate()`

In this exercise, you will prevent unnecessary calls to the `render()` method with the `shouldComponentUpdate()` component lifecycle method and `React.memo()`.

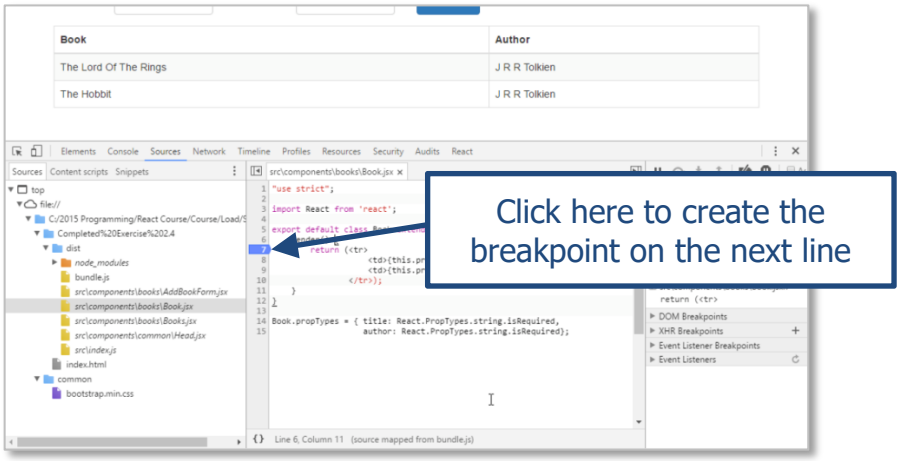
This start point for this exercise is the solution to the previous exercise with the `Book` component rewritten as a class component. At this point, `BookList` and `Book` are class components, while `ReviewList` and `Review` are functional components. This allows us to explore both approaches to lifecycle events going forward.

1. Open `React/Exercises/exercise-2.4` with VS Code, add the `node_modules`, and use `npm start` to run the application.

2. View the page in Google Chrome, open the developer tools, select the sources pane, and navigate down the treeview to find the `Book.js` file. Double-click the file to open it inside the developer tools.

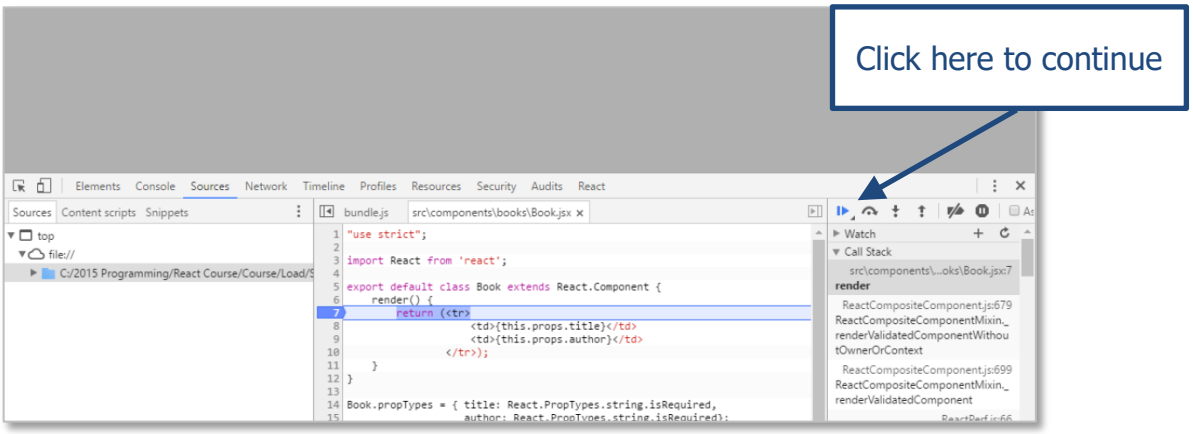
Note that `BookList` has been set as the child of `App.js`. Later, we will use routing to avoid hard-coding what is displayed.

3. Click the line number in the side bar beside the line with the `return` statement inside the `render()` method declaration. This will create a breakpoint.



4. Refresh the page and observe how often the `render()` method is called.

You can resume after pausing at the breakpoint by clicking the blue arrow above the right pane.



