

Course syllabus for React Basics

By the end of this reading, you will have learned about the scope of things you will cover in this course.

Prerequisites

To take this course, you should understand the basics of HTML, CSS, and JavaScript. Additionally, it always helps to have a can-do attitude!

Course content

This course is an introduction to React development. You'll learn enough basic concepts to empower you to build simple user interfaces in React.

This course consists of four modules. They cover the following topics.

Module 1: Anatomy of React

In this introductory module, you'll learn about what React is and where it is used. You'll also learn how to set up your coding environment so that you have as productive a learning experience as possible. So, the purpose of this module is to understand the what and the why, and to get set up for the modules that follow.

Components are one of the foundations of React. In React, everything revolves around components. You'll learn how to build components, how to structure and customize your React projects, and how to compose layouts by importing components into other components.

You'll learn about passing data from one component to another. You'll learn about JSX syntax in React and how to use it to structure and style your components.

By the end of this module you will be able to:

- Explain the concepts behind React and component architecture.
- Describe how to use assets within an app to apply styling and functional components.
- Create a component to service a specific purpose.
- Create a folder and demonstrate how to create and import files within that folder.
- Use and manipulate props and components to effect visual results.

Module 2: Data and State

The second module of this course deals with working with events and errors in React. You'll learn how events work and how you can handle them in React. Handling events can sometimes get a bit tricky, so you'll also learn about dealing with errors related to events in React.

By the end of this module you will be able to:

- Use common methods to manage state in React.
- Detail the concept and nature of state and state change.
- Describe the hierarchical flow of data in React.
- Describe how data flows in both stateful and stateless components.
- Use an event to dynamically change content on a web page.
- Describe some common errors associated with events and the syntax required to handle them.

Module 3: Navigation Updating and Assets in React

In this module, you'll learn about routing and navigation in React. You'll learn how to render partial views and how to update routes in your React apps. You'll understand how assets are used, bundled and embedded.

By the end of this module you will be able to:

- Use media assets, such as audio and video, with React.
- Demonstrate how to manipulate image assets using reference paths.
- Explain the folder structure of a React project in terms of embedded or referenced assets.
- Demonstrate the conditional implementation and rendering of multiple components.
- Create and implement a route in the form of a navbar.

Describe navigation design in React, with a focus on single and multi-page navigation.

-

Module 4: Portfolio Mini-Project (Calculator App)

This module is focused on a practical mini project of building a calculator app in React. Upon completing this module, you'll have coded your own mini project in React, as a starting point for building your React portfolio.

You have now learned about the scope of things you will cover in this course.

By the end of this module you will be able to:

- Synthesize the skills from this course to create and style a React component.
- Reflect on this course's content and on the learning path that lies ahead.

Before you learn React

Do you know the fundamentals of HTML, CSS and JavaScript? Perhaps you learned about these technologies from another course. Either way, a quick summary will be useful so let's explore some fundamental HTML, CSS and JavaScript principles and practices.

In this reading, let's take a practical approach, and revisit some of the development techniques you'll need to be comfortable with before learning React.

To get the most out of this course on React basics, you should first understand the fundamental methods and concepts of JavaScript. Otherwise, you may feel like you're a child learning to run before you can walk. React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called "components".

React apps are built using modern JavaScript features, which are commonly known as ES6. Developers use React to develop Single Page Applications. And you can also develop mobile applications with React Native.

As an aspiring developer, you may opt for a 'learn as you go approach' regarding JavaScript and React. But this may not help your productivity and even at times frustrate you. This is because you may confuse code or functionality that is plain JavaScript, or code that is React.

For example, with a solid foundational knowledge of JavaScript, you can quickly identify code that is JavaScript ES6 and code that is React. And throughout this course, there will be help for you along the way with some friendly reminders.

Also, keep in mind that you are using React to build user interfaces which also include HTML and CSS code.

So let's begin with the fundamental HTML knowledge needed to learn React.

HTML

Recall that HTML is used to describe the structure of Web pages. Developers use HTML elements with their opening and closing tags to "mark up" an HTML document.

These elements form the structure of a web page and describe what to display to the web browser. When it comes to HTML it's important to know about:

1. The purpose of HTML in the web browser,

2. the use of HTML tags and correct syntax,
3. and how HTML elements are used in a web document.

Another important concept to know about when you're talking about HTML is the Document Object Model, or DOM.

Users need to be able to interact with elements on a web page. This means that an HTML document must be represented in a way that JavaScript code can query and update it. And that's the function of the DOM. It's a model of the objects in your HTML file.

And web developers interact with the DOM through JavaScript to update content, set up events and animate HTML elements.

Before you learn React, it's advisable that you are comfortable with the following HTML tags and concepts

Layout & Style

- `<html>`
- `<head>`
- `<body>`
- `<div>`

Text formatting & lists

- `<h1>...<h6>`
- `<p>`
- ``
- `<i>`

Images and links

- ``
- ``

Linking and Meta

- `<link>`
- `<title>`

- `<meta>`

Semantic

- `<header>`

CSS

CSS (Cascading Style Sheets) is the code that you use to style HTML. You need to be familiar with basic CSS concepts before you start learning React. This is because you will need to style your React components as well, and basic CSS knowledge will help your learning journey.

Before you learn React, make sure you are comfortable with these CSS styling options:

- Font styling (font size, font color, etc.)
- Flex Box Layout (Layout of items using CSS Flex Box Layout)
- CSS Selectors
- Position, Padding, Margins and Display
- Colors, Background and Icons

You can refresh your knowledge of HTML and CSS in the Meta course titled: [Introduction to Front-End Development](#)

JavaScript fundamentals and ES6

React is completely written in JavaScript and uses the more modern version of JavaScript which is ES6. While learning React, you should already know JavaScript fundamentals.

JavaScript is the programming language and React is a JavaScript UI library. This means the first step is to be proficient at JavaScript.

Here are some of the JavaScript topics that you need to be comfortable with before you begin your journey learning React.

- Data types
- Using var, let and const
- Conditionals and Loops
- Using objects, arrays and functions
- ES6 Arrow functions
- In-built functions such as map(), forEach() and promises.

- Destructuring Arrays and Objects
- Error Handling

Package Manager (Node + npm)

React is a UI library, and you will encounter that many times you will need to add other packages to your React application. A package in JavaScript contains all the files needed for a module. To install these packages effectively and manage their dependencies you can use a package manager like NPM (Node Package Manager).

You can install npm by installing Node.js, which will then automatically install npm.

You need to be comfortable with using npm as your package manager, since you will be using npm to install packages within your React application. Make sure you are aware of how to do the following with npm before you get started on this course.

- Installation command to install npm modules in your project
- Installing a package as a dev dependency
- Start command
- Updating npm version
- Navigating around the package.json file

Once you have become confident with these skills, you'll be in a better position to learn and apply React concepts and prepare yourself for development of React apps.

To refresh your knowledge of JavaScript and the basics of Node and npm, please visit Meta course titled: [Programming with JavaScript](#).

JavaScript modules, imports - exports

Before you start creating the next great app, let's explore a little more about modules.

Modules can help you to save and access your code in a more structured way, and in this reading, you'll learn about some foundational concepts of working with JavaScript modules.

This knowledge is crucial in order to understand the syntax and the logic behind how the example React apps in this course are put together.

This reading will cover the three main concepts:

1. JavaScript modules

2. Module exports
3. Module imports

JavaScript Modules

In JavaScript, a module is simply a file.

The purpose of a module is to have more modular code, where you can work with smaller files, and import and export them so that the apps you build are more customizable and have more composable parts.

A module can be as simple as a single function in a separate file.

Consider the following function declaration:

```
Ex. function addTwo(a, b) {  
  console.log(a + b);  
}
```

Say that you have a file named `addTwo.js` that contains only the above code.

How would you make this file a JavaScript module?

All that you would need to do to make it a JavaScript module is use the export syntax.

Module Exports

There is more than one way to export a module in JavaScript.

While all the various syntactical differences are not listed, here are a few examples that will cover all the ways that the importing and exporting of JavaScript modules will be done in this course.

In general, there are two ways to export modules in JavaScript:

1. Using default exports
2. Using named exports

Default Exports

You can have one default export per JavaScript module.

Using the above `addTwo.js` file as an example, here are two ways to perform a default export:

```
Ex. export default function addTwo(a, b) {
```

```
    console.log(a + b);  
  }  
}
```

So, in the above example, you're adding the **export default** keywords in front of the **addTwo** function declaration.

Here's an alternative syntax:

```
Ex. function addTwo(a, b) {  
    console.log(a + b);  
}  
  
export default addTwo;
```

Named Exports

Named exports are a way to export only certain parts of a given JavaScript file.

In contrast with default exports, you can export as many items from any JavaScript file as you want.

In other words, there can be only one default export, but as many named exports as you want.

For example:

```
Ex. function addTwo(a, b) {  
    console.log(a + b);  
}  
  
function addThree(a, b, c) {  
    console.log(a + b + c);  
}
```

If you want to export both the **addTwo** and the **addThree** functions as named exports, one way to do it would be the following:

```
Ex. export function addTwo(a, b) {  
    console.log(a + b);  
}  
  
export function addThree(a, b, c) {  
    console.log(a + b + c);  
}
```


Here's another way you could do it:

```
Ex. function addTwo(a, b) {  
  
    console.log(a + b);  
  
}  
  
function addThree(a, b, c) {  
  
    console.log(a + b + c);  
  
}  
  
export { addTwo, addThree };
```

Importing Modules

Just like when exporting modules in JavaScript, there are several ways to import them.

The exact syntax depends on how the module was exported.

Say that you have two modules in a folder.

The first module is addTwo.js and the second module is mathOperations.js.

You want to import the addTwo.js module into the mathOperations.js module.

Importing a Module that was Exported as Default

Consider the previous example of exporting the **addTwo** function as a default module:

```
Ex.// addTwo.js module:  
  
function addTwo(a, b) {  
  
    console.log(a + b);  
  
}  
  
export default addTwo;
```

To import it into the mathOperations.js module, you could use the following syntax:

```
Ex.import addTwo from './addTwo';  
  
// the rest of the mathOperations.js code goes here
```

So, you could start this import with the **import** keyword, then the name under which you'll use this imported code inside the mathOperations.js file. You would then type the keyword **from**, and finally the location of the file, *without the .js extension*.

Contrast the above import of the default `addTwo export` with the different import syntax if the `addTwo` function was instead a named export:

```
Ex. import { addTwo } from "./addTwo";
```

```
// the rest of the mathOperations.js code goes here
```

Conclusion

In this reading, you've learned about the very basics of what modules are in JavaScript, why they are used and how they get exported and imported.

The examples you've seen here are the core of how you'll deal with imports and exports of various modules in the example React apps on this course.

However, please note that there are many more caveats, rules, and implementations of working with modules in JavaScript. The examples given in this reading are there just to make it easier to comprehend what is happening in React apps that you'll be building in this course. The intent of this reading was just to get you familiar with the most common syntax used - not as a comprehensive overview of modules in JavaScript.

Additional reading

Below you will find links to helpful additional readings.

- nodejs.org
- npmjs.com
- <https://docs.npmjs.com/common-errors>
- reactjs.org
- <https://create-react-app.dev/>
- [VS Code](#)

Transpiling JSX

By the end of this reading, you will have learned how a component is built.

Introduction

Components are a nice way to build websites in React because they allow you to build more modular apps. However, how do you build components using React, JSX, and JavaScript? You'll learn how this works in this lesson item.

A browser cannot understand JSX syntax.

This means that making a browser understand React code requires a lot of supporting technologies.

An example of such a technology is a transpiler.

A transpiler takes a piece of code and transforms it into some other code.

To understand why this is done, here is an example of an ES6 variable declaration:

Ex. `const PI = 3.14`

This is perfectly valid ES6 syntax.

However, if you were using a very old computer, that computer will have an old browser. Perhaps that browser was built before ES6 came out in 2015.

This means that the JavaScript engine that is built into your old computer's browser is likely to be an ES5 JavaScript engine.

In ES5, the only way to declare a variable is the following:

Ex. `var pi = 3.14`

What this means is that for this old browser to understand the ES6 code, the only way to do it is by transpiling it.

This means you need a way of essentially translating the modern js code into a format that an older js engine can understand.

Let's say that you want to use a brand new, most modern ECMAScript syntax in an app. The only problem is that this new syntax is currently not supported by any browser; even an up-to-date browser.

However, by transpiling the new most-modern JavaScript syntax into something that modern browsers can understand, it is able to convert some code that the browser cannot comprehend, into code that it can comprehend, run, and produce a result from.

Likely the most popular site that shows off how this works is [Babel](#). As the heading of the website reads, "Babel is a JavaScript Compiler".

This finally brings you to the point of this discussion about transpiling JavaScript code.

What Babel does is this: it allows you to transpile JSX code (which cannot be understood by a browser) into plain JavaScript code (which can be understood by a browser).

This is where React and JSX come in.

For React code to be understood by a browser, you need to have a transpiling step in which the JSX code gets converted to plain JavaScript code that a modern browser can work with.

To demonstrate how this works, let's use the **Heading** component from the previous lesson.

Add the JSX code into [the online Babel repl](#). Repl stands for "read-eval-print loop" and it accepts code you write, evaluates it, and produces some result. In the specific case of [the online Babel repl](#), that result is some transpiled code. Here's a more detailed explanation.

If you've visited the above-linked URL, you'll find a web page that has two panels. On the left, there's source JSX code:

```
Ex. function Heading(props) {  
  
  return <h1>{props.title}</h1>  
  
}
```

... and on the right, there's the transpiled, plain JavaScript code. However, ensure that you select the classic runtime for React in the left sidebar.

```
Ex. function Heading(props) {  
  
  return /*#__PURE__*/React.createElement("h1", null, props.title);  
  
}
```

If you now analyze the difference between the source JSX code and the transpiled, plain JavaScript code, dis-regarding the comment, here's the body of the Heading function:

```
Ex. React.createElement("h1", null, props.title);
```

So, here you have a React object, and this object has a **createElement()** method on it. The method is invoked with three arguments:

1. **"h1"**
2. **null**
3. **props.title**

The first argument is the DOM element to render - in this case, an **h1** element. The second property is any HTML attribute that should be added, and there's a null here - meaning, there should be an object with some data, but there isn't any data so instead of the object there's the null value. The third property is the contents of the inner HTML of the DOM element specified as the first argument - in this case, the contents of the inner HTML of the **h1** element.

Now let's use Babel again, and this time transpile the **render** syntax for the **Heading** component:

```
Ex. <Heading title="This is the heading text!"></Heading>
```

Again using [the Babel repl](#), and as can be confirmed in [the link](#), the output of the transpilation is the following code. Ensure that you select the classic runtime for React in the left sidebar.

```
Ex. /*#__PURE__*/  
  
React.createElement(Heading, {  
  
  title: "This is the heading text!"  
  
});
```

Again, you have the **React.createElement()** method call, and this time, the first item to render is **Heading**, and then you have an object as the second argument (instead of a null that you had in the previous transpilation example).

This brings me to an interesting question: What is the minimum code that a component must have to be able to show something on the screen when rendered?

You can see the answer below:

```
Ex. function Example() {  
  
  return <div>An element</div>  
  
}  
  
export default Example
```

Your First Component

Task:

You've learned that you can add another component inside the App component, and then render that new component inside the App component's return statement.

In this exercise, you'll practice doing just that. You will be instructed, step by step, to build a new component inside the App component, and to have it rendered on the screen.

Note: Before you begin, make sure you understand how to work with the [Coursera Code Lab for the React Basics](#) course.

Steps:

Step 1

Add a new component to the App component, named Heading.

- Navigate to the src folder and find the App.js file. Click on the **App.js** file in the EXPLORER to have it open.

- Now you're ready to add a new component to the App component and name that component **Heading**. Place it at the very top of the **App.js** file.

Step 2:

Inside the Heading component, include a return statement, which will return the JSX code.

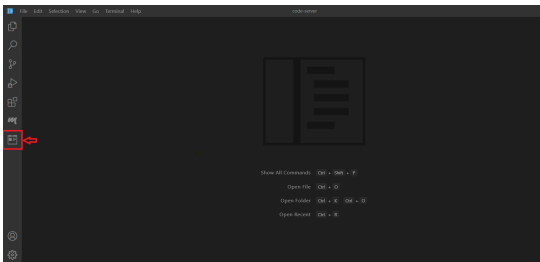
Step 3

To the right of the **return** keyword, add the following JSX code: **<h1>This is an h1 heading</h1>** that displays the H1 element on the screen.

Step 4

Running your code:

- At the top of the lab environment, locate the Terminal menu. Click on it to open a dropdown, then select New Terminal. Use the **npm start** command to start the development server.
- If you encounter errors like *File not found* or *Unexpected token*, in the terminal, stop the server with **Ctrl + C** and restart it using **npm start**.
- You can now view the App in your lab browser. To view the output, click on the **Browser Preview** icon located on the left panel. It is the last icon in the panel.