5. Use the `BookForm` in the web page to add a new book, and again observe the number of times the `render()` method is called.

You should see that the `render()` is called twice for every row, including those that have already been added to the DOM. The double call only happens in development mode and is part of React's strict mode, so in production, render will be called once for every row. The diffing engine will prevent unnecessary writes to the DOM, but this is still a waste of resources. You will now write code to ensure that the `render()` method is only called if there are relevant changes.

6. Add the `shouldComponentUpdate()` method to the `Book class`, accepting a single argument, `nextProps`.

```
shouldComponentUpdate(nextProps) {}
```

You could also add the `nextState` argument, but it is not needed in this instance.

7. Within the method, return the result of testing whether `nextProps.title` is not equal to `this.props.title`.

```
return (this.props.title !== nextProps.title);
```

8. Save the file, remove your breakpoint in Google Chrome, and refresh the page. Set the breakpoint again at the same point in your code. Bear in mind that the line number may have changed in the new version. Refresh the page again and watch the code break on the breakpoints. Initially, it should behave exactly as it did before you added the `shouldComponentUpdate()` method.

9. Again, add a new book using the form. This time, your breakpoint should be hit only for the new table row that has not previously created in the DOM.

10. Add another new book. Once again, the `render` method should run only twice, demonstrating the benefits of using `shouldComponentUpdate` to eliminate unnecessary processing.

You have prevented unnecessary updates in a class component. Now it is time to do the same in a functional component. You will begin by resetting `App.js` to display the book list component.

11. Open `App.js` and comment out the `import` for `BookList`. Then uncomment the `ReviewList import` and modify the JSX to reference `ReviewList` instead of `BookList`. Save your file and make sure that you are now seeing reviews in the browser.

12. Use the browser tools to set a breakpoint inside the JSX returned from `Review.js` and try refreshing the page and then adding reviews.

You should see the same behavior you saw with `Book.js`.

13. Next, add an `import` for the default export `React` from `react`, and then wrap the export of `Review` in a call to `React.memo()`. The code for this step follows, with the intermediate code omitted to save space.

```
import React from 'react';
```

```
// code ommitted

export default React.memo(Review);
```

14. Once again use the browser tools to check the result. You should see that `React.memo()` provides the same behavior as `shouldComponentUpdate()`.

15. While your application is still running, click your breakpoints in the Google Chrome developer tools to remove them and prevent unexpected interruptions in later exercises.

16. Press CTRL+C to stop the React application.

Congratulations! You have completed the exercise.

Exercise 3.1: Unit Testing React

In this exercise, you will create automated unit tests for React components.

1. Open `React/Exercises/exercise-3.1` with VS Code and add the `node_modules`. Do NOT use `npm start` to run the application.

2. Run `npm test` in the VS Code command window and examine the output from the single existing test.

You should see that a single test runs and fails. Testing is pre-configured by `create-react-app` and uses `jest` and `react-testing-library`. The pre-created test tested the original version of `App.js`, which has changed considerably over the previous exercises.

You may also notice that the console output includes content from child components. React Testing Library deliberately avoids shallow rendering. If you wish to test a component in isolation, you can do so using `jest` mocks. There is a bonus exercise to modify this test to use mocks.

3. Open the file `src/App.test.js` and replace the string in the `test()` method with the following: `renders application without crashing`. Then remove everything inside the method except `render(<App />);` and save the file.

You should see that the test runs automatically, and that this time it passes. The test process remains active, watching for changes to your code as you work. That wasn't much of a test, but the root file is essentially a container. You will proceed to test components that have actual content.

4. Add a new file `Book.test.js` in the `src/components/books` folder.

You should see a fail message in the console. Expect to see fails as you add new test files, as an empty test file automatically fails.

5. Next, copy all the content from `App.test.js` and paste it into the file. Then replace all instances of `App` with `Book`. At this point, your code should be as follows:

```
import { render, screen } from '@testing-library/react';
import Book from './Book';

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

6. Save your work and note that while your test passes, there is a console error message. Examine the error message in the console.

By default, `react-testing-library` wraps components in a `div`, but your component is returning a `tr`, which means the resulting HTML is invalid.

7. Modify the call to the `render()` method so that your `Book` component is wrapped inside valid HTML, save your work, and examine the result in the console.

```
render(<table><tbody><Book /></tbody></table>);
```

Your test now works, but it is not yet very useful. You will now modify the test to check if the component is correctly displaying the `props` passed into it.

8. Modify the text for the test so that it reads `it should build the table row from strings passed as props`. Then define a `const input` inside the test function. Set `input` equal to a string literal with `title` and `author` properties.

```
const input = {
    title: 'Title 1',
    author: 'Author 1'
}
```

9. Next, add `title` and `author` attributes to the `Book` element, passing in the appropriate properties of `input`.

```
render(<table><tbody><Book author={input.author}
      title={input.title} /></tbody></table>);
```

Your test is set up correctly. All that remains is to test that the values were correctly set and displayed. For that, you need to find the content in the rendered component.

10. Use `screen.getByText()` to look for `input.title` and assign the result to a `const title`. Then call `expect()` passing in `title`, and call the `toBeInTheDocument()` method. Save your work. Your test should pass.

```
const title = screen.getByText(input.title);
expect(title).toBeInTheDocument();
```

11. Repeat the above step, but this time for `author` rather than the title. Save your file. The two tests should pass.

The complete code for `Book.test.js` follows on the next page.

```
import { render, screen } from '@testing-library/react';
import Book from './Book';

test('it should build the table row from strings passed
as props', () => {
    const input = {
        title: 'Title 1',
        author: 'Author 1'
    };
    render(<table><tbody><Book author={input.author}
        title={input.title} /></tbody></table>);
    const title = screen.getByText(input.title);
    expect(title).toBeInTheDocument();
    const author = screen.getByText(input.author);
    expect(author).toBeInTheDocument();
});
```

You have tested that a component is displaying correctly. Next, you will test the behavior of the `ReviewForm` component.

12. Create a new file `ReviewForm.test.js` in the `forms` folder. Copy over the content of `App.test.js`, change all references to `App` to `ReviewForm`, and save the file.

At this point, your test should pass, but you will see a console error because `ReviewForm` has not been passed a required `prop`.

13. Use `jest.fn()` to mock a `const onSubmit`. Then modify the r`ender()` method to pass `onSubmit` as the value of the `addReview` property. Save your work. At this point the test should pass with no errors, though it is not yet doing anything especially useful.

```
const onSubmit = jest.fn();
render(<ReviewForm addReview={onSubmit}  />);
```

You are about to simulate clicking the submit button inside `ReviewForm`, and then checking that the `onSubmit` method passed from the parent component (or in this case, the testing harness) is called appropriately. You will begin by providing a more suitable description of the test.

14. First, add `fireEvent` to the imports from `@testing-library/react`. Then modify the text describing the test so that it reads: `it should call the parent function when Add Review is clicked`.

15. Next, assign the return from the existing call to `render()` to a destructured `const getByText`. Then, call `fireEvent.click()`, locating the button to `click` by finding `Add Review` using `getByText()`.

```
const { getByText } = render(<ReviewForm addReview={onSubmit} />);
fireEvent.click(getByText(/Add Review/i));
```

16. Finally, add an `expect()` that checks that `onSubmit` was called. Save your work. The test should pass.

The complete code for `ReviewForm.test.js` so far is as follows.

```
import { render, screen, fireEvent } from '@testing-
library/react';
import ReviewForm from './ReviewForm';

test('it should call the parent function when Add Review
is clicked', () => {
    const onSubmit = jest.fn();
    const { getByText }
        = render(<ReviewForm addReview={onSubmit}  />);
    fireEvent.click(getByText(/Add Review/i));
    expect(onSubmit).toBeCalled();
});
```

This has demonstrated that the child component calls the method passed from the parent. Next, you will check not only that an event is called, but that it behaves correctly. You will begin by importing `userEvent` from a separate `react-testing-library` module.

17. Copy the current test and paste it lower down inside the same file. You will modify this copy to create your new test. Next, add a new `import` to the file for `userEvent` from `@testing-library/user-event`.

18. Change the description of your new test to: `it should allow users to enter a review`. Then add a new `const input` assigning the value as the string literal `A great book` and remove the final two lines of the test. The complete code for the test so far follows:

```
test("it should allow users to enter a review", () => {
  const onSubmit = jest.fn();
  const input = "A great book";
  const { getByText } =
        render(<ReviewForm addReview={onSubmit} />);
});
```

19. After the `render()`, call the `userEvent.type()` method to simulate typing. Use `screen.getByLabel()` to locate the `input` with the `label Add Review` and pass the string `input` you created earlier as the second argument.