- When your lab browser has launched, enter: **http://localhost:3000** in the address bar to see the output. Run your code using the play button at the top right of your lab. You should see the heading, "This is an h1 heading", rendered on the screen.

Tip:

If you're having trouble with this lab, please review the "Create React components" video. This video covers all of the concepts that you'll need to successfully complete this lab.

Solution: Your first component

Here is the completed solution code for the App.js file:

Ex. function Heading() {

```

return (

  <h1>This is an h1 heading.</h1>

)

}

function App() {

  return (

    <div className="App">

      This is the starting code for "Your first component" ungraded lab

      <Heading />

    </div>

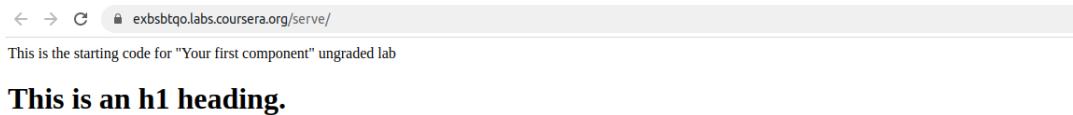
  );

}

export default App;

```

Here is the output from the solution code for the App.js file:



< > ↻ exsbtqo.labs.coursera.org/serve/
 This is the starting code for "Your first component" ungraded lab
This is an h1 heading.

Step 1: In the starting code, you already had a JSX element named `<Heading />`, being rendered from the App component, since it is a part of the App component's return statement.

```

Ex. function App() {

  return (

    <div className="App">

      This is the starting code for "Your first component" ungraded lab

      <Heading />

    </div>

  );

}

export default App;

```

Then, you added a new function to the App component, and named that function **Heading ()**. You placed it at the very top of the **App.js** file.

```
Ex. function Heading() {  
  
}
```

Step 2: Next, in the body of the **Heading** component, you added a return statement and spread it over several lines by following it up with an opening and a closing parenthesis.

```
Ex. function Heading() {  
  
  return (  
  
  )  
  
}
```

Step 3: Then, inside the parentheses, you added the following code: **<h1>This is an h1 heading</h1>**

```
Ex. function Heading() {  
  
  return (  
  
    <h1>This is an h1 heading.</h1>  
  
  )  
  
}
```

Step 4: Finally, you saved your changes and viewed the app in the browser.

Customizing the project

So far, you've learned about React components, but now you will focus on learning how to customize the project. You will learn about the software development approach, detailing the creation of separate associated files, the requirements gathering and the subsequent folder structure to be created.

Building a Layout

Imagine that you've been given the task of building a somewhat more complex website layout using React.

At this point, you still don't know too much about how React works, but even with your limited knowledge, you can still build some relatively interesting designs.

Currently, you need to build a simple typography-focused layout for a coding blog.

This means that you will not have to use images, which simplifies your task significantly.

The layout you're supposed to build will consist of the following sections:

- Main navigation
- Promo (main advertisement)
- A list of newest posts' previews (intros)
- The footer

Organizing Your Code

Keeping in mind the above structure, how would you organize your code?

This is where React docs can help. They suggest two approaches:

1. Grouping by features
2. Grouping by file type

They also advise not to nest folders too deep, and to keep things simple and not overthink it.

They even say that if you're just starting out, you shouldn't spend more than five minutes setting up a project.

Taking this advice into account, you might say that for a small project like this, you could keep it as simple as just adding a components folder and moving all your components into it. This is exactly what you'll do next.

Building The App

Since this app's focus is on customization, let's name the app customizing-example.

What follows is the command to run in a suitable folder on your own computer. By "a suitable folder", I mean: "a folder where you feel comfortable installing a boilerplate React application". This also includes that the folder you chose will need to be accessible for your user on your OS (Operating System).

Ex. `npm init react-app customizing-example`

This will produce a brand-new starter app with a familiar structure.

Inspecting the src folder of the starter app, it looks like this:

Ex.

Then simply add a components folder to it, like this:

Ex. src/

App.js

App.test.js

index.css

index.js

logo.svg

reportWebVitals.js

setupTests.js

Since the components folder is currently empty, you can add a component for each of the sections of the typography-focused blog. Here's the structural update:

Ex. src/

components/

App.js

App.test.js

index.css

index.js

logo.svg

reportWebVitals.js

setupTests.js

At this point, there's no need to complicate things. You have the Nav component, the Promo component, the Intro1, Intro2, and the Intro3 component. Finally, there's also a Footer.js component.

This means you've fully planned the app, based on some best practices as suggested by the official React docs website, and based on the level of complexity of the project itself. Since this project is relatively simple, this structure feels right.

In this reading, you'll just build all the components inside the components folder, and then, in the upcoming lesson items, import them into the App.js file.

Building Components

For now, let's just build those components. After you've added the components folder, you've also added all the functional component files. Since they are all currently empty, you can start adding them, one by one.

Here' s the contents of the Nav.js file:

```
Ex. function Nav() {  
  
  return (  
  
    <nav className="main-nav">  
  
      <ul>  
  
        <li>Home</li>  
  
        <li>Articles</li>  
  
        <li>About</li>  
  
        <li>Contact</li>  
  
      </ul>  
  
    </nav>  
  
  );  
  
};  
  
export default Nav;
```

Next, you can focus on the Promo.js file:

```
Ex. function Promo() {  
  
  return (  
  
    <div className="promo-section">  
  
      <div>  
  
        <h1>Don't miss this deal!</h1>  
  
      </div>  
  
      <div>  
  
        <h2>Subscribe to my newsletter and get all the shop items at 50% off!</h2>  
  
      </div>  
  
    </div>  
  
  );  
  
};  
  
export default Promo;
```

Once you've finished the promo section, you can focus on the Intro components.

Here's Intro1.js:

```
Ex. function Intro1() {  
  
  return (  
  
    <div className="blog-post-intro">  
  
      <h2>I've become a React developer!</h2>  
  
      <div>  
  
        <p>I've completed the React Basics course and I'm happy to announce that I'm now a Junior React Developer!</p>  
  
        <p className="link">Read more...</p>  
  
      </div>  
  
    </div>  
  
  );  
};  
  
export default Intro1;
```

Here's the code for the Intro2.js component:

```
Ex. function Intro2() {  
  
  return (  
  
    <div className="blog-post-intro">  
  
      <h2>Why I love front-end web development</h2>  
  
      <div>  
  
        <p>In this blog post, I'll list 10 reasons why I love to work as a front-end developer.</p>  
  
        <p className="link">Read more...</p>  
  
      </div>  
  
    </div>  
  
  );  
};  
  
export default Intro2;
```

You can finish the previews for my blog posts with the code for Intro3.js component:

```
Ex. function Intro3() {  
  
  return (  

```

```

    <div className="blog-post-intro">

      <h2>What's the best way to style your React apps?</h2>

      <div>

        <p>There are so many options to choose from. Here's a high-level overview of the popular ones.</p>

        <p className="link">Read more...</p>

      </div>

    </div>

  );
};

export default Intro3;

```

There's just one more thing left to code, the Footer component, so here it is:

```

Ex. function Footer() {

  return (

    <div className="copyright">

      <p>Made with love by Myself</p>

    </div>

  );

};

export default Footer;

```

Now that you have completed all the components for the app, here are a few more interesting things about the syntax.

These are:

- The use of the **className** attribute in JSX
- The use of separate components for repetitive code
- Where are all the props?
- Why was I not using the **<a>** element for empty links?

Discussing the Syntax

Now let's briefly discuss the four bullet points above.

Why use the `className` attribute in the JSX syntax?

Well, with JSX, it looks like HTML so much that it's easy to forget that it's actually JavaScript code - not HTML.

While regular HTML does indeed have a `class` attribute, which is used to list one or more CSS classes to be used on a given HTML element, this cannot really work in JSX. The reason is that JSX is a special kind of JavaScript syntax, and the word `class` is a reserved keyword in JSX. That's why the React team had to make a compromise and so `className` is used in JSX to list one or more CSS classes to be used on a given element or component.

But why use `Intro1.js`, `Intro2.js`, and `Intro3.js`? Isn't one of the tenets of coding the DRY approach - that is, the "Don't repeat yourself" approach?

Indeed, it is. However, there are still a few concepts to discuss before you learn how to re-use a single component with variations in its content. This has to do with data in components, but don't worry, we'll be getting to that later.

The third question is about the props object. It has been mentioned before, but so far it hasn't been used. It hasn't been used in this example either.

The answer to this question has to do with the next lesson, titled *Component Use and Styling*. In this lesson, you'll see in practice how you can make components work better, with the help of props.

The final question is about not using the `<a>` element for empty links in my app.

The answer here depends on whether those links are "internal" - inside an app, or "external", meaning, leading to some external link, such as <https://www.coursera.org>. If the links are internal to the app - as they are envisioned here - using the `<a>` tag is simply not the React way of doing things. You'll learn why that is the case when discussing the use of React Router.

Conclusion

Having finished this reading, you have now learned about the software development approach, detailing the creation of separate associated files, the requirements gathering, and the subsequent folder structure to be created.

Solution: Creating and importing components

Here are the contents of the `Heading.js` file:

```
Ex. function Heading() {  
  
  return (  
  
    <h1>This is an h1 heading</h1>
```

```
)  
}
```

export default Heading;

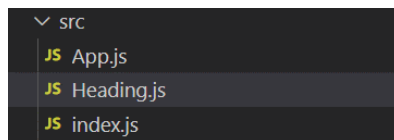
Here are the contents of the App.js file:

Ex. import Heading from "../Heading";

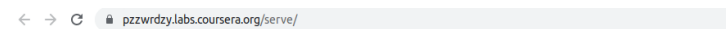
```
function App() {  
  
  return (  
  
    <div className="App">  
  
      <Heading />  
  
    </div>  
  
  );  
}
```

export default App;

Here is a screenshot of the src folder:



Here is the output from the solution code for the App.js file:



This is an h1 heading

Step 1: You moved the **Heading** function from App to a separate component file, named “Heading.js”.

```
Ex. function Heading() {  
  
  return (  
  
    <h1>This is an h1 heading</h1>  
  
  )  
}
```

```
}
```

```
export default Heading;
```

Step 2: Next, you imported the **Heading** component into the **App** component.

Ex. import Heading from "../Heading";

Step 3: Finally, you removed the sentence that reads: *This is the starting code for “Your first component” ungraded lab* - so that only the **Heading** JSX element remains in the return statement of the App component.

Ex. import Heading from "../Heading";

```
function App() {
```

```
  return (
```

```
    <div className="App">
```

```
      <Heading />
```

```
    </div>
```

```
  );
```

```
}
```

```
export default App;
```

Additional resources for React components and where they live

Below you will find links to helpful additional resources.

- [Basic Concepts of Flexbox](#)
- [Importing a Component](#)
- [Babeljs.io](#)
- [NPM docs: package.json](#)
- [git docs: gitignore](#)
- [NPM docs: node modules folder](#)
- [webpack docs: DevServer](#)
- [webpack/webpack-dev-server on GitHub](#)
- [Visual Studio Code keyboard shortcuts \(Windows\)](#)

- Visual Studio Code keyboard shortcuts (macOS)

Setting up a React project in VS Code (Optional)

To complete the exercises in this course you have been provided with a dedicated lab environment set up specifically for you to apply the skills that you have learned. You can find out more about Working with Labs in this course by accessing the link below:

Working with Labs in this course

You can also use VS Code to practice these exercises on your local machine as an alternative option.

To follow along in this reading, you need to have Node.js and VS Code already installed on your computer. If you don't have this setup, please refer to the Programming with JavaScript course:

- Setting up VS Code
- Installing Node and NPM

In VS Code, you're ready to start a brand new React project.

You can do it using npm.

What is npm?

When Node.js is installed on a computer, npm comes bundled with it.

With npm, you can:

1. Author your own Node.js modules ("packages"), and publish them on the npm website so that other people can download and use them
2. Use other people's authored modules ("packages")

So, ultimately, npm is all about *code sharing* and *reuse*. You can use other people's code in your own projects, and you can also publish your own Node.js modules so that other people can use them.

An example npm module that can be useful for a new React developer is create-react-app. While this npm module comes with its own website, you can also find some info on the create-react-app project on GitHub.

Whenever you run the npm command to add other people's code, that code, and all other Node modules that depend on it, get downloaded to your machine.

However, although it's possible to do so, this is not really necessary, at least in the case of the **create-react-app** Node module.

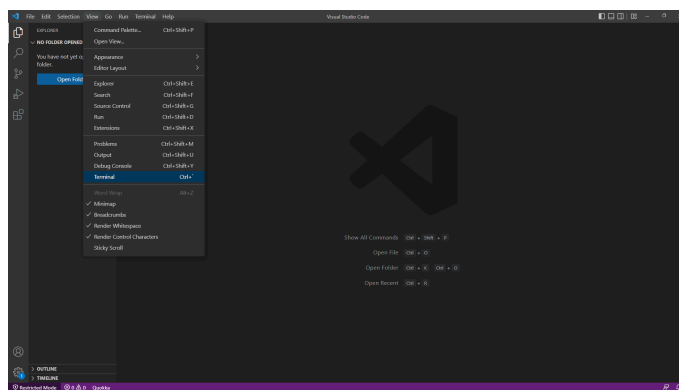
In other words, you can avoid installing the **create-react-app** package but still use it.

You can do that by running the following command: **npm init react-app example**, where “example” is the actual name of your app. You can use any name you’d like, but it’s always good to have a name that is descriptive and short.

In the next section, you'll learn how to build a brand new app that you can name: firstapp.

Opening the built-in VS Code terminal and running *npm init react-app* command

In VS Code, click on *View, Terminal* to open the built-in terminal.



Now run the command to add a brand new React app to the machine:

Ex.

The installation and setup might take a few minutes.

Here's the output of executing the above command:

Ex.

If you follow the suggestions from the above output, you'll run: **cd firstapp**, and then **npm start**.

This will end up with the following output in the built-in terminal:

Compiled successfully!

You can now view firstapp in the browser.

Local: <http://localhost:3000>

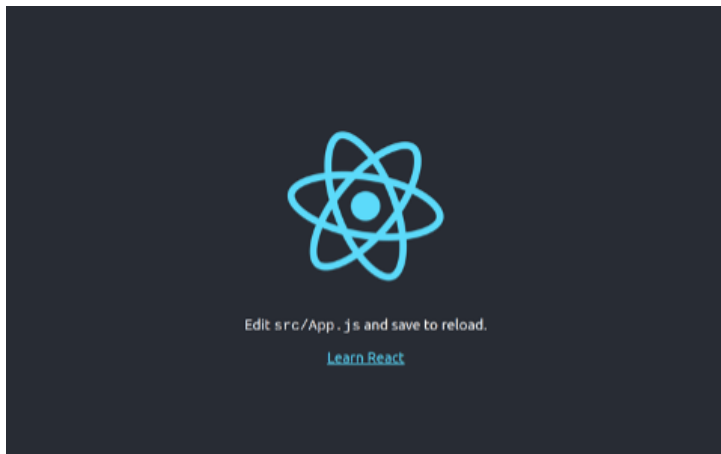
On Your Network: <http://192.168.1.167:3000>

Note that the development build is not optimized.

To create a production build, use `npm run build`.

webpack compiled successfully

Again, following the instructions, opening a browser with the address bar pointing to <http://localhost:3000>, will show the following page in your browser:



This means that you've successfully:

- Set up your local development environment
- Run the create-react-app npm package (without installing it!)
- Built a starter React app on your local machine
- Served that starter React app in your browser

After you've built your starting setup, in Module 2 you'll start working with the basic building blocks of React: components.

Dissecting props

Recall that much like parameters in a JavaScript function which allow you to pass in values as arguments, React uses properties, or props, to pass data between components. But how exactly do they work?

In this reading, you'll use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Remember first that JSX code in React is just syntactic sugar - meaning, a nicer way to write some hard-to-read code.

For the browser to understand this syntactic sugar, you need to transpile JSX down to plain JavaScript code. You have a resource online, at the URL of babeljs.io, which allows you to inspect the results of this transpiling. Once you visit the website, make sure to navigate to the *Try it out* link in the main navigation.

For example, let's say you have a component that returns a piece of JSX:

```
Ex. function App() {  
  return <h1>Hello there</h1>  
}
```

... if you used the Babel transpiler to transpile this JSX syntactic sugar code down to plain JavaScript code, you'd get back some unusual code:

```
Ex. output: "use strict";  
  
function App() {  
  return /*#__PURE__*/React.createElement("h1", null, "Hello there");  
}
```

You just want to focus on the `React.createElement("h1", null, "Hello there");` part. You can ignore the rest.

This means that the `createElement` function receives three arguments:

1. The wrapping element to render.
2. A null value (which is there to show an absence of an expected JavaScript object value).
3. The inner content that will go inside the wrapping element.

Interestingly, the inner content that will go inside the wrapping element can also be a call to the `createElement` function.