```
userEvent.type(screen.getByLabelText(/Review:/i), input);
```

20. Finally, complete the test by adding an `expect()` that checks that the textbox has been updated with the `input` string. This will ensure that the state will be correctly set inside the component as the user types.

```
expect(screen.getByLabelText(/Review:/i))
              .toHaveValue(input);
```

21. Save the file. All your tests should now run successfully.

You have successfully written React unit tests. There are two optional bonus exercises, one using mocking for a truly isolated unit test, and the

second allowing you to practice what you have learned with much less direction.

Bonus Exercise (to be attempted if time permits):

You will add mocks for shallow rendering to ensure a truly isolated unit test. You will begin by using debug to write the component's generated HTML to the screen even when a test passes.

22. Return to `App.test.js` and assign the destructured `const debug` to the return from the call to `render()`. Then add a call to `debug()` at the bottom of the test. Save your file and examine the result in the console.

```
const {debug } = render(<App />);
debug();
```

You should see the generated HTML in the console. Note that all the children and grandchildren of `App` have been rendered. You will change this behavior using `jest.mock`.

23. Add the following code above the test to mock both `Navigation` and `BookList` elements.

```
jest.mock('./components/common/Navigation',
              () => () => (<div>navigation</div>));
jest.mock('./components/books/BookList',
              () => () => (<div>book list</div>));
```

24. Save your file and examine the console messages.

You should see that the generated HTML is using your mock implementations of the components and is therefore performing a truly isolated unit test.

25. Finally, write your own tests for `BookForm` using `ReviewForm.test.js` as a model.

Congratulations! You have completed the exercise.

Exercise 4.1: Retrieving Data with REST

In this exercise, you will retrieve book data from a RESTful web service and display it inside your React application.

This and all subsequent labs rely on the web service to provide data to your React application. If the web service is not running, return to the Express folder and run `npm start`.

1. Open `React/Exercises/exercise-4.1` with VS Code and add the `node_modules`. Use `npm start` to run the application, and then right-click the `src` folder and create a new sub-folder `api`. Then right-click inside the `api` folder and create `index.js`.

`index.js` will eventually contain all the code to make RESTful requests for the Book Reactions application.

2. Next, declare a `const url` and set it equal to a string contining the base URL of the RESTful service.

```
const url = "http://localhost:3030/api/bookreactions/";
```

3. Next, create and `export` a `const fetchAllBooks`, assigning an empty arrow function as the value.

```
export const fetchAllBooks = () => {}
```

4. Inside the arrow function, `return` the result of calling `fetch()`, passing in a string combining the `url` const with the string literal `"Books"`.

```
return fetch(url + "Books")
```

5. Chain a `then()` method to the `fetch()`. The `then()` should contain an arrow function that accepts the single argument `response`, and returns `response.json()`.

The complete code for the method follows, with the last step highlighted in bold.

```
export const fetchAllBooks = () => {
 return fetch(url + 'Books').then((response) => {
     return response.json();
 });
}
```

You have now written to code to call the REST server and extract the array of books from the JSON data returned from the return. Later, you will integrate Redux and call the `api` from inside Redux actions. For now, you will call it directly from inside a component: this works but is not a best practice.

6. Open `BookList` and add a wildcard `import` statement for all methods `from ../../api` as `api`.

```
import * as api from '../../api';
```

7. Add a new `getBooks()` method to the class that accepts no arguments. Inside the method, `return` the result from calling `api.fetchAllBooks()`.

```
getBooks() {
   return api.fetchAllBooks()
}
```

8. Complete the method by chaining a `then()` to the call to `fetchAllBooks()`. Add an arrow function inside the `then()`, with a single argument `response`. In the body of the method, call `setState()` and set `books` equal to `response`.