For example, let's say you have a slightly more complex JSX element structure:

```
Ex. function App() {  
  
  return (  
  
    <div>  
  
      <h1>Hello there</h1>  
  
    </div>  
  
  )  
}
```

... the transpiled return statement in plain JavaScript again returns two **createElement** functions:

```
Ex. "use strict";  
  
function App() {  
  
  return /*#__PURE__*/React.createElement("div", null, /*#__PURE__*/React.createElement("h1", null, "Hello there"));  
  
}
```

If you format this output, remove the **"use strict"** line, and remove the **__PURE__** comments, you get a more readable output:

```
Ex. function App() {  
  
  return React.createElement(  
  
    "div",  
  
    null,  
  
    React.createElement("h1", null, "Hello there")  
  
  );  
}
```

So now the third argument of the outer-most **React.createElement** call is another **React.createElement** call.

This is how you can nest as many elements as you want.

This means that a nested JSX structure is just a bunch of nested **React.createElement** calls, passed in to other **React.createElement** calls as their third argument.

The second – **null** – argument

The second argument of **null** can – in this case – be replaced with an empty object.

In that case, your code would contain a pair of curly braces instead of the word `null`:

Ex. `"use strict";`

```
function App() {  
  
  return React.createElement(  
  
    "div",  
  
    {},  
  
    React.createElement("h1", {}, "Hello there")  
  
  );  
}
```

This object is referred to as the *props* object. It is the main mechanism of sending data from a parent component to a child component in React.

The way this works is described in React docs using the following code:

```
Ex. React.createElement(  
  
  type,  
  
  [props],  
  
  [...children]  
  
)
```

The third argument (...children)

This is the inner content that will go inside the wrapping element. It's what makes it possible to nest elements inside other elements, mimicking the way that HTML works.

In this reading you've learned how to use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Task

You've learned that you can send data in React through the use of props in components. Props allow you to create components that are reusable and contain dynamic data.

Let me demonstrate this with a basic example. In this lab, you're going to create a component with some simple "Hello, ____" text. If you know in advance that you're always going to be saying hello to "Bob", then you could simply hard-code the name "Bob" into your component. But what if the name of the person you're saying "hello" to could change? Or what if you want

to say “hello” to multiple different people at different places in your application? In these cases, hard-coding a name wouldn’t work very well. Instead, you'll need to create a component that uses props to pass this name data.

Before you begin, if you run `npm start` and view the app in the browser you'll notice that the starting React app works as is. The app outputs a rather simple user interface. You'll build from this simple starting point.

← → ⓘ yhuqzhr:labs.courses.org/serve/

Hello,

STEPS

Step 1

Open the `Heading.js` file. You’ll see the **Heading** component has already been created.

However, the props object is not passed to it. In order for the component to receive data as props, the props object needs to be passed in to the function.

You’ll need to pass the props object as a parameter.

Step 2

Now you’ll want to use the props data in the **Heading** component.

You’ll need to use JSX to access a property of the props object.

Inside the **Heading** component’s body, locate the **return** statement. Inside the **return** statement's **h1**, after **Hello**, add a JSX expression that accesses the **firstName** property of the props object.

Step 3

Now that the **Heading** component is all setup to receive and use props data, you’ll need to create some data to send to it.

Open the `'App.js'` file, and inside of its return statement, locate the **<Heading />** JSX element. Add the attribute of **firstName** and give it the value of **Bob**.

Step 4

Save all the changes and run the app. Preview the updates in the browser, and confirm that the page shows an **h1** element with the text that reads "Hello, Bob".

Step 5

If the name changes, you don't have to change the component. The component output updates based on the data passed to it.

Change the **firstName** attribute to any name other than 'Bob'. Confirm that the served web page updates with the new name.

Step 6

You can use the **Heading** component multiple times for multiple different people.

Add a second **<Heading />** after the first one. Again, add the **firstName** attribute and choose another name for this value.

Step 7

Save and view the app in the browser

Tip

If you're having trouble with this lab, please review the "Principles of Components: Props" video. This video covers all of the basics of props that you'll need to successfully complete this lab.

Solution: Passing props

Here is the completed App.js file:

```
Ex. import Heading from "../Heading";

function App() {

  return (

    <div className="App">

      <Heading firstName="Bob" />

      <Heading firstName="Any name other than Bob" />

    </div>

  );

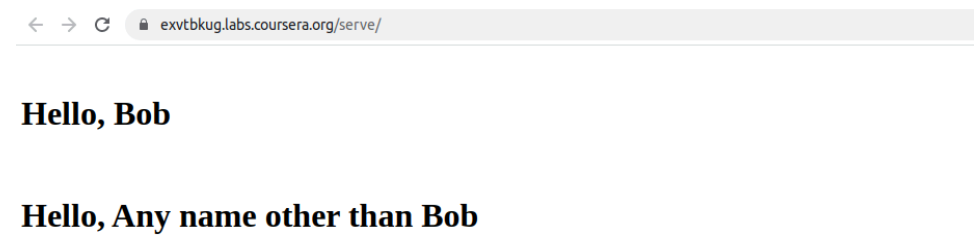
}

export default App;
```


And here is the completed Heading.js file:

```
Ex. function Heading(props) {  
  
  return (  
  
    <h1>Hello, {props.firstName}</h1>  
  
  )  
  
}  
  
export default Heading;
```

Here is the output from the solution code for the App.js file:



Step 1: First, you passed the props object as a parameter to the **Heading** component in the 'Heading.js' file.

```
Ex. function Heading(props) {  
  
  return (  
  
    <h1>Hello, </h1>  
  
  )  
  
}  
  
export default Heading;
```

Step 2: Next, inside the **Heading** component's body, you located the **return** statement, and added a JSX expression that accesses the **firstName** property of the props object, inside the **return** statement's **h1**, after **Hello**.

```
Ex. function Heading(props) {  
  
  return (  
  
    <h1>Hello, {props.firstName}</h1>  
  
  )  
  
}
```

```
export default Heading;
```

Step 3: Then, inside the App component's return statement, you located the `<Heading />` JSX element, and added the attribute of `firstName` and give it the value of `Bob`.

```
Ex. import Heading from "../Heading";
```

```
function App() {  
  
  return (  
  
    <div className="App">  
  
      <Heading firstName="Bob" />  
  
    </div>  
  
  );  
}
```

```
export default App;
```

Step 4: You saved all your changes and ran the app to preview the updates in the browser, and confirm that the page shows an `h1` element with the text that reads "Hello, Bob".

Step 5: Then, you changed `firstName` to any name other than 'Bob' and see how the page updates with the new name.

```
Ex. import Heading from "../Heading";
```

```
function App() {  
  
  return (  
  
    <div className="App">  
  
      <Heading firstName="Any name other than Bob" />  
  
    </div>  
  
  );  
}
```

```
export default App;
```

Step 6: Finally, you added a second `<Heading />` after the first one. Again, adding the `firstName` attribute and choosing another name for this value.

```
Ex. import Heading from "../Heading";
```

```
function App() {  
  
  return (  
  

```

```

<div className="App">

  <Heading firstName="Any name other than Bob" />

  <Heading firstName="Jack" />

</div>

);

}

export default App;

```

Props and children

Previously, you learned to pass props to and within a component. But there is also a special prop called **props.children**, which is automatically passed to every component. In this reading, you'll learn about **props.children** and its purpose.

To understand the concept of **props.children**, consider the following real-life situation: you have a couple of apples, and you have a couple of pears. You'd like to carry the apples some distance, so obviously, you'll use a bag.

It's not a "bag for apples", and it's not a "bag for pears." It's just a bag. Nothing about this bag makes it such that it needs to be referred to as a bag in which you'd only and always carry apples, nor a bag in which you'd only and always carry pears.

In a way, the bag "doesn't care" if it is used to carry apples or pears. Nothing about the bag changes. There are no changes in the bag's material, size, shape, or color - because it can handle apples or pears being carried inside, without issues.

Now, consider the following component:

```

Ex. function Apples(props) {

  return (

    <div className="promo-section">

      <div>

        <h2>These apples are: {props.color}</h2>

      </div>

      <div>

        <h3>There are {props.number} apples.</h3>

      </div>

    </div>

  )
}

```

```
)  
}  
  
export default Apples
```

There is also a **Pears** component:

```
Ex. function Pears(props) {  
  
  return (  
  
    <h2>I don't like pears, but my friend, {props.friend}, does</h2>  
  
  )  
}
```

Now, the question is this: Let's say you want a **Bag** component, which can be used to "carry" **Apples** or **Pears**. How would you do that?

This is where **props.children** comes in.

You can define a **Bag** component as follows:

```
Ex. function Bag(props) {  
  
  const bag = {  
  
    padding: "20px",  
  
    border: "1px solid gray",  
  
    background: "#fff",  
  
    margin: "20px 0"  
  
  }  
  
  return (  
  
    <div style={bag}>  
  
      {props.children}  
  
    </div>  
  
  )  
}  
  
export default Bag
```

So, what this does in the **Bag** component is: it adds a wrapping **div** with a specific styling, and then gives it **props.children** as its content.

But what is this `props.children`?

Consider a very simple example:

```
Ex. <Example>
```

```
  Hello there
```

```
</Example>
```

The **Hello there** text is a child of the Example JSX element. The Example JSX Element above is an "invocation" of the Example.js file, which, in modern React, is usually a function component.

This **Hello there** text can be passed as a named prop when rendering the **Example** component.

Here's what that would look like:

```
Ex. <Example children="Hello there" />
```

There are two ways to perform this action. But this is just the beginning.

What if you, say, wanted to surround the **Hello there** text in an **h3** HTML element?

Obviously, in JSX, that is easily achievable:

```
Ex. <Example children={<h3>Hello there</h3>} />
```

What if the `<h3>Hello there</h3>` was a separate component, for example, named **Hello**?

In that case, you'd have to update the code like this:

```
Ex. <Example children={<Hello />} />
```

You could even make the **Hello** component more dynamic, by giving it its own prop:

```
Ex. <Example children={<Hello message="Hello there" />} />
```

So, how can you make the Bag, Apples, and Pears examples from the beginning of this reading work?

Here's how you'd render the **Bag** component with the **Apples** component as its `props.children`:

```
Ex. <Bag children={<Apples color="yellow" number="5" />} />
```

And here's how you'd render the **Bag** component, wrapping the **Pears** component:

```
Ex. <Bag children={<Pears friend="Peter" />} />
```

While the above syntax might look strange, it's important to understand what is happening.

Effectively, the above syntax is the same as the two examples below.

Ex. <Bag>

```
<Apples color="yellow" number="5" />
```

```
</Bag>
```

```
<Bag>
```

```
<Pears friend="Peter" />
```

```
</Bag>
```

You can even have multiple levels of nested JSX elements, or a single JSX element having multiple children, such as, for example:

Ex. <Trunk>

```
<Bag>
```

```
<Apples color="yellow" number="5" />
```

```
<Pears friend="Peter" />
```

```
</Bag>
```

```
</Trunk>
```

So, in the above structure, there's a **Trunk** JSX element, inside of which is a single **Bag** JSX element, holding an **Apples** and a **Pears** JSX element.

Before the end of this reading, consider this JSX element again:

Ex. <Bag>

```
<Apples color="yellow" number="5" />
```

```
</Bag>
```

What is Apples to Bag in the above code?

In the above code, **Apples** is a prop of the **Bag** component. To explain further, the Bag component can wrap the Apples component, or *any* other component, because I used the **{props.children}** syntax in the **Bag** component function declaration. In other words, just like in the real world, when you take a bag to a grocery store, you can “wrap” a wide variety of groceries inside the bag, you can do the same thing in React: wrap a wide variety of components inside the **Bag** component, using the children prop to achieve this. Of course, don't carry your friends around in a bag! This example shows that the values being wrapped can be anything and that they can be rendered with the same styling by using a particular component.

It's crucial to understand this when working with React.

Before the end of this reading, there's another important concept that you need to be aware of: *finding the right amount of modularization*. What does this mean? Imagine, for example, that

you had a number of small bags, and that each bag could only carry a single apple or pear. You'd have to wrap each "apple" inside a "bag". That doesn't make much sense. You can think about components making your layouts modular in a similar way. You don't want to have an entire layout contained in a single component, because that would be very difficult to work with. On the flip side, if you made each HTML element in your layout a separate component, it would be very hard to work with, although such layout would be modular. So it's all about moderation. You need to organize your layouts by splitting them into meaningful areas of the page, and then code those meaningful areas as separate components. that would constitute the right amount of modularity. To reinforce this point, It might help to think of how a person would describe a website: a menu, a footer, a shopping cart, etc.

In conclusion, when you see a JSX element wrapping another JSX element, you can easily understand that it's all just **props.children** in the background.

Styling JSX elements

You've observed that JSX is incredibly versatile, and can accept a combination of JavaScript, HTML and CSS. In this reading, you'll learn some approaches for styling JSX elements and doing so in a way that achieves both a functional and visual aspect within an app.

There are various ways to style JSX elements.

Probably the simplest way to do this is using the **link** HTML element in the head of the index.html file in which your React app will mount.

The **href** attribute loads some CSS styles, probably with some CSS classes, and then, inside the function component's declarations, you can access those CSS classes using the **className** attribute.

```
Ex. function Promo(props) {  
  
  return (  
  
    <div className="promo-section">  
  
      <div>  
  
        <h1>{props.heading}</h1>  
  
      </div>  
  
      <div>  
  
        <h2>{props.promoSubHeading}</h2>  
  
      </div>  
  
    </div>  
  
  );  
}
```

```
}
```

In CSS:

```
Ex. .promo-section {  
  
  font-weight: bold;  
  
  line-height: 20px;  
  
}
```

Another way to add CSS styles to components is using inline styles.

The syntax of inline styles in JSX is a bit custom.

Consider a starting **Promo** component, containing code that you encountered earlier:

```
Ex. function Promo(props) {  
  
  return (  
  
    <div className="promo-section">  
  
      <div>  
  
        <h1>{props.heading}</h1>  
  
      </div>  
  
      <div>  
  
        <h2>{props.promoSubHeading}</h2>  
  
      </div>  
  
    </div>  
  
  );  
  
}  
  
export default Promo;
```

Now you can add some inline styles to it:

```
Ex. function Promo(props) {  
  
  return (  
  
    <div className="promo-section">  
  
      <div>  
  
        <h1 style={{color:"tomato", fontSize:"40px", fontWeight:"bold"}}>
```



```

        {props.heading}

      </h1>

    </div>

    <div>

      <h2>{props.promoSubHeading}</h2>

    </div>

  </div>

);
}

export default Promo;

```

You can start updating the **Promo** component by adding the JavaScript expression syntax:

Ex. `<h1 style={} >`

As explained previously, this means that whatever code you add inside these opening and closing curly braces is to be parsed as regular JavaScript. Now let's add a style object literal inside of these curly braces:

Ex. `<h1 style={{color:"tomato",fontSize:"40px"}} >`

You can then re-write this object literal:

Ex. `{`

```

  color: "tomato",

  fontSize: "40px"

}
```

So, there's nothing special about this object, except for the fact that you've inlined it and placed it inside a pair of curly braces. Additionally, since it's just JavaScript, those CSS properties that would be hyphenated in plain CSS, such as, for example, **font-size:40px**, become camelCased, and the value is a string, making it look like this: **fontSize:"40px"**.

Besides inlining a *style object literal*, you can also save it in a variable, and then use that variable instead of passing an object literal.

That gives you an updated **Promo** component, with the styles object saved as a JavaScript variable:

Ex. `function Promo(props) {`

```

const styles = {
```

```

    color: "tomato",

    fontSize: "40px"

  }

  return (
    <div className="promo-section">
      <div>
        <h1 style={styles}>
          {props.heading}
        </h1>
      </div>
      <div>
        <h2>{props.promoSubHeading}</h2>
      </div>
    </div>
  );
}

```

Using this approach makes your components more self-contained, because they come with their own styles built-in, but it also makes them a bit harder to maintain.

JSX syntax and the arrow function

Components as Function Expressions

Up to this point, you've likely only observed ES5 function declarations used to define components in React. However, this is not the only way to do it.

In this reading, you learn about some alternative approaches, specifically by using function expressions and arrow functions.

Function Expressions

Let's start with a function declaration used as a component in React:

Ex. `function Nav(props) {`

```

return (
  <ul>
    <li>{props.first}</li>
  </ul>
)
}

```

This component's code returns a list item containing the value of the 'first' prop.

Now, let's change this function declaration to a function expression:

```

Ex. const Nav = function(props) {
  return (
    <ul>
      <li>{props.first}</li>
    </ul>
  )
}

```

The component is, for the most part, the same. The only thing that's changed is that you're now using an anonymous (nameless) function, and assigning this anonymous function declaration to a variable declared using the **const** keyword, and the name **Nav**. The rest of the code is identical.

Changing a component from a function declaration to a function expression doesn't change its behavior, or how you write the code to render the **Nav** component. It's still the same:

```

Ex. <Nav first="Home" />

```

You can also take this concept a step further, using arrow functions.