```
   return api.fetchAllBooks().then((response) => {
       this.setState({ books: response });
  });
```

Your REST call is now ready. All that remains is to remove the existing mock data from the class and call the new method.

WARNING! The next step will work but will trigger a warning message in the browser console. As this is a class component, you will fix it using a lifecycle method.

9. Move to the constructor and delete all the content of the `books` state array, leaving an empty array. Then call `this.getBooks()` and save the file.

```
constructor() {
      super();
      this.state = {
          books: []
      };
      this.getBooks();
}
```

10. Examine the page inside Google Chrome. It should display as before, but this time with data from the server. Open the browser tools and note the error message.

The problem is that you are calling `setState` in the `getBooks()` method, and then referencing `getBooks()` inside the constructor before the component is mounted. You will fix the problem using `componentDidMount`.

11. Add a `componentDidMount()` method to your component, and move the call to `this.getBooks()` into the new method. Then save your work. This time, there should not be any error messages in the browser console.

```
componentDidMount() {
    this.getBooks();
}
```

You will have a similar problem when fetching reviews—but as `ReviewList` is a functional component, the solution to the lifecycle problem will be a hook, not a lifecycle method.

12. Open `src/api/index.js` and copy the existing `fetchAllBooks()` function. Make the following changes to the new function (highlighted in bold in the code sample below):
    a. Name the new function `fetchReviews`.
    b. Modify the signature to accept a single argument, `bookId`.
    c. Modify `fetch()` so that `url` is concatenated with `/Reviews/` and `bookId`.

```
export const fetchReviews = (bookId) => {
    return fetch(url + "Reviews/" + bookId)
        .then(function (response) {
            return response.json();
        });
}
```

13. Next, open `App.js` and modify the JSX to reference `ReviewList` rather than `BookList`. Check that the review page and hard-coded reviews are currently showing in the browser.

14. Open `ReviewList.js` and add `useEffect` to the existing `import` from `react`. Also, import `* as api` from `../../api`.

The `api` method for reviews expects a `bookId`. For now, you will hard-code this as a `const`.

15. Add a `const bookId` inside the component. Set its value to `1` and then modify the existing `useState` definition to receive an empty array instead of hard-coded reviews.

```
const bookId = 1;
const [reviews, setReviews] = useState([]);
```

1 is the id of The Lord of The Rings.

16. Next, add a `useEffect()` method to the component with two arguments. The first should initially be an empty arrow function, the second an array with the single value `bookId`.

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

Recall that `useEffect()` will run after every render. That is usually the desired behavior, but here it would lead to an infinite loop where renders trigger API calls which trigger renders. The second, array argument ensures the effect will only run again if the `bookId` changes. Later, it will change based upon routing, which is exactly the behavior our application needs.

17. Complete `useEffect()` by providing a call to `api.fetchReviews()` in the empty method body, passing in the single argument `bookId`. Add a `then()` with the single argument `response`, and pass `response` to `setReviews()`.

```
api.fetchReviews(bookId).then((response) => {
          setReviews(response);
});
```

18. Save your work and view the result in the browser. You should see a single review for The Lord of The Rings.

You have retrieved data from an API from both class and functional components. The bonus exercise displays an image for the book cover and illustrates a potential problem with string comparisons.

Bonus Exercise (to be attempted if time permits):

The JSON book data returned from the server contains additional information, including a string that can be used to set the `src` property of an `<img>` element. You will update your React components to display that image, and also to deal with situations where the value is set to an empty string.

19. Revert `App.js` to reference `BookList` and use Google Chrome developer tools to examine the `http` request to `Books`, and the JSON data returned from the request. Note that the individual objects in the array have a `cover` property that contains the string for the image.

20. Open `Book.js` and add a required `cover` prop to the `propTypes`. Then add an additional `<td>` after the `title` `<td>`, and use `this.props.cover` to set the `src` attribute of an `<img />` tag, and `this.props.title` to set the `alt` attribute.

21. Next, inside `BookList.js` add an additional `<th>` for the `cover`, and pass `item.cover` to `<Book />` as a `prop`.

22. Check the page in Google Chrome. What happens when an empty string is passed to the `Book` component?

_____

_____

_____

The empty string leads to badly formed HTML where the `<img>` has no `src` property. This is a little more complicated to correct than it appears at first sight. You will explore and fix the problem over the next few steps. First, you will add a default 'no image' image to the public folder so that you can substitute it where there is no cover.

23. Copy the file `NoImage.png` from the `React/Materials` folder into the `public` folder of your application.

WARNING! The next step will not fix the problem. You will need to take additional steps to display the default image.

24. Add `defaultProps` to the `Book` component, setting the cover to `/NoImage.png` and `title` and `author` to suitable string values.

```
Book.defaultProps = {
    title: 'unknown',
    author: 'unknown',
    cover: '/NoImage.png'
};
```

25. Test your page again. This does not fix the problem. Why?

_____

_____

The problem is that an empty string is still valid, so the `defaultProp` for the cover is not being applied.

26. Try fixing your problem by changing the way in which the `cover prop` is passed into `Book`. Return to the `BookList` component, and add a JavaScript OR after `item.cover`, returning `null` if `item.cover` is not true.

```
cover={item.cover || null}
```

27. Test your page again. Does this fix the problem?

The problem remains: nulls on string props are the same as empty strings.

28. Finally, instead of returning `null`, return `undefined`. This time when you test your page, you should see the 'no image available' default image wherever there the book has no cover defined.

```
cover={item.cover || undefined}
```

Congratulations! You have completed the exercise.

Exercise 4.2: Inserting Data with REST

In this exercise, you will add a new book by making a RESTful call to a web service.

1. Open `React/Exercises/exercise-4.2` with VS Code and add the `node_modules`, then run `npm start`.

You are going to add a new method to `api/index.js`, and update the `addBook()` method inside `BookList` to call the new method. The key differences between `addBook()` and `fetchAllBooks()` will be the HTTP method called, and the need to supply additional headers to tell the server what type of content to accept and expect. You will define these header values in a `const` and pass them to a more complex version of the `fetch()` method.

2. Open `src/api/index.js` in VS Code and define a new `const headers`, setting it equal to an object literal. The object literal should have two properties, `accept` and `content-type`. Both should be set equal to `application/json`. Be sure to enclose the `content-type` property name in quotes to allow the use of a hyphen in the property name.

```
const headers = {
    accept: 'application/json',
    'content-type': 'application/json'
};
```

3. Next, copy the existing `fetchAllBooks()` method and paste it inside `index.js`. Rename the new method to `addBook`() and add a single argument, `book`.

```
export const addBook = (book) => {
    return fetch(url + 'Books').then((response) => {
        return response.json();
    });
}
```

4. Modify the `fetch()` method by adding an object literal as a second argument to the `fetch()` method in the pasted code.

```
return fetch(url + 'Books', {    })
      .then(function (response) {
        return response.json();
    });
```

5. The object literal should have four arguments, `method`, `mode`, `headers`, and `body`. Both method and mode arguments are strings. Set the values as follows:
   a. `method: post`
   b. `mode: cors`
   c. `headers: constants.headers`
   d. `body: JSON.stringify(book)`

```
{
     method: 'post',
     mode: 'cors',
     headers: headers,
     body: JSON.stringify(book)
}
```

You are `POST`ing data to a server other than the originating server for this application. This is only possible if cross origin resource sharing (CORS) is enabled.

You have made all the necessary changes to the `api`. Now, you will update `BookList` to call the API method.

6. Open `BookList.js` and modify the `addBook()` method as follows:
   a. Remove the call to `this.setState()`.
   b. Replace `this.state.books.push` with `api.addBook`.

```
addBook = (title, author) => {
    api.addBook({
           title: title,
            author: author,
            cover: ''
    });
}
```

You have written the code to add a book. Now you need to display the new list. The `api.addBook()` method returns the `book` object, along with its server-assigned `bookId`. You could push the new object into the existing array and called `setState()`, but it is simpler to re-fetch the data—and doing so ensures that in the real world, your users would also see any changes made by other users.

7. Chain a `then()` method after the call to `api.addBook()`. Add a no arguments arrow function inside the method, and use it to call `this.getBooks()`.