Components as Arrow Functions

Arrow functions are a core feature of the ES6 version of JavaScript.

One of the main benefits of using arrow functions is its shorter syntax.

Consider the Nav function expression written as an arrow function:

```

Ex. const Nav = (props) => {
  return (
    <ul>

```

```
        <li>{props.first}</li>
      </ul>
    )
  }
```

So, the way to think about this is the following:

- The arrow itself can be thought of as the replacement for the **function** keyword.
- The parameters that this arrow function accepts are listed before the arrow itself.

To reiterate, take the smallest possible anonymous ES5 function:

```
Ex. const example = function() {}
```

And then observe how this is written as an arrow function:

```
Ex. const example = () => {}
```

Another important rule regarding arrow functions is that using the parentheses is optional if there's a single parameter that a function accepts.

In other words, another correct way to write the previous Nav arrow function component would be to drop the parentheses around 'props':

```
Ex. const Nav = props => {
  return (
    <ul>
      <li>{props.first}</li>
    </ul>
  )
}
```

In all other cases, when you write arrow functions, for any number of parameters other than a single parameter, using parentheses around parameters is compulsory.

For example, if your **Nav** component wasn't accepting any parameters, you'd code it with empty parentheses:

```
Ex. const Nav = () => {
  return (
    <ul>
```

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

Another interesting thing about arrow functions is the implicit return. However, it only works if it's on the same line of code as the arrow itself. In other words, the implicit return works if your entire component is a single line of code.

To demonstrate how this works, let's re-write the **Nav** component as a one-liner:

```
Ex. const Nav = () => <ul><li>Home</li></ul>
```

Note that with the implicit return, you don't even have to use the curly braces that are compulsory function body delimiters in all other cases.

Using Arrow Functions in Other Situations

In React, just like in plain JavaScript, arrow functions can be used in many different situations. One such situation is using it with, for example, the **forEach()** built-in array method.

For example:

```
Ex. [10, 20, 30].forEach(item => item * 10)
```

The output of the above vanilla JavaScript line of code would be three number values:

100

200

300

As a side-note, the term "vanilla JavaScript" is often used to describe the plain, regular JavaScript language syntax, without any framework-specific or library-specific code. For example, React is a library, so in this context, saying that a piece of code is "vanilla JavaScript" means that it doesn't need any special library to run. It can run in "plain" JavaScript without any additional dependencies.

You could also write this code in ES5 syntax:

```
Ex. [10, 20, 30].forEach(function(item) {
    return item * 10
})
```

)

Regardless of how you write it, the **forEach()** method can be run on an array. The **forEach()** method accepts a single parameter: an anonymous function. If you write this anonymous function in ES5 syntax, then it would contain a return statement:

```
Ex. function(item) {  
  return item * 10  
}
```

If you write it as an ES6 function instead, it can be simplified as one line:

```
Ex. item => item * 10
```

Both these functions perform the exact same task. Only the syntax is different. The ES6 function is a lot shorter because:

- The arrow function has a single parameter, so you do not need to add parentheses around the item parameter (to the left of the arrow)
- Since the arrow function fits on one line of code, you don't need to use curly braces around the function body, or the return keyword; it's implicit

Arrow functions are used extensively in JSX in React, and getting used to their syntax and being able to "mentally parse" it as you read it is an important skill to have and helps you get better at writing React apps.

Now that you have completed this reading, you've learned about some alternative approaches, specifically by using function expressions and arrow functions.

Ternary operators and functions in JSX

So you've explored several ways to define components in React; this includes function declarations, function expressions and arrow functions.

As you continue with building your knowledge of React syntax, you'll learn to make more use of JSX and embedded JSX expressions.

In this reading, you will become familiar with how to use ternary expressions to achieve a random return, as well as how to invoke functions inside of JSX expressions.

A different way of writing an if...else conditional

You are likely familiar with the structure of an if...else conditional. Here is a quick refresher:

```
Ex. let name = 'Bob';
```

```
if (name === 'Bob') {  
  
    console.log('Hello, Bob');  
  
} else {  
  
    console.log('Hello, Friend');  
  
};
```

The above code works as follows:

1. First, I declare a `name` variable and set it to a string of `"Bob"`.
2. Next, I use the `if` statement to check if the value of the `name` variable is `"Bob"`. If it is, I want to `console.log` the word `"Bob"`.
3. Otherwise, if the `name` variable's value is not `"Bob"`, the `else` block will execute and output the words `"Hello, Friend"` in the console.

Above, I gave you an example of using an `if...else` conditional. Did you know that there is another, different way, to effectively do the same thing? It's known as the ternary operator. A ternary operator in JavaScript uses two distinct characters: the first one is the question mark, that is, the `?` character. To the left of the `?` character, you put *a condition that you'd like to check for*. Just like I did in the above `if...else` statement, the condition I'm checking is `name === 'Bob'`. In other words, I'm asking the JavaScript engine to look at the value that's stored inside the `name` variable, and to verify if that value is the same as `'Bob'`. If it is, then the JavaScript engine will return the boolean value of `true`. If the value of the `name` variable is something different from `'Bob'`, the value that the JavaScript engine returns will be the boolean value of `false`.

Here is the code that reflects the explanation in the previous paragraph:

```
Ex. name === 'Bob' ?
```

Note that the above code is incomplete. I have the condition that I'm checking (the `name === 'Bob'` part). I also have the `?` character, that is, the first of the two characters needed to construct a syntactically valid ternary operator. However, I still need the second character, which is the colon, that is the `:` character. This character is placed after the question mark character. I can now expand my code to include this as well:

```
Ex. name === 'Bob' ? :
```

This brings me a step closer to completing my ternary operator. Although I've added the characters needed to construct the ternary operator, I still need to add the return values. In other words, if `name === 'Bob'` evaluates to true, I want to return the words, "Yes, it is Bob!". Otherwise, I want to return the words "I don't know this person".

```
Ex. name === Bob ? "Yes, it is Bob" : "I don't know this person";
```

This, in essence, is how the ternary operator works. It's just some shorthand syntax that I can use as a replacement for the **if** statement. To prove that this is really the case, here's my starting if...else example, written as a ternary operator:

```
Ex. let name = 'Bob';

name == 'Bob' ? console.log('Hello, Bob') : console.log('Hello, Friend');
```

Using ternary expressions in JSX

Let's examine an example of a component which uses a ternary expression to randomly change the text that is displayed.

```
Ex. function Example() {

  return (

    <div className="heading">

      <h1>{Math.random() >= 0.5 ? "Over 0.5" : "Under 0.5"}</h1>

    </div>

  );

};
```

Inside the **<h1>** element, the curly braces signal to React that you want it to parse the code inside as regular JavaScript.

Then, inside the curly braces, you can add a ternary statement. Every ternary statement conceptually, expressed in pseudo-code, works like this:

```
Ex. comparison ? true : false
```

In the actual code example at the start of this lesson item, the comparison part, which goes to the left of the question mark, is using the **>=** (greater-than-or-equal-to operator), to return a Boolean value. If the result of the comparison evaluates to **true**, then the string "Over 0.5" gets returned. In other words, whatever sits between the question mark and the semi-colon character will get returned. Otherwise, if the result of the comparison evaluates to **false**, then the string "Under 0.5" gets returned. In other words, the value that sits to the right of the colon character will get returned from the ternary expression.

This is how you can use a ternary expression to check for a condition right inside a component and return a value dynamically.

Using function calls in JSX

Another way to work with an expression in JSX is to invoke a function. Function invocation is an expression because every expression returns a value, and function invocation will always return a value, even when that return value is **undefined**.

Like the previous example, you can use function invocation inside JSX to return a random number:

```
Ex.function Example2() {  
  
  return (  
  
    <div className="heading">  
  
      <h1>Here's a random number from 0 to 10:  
  
        { Math.floor(Math.random() * 10) + 1 }  
  
      </h1>  
  
    </div>  
  
  );  
};
```

In the **Example2** component, built-in **Math.floor()** and **Math.random()** methods are being used, as well as some number values and arithmetic operators, to display a random number between 1 and 10.

You can also extract this functionality into a separate function:

```
Ex. function Example3() {  
  
  const getRandomNum = () => Math.floor(Math.random() * 10) + 1  
  
  return (  
  
    <div className="heading">  
  
      <h1>Here's a random number from 1 to 10: { getRandomNum() }</h1>  
  
    </div>  
  
  );  
};
```

The **getRandomNum()** function can also be written as a function declaration, or as a function expression. It does not have to be an arrow function.

But let's observe both alternatives: the function expression *and* the function declaration.

Function expression:

```
Ex. const getRandomNum = function() {  
  
  return Math.floor(Math.random() * 10) + 1  
  
};
```

Function declaration:

```
Ex. function getRandomNum() {  
    return Math.floor(Math.random() * 10) + 1  
};
```

Of course, there are many other examples. The ones used here are there to help you understand how versatile and seamless the JSX syntax is. As you improve your React skills, you will find many creative ways of using JavaScript expressions in JSX.

Now that you have completed this reading, you have learned about a few more ways that you can use expressions in JSX.

Expressions as props

You've already learned a bit about using expressions as props. These can be, among other things, ternary operators, function calls, or some arithmetic operations.

However, you can pass almost any kind of expression as a prop.

For example:

```
Ex. const bool = false;  
  
function Example(props) {  
    return (  
        <h2>The value of the toggleBoolean prop is: {props.toggleBoolean.toString()}</h2>  
    );  
};  
  
export default function App() {  
    return (  
        <div className="App">  
            <Example toggleBoolean={!bool} />  
        </div>  
    );  
};
```

In the example above, you're using the **!bool**, that is, the NOT operator, which evaluates to **true**, since **!false** is true.

Also, for the **toggleBoolean** prop to be rendered on the page, you're converting its boolean value to a string using the JavaScript's built-in `toString` method.

Here's an extension of the above code which shows more ways to work with expressions as props in React.

What is happening here is several props are being passed to the **Example** component, and rendering each of these props' values to the screen.

Ex. `const bool = false;`

`const str1 = "just";`

```
function Example(props) {  
  return (  
    <div>  
      <h2>  
        The value of the toggleBoolean prop is: {props.toggleBoolean.toString()}  
      </h2>  
      <p>The value of the math prop is: <em>{props.math}</em></p>  
      <p>The value of the str prop is: <em>{props.str}</em></p>  
    </div>  
  );  
};  
  
export default function App() {  
  return (  
    <div className="App">  
      <Example  
        toggleBoolean={!bool}  
        math={(10 + 20) / 3}  
        str={str1 + ' another ' + 'string'}  
      />  
    </div>  
  );  
};
```

In this improvement to the **Example** component, three props are being passed to it: **toggleBoolean**, **math**, and **str**. The **toggleBoolean** is unchanged, and the **math** prop and the **str** prop have been added.

The **math** prop is there to show that you can add arithmetic operators and numbers inside JSX, and it will be evaluated just like it does in plain JavaScript.

The **str** prop is there to show that you can concatenate strings, as well as strings and variables – which is shown by adding string literals of “another ” and “string” to the **str1** variable.

In summary, just like you can use expressions inside function components, you can also use them as prop values inside JSX elements, when rendering those function components.

Additional resources

Below you will find links to helpful additional resources.

- [Components and props](#)
- [Introducing JSX](#)
- [Styling and CSS in React](#)
- [Introducing expressions in JSX](#)

Eventful issues

You’re now aware that React can work with most of the same events found in HTML, although React handles them differently.

This means that you may encounter unfamiliar errors when you run your event-driven React code. However, in this reading, you’ll learn about some of the most common errors associated with events and how you can deal with them.

Event Errors

When you work in any programming environment, language, or framework, you are bound to write code that throws errors, for a variety of reasons.

Sometimes it's just about writing the wrong syntax. Other times it's about not thinking of all the possible scenarios and all the possible ways that things can go wrong in your code.

Regardless of what causes them, errors are a part of everyday life for a developer.

The JavaScript language comes with a built-in error handling syntax, the try...catch syntax.

Let's examine an example of an error in JavaScript:

Ex. `(5).toUpperCase()`

Obviously, you cannot uppercase a number value, and thus, this throws the following error:

Ex. `Uncaught TypeError: 5.toUpperCase is not a function`

To handle this `TypeError`, you can update the code with a `try...catch` block that instructs the code to continue running after the error is encountered:

```
Ex. try {  
    (5).toUpperCase();  
}  
catch(e) {  
    console.log(`Oops, you can't uppercase a number.  
    Trying to do it resulted in the following`, e);  
}
```

The `try-catch` block will output some text in the console:

Oops, you can't uppercase a number. Trying to do it resulted in the following `TypeError`:
`5.toUpperCase is not a function`

It is assumed that if you are taking this course that you are already familiar with how the `try...catch` syntax works, so I won't go into any details after this quick refresher.

Back to React, here's an example of a simple error in a React component:

```
Ex. function NumBillboard(props) {  
    return (  
        <>  
        <h1>{prop.num}</h1>  
        </>  
    )  
}  
export default NumBillboard;
```

In React, an error in the code, such as the one above, will result in the error overlay showing in the app in the browser.

In this specific example, the error would be:

ReferenceError

prop is not defined

Note: *You can click the X button to close the error overlay.*

Since event-handling errors occur after the UI has already been rendered, all you have to do is use the error-handling mechanism that already exists in JavaScript – that is, you just use the `try...catch` blocks.

Event handling and embedded expressions

In this reading, you'll learn the different ways to embed expressions in event handlers in React:

- With an inline anonymous ES5 function
- With an inline, anonymous ES6 function (an arrow function)
- Using a separate function declaration
- Using a separate function expression

You may find this reading useful as a reference sheet.

For clarity and simplicity, a function will simply console log some words. This will allow you to compare the difference in syntax between these four approaches, while the result of the event handling will always be the same: just some words output to the console.

Handling events using inline anonymous ES5 functions

This approach allows you to directly pass in an ES5 function declaration as the `onClick` event-handling attribute's value:

Ex. `<button onClick={function() {console.log('first example')}}>`

An inline anonymous ES5 function event handler

`</button>`

Although it's possible to write your click handlers using this syntax, it's not a common approach and you will not find such code very often in React apps.

Handling events using inline anonymous ES6 functions (arrow functions)

With this approach, you can directly pass in an ES6 function declaration as the `onClick` event-handling attribute's value:

Ex. `<button onClick={() => console.log('second example')}>`

An inline anonymous ES6 `function` event handler

`</button>`

This approach is much more common than the previous one. If you want to keep all your logic inside the JSX expression assigned to the `onClick` attribute, use this syntax.

Handling events using separate function declarations

With this approach, you declare a separate ES5 function declaration, and then you reference its name in the event-handling `onClick` attribute, as follows:

Ex. `function App() {`

`function thirdExample() {`

`console.log('third example');`

`};`

`return (`

`<div className="thirdExample">`

`<button onClick={thirdExample}>`

using a separate `function` declaration

`</button>`

`</div>`

`);`

`};`

`export default App;`

This syntax makes sense to be used when your `onClick` logic is too complex to easily fit into an anonymous function. While this example is not really showing this scenario, imagine a function that has, for example, 20 lines of code, and that needs to be run when the click event is triggered. This is a perfect use-case for a separate function declaration.

Handling events using separate function expressions

Tip: A way to determine if a function is defined as an expression or a declaration is: if it does not start the line with the keyword `function`, then it's an expression.

In the following example, you're assigning an anonymous ES6 arrow function to a `const` variable – hence, this is a function expression.

You're then using this const variable's name to handle the `onClick` event, so this is an example of handling events using a separate function expression.

```
Ex. function App() {  
  
  const fourthExample = () => console.log('fourth example');  
  
  return (  
  
    <div className="fourthExample">  
  
      <button onClick={fourthExample}>  
  
        using a separate function expression  
  
      </button>  
  
    </div>  
  
  );  
};  
  
export default App;
```

The syntax in this example is very common in React. It uses arrow functions, but also allows us to handle situations where our separate function expression spans multiple lines of code.

In this reading lesson item, you've learned the several types of functions you can use to handle events in React. Some of those are more common than others, but now that you know all the different ways of doing this, you can understand other people's code more easily, as well as choose the syntax that best suits your given use case, such as a specific company coding style guide.

Solution: Dynamic events

Here is the completed App.js file:

```
Ex. function App() {  
  
  function handleClick() {  
  
    let randomNum = Math.floor(Math.random() * 3) + 1;  
  
    console.log(randomNum);  
  
    let userInput = prompt('type a number');  
  
    alert(`Computer number: ${randomNum}, Your guess: ${userInput}`);  
  
  }  
}
```



```

return (

  <div>

    <h1>Task: Add a button and handle a click event</h1>

    <button onClick={handleClick}>Guess the number between 1 and 3</button>

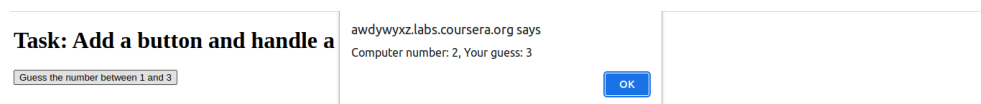
  </div>

);
}

export default App;

```

Here is the output from the solution code for the App.js file:



Step 1. First, you added a **button** element, with an opening and a closing **button** tag, to the App component's **h1** element .

```

Ex. function App() {

  return (

    <div>

      <h1>Task: Add a button and handle a click event</h1>

      <button></button>

    </div>

  );

}

export default App;

```

Step 2. In between the opening and closing **button** tags, you added the following text: Guess the number between 1 and 3.

```

Ex. function App() {

  return (

    <div>

      <h1>Task: Add a button and handle a click event</h1>

```

```
    <button>Guess the number between 1 and 3</button>

  </div>

);

}
```

```
export default App;
```

Step 3. Next, inside the opening **button** tag, you added the **onClick** event-handling attribute, and passed it the following JSX expression: **{handleClick}**.

```
Ex. function App() {

  return (

    <div>

      <h1>Task: Add a button and handle a click event</h1>

      <button onClick={handleClick}>Guess the number between 1 and 3</button>

    </div>

  );

}

export default App;
```

Step 4. Then, above the **return** statement of the App component - but still inside the App function - you added the following ES5 function declaration:

```
Ex. function App() {

  function handleClick() {

    let randomNum = Math.floor(Math.random() * 3) + 1;

    console.log(randomNum);

    let userInput = prompt('type a number');

    alert('Computer number: ${randomNum}, Your guess: ${userInput}');

  }

  return (

    <div>

      <h1>Task: Add a button and handle a click event</h1>

      <button onClick={handleClick}>Guess the number between 1 and 3</button>
```

```
</div>

);

}

export default App;
```

Step 5. Finally, you saved your changes and ran the app to preview it in the browser. You should then be able to click a button, which will show a prompt pop up which you can type into. After that, an alert pop up will show computer's "choice" and your guess. After you click "ok" to close the alert, you'll be able to click the button again and try matching the number "chosen" by the computer one more time.

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [Handling Events](#)
- [Supported Events](#)
- [SyntheticEvent](#)
- [How do I pass an event handler \(like onClick\) to a component?](#)
- [JavaScript Expressions as Props](#)
- [JavaScript Expressions as Children](#)

Data flow in React

You've just learned how the parent-child relationship can be set up so that data flows from parent to child.

In this reading, you'll learn how to detail the flow of data from parent to child. You will then learn why code samples need to be clear and concise. Finally, you will explore data flow in greater detail by looking at more examples. This should act as a refresher to knowledge gained in previous courses.

Parent-child data flow

In React, data flow is a one-way street. Sometimes it's said that the data flow is unidirectional. Put differently, the data in React flows from a parent component to a child component. The data flow starts at the root and can flow to multiple levels of nesting, from the root component (parent component) to the child component, then the grandchild component, and further down the hierarchy.

A React app consists of many components, organized as a component tree. The data flows from the root component to all the components in the tree structure that require this data, using props.

Props are immutable (cannot be changed).

The two main benefits of this unidirectional data flow are that it allows developers to:

1. comprehend the logic of React apps more quickly and
2. simplify the data flow.

Here's a practical example of this:

Imagine that the parent component passes a prop (name) to the child component. The child component then uses this prop to render the name in the UI.

Parent component:

```
Ex. function Dog() {  
  
  return (  
  
    <Puppy name="Max" bowlShape="square" bowlStatus="full" />  
  
  );  
};
```

Child component:

```
Ex. function Puppy(props) {  
  
  return (  
  
    <div>  
  
      {props.name} has <Bowl bowlShape="square" bowlStatus="full" />  
  
    </div>  
  
  );  
};
```

Grandchild component:

```
Ex. function Bowl(props) {  
  
  return (  
  
    <span>  
  
      {props.bowlShape}-shaped bowl, and it's currently {props.bowlStatus}  
  
    </span>  
  
  );  
};
```

```
</span>

);

};
```

Having data move through props in only one direction makes it simpler to understand the logic of how the components interact. If data were moving everywhere, all the time, then it would be much harder to comprehend its logical flow. Any optimization you tried to implement would likely not be as efficient as it could be, especially in modern React.

Data flow in React

You've just learned how the parent-child relationship can be set up so that data flows from parent to child.

In this reading, you'll learn how to detail the flow of data from parent to child. You will then learn why code samples need to be clear and concise. Finally, you will explore data flow in greater detail by looking at more examples. This should act as a refresher to knowledge gained in previous courses.

Parent-child data flow

In React, data flow is a one-way street. Sometimes it's said that the data flow is unidirectional. Put differently, the data in React flows from a parent component to a child component. The data flow starts at the root and can flow to multiple levels of nesting, from the root component (parent component) to the child component, then the grandchild component, and further down the hierarchy.

A React app consists of many components, organized as a component tree. The data flows from the root component to all the components in the tree structure that require this data, using props.

Props are immutable (cannot be changed).

The two main benefits of this unidirectional data flow are that it allows developers to:

1. comprehend the logic of React apps more quickly and
2. simplify the data flow.

Here's a practical example of this:

Imagine that the parent component passes a prop (name) to the child component. The child component then uses this prop to render the name in the UI.

Parent component:

Ex. `function Dog() {`

```
return (  
  <Puppy name="Max" bowlShape="square" bowlStatus="full" />  
);  
};
```

Child component:

```
Ex. function Puppy(props) {  
  return (  
    <div>  
      {props.name} has <Bowl bowlShape="square" bowlStatus="full" />  
    </div>  
  );  
};
```

Grandchild component:

```
Ex. function Bowl(props) {  
  return (  
    <span>  
      {props.bowlShape}-shaped bowl, and it's currently {props.bowlStatus}  
    </span>  
  );  
};
```

Having data move through props in only one direction makes it simpler to understand the logic of how the components interact. If data were moving everywhere, all the time, then it would be much harder to comprehend its logical flow. Any optimization you tried to implement would likely not be as efficient as it could be, especially in modern React.

Using hooks

Now that you understand what hooks are in React and have some basic knowledge on the **useState** hook, let's dive in deeper. In this reading, you will learn how to use hooks in React components and understand the use-cases for the **useState** hook.

Let's say you have a component with an input text field. The user can type into this text field. The component needs to keep track of what the user types within this text field. You can add state and use the **useState** hook, to hold the string.

As the user keeps typing, the local state that holds the string needs to get updated with the latest text that has been typed.

Let's discuss the below example.

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

export default function InputComponent() {

  const [inputText, setText] = useState('hello');

  function handleChange(e) {

    setText(e.target.value);

  }

  return (

    <>

    <input value={inputText} onChange={handleChange} />

    <p>You typed: {inputText}</p>

    <button onClick={() => setText('hello')}>

      Reset

    </button>

    </>

  );

}
```

To do this, let's define a React component and call it **InputComponent**. This component renders three things:

- An input text field
- Any text that has been entered into the field
- A Reset button to set the field back to its default state

As the user starts typing within the text field, the current text that was typed is also displayed.

You typed: Welcome

The state variable **inputText** and the **setText** method are used to set the current text that is typed. The **useState** hook is initialized at the beginning of the component.

Ex. `const[inputText, setText] = useState('hello');`

By default, the **inputText** will be set to “hello”.

As the user types, the **handleChange** function, reads the latest input value from the browser’s input DOM element, and calls the **setText** function, to update the local state of **inputText**.

Ex. `function handleChange(e) {
 setText(e.target.value);
};`

Finally, clicking the reset button will update the **inputText** back to “hello”.

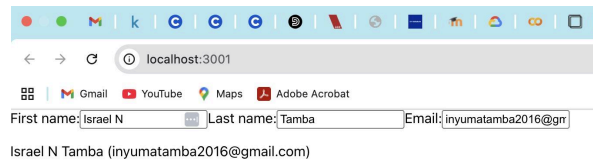
Isn’t this neat?

Keep in mind that the **inputText** here is local state and is local to the **InputComponent**. This means that outside of this component, **inputText** is unavailable and unknown. In React, state is always referred to the local state of a component.

Hooks also come with a set of rules, that you need to follow while using them. This applies to all React hooks, including the **useState** hook that you just learned.

- You can only call hooks at the top level of your component or your own hooks.
- You cannot call hooks inside loops or conditions.
- You can only call hooks from React functions, and not regular JavaScript functions.

To demonstrate, let’s extend the previous example, to include three input text fields within a single component. This could be a registration form with fields for first name, last name and email.



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

export default function RegisterForm() {

  const [form, setForm] = useState({

    firstName: 'Israel N',

    lastName: 'Tamba',

    email: 'inyumatamba2016@gmail.com',

  });

  return (

    <div>

      <label>

        First name:

        <input

          value={form.firstName}

          onChange={e => {

            setForm({

              ...form,

              firstName: e.target.value

            });

          }}

        />

      </label>

      <label>

        Last name:

        <input
```

```
      value={form.lastName}

      onChange={e => {

        setForm({

          ...form,

          lastName: e.target.value

        });

      }}

    />

  </label>

  <label>

    Email:

    <input

      value={form.email}

      onChange={e => {

        setForm({

          ...form,

          email: e.target.value

        });

      }}

    />

  </label>

  <p>

    {form.firstName} {' '}

    {form.lastName} {' '}

    ({form.email})

  </p>

</>

);

}
```

Notice that you are using a **form** object to store the state of all three text input field values:

```
Ex. const [form, setForm] = useState({  
  firstName: 'Israel N',  
  lastName: 'Tamba',  
  email: 'inyumatamba2016@gmail.com',  
});
```

You do not need to have three separate state variables in this case, and instead you can consolidate them all together into one **form** object for better readability.

In addition to the **useState** hook, there are other hooks that come in handy such as **useContext**, **useMemo**, **useRef**, etc. When you need to share logic and reuse the same logic across several components, you can extract the logic into a custom hook. Custom hooks offer flexibility and can be used for a wide range of use-cases such as form handling, animation, timers, and many more.

Next, I'll give you an explanation of how the **useRef** hook works.

The **useRef** hook

We use the **useRef** hook to access a child element directly.

When you invoke the **useRef** hook, it will return a **ref** object. The **ref** object has a property named **current**.

```
Ex. function TextInputWithFocusButton() {  
  const inputEl = useRef(null);  
  const onButtonClick = () => {  
    // `current` points to the mounted text input element  
    inputEl.current.focus();  
  };  
  return (  
    <>  
    <input ref={inputEl} type="text" />  
    <button onClick={onButtonClick}>Focus the input</button>  
    </>  
  );  
}
```

}

Using the ref attribute on the input element, I can then access the current value and invoke the focus() method on it, thereby focusing the input field.

There are situations where accessing the DOM directly is needed, and this is where the useRef hook comes into play.

Conclusion

In this reading, you have explored hooks in detail and understand how to use the **useState** hook to maintain state within a component. You also understand the benefits of using hooks within a React component.

Instructions

Lab instructions

Introduction:

The lab will focus on how to build a simple counter application using React. This counter will start with a value of 0 and have buttons to increase and decrease the value by 1. You'll practice using state in React with the useState hook and handle basic user interactions through event handlers. By the end of this lab, you will have a functional counter app and a better understanding of how to manage state in React.

Goal:

To build a simple counter app in React that:

- Starts with an initial value of 0.
- Increases the value by 1 when the "Add 1" button is clicked.
- Decreases the value by 1 when the "Subtract 1" button is clicked.

Objectives:

By the end of this lab, learners will:

1. Learn how to use the useState hook to create and manage state in functional components.
2. Practice handling user input (button clicks) to trigger state changes.
3. Understand how to update the state dynamically using event handlers.
4. Learn how to use JSX to display and render updated state in a React component.
5. Apply best practices for building interactive and dynamic React applications.

Problem Statement

You will create a simple Counter component that:

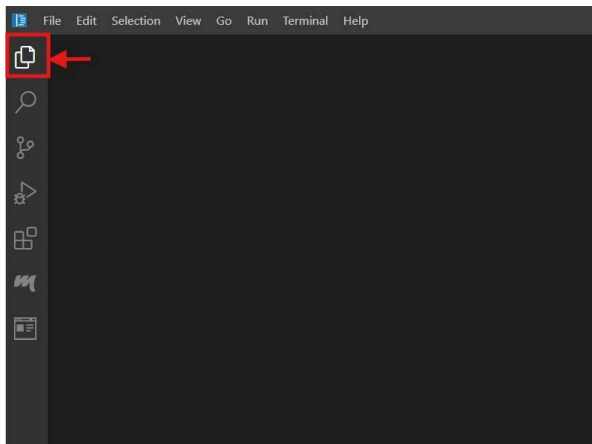
- Displays the current number.
- Includes two buttons: one to add 1 to the number and one to subtract 1 from the number.
- Uses React's useState hook to keep track of the number's state and update it accordingly.

Instructions:

Step 1: Create the React App

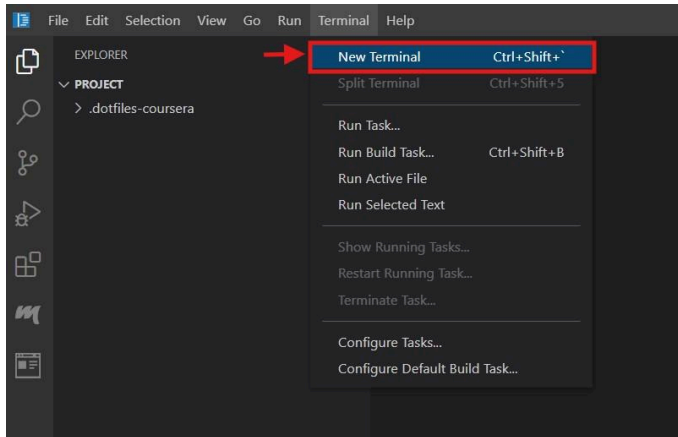
1. Open the Project Folder

- Click on the Explorer icon displayed below.
- Select Open Folder from the options.
- Navigate to and click on your Project Folder.

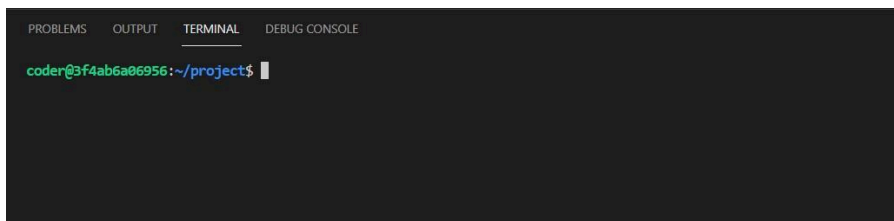


2. Opening a new terminal

- Click on the Terminal menu.
- From the dropdown, select New Terminal.



3. After opening the new terminal, the terminal panel will appear as shown below, defaulting to the project directory.



4. Run the following command to create the React app named counter_app:

Ex. `npx create-react-app counter_app`

5. The installation process may take some time. Once completed, navigate to the project folder using the following command:

Ex. `cd counter_app`

Step 2: Start the React Development Server

1. Use the following command to start the development server and test the app:

Ex. `npm start`

2. You can now view the counter_app in your browser by navigating to localhost:<exposed port> (eg:3000).

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Compiled successfully!

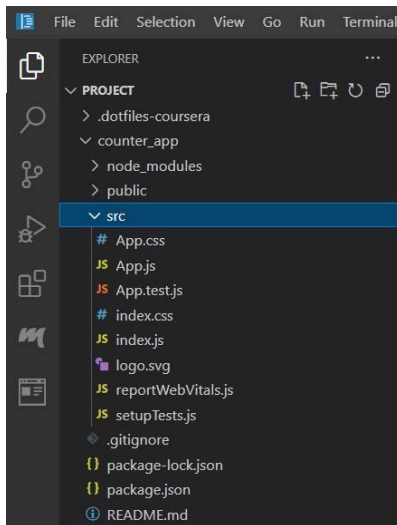
You can now view counter_app in the browser.

  Local:      http://localhost:3000
  On Your Network: http://172.18.0.68:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

3. After successfully installing the React app, the project structure will appear as shown below.



Step 3: Set Up the App Component

1. The App.js file is located in the src folder. It already contains some code; delete the existing code.

2. Import the necessary modules:

- Import `useState` from `React`. This hook will allow us to manage the counter state.
- Import the `App.css` file to apply styles to the app. The necessary code is already included in the `App.css` file.