```
.then(() => {
      this.getBooks();
});
```

8. You should now be able to add new books using REST.

This is still a very naïve implementation (e.g., the lack of validation). You will improve it in later exercises.

Your books don't have covers yet, either. We will also fix that in a later exercise.

Congratulations! You have completed the exercise.

This page intentionally left blank.

Exercise 5.1: Adding Routes to the Application

In this exercise, you will add routing to your Single Page Application.

1. Open `React/Exercises/exercise-5.1` with VS Code and add the `node_modules`. Do NOT use `npm start` to run the application.

React routing has been broken into multiple packages, with the native and browser packages not installed by default. You will begin by installing the appropriate package for your environment.

2. Run the following command to install `react-router-dom`.

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

3. Use `npm start` to run the application, and then create a new `about` folder under `src/components`. Create a file `About.js` inside the new folder.

You are going to create a simple About page and create a route and navigation to allow you to switch between the About and BookList pages.

4. Inside `About.js`, create a functional component `About` with appropriate text and JSX.

You've had some practice at doing this. See if you can create the component without detailed instructions. If you want help, there is a sample of what your component might look like on the next page. It uses the outer Fragment element `<></>` for clean HTML.

```
const About = () => (
    <>
        <div className="row">
            <h1>Book Reactions</h1>
        </div>
        <div className="row">
            <h2>Where you react to books</h2>
        </div>
    </>
)

export default About;
```

You now have a page you can navigate to; the next step is to add routing to your `App.js` file.

5. Open `App.js` and add an `import` statement for `BrowserRouter` and `Route` from `react-router-dom`, as well as importing your new `About` component.

```
import { BrowserRouter, Route } from 'react-router-dom';
import About from './components/about/About';
```

Recall that the *`{ name, name }`* syntax retrieves the named, as opposed to default, exports from the `react-router-dom` module.

6. Next, make the following changes to the JSX in `App.js`:
    a. Add a new root element `BrowserRouter`.
    b. Remove the `BookList` element, and replace it with a `Route` element with the following properties:
        i. `path: /`
        ii. `component: BookList`
        iii. `exact` [this property does not have a value assigned]

The code for this step is on the next page.

```
<BrowserRouter>
    <div className="container-fluid">
      <Navigation />
      <div className="container">
          <Route exact path="/" component={BookList} />
      </div>
    </div>
</BrowserRouter>
```

7. Check your page in Google Chrome. If everything is working, it should look and behave exactly the same as it did before. If not, fix any errors and then proceed to the next step.

The routing infrastructure is now in place; it is time to add a second route.

8. Add a second `Route` element immediately below the first. It should have the following properties:

   a. `path: about`

   b. `component: About`

```
<Route path="/about" component={About} />
```

Now that you have a second route, it is time to add a link to the navigation so that you can access it.

9. Open the `Navigation.js` file in the `components/common` folder and add an `import` for `Link` from `react-router-dom`. `Link` is not a default export from `react-router-dom`.

```
import { Link } from 'react-router-dom';
```

10. Underneath the Book Reactions branding `<a>` element, add a `<ul>` with the CSS classes `nav` and `navbar-nav`, and two nested empty `<li>` elements with the CSS class `nav-item`.

```
<ul className="navbar-nav">
        <li className="nav-item"></li>
        <li className="nav-item"></li>
</ul>;
```

11. Inside the two **`<li>`** elements, add `Link` elements with the `className` `nav-link`. The first link should have the content `Home` and the `to prop` set to `/`. The second link should have the content `About` and the `to prop` set to `/about`

```
<li className="nav-item">
    <Link className="nav-link" to="/">Home</Link>
</li>
<li className="nav-item">
    <Link className="nav-link" to="/about">About</Link>
</li>
```

12. Test your application. You should now be able to navigate between the Home and About pages using the navigation bar.

You may have noticed that we currently have no way of reaching the Reviews page. That route requires parameters and will be implemented in the next exercise.

Bonus Exercise (to be attempted if time permits):

13. Improve the navigation bar by converting the `<a>` element used for branding into a `Link` that navigates to the home page.

Congratulations! You have completed the exercise.

Exercise 5.2: Passing and Receiving Route Parameters

In this exercise, you will set and retrieve route parameters.

1. Open `React/Exercises/exercise-5.2` with VS Code, add the `node_modules`, and use `npm start` to run the application.

2. Open `App.js` and uncomment the `import` statement for `ReviewList`.

You are going to add a `Route` that passes the book `title` and `id` to the `ReviewList` component. You will then modify the code in `Book` to build links to the new route dynamically.

3. Add a third `Route` inside the JSX. The new route should use the `ReviewList` component, and should comprise three segments: `reviews`, and `bookId` and `title` parameters.