You should have the following imports:

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

import './App.css';
```

Step 4: Initialize the State

1. Inside the App function, use the useState hook to create a state variable called num with an initial value of 0. useState returns an array with two elements:

- The current state (num).
- A function to update that state (setNum).

Ex. `const [num, setNum] = useState(0);`

2. Now, you have the state set up, and you can use num to display the current number and setNum to update it when a button is clicked.

Step 5: Render the Current Number

- Inside the JSX of the App component, render the current number using {num}

Ex. `<h1>Current number: {num}</h1>`

Step 6: Add Event Handlers for Buttons

Create two buttons:

- One button will call setNum(num + 1) when clicked to increase the number.
- Another button will call setNum(num - 1) when clicked to decrease the number.

Ex. `<button onClick={() => setNum(num + 1)}>Add 1</button>`

`<button onClick={() => setNum(num - 1)}>Subtract 1</button>`

Step 7: Wrap Everything in JSX

Finally, wrap all your JSX inside the return statement to render the updated number and the buttons

Ex. `return (`

`<>`

`<h1>Current number: {num}</h1>`

`<button onClick={() => setNum(num + 1)}>Add 1</button>`

`<button onClick={() => setNum(num - 1)}>Subtract 1</button>`

`</>`

`);`

Complete code

```
Ex.import { useState } from 'react'; // Import the useState hook from React to
manage state in the component

import './App.css'; // Import the CSS file for styling the app

function App() {

  // Declare a state variable 'num' and a function 'setNum' to update the value of
  'num'.

  // Initial value of num is set to 0.

  const [num, setNum] = useState(0);

  return (

    // The main container for the app content

    <div className="app-container">

      {/* Display the current value of the 'num' variable */}

      <h1 className="counter-heading">Current Number: {num}</h1>

      {/* Container for buttons, to allow adding or subtracting 1 */}

      <div className="button-container">

        {/* Button to increment the 'num' value by 1 */}

        <button

          className="counter-button"

          onClick={() => setNum(num + 1)} // onClick calls setNum with num + 1 to
increment the value

        >

          Add 1

        </button>

        {/* Button to decrement the 'num' value by 1 */}

        <button
```

```

        className="counter-button"

        onClick={() => setNum(num - 1)} // onClick calls setNum with num - 1 to
decrement the value

    >

        Subtract 1

    </button>

</div>

</div>

);

}

// Export the App component to be used in other parts of the app

export default App;

```

Explanation:

1. useState Hook:

- The useState hook is used to declare state in functional components. In this case, the num state variable is used to hold the current number, and setNum is the function that updates this number.
- Initially, num is set to 0 (useState(0)).

2. Buttons:

- There are two buttons: "Add 1" and "Subtract 1".
- When the "Add 1" button is clicked, the value of num is increased by 1. This is achieved by calling setNum(num + 1).
- When the "Subtract 1" button is clicked, the value of num is decreased by 1 using setNum(num - 1).

3. JSX Structure:

- The app is wrapped inside a div with a class name app-container for styling.
- The value of num is displayed inside an h1 tag.

- The two buttons are inside a div with the class button-container to group them together.

Step 8: Style the App

The `App.css` file is located in the `src` folder. It already has some existing code; remove it and replace it with the following code for styling and improved visual appeal.

```
Ex.. /* App.css */

/* Centering the app content */

.app-container {

  display: flex;

  flex-direction: column;

  justify-content: center;

  align-items: center;

  height: 100vh;

  background-color: #f0f4f8;

  font-family: Arial, sans-serif;

}

/* Style the heading */

.counter-heading {

  font-size: 2rem;

  font-weight: bold;

  color: #333;

  margin-bottom: 20px;

}

/* Style the buttons container */

.button-container {
```

```
display: flex;

gap: 10px;

}

/* Style for buttons */

.counter-button {

padding: 10px 20px;

font-size: 1rem;

background-color: #007bff;

color: white;

border: none;

border-radius: 5px;

cursor: pointer;

transition: background-color 0.3s ease;

}

/* Hover effect for buttons */

.counter-button:hover {

background-color: #0056b3;

}

/* Style for buttons on active state */

.counter-button:active {

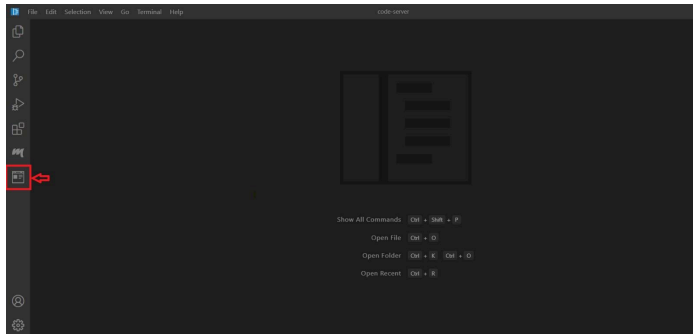
background-color: #004085;

}

}
```

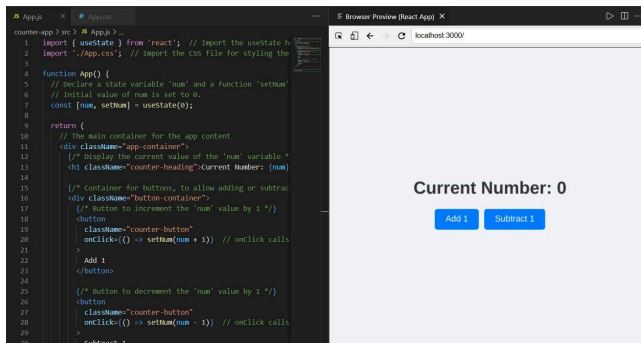
Step 9: view the output

1. To view the output, click on the Browser Preview icon located on the left panel. It is the last icon in the panel.



2. In your browser, enter the following URL format, replacing <exposed port> with the actual port number displayed in the terminal when the server was initially started:
`http://localhost:<exposed port>`.

3. After entering the URL correctly, the webpage will load, displaying a preview of the webpage as shown below.



Step 10: Close the server after completing the lab:

Once you're done with the lab, make sure to close the server to free up the port:

- You can stop the server by pressing `Ctrl + C` in the terminal.

Key Takeaways:

- **useState Hook:** Learners should understand how to create and manage state in functional components using `useState`.
- **Event Handling:** They will practice using event handlers (such as `onClick`) to trigger state updates.
- **Conditional Rendering:** This app involves dynamic rendering based on state, so learners will get hands-on practice with it.

- React Basics: This lab reinforces basic React concepts such as state, props, and functional components.

Prop drilling

As you've learned previously, prop drilling is a situation where you are passing data from a parent to a child component, then to a grandchild component, and so on, until it reaches a more distant component further down the component tree, where this data is required.

Here is a very simple app that focuses on the process of props passing through several components.

Please note that the goal here is not to build an app that would exist in the real world. The goal of this app is to examine the practice of prop drilling, so that you can focus on it and understand it in isolation.

Here is the code for the app:

```
Ex. function Main(props) {  
  
  return <Header msg={props.msg} />;  
  
};  
  
function Header(props) {  
  
  return (  
  
    <div style={{ border: "10px solid whitesmoke" }}>  
  
      <h1>Header here</h1>  
  
      <Wrapper msg={props.msg} />  
  
    </div>  
  
  );  
  
};  
  
function Wrapper(props) {  
  
  return (  
  
    <div style={{ border: "10px solid lightgray" }}>  
  
      <h2>Wrapper here</h2>
```

```

        <Button msg={props.msg} />

    </div>

    );

}

function Button(props) {

    return (

        <div style={{ border: "20px solid orange" }}>

            <h3>This is the Button component</h3>

            <button onClick={() => alert(props.msg)}>Click me!</button>

        </div>

    );

};

function App() {

    return (

        <Main

            msg="I passed through the Header and the Wrapper and I reached the Button
component"

        />

    );

};

export default App;

```

This app is simple enough that you should be able to understand it on your own. Let's address the main points to highlight what is happening in the code above.

The top-most component of this app is the **App** component. The **App** component returns the **Main** component. The **Main** component accepts a single attribute, named **msg**, as in "message".

At the very top of the app, the **Main** function declares how the **Main** component should behave. The **Main** component is responsible for rendering the **Header** component. Note that when the **Header** component is rendered from inside **Main**, it also receives the **msg** prop.

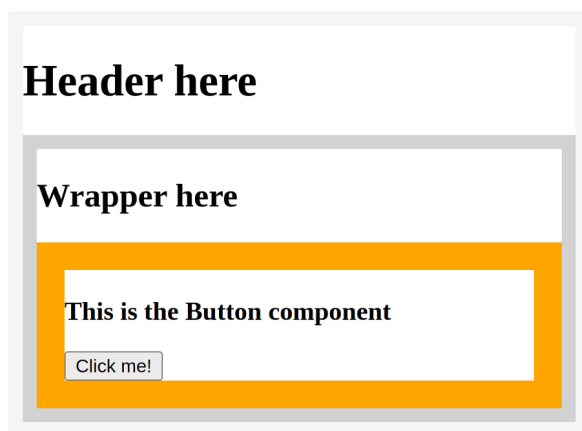
The **Header** component's function declaration renders an **h1** that reads "Header here", then another component named **Wrapper**. Note that the naming here is irrelevant – the components **Header** and **Wrapper** are named to make it a bit more like it might appear in a real app – but ultimately, the focus is on having multiple components, rather than describing specific component names properly.

So, the **Header** component's function declaration has a return statement, which renders the **Wrapper** component with the **msg** prop passed to it.

In the **Wrapper** component's function declaration, there's an **h2** that reads "Wrapper here", in addition to the rendering of the **Button** component, which also receives the **msg** attribute.

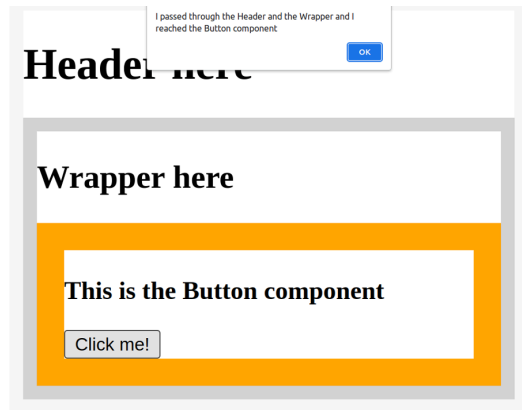
Finally, the **Button** component's function declaration is coded to receive the props object, then inside of the wrapping **div**, show an **h3**. The **h3** reads "This is the Button component", and then, under that, there's a button element with an **onClick** event-handling attribute. This is passed to an arrow function which should alert the string that comes from the **props.msg** prop.

All this code results in the following UI rendered on the screen:



This screenshot illustrates the boundaries of each component. The **Main** component can't be found in the UI because it's just rendering the **Header** component. The **Header** component then renders the **Wrapper** component, and the **Wrapper** component then renders the **Button** component.

Note that the string that was passed on and on through each of the children component's props' objects is not found anywhere. However, it will appear when you click the "Click me!" button, as an alert:



The alert's message reads "I passed through the Header and the Wrapper and I reached the Button component".

That's really all there is to it. Props drilling simply means passing a prop through props objects through several layers of components. The more layers there are, the more repetitive and unnecessary this feels. There are various ways to deal with this, as you'll learn in the lesson items that follow.

Lab instructions

Tasks

In the starter code of this code lab, you are given a **Fruits** component that has its own state. Based on this state, it outputs three fruits on the screen. Additionally, you have a **FruitsCounter** component which shows a message that reads: "Total fruits: 2".

Your task is to lift state up from the **Fruits** component to the **App** component, so that you can then pass the state information to both the **Fruits** component and the **FruitsCounter** component.

This change to the app should fix the previously incorrect message of "Total fruits: 2". The new message should be "Total fruits: 3". However, the new message will not be just a hard-coded string. Instead, it should reflect the number of fruits that exist in the state variable, so based on how many fruits there are in the state array, this information should affect the output of the total number of fruits - as returned from the **FruitsCounter** component.

Where should the state go?

```
apple
apple
plum
Total fruits: 2
```

Steps

Step 1. This task's starting point is the stateless **App** component's code:

```
Ex. import Fruits from "../Fruits";

import FruitsCounter from "../FruitsCounter";

function App() {

  return (

    <div className="App">

      <h1>Where should the state go?</h1>

      <Fruits />

      <FruitsCounter />

    </div>

  );

};

export default App;
```

The first step of this task is to move the state from the **Fruits** component to App component.

```
Ex. import React from "react";

function Fruits() {

  const [fruits] = React.useState([

    {fruitName: 'apple', id: 1},

    {fruitName: 'apple', id: 2},

    {fruitName: 'plum', id: 3},

  ]);

  return (

    <div>

      {fruits.map(f => <p key={f.id}>{f.fruitName}</p>)}

    </div>

  );

};

export default Fruits;
```

Update the App component to pass this fruits state as props to both the Fruits and FruitsCounter components.

Step 2.

Once you've moved the state up from the Fruits component to the App component, Remove the state from the Fruits component and update it to receive the fruits as prop and display the list of fruits dynamically based on the passed prop.

```
Ex. import React from "react";

function Fruits({ fruits }) {

  return (

    <div>

      {fruits.map((f) => (

        <p key={f.id}>{f.fruitName}</p>

      ))}

    </div>

  );

}

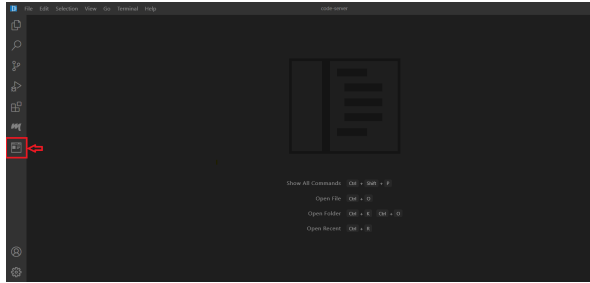
export default Fruits;
```

Step3.

Now, Modify the FruitsCounter component to accept the fruits prop and calculate the total fruits dynamically using the length property of the array.

Step 4.

- At the top of the lab environment, locate the Terminal menu. Click on it to open a dropdown, then select New Terminal. Use the **npm start** command to start the development server.
- If you encounter errors like *File not found* or *Unexpected token*, in the terminal, stop the server with **Ctrl + C** and restart it using **npm start**.
- You can now view the App in your lab browser. To view the output, click on the **Browser Preview** icon located on the left panel. It is the last icon in the panel.



- When your lab browser has launched, enter: **http://localhost:3000** in the address bar to see the output.
- The app displays the correct list of fruits. The total fruits count dynamically reflects the length of the fruits array.

Solution: Managing state in React

Here is the completed App.js file:

```
Ex. import React from "react";

import Fruits from "./Fruits";

import FruitsCounter from "./FruitsCounter";

function App() {

  const [fruits] = React.useState([

    {fruitName: 'apple', id: 1},

    {fruitName: 'apple', id: 2},

    {fruitName: 'plum', id: 3},

  ]);

  return (

    <div className="App">

      <h1>Where should the state go?</h1>

      <Fruits fruits={fruits} />

      <FruitsCounter fruits={fruits} />

    </div>

  );
}
```

```
        </div>

    );

}

export default App;
```

Here is the completed Fruits.js file:

```
Ex. function Fruits(props) {

    return (

        <div>

            {props.fruits.map(f => <p key={f.id}>{f.fruitName}</p>)}

        </div>

    )

}

export default Fruits
```

Here is the completed FruitsCounter.js file:

```
Ex. function FruitsCounter(props) {

    return (

        <h2>Total fruits: {props.fruits.length}</h2>

    )

}

export default FruitsCounter;
```

The completed app should look as follows:

Where should the state go?

apple
apple
plum

Total fruits: 3

Step 1. Move the state from the **Fruits** component to the **App** component.

To complete this step, you need to go to the **Fruits** component and cut the **useState** call, namely this piece of code:

```
Ex. const [fruits] = React.useState([
  {fruitName: 'apple', id: 1},
  {fruitName: 'apple', id: 2},
  {fruitName: 'plum', id: 3},
]);
```

You also need to cut the **import React from "react";** at the very top of the **Fruits** component, since you no longer need to access the **useState** method on the React object from the Fruits file.

Additionally, you need to add the import statement to the **App** component, which means that you should inject a new import at the very top of App.js:

```
Ex. import React from "react";
```

Once you've done that, you need to update the **App** component's return statement so that it sends the fruits data to the **Fruits** and **FruitsCounter** component - since both of these components need to get this state's data via props.

Ex.// The updated return statement in App.js:

```
return (
  <div className="App">
    <h1>Where should the state go?</h1>
    <Fruits fruits={fruits} />
    <FruitsCounter fruits={fruits} />
  </div>
);
```

Step 2.

The **Fruits** component should be updated so that it accepts state from the **App** component.

Now all that you need to do is to update the code in the **Fruits** components to accept the props object and render the fruits property where appropriate.

That means that the **Fruits** component will end up having the following code:

```
Ex.function Fruits(props) {  
  
  return (  
  
    <div>  
  
      {props.fruits.map(f => <p key={f.id}>{f.fruitName}</p>)}  
  
    </div>  
  
  )  
  
}  
  
export default Fruits
```

Step 3.

Once you've lifted the state up from the **Fruits** component to the **App** component, you also need to update the **FruitsCounter** component.

Just like the **Fruits** component, the **FruitsCounter** component should also receive state from the **App** component, so that it can display the number of the available fruits using the **length** property of the array of fruits from the **fruits** state variable.

The **FruitsCounter** component will end up having the following code:

```
Ex.function FruitsCounter(props) {  
  
  return (  
  
    <h2>Total fruits: {props.fruits.length}</h2>  
  
  )  
  
}  
  
export default FruitsCounter;
```

That completes this ungraded lab's solution.

Additional resources

Below you will find links to helpful additional resources.

- [React docs website URL which discusses the issue in depth](#)

- [Data flows down](#)
- [The Power Of Not Mutating Data](#)
- [Add Inverse Data Flow](#)
- [Component state](#)
- [State: A Component's Memory](#)
- [Sharing State Between Components](#)
- [State as a Snapshot](#)
- [Basic useState examples](#)
- [Synchronizing with effects - putting it all together](#)
- [Fetch API](#)
- [The event loop in JavaScript](#)

Navigation

In this reading, you'll learn about the differences between traditional web pages and React-powered web pages (SPAs – single page applications).

Once you understand the difference between these two ways of building web pages, you will be able to understand the necessary difference between how navigation works in traditional web apps versus how it works in modern SPA websites.

Before Single-Page Apps

Before the advent of modern JavaScript frameworks, most websites were implemented as multi-page applications. That is, when a user clicks on a link, the browser navigates to a new webpage, sends a request to the web server; this then responds with the full webpage and the new page is displayed in the browser.

This can make your application resource intensive to the Web Server. CPU time is spent rendering dynamic pages and network bandwidth is used sending entire webpages back for every request. If your website is complex, it may appear slow to your users, even slower if they have a slow or limited internet connection.

To solve this problem, many web developers develop their web applications as Single Page Applications.

Single-Page Apps

You're using many Single Page Applications every day. Think of your favorite social network, or online email provider, or the map application you use to find local businesses. Their excellent user experiences are driven by Single Page Applications.

A Single Page Application allows the user to interact with the website without downloading entire new webpages. Instead, it rewrites the current webpage as the user interacts with it. The outcome is that the application will feel faster and more responsive to the user.

How Does a Single-Page App Work?

When the user navigates to the web application in the browser, the Web Server will return the necessary resources to run the application. There are two approaches to serving code and resources in Single Page Applications.

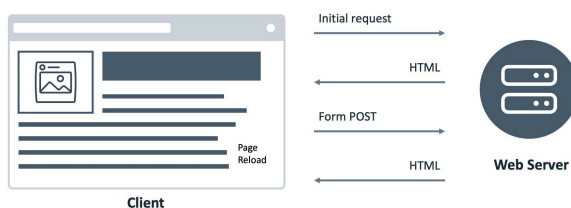
1. When the browser requests the application, return and load all necessary HTML, CSS and JavaScript immediately. This is known as *bundling*.
2. When the browser requests the application, return only the minimum HTML, CSS and JavaScript needed to load the application. Additional resources are downloaded as required by the application, for example, when a user navigates to a specific section of the application. This is known as *lazy loading* or *code splitting*.

Both approaches are valid and are used depending on the size, complexity and bandwidth requirements of the application. If your application is complex and has a lot of resources, your bundles will grow quite large and take a long time to download – possibly ending up slower than a traditional web application!

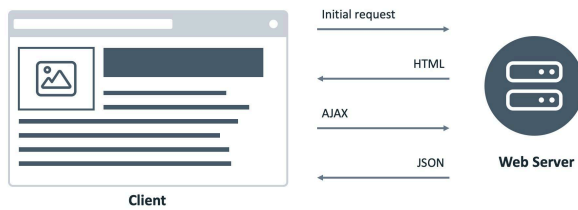
Once the application is loaded, all logic and changes are applied to the current webpage.

Let's look at an example.

Traditional Page Lifecycle



SPA Page Lifecycle



An Example of a Single-Page App

Imagine there is a webpage that has a Label and a Button. It will display a random movie name when the button is clicked.

In a traditional website, when the button is clicked, the browser will send a POST request to the web server. The web server will return a new web page containing the button and movie name, and the web browser renders the new page.

In a Single Page Application, when the button is clicked, the browser will send a POST request to a web server. The web server will return a JSON object. The application reads the object and updates the Label with the movie name.

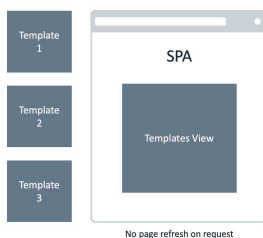
See, more efficient!

But what if we need to have multiple pages with different layouts in our application?

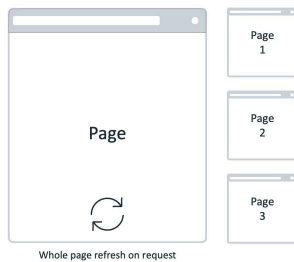
Let's look at another example.

Practical Differences Between Single-Page Apps and Multi-Page Apps

Single Page Application



Traditional Page Application



You have a web application that has a navigation bar on top and two pages. One page shows the latest news, and the other shows the current user's profile page. The navigation bar contains a link for each page.

In a traditional website, when the user clicks the Profile link, the web browser sends the request to the web server. The web server generates the HTML page and sends it back to the web browser. The web browser then renders the new web page.

In a Single Page Application, different pages are broken into templates (or views). Each view will have HTML code containing variables that can be updated by the application.

The web browser sends the request to the web server, and the web server sends back a JSON object. The web browser then updates the web page by inserting the template with the variables replaced by the values in the JSON object.

Anchor Tag Elements in Single-Page Elements

A single-page application can't have regular anchor tag elements as a traditional web app can.

The reason for this is that the default behavior of an anchor tag is to load another HTML file from a server and refresh the page. This page refresh is not possible in a SPA that's powered by a library such as React because a total page refresh is not the way that a SPA works, as explained earlier in this lesson item.

Instead, a SPA comes with its own special implementation of anchor tags and links, which only give an illusion of loading different pages to the end user when in fact, they simply load different components into a single element of the real DOM into which the virtual DOM tree gets mounted and updated.

That's why navigation in a single-page app is fundamentally different from its counterpart in a multi-page app. Understanding the concepts outlined in this lesson item will make you a more well-rounded React developer.

Task

You are using the code similar to the lesson item titled *The Navbar*. Your goal is to add another link to the existing code. This link should show a brand new component, named Contact.

Important

First, install the react-router package by executing the command **npm install react-router-dom@6**. Ensure you do this before proceeding with any other steps.

Steps

Step 1

Add a new file, **Contact.js**, to the root of the **src** folder.

Step 2

Inside the Contact.js file, add an ES5 function, named Contact. Add the **export default Contact** after the Contact function's closing curly brace.

Step 3

Inside the body of the Contact function, add a **return** statement with the following code:
<h1>Contact Little Lemon on this page.</h1>.

Step 4

Inside the App.js file, import the newly-built Contact component.

Step 5

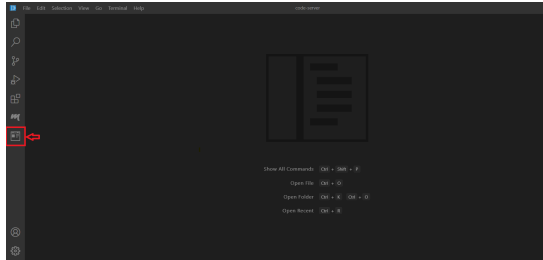
Inside the App.js file's App function's return statement, locate the **nav** element, and inside of it, add another **<Link>** element, with the **to** attribute pointing to **"/contact"**, the **className** set to **"nav-item"**, and the the text inside the **Link** element's opening and closing tags set to **Contact**.

Step 6

Inside the **Routes** element, add a third route, with the path attribute pointing to **"/contact"**, and the element attribute set to **{<Contact />}**.

Step 7

- At the top of the lab environment, locate the Terminal menu. Click on it to open a dropdown, then select New Terminal. Use the **npm start** command to start the development server.
- If you encounter errors like *File not found* or *Unexpected token*, in the terminal, stop the server with **Ctrl + C** and restart it using **npm start**.
- You can now view the App in your lab browser. To view the output, click on the **Browser Preview** icon located on the left panel. It is the last icon in the panel.



- When your lab browser has launched, enter: **http://localhost:3000** in the address bar to see the output.
- The navigation bar displays three links: Home, About, and Contact. Clicking the Contact link updates the page content to show the message: "Contact Little Lemon on this page."

Tip

If you're having trouble with this lab, please review the "The Navbar" video lesson item. That video explains how to work with routes in React.

Solution: Creating a route

Here is the Contact.js file:

```
Ex. function Contact() {  
  
    return <h1>Contact Little Lemon on this page.</h1>  
  
}  
  
export default Contact
```

Here is the completed App.js file:

```
Ex. import './App.css';  
  
import Homepage from './Homepage';  
  
import AboutLittleLemon from './AboutLittleLemon';  
  
import Contact from './Contact';  
  
import { Routes, Route, Link } from 'react-router-dom';  
  
function App() {  
  
    return (  
  
        <div>
```

```

<nav>

  <Link to="/" className="nav-item">Homepage</Link>

  <Link to="/about" className="nav-item">About Little Lemon</Link>

  <Link to="/contact" className="nav-item">Contact</Link>

</nav>

<Routes>

  <Route path="/" element={<Homepage />}></Route>

  <Route path="/about" element={<AboutLittleLemon />}></Route>

  <Route path="/contact" element={<Contact />}></Route>

</Routes>

</div>

);

};

export default App;

```

Here is the output from the completed solution code:



Step 1

First, you added a new file, Contact.js, to the root of the src folder.

Step 2

Inside the Contact.js file, you added an ES5 function, named Contact. And then, added the **export default Contact** after the Contact function's closing curly brace.

```
Ex. function Contact() {
```

```
};  
  
export default Contact;
```

Step 3

Next, inside the body of the Contact function, you added a **return** statement with the following code: **<h1>Contact Little Lemon on this page.</h1>**.

```
Ex. function Contact() {  
  
    return <h1>Contact Little Lemon on this page.</h1>  
  
};  
  
export default Contact;
```

Step 4

Inside the App.js file, you imported the newly-built Contact component.

```
Ex. import './App.css';  
  
import Homepage from './Homepage';  
  
import AboutLittleLemon from './AboutLittleLemon';  
  
import Contact from './Contact';  
  
import { Routes, Route, Link } from 'react-router-dom';  
  
function App() {  
  
    return (  
  
        <div>  
  
            <nav>  
  
                <Link to="/" className="nav-item">Homepage</Link>  
  
                <Link to="/about" className="nav-item">About Little Lemon</Link>  
  
            </nav>  
  
            <Routes>  
  
                <Route path="/" element={<Homepage />}></Route>
```

```

        <Route path="/about" element={<AboutLittleLemon />}></Route>

    </Routes>

</div>

);

};

export default App;

```

Step 5

Inside the App.js file's App function's return statement, locate the **nav** element, and inside of it, add another **<Link>** element, with the **to** attribute pointing to contact, the **className** set to "nav-item", and the the text inside the **Link** element's opening and closing tags set to **Contact**.

```

Ex. import "./App.css";

import Homepage from "./Homepage";

import AboutLittleLemon from "./AboutLittleLemon";

import Contact from "./Contact";

import { Routes, Route, Link } from "react-router-dom";

function App() {

    return (

        <div>

            <nav>

                <Link to="/" className="nav-item">Homepage</Link>

                <Link to="/about" className="nav-item">About Little Lemon</Link>

                <Link to="/contact" className="nav-item">Contact</Link>

            </nav>

            <Routes>

                <Route path="/" element={<Homepage />}></Route>

```



```

        <Route path="/about" element={<AboutLittleLemon />}></Route>

    </Routes>

</div>

);

};

export default App;

```

Step 6

Inside the **Routes** element, add a third route, with the path attribute pointing to **"/contact"**, and the element attribute set to **{<Contact />}**.

```

Ex. import "./App.css";

import Homepage from "./Homepage";

import AboutLittleLemon from "./AboutLittleLemon";

import Contact from "./Contact";

import { Routes, Route, Link } from "react-router-dom";

function App() {

    return (

        <div>

            <nav>

                <Link to="/" className="nav-item">Homepage</Link>

                <Link to="/about" className="nav-item">About Little Lemon</Link>

                <Link to="/contact" className="nav-item">Contact</Link>

            </nav>

            <Routes>

                <Route path="/" element={<Homepage />}></Route>

                <Route path="/about" element={<AboutLittleLemon />}></Route>

```

```

    <Route path="/contact" element={<Contact />}></Route>

  </Routes>

</div>

);

};

```

Step 7

You saved all your changes and viewed your updates in the served app. You should have had three links in the top navbar, and the third link should have been Contact. Once you clicked the link, the sentence "Contact Little Lemon on this page" should have replaced whatever other content was under the navbar previously.

Applying conditional rendering

State is all the data your app is currently working with. With this in mind, you can decide to conditionally render specific components in your app, based on whether specific state data has specific values. To make this possible, React works with the readily available JavaScript syntax and concepts.

Consider a minimalistic productivity app.

The app takes the client computer's current datetime, and based on the data, displays one of two messages on the screen:

1. For workdays, the message is: "Get it done"
1. For weekends, the message is: "Get some rest"

There are a few ways you can achieve this in React.

One approach would include setting a component for each of the possible messages, which means you'd have two components. Let's name them **Workdays** and **Weekends**.

Then, you'd have a **CurrentMessage** component, which would render the appropriate component based on the value returned from the **getDay()** function call.

Here's a simplified **CurrentMessage** component:

```

Ex.function CurrentMessage() {

  const day = new Date().getDay();

  if (day >= 1 && day <= 5) {

```

```

        return <Workdays />
    }

    return <Weekends />
}

```

Instead of calculating it directly, you could use some historical data instead, and perhaps get that data from a user via an input, from a parent component.

In that case, the **CurrentMessage** component might look like this:

```

Ex. function CurrentMessage(props) {

    if (props.day >= 1 && props.day <= 5) {

        return <Workdays />

    }

    return <Weekends />

}

```

Conditional rendering with the help of element variables

To further improve your **CurrentMessage** component, you might want to use element variables. This is useful in some cases, where you want to streamline your render code - that is, when you want to separate the conditional logic from the code to render your UI.

Here's an example of doing this with the **CurrentMessage** component:

```

Ex. function CurrentMessage({day}) {

    const weekday = (day >= 1 && day <= 5);

    const weekend = (day >= 6 && day <= 7);

    let message;

    if (weekday) {

        message = <Workdays />

    } else if (weekend) {

        message = <Weekends />

    } else {

```

```

        message = <ErrorComponent />

    }

    return (

        <div>

            {message}

        </div>

    )

}

```

The output of the **CurrentMessage** component will depend on what the received value of the day variable is. On the condition of the day variable having the value of any number between 1 and 5 (inclusive), the output will be the contents of the **Workdays** component. Otherwise, on the condition of the day variable having the value of either 6 or 7, the output will be the contents of the **Weekends** component.

Conditional rendering using the logical AND operator

Another interesting approach in conditional rendering is the use of the logical **AND** operator **&&**.

In the following component, here's how the **&&** operator is used to achieve conditional rendering:

```

Ex. function LogicalAndExample() {

    const val = prompt('Anything but a 0')

    return (

        <div>

            <h1>Please don't type in a zero</h1>

            {val &&

                <h2>Yay, no 0 was typed in!</h2>

            }

        </div>
    )
}

```

```
)  
  
}
```

There are a few things to unpack here, so here is the explanation of the **LogicalAndExample** component, top to bottom:

1. First, you ask the user to type into the prompt, specifying that you require anything other than a zero character; and you save the input into the **val** value.
2. In the return statement, an **h1** heading is wrapped inside a **div** element, and then curly braces are used to include a JSX expression. Inside this JSX expression is a single **&&** operator, which is surrounded by some code both on its left and on its right sides; on the left side, the val value is provided, and on the right, a piece of JSX is provided.

To understand what will be output on screen, consider the following example in standard JavaScript:

```
Ex.true && console.log('This will show')
```

If you ran this command in the browser's console, the text 'This will show' will be output.

On the flip side, consider the following example:

```
Ex.false && console.log('This will never show')
```

If you ran *this* command, the output will just be the boolean value of **false**.

In other words, if a prop gets evaluated to **true**, using the **&&** operator, you can render whatever JSX elements you want to the right of the **&&** operator.

Conditional components

Have you ever visited a website that required a user account? To log in you click on a Log in button and once you've logged in, the Log in button changes to a Log out button.

This is often done using something called conditional rendering.

In a previous course, you've already learned about simple conditions using if and switch statements.

Using these statements allows you to change the behaviour of code based on certain conditions being met.

For example, you can set a variable to a different value based on the result of a condition check.

```
Ex.let name;  
  
if (Math.random() > 0.5) {
```

```

    name = "Mike"

} else {

    name = "Susan"

}

//

let name;

let newUser = true;

if (Math.random() > 0.5 && newUser) {

    name = "Mike"

} else {

    name = "Susan"

}

```

Conditional rendering is built on the same principle. By using conditions, you can return different child components. This is often done using the props that are passed into the parent component, but can also be done based on component state.

Let's take a look at a simple example.

Let's say you have two child components called **LoginButton** and **LogoutButton**; each displaying their corresponding button.

In the parent component, named **LogInOutButton**, you can check the props passed into the parent component and return a different child component based on the value of the props.

In this example, the props contains a property named **isLoggedIn**. When this is set to **true**, the **LogoutButton** component is returned. Otherwise, the **LoginButton** component is returned.

```

Ex. function LogInOutButton(props) {

const isLoggedIn = props.isLoggedIn;

    if (isLoggedIn) {

        return <LogoutButton />;

    } else {

```

```
    return <LoginButton />;  
  }  
}
```

Then when the **LogInOutButton** parent component is used, the prop can be passed in.

```
Ex. <LogInOutButton isLoggedIn={false} />
```

This is a simple example showing how you can change what is displayed based on a condition check. You will use this often when developing React applications.

Dynamic Component Rendering in React: A Login/Logout App

Introduction:

In this lab, learners will build a simple React app to understand key concepts like state management, props passing, and conditional rendering. State management in React allows components to track and update dynamic data, such as whether a user is logged in or not. Props passing enables parent components to send data or functions to child components, like passing login and logout functions to buttons. Conditional rendering controls what appears on the screen based on the current state, such as showing a login button when logged out or a home page with a logout button when logged in.

Goal:

To practice using React's `useState` hook to manage the logged-in state and conditionally render components based on that state. Learners will learn how to create and manage interactions between different components and how to pass functions as props to handle user actions (such as login and logout).

Objectives:

By the end of this lab, learners will:

- Understand how to use `useState` to manage state in a React application.
- Learn how to conditionally render components based on state.
- Pass functions as props between components.
- Practice component-based architecture and organizing code into multiple files.

Problem Statement

In this lab, learners will be building a simple application where the user can toggle between a logged-in and logged-out state. Based on the state, different components will be displayed. If

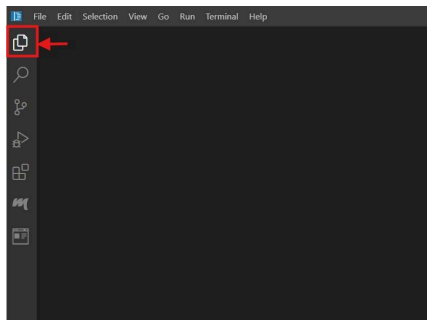
the user is not logged in, they will see a Login button. Once they click Login , the app will display a home page along with a Logout button. By clicking Logout, the app will return to the logged-out state.

Instructions:

Step 1: Create the React App

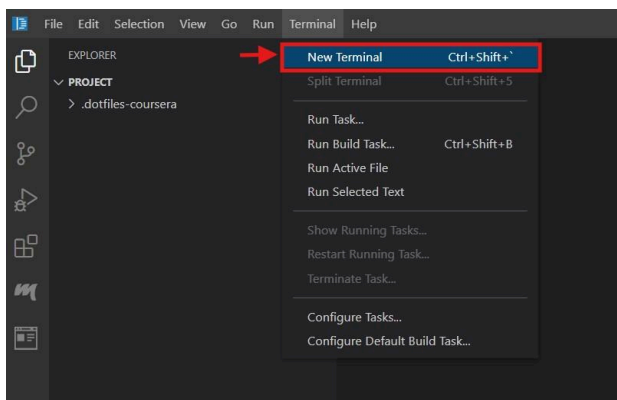
1. Open the Project Folder

- Click on the Explorer icon displayed below.
- Select Open Folder from the options.
- Navigate to and click on your Project Folder.



2. Opening a new terminal

- Click on the Terminal menu.
- From the dropdown, select New Terminal.



3. After opening the new terminal, the terminal panel will appear as shown below, defaulting to the project directory.



4. Run the following command to create the React app named `toggle_app`:

Ex.

5. The installation process may take some time. Once completed, navigate to the project folder using the following command:

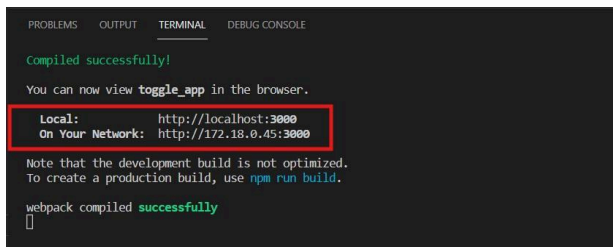
Ex.

Step 2: Start the React Development Server

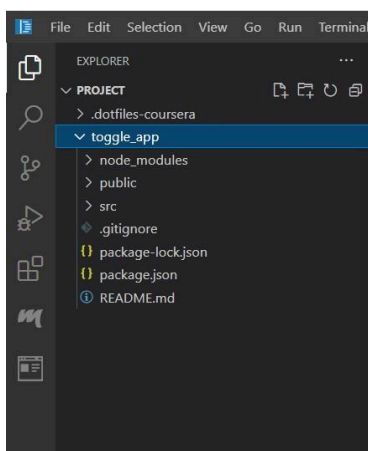
1. Use the following command to start the development server and test the app:

Ex.

2. You can now view the `toggle_app` in your browser by navigating to `localhost:<exposed port>` (eg:3000).



3. After successfully installing the React app, the project structure will appear as shown below.



Step 3: Create LoginButton.js and Add Code

Creating LoginButton.js

1. Right-click on the src folder inside the toggle_app directory.
2. Select New File from the context menu.
3. Name the file LoginButton.js and press Enter.
4. The file is now created, and you can start writing the code.

Objective

- This component renders a button labeled "Login".
- The button uses a login function (passed as a prop) to update the state in the parent component when clicked.

Steps to Implement

- Create a functional component named LoginButton.
- Accept props and use the props.login function for the button's onClick event.
- Add a meaningful label for the button (e.g., "Login").

code

Ex.

Explanation:

- The LoginButton component is a simple functional component.
- It receives a login function as a prop and binds it to the onClick event of the button.
- When clicked, it triggers the login function to change the application state.

Step 4: Create LogoutButton.js and Add Code

Creating LogoutButton.js

1. Right-click on the src folder inside the toggle_app directory.
2. Select New File from the context menu.
3. Name the file LogoutButton.js and press Enter.
4. The file is now created, and you can start writing the code.

Objective

- This component renders a button labeled "Logout".
- The button uses a logout function (passed as a prop) to update the state in the parent component when clicked.

Steps to Implement

- Create a functional component named LogoutButton.
- Accept props and use the props.logout function for the button's onClick event.
- Add a meaningful label for the button (e.g., "Logout").

code

Ex.

Explanation

- Similar to LoginButton, the LogoutButton receives a logout function as a prop.
- It binds this function to the onClick event of the button to handle logout logic when clicked.

Step 5: Create HomePage.js and Add Code

Creating HomePage.js

1. Right-click on the src folder inside the toggle_app directory.
2. Select New File from the context menu.
3. Name the file HomePage.js and press Enter.
4. The file is now created, and you can start writing the code.

Objective

- This component renders a welcome message when the user is logged in.
- It doesn't need to handle any functionality, as it only displays static content.

Steps to Implement

- Create a functional component named HomePage.
- Use JSX to render a <h1> element with a welcome message.

code

Ex.

Explanation

- The HomePage component is a functional component that renders a single line of text inside an `<h1>` tag.
- It doesn't require props or state, as it only displays static content.

Step 6: Modify App.js and Add Code

The App.js file is located in the src folder. It already contains some code; delete the existing code.

Objective

- The App component manages the main logic and state for the application.
- It determines whether the user is logged in and renders either the LoginButton or the HomePage with a LogoutButton.

Steps to Implement

- Import the three components created (LoginButton, LogoutButton, HomePage).
- Use the useState hook to manage the loggedIn state.
- When loggedIn is false, render the LoginButton.
- When loggedIn is true, render the HomePage and the LogoutButton.
- Define login and logout functions to toggle the state.

code

Ex.

Explanation:

State Management:

- The useState hook is used to define and manage the loggedIn state.
- loggedIn is a boolean that determines the current login status.

Functions:

- The login function sets loggedIn to true, and the logout function sets it to false.

Conditional Rendering:

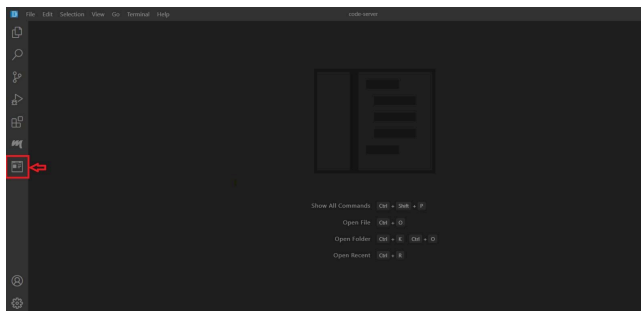
- When loggedIn is false, the LoginButton is displayed.
- When loggedIn is true, the HomePage and LogoutButton are displayed.

Props Passing:

- The login and logout functions are passed as props to LoginButton and LogoutButton, respectively.

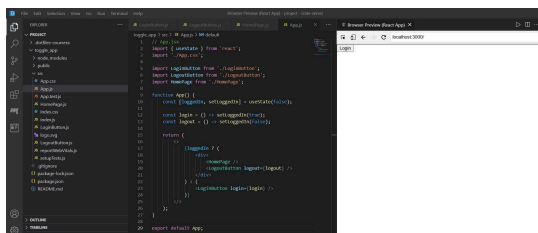
Step 7: view the output

1. To view the output, click on the Browser Preview icon located on the left panel. It is the last icon in the panel.



2. In your browser, enter the following URL format, replacing <exposed port> with the actual port number displayed in the editor when the server was initially started:
http://localhost:<exposed port>.

3. After entering the URL correctly, the webpage will load, displaying a preview of the webpage as shown below.



Step 8: Close the server after completing the lab:

Once you're done with the lab, make sure to close the server to free up the port:

- You can stop the server by pressing Ctrl + C in the terminal.

Key Takeaways:

State Management: Using the useState hook in React helps manage dynamic data, like the logged-in state.

Conditional Rendering: Components can be conditionally rendered based on the state, making the UI interactive.

Props: Functions can be passed as props to child components, enabling actions like login and logout to be handled by different components.

Component-Based Design: Dividing the app into separate components (App, HomePage, LoginButton, and LogoutButton) allows for better organization and reusability.

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [React Router v6](#)
- [nav: The Navigation Section element](#)
- [Conditional \(ternary\) operator in JavaScript](#)
- [if...else](#)

Bundling assets

Earlier, you learned what assets are in React and the best practices for storing them in your project folders.

In this reading, you will learn about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You will also learn about the trade-offs inherent in using asset-heavy apps.

The app's files will likely be bundled when working with a React app. Bundling is a process that takes all the imported files in an app and joins them into a single file, referred to as a bundle.

Several tools can perform this bundling. Since, in this course, you have used the **create-react-app** to build various React apps, you will focus on webpack. This is because webpack is the built-in tool for the **create-react-app**.

Let's start by explaining what webpack is and why you need it.

Simply put, webpack is a module bundler.

Practically, this means that it will take various kinds of files, such as SVG and image files, CSS and SCSS files, JavaScript files, and TypeScript files, and it will bundle them together so that a browser can understand that bundle and work with it.

Why is this important?

When building websites, you could probably do without webpack since your project's structure might be straightforward: you may have a single CSS library, such as Bootstrap, loaded from a

CDN (content delivery network). You might also have a single JavaScript file in your static HTML document. If that is all there is to it, you do not need to use webpack in such a scenario.

However, modern web development can get complex.

Here is an example of the first few lines of code in a single file of a React application:

Ex.

The imports here are from fictional libraries and resources because the specific libraries are not necessary. All these different imports can be of various file types: .js, .svg, .css, and so on.

In turn, all the imported files might have their own imported files, and even those might have their imports.

This means that depending on other files, all of these files can create a dependency graph. The order in which all these files are loading is essential. That dependency graph can get so complex that it becomes almost impossible for a human to structure a complex project and bundle all those dependencies properly.

This is the reason you need tools like webpack.

So, webpack builds a dependency graph and bundles modules into one or more files that a browser can consume.

While it is doing that, it also does the following:

- It converts modern JS code - which can only be understood by modern browsers - into older versions of JavaScript so that older browsers can understand your code. This process is known as *transpiling*. For example, you can transpile ES7 code to ES5 code using webpack.
- It optimizes your code to load as quickly as possible when a user visits your web pages.
- It can process your SCSS code into the regular CSS, which browsers can understand.
- It can build source maps of the bundle's building blocks
- It can produce various kinds of files based on rules and templates. This includes HTML files, among others.

Another significant characteristic of webpack is that it helps developers create modern web apps.

It helps you achieve this using two modes: production mode or development mode.

In development mode, webpack bundles your files and optimizes your bundles for updates - so that any updates to any of the files in your locally developed app are quickly re-bundled. It also builds source maps so you can inspect the original file included in the bundled code.

In production mode, webpack bundles your files so that they are optimized for speed. This means the files are minified and organized to take up the least amount of memory. So, they are optimized for speed because these bundles are fast to download when a user visits the website online.

Once all the source files of your app have been bundled into a single bundle file, then that single bundle file gets served to a visitor browsing the live version of your app online, and the entire app's contents get served at once.

This works great for smaller apps, but if you have a more extensive app, this approach is likely to affect your site's speed. The longer it takes for a web app to load, the more likely the visitor will leave and move on to another unrelated website. There are several ways to tackle this issue of a large bundle.

One such approach is code-splitting, a practice where a module bundler like webpack splits the single bundle file into multiple bundles, which are then loaded on an as-needed basis. With the help of code-splitting, you can lazy load only the parts that the visitor to the app needs to have at any given time. This approach significantly reduces the download times and allows React-powered apps to get much better speeds.

There are other ways to tackle these problems.

An example of a viable alternative is SSR (Server-side rendering).

With SSR, React components are rendered to HTML on the server, and the visitor downloads the finished HTML code. An alternative to SSR is client-side rendering, which downloads the index.html file and then lets React inject its own code into a dedicated HTML element (the root element in **create-react-app**). In this course, you've only worked with client-side rendering.

Sometimes, you can combine client-side rendering and server-side rendering. This approach results in what's referred to as isomorphic apps.

In this reading, you learned about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You also learned about the trade-offs inherent in the use of asset-heavy apps.

Media packages

In this reading, you'll learn how to install the react-player npm package.

You can find this package on the npmjs.org website at the following URL:

<https://www.npmjs.com/package/react-player>

To install this package you'll need to use the following command *in the terminal*:

```
Ex. npm install react-player
```


Once you have this package installed, you can start using it in your project.

There are a few ways that you can import and use the installed package. For example, to get the entire package's functionality, use the following import:

```
Ex. import ReactPlayer from "react-player";
```

If you are, for example, only planning to use videos from a site like YouTube, to reduce bundle size, you can use the following import:

```
Ex. import ReactPlayer from "react-player/youtube";
```

Here's an example of using the react-player packaged in a small React app:

```
Ex. import React from "react";

import ReactPlayer from "react-player/youtube";

const App = () => {

  return (

    <div>

      <MyVideo />

    </div>

  );

};

const MyVideo = () => {

  return (

    <ReactPlayer url='https://www.youtube.com/watch?v=T8TZQ6k4SLE' />

  );

};

export default App;
```

In this reading, you learned how to install and use the react-player npm package.

Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

- [webpack docs](#)
- [webpack asset management](#)
- [npm docs](#)
- [ReactPlayer on npm](#)
- [Video and audio content on the web](#)

About this graded assessment: Calculator app

The purpose of this graded assessment

The primary purpose of a graded assessment is to check your knowledge and understanding of the key learning objectives of the course you have just completed. Most importantly, graded assessments help you establish which topics you have mastered and which require further focus before completing the course. Ultimately, the graded assessment is designed to help you make sure that you can apply what you have learned. This assessment's learning objective is to allow you to create a React application or App.

Prepare for this graded assessment

You will have already encountered exercises, knowledge checks, in-video questions and other assessments as you have progressed through the course. The 'styling a page' ungraded lab from Module 2 is the foundation for this assessment.

The graded assessment requires you to complete a calculator in React. You will be provided with code snippets, and your task is to use these, plus any of your code to complete the calculator that can perform the four basic mathematical operations: addition, subtraction, multiplication, and division.

It will also have a single input button, which will accept user input (any number) and a total starting with a zero.

Once a user types into the input field, they will then have to update the total by pressing any of the four math operation buttons:

- addition
- subtraction
- multiplication
- division

Here's a diagram of the completed calculator app:

← → 🔍 vmtchp10labcalculator.org/mini/ 🔍 < > 🌐

Simplest Working Calculator

8760

add

subtract

multiply

divide

reset input

reset result

Nothing in the graded assessment will be outside what you have covered already, so you should be well placed to succeed.