```
<Route path="/reviews/:bookId/:title" component={ReviewList} />
```

4. Next, open `Book.js` and `import Link` from `react-router-dom`.

You are going to add a link to the book title and pass through the `bookId` and `title` parameters to the `ReviewList` page. For this, you need to ensure that the `bookId` is added to the list of required props.

5. Update the required `propTypes` to include `bookId` of type `number`.

```
bookId: PropTypes.number.isRequired
```

6. Next, wrap a `Link` component around the content of the `<td>` containing the `title`. Set the `to` prop equal to an ES6 template literal that matches the pattern expected by your route, with the `bookId` param provided by `this.props.bookId` and the `title` by `this.props.title`.

```
<td>
  <Link to={`/reviews/${this.props.bookId}/${this.props.title}`}>
        {this.props.title}
  </Link>
</td>
```

ES6 template literals are created using the back-tick or grave accent character, `` ` ``, and allow the interpolation of variable values into strings.

7. Next, open `BookList.js` and pass `item.bookId` to the `Book` element as the value of `bookId`.

```
bookId={item.bookId}
```

You have set up the parameterized route. Now it's time to use it in the `ReviewList` page to retrieve matching reviews and show the title on the page. First, you will need access to the parameters. The simplest way to do this is to destructure the argument to the function to retrieve the necessary properties.

8. Open `ReviewList.js` and examine the code. At the moment, no argument is specified for the function. You will modify this by destructuring an object passed in to the function. You need to access the `title` and `bookId` properties from `match.params`. See if you can work out how to do this. If not, the code is on the next page.

```
function ReviewList({ match: { params: { title, bookId } } }) {
```

This code makes `title` and `bookId` available throughout the function.

9. Next, modify the component by removing the `bookId const`, and writing the `title` into the `<th>` element in the JSX.

10. Save any unsaved files and test your page. You should now be able to access the list of reviews by clicking a book title, and the page should correctly display any reviews and the title of the book that was clicked. The `useEffect()` method ensures that the API call happens *every* time the parameter changes, but *only* when it changes.

At this point, your components display books and reviews, and you can add new books. However, there is a bug in your code for adding new reviews. It appears to work, but reviews are not being saved properly on the server and will not be retrieved if you navigate away and back. You will fix this in a bonus exercise.

Bonus Exercise (to be attempted if time permits):

Now that you are retrieving review data from the server, your existing `ReviewForm` is too simple. Reviews have a `bookId` property, as well as `content`. You will refactor your code to pass `bookId` to `ReviewForm`, and modify the `createReview()` method appropriately.

11. Modify the `<ReviewForm />` element in `ReviewList.js` by adding a `bookId` property with the value `bookId`. Then modify the `props` destructuring code in `ReviewForm.js` to add `bookId` as a destructured variable.

12. Still inside `ReviewForm.js`, modify `createReview()` by adding a `bookId` property to the object passed to `addReview()`. At the same time, simplify your code by using ES6 shorthand syntax to remove the redundancy of repeating identical properties and values.

```
const createReview = (e) => {
    e.preventDefault();
    addReview({
        content,
        bookId
    });
}
```

13. Test your work. You should now be able to add reviews with the correct `bookId`.

There are some small improvements that can be made in `Book.js`. You will modify the key iterator to use the `bookId`, as non-unique iterator keys can cause UI update issues. You will also take an opportunity to simplify the code.

14. Now that the `Book` component has access to `bookId`, improve your code by having `shouldComponentUpdate` check the `bookId`, which is guaranteed to be unique, rather than the `title`, which is not.

15. Make your `Book.js` code simpler by destructuring `this.props`. The complete content of the revised `render()` method follows:

```
render() {
     const { title, bookId, author, cover } = this.props;
      return (<tr>
          <td><Link to={`/reviews/${bookId}/${title}`}>
                        {title}</Link> </td>
          <td>{author}</td>
          <td><img src={cover} alt={title} /></td>
        </tr>);
 }
```

Congratulations! You have completed the exercise.