Introduction to React

Goals

By the end of this lesson, you should understand:
- What React is, and what single page applications (SPAs) are
- The purpose of React and why it was created
- Several key concepts of React such as components, the virtual DOM, and one-way data binding

Introduction

In this lesson, we will begin learning about the immensely popular and powerful frontend JavaScript library React. First, we'll delve a bit into the fascinating evolution of frontend development, charting the progression from early vanilla JavaScript to the emergence of libraries and frameworks that have revolutionized the way web applications are built today. Additionally, we will discuss the concept of single-page applications (SPAs) in greater depth, exploring their advantages, challenges, and the role of React in facilitating their development. By understanding the historical context and the advancements that led to the creation of React, you will gain a deeper appreciation for its significance in modern web development and its role in shaping the future of frontend technologies.

Evolution of Frontend Development

To better understand the significance of React in modern web development, it's helpful to examine the evolution of frontend development over the years. Frontend development has come a long way from using only basic vanilla JavaScript to now having a plethora of advanced libraries and frameworks, shaping the way we build applications today.

Vanilla JavaScript

In the early days of web development, developers were limited to using vanilla JavaScript, which refers to plain JavaScript without any additional libraries or frameworks, to interact with the DOM. While this early form of JavaScript allowed for basic interactivity and DOM manipulation, it was very limited in terms of functionality and often required a lot of repetitive code. As web applications became more complex, developers sought more efficient ways to build dynamic user interfaces.

jQuery

jQuery, released in 2006, was a game-changer for frontend development. It provided a simple, powerful API for DOM manipulation, event handling, and AJAX requests, all while ensuring cross-browser compatibility. jQuery significantly reduced the amount of code required to perform common tasks and quickly became the go-to library for frontend developers.

Early frontend frameworks

As web applications continued to grow in complexity, developers needed more robust solutions for organizing and structuring their code. In response, libraries like Backbone.js and frameworks like the original Angular.js emerged, offering more sophisticated tools for building web applications.

Backbone.js, released in 2010, introduced a lightweight, flexible structure based on the Model-View-Controller (MVC) pattern. It provided developers with a way to manage application state and encouraged code modularity.

Angular.js, introduced by Google in 2010, took frontend development yet a step further with its two-way data binding, dependency injection, and built-in directives. Angular.js enabled developers to create complex, feature-rich applications with less code and effort than ever before.

The Birth of React

As web applications continued to evolve, the need for a more efficient and performant solution became apparent – one that could adequately address the challenges of maintaining and scaling large, complex applications with rapidly changing data. This solution emerged at Facebook, of all places!

Facebook's applications grew, and the engineering team realized that traditional methods for building user interfaces were becoming increasingly inefficient and difficult to manage. To solve this problem, Facebook engineer Jordan Walke began developing React in 2011. After several iterations and improvements, React was eventually open-sourced in 2013, allowing the wider development community to contribute to and utilize the library. Today, React is used by a vast number of companies and developers, making it one of the most popular JavaScript libraries for building UIs.

React is primarily used for developing what are known as single-page applications (SPAs), where the user interacts with and navigates through the website without requiring full-page refreshes. We'll delve deeper into single-page applications shortly. For now, suffice it to say that one of React's core strengths is its ability to handle rapidly changing data and seamlessly update the UI without causing unnecessary re-rendering or page refreshes. This is particularly useful when building applications that need to display real-time information or have numerous user interactions.

Single Page Applications (SPAs)

Typically, as you may remember from the explanation of the client-server model in Week One, the way the vast majority of websites used to work is as follows: a user entered a URL in a browser's address bar, which then made a request to that website's server. The server, in turn, sent back to the user all the HTML, CSS, and JavaScript necessary to display the requested page. If the user clicked a link to go to a different page on that website, another request was sent to the server, which then sent back another HTML document with its associated CSS and JavaScript, and so on.

Single-page applications (SPAs), on the other hand, are web applications that load a single HTML page and dynamically update the content as the user interacts with the application – including navigating to a new "page." This approach provides a smoother, more seamless user experience compared to traditional multi-page applications.

There are several advantages to using SPAs:

1. **Improved performance**: By reducing the need for full-page refreshes, SPAs can deliver faster, more responsive user experiences.
2. **Simplified deployment and maintenance**: With a single HTML file and separate assets (JavaScript, CSS, images), SPAs can be easier to deploy and maintain compared to traditional multi-page applications.
3. **Enhanced user experience**: The app-like behavior of SPAs can lead to a more engaging and user-friendly experience, as users can navigate and interact with the application more quickly and smoothly.

However, there are also some challenges associated with SPAs:

1. **SEO**: Due to their reliance on JavaScript for rendering content, SPAs can sometimes face issues with search engine optimization (SEO). Modern search engines have improved their ability to index SPAs, but developers still need to take extra precautions to ensure their content is crawlable and indexable.
2. **Initial load time**: As SPAs load most of the application resources upfront, the initial load time can be longer compared to traditional multi-page applications. However, once the initial resources are loaded, subsequent interactions are usually much faster.
3. **Browser history management**: Since SPAs don't rely on traditional page navigation, managing browser history and enabling the back button functionality can be more complex. Developers often use client-side routing libraries to handle this aspect of SPAs.

Of course, as React has only grown in popularity over the years, and the number of React developers has exponentially multiplied, there exists a wealth of different solutions within the React ecosystem to help address these challenges (as there are in other SPA frameworks like Angular and Vue). We'll cover a couple of them later this week – namely, React Router and React's lazy loading functionality.

Core Concepts in React

Let's look at some of React's core concepts which are important to understand before we begin using it in order to fully appreciate why React is so advantageous for frontend development.

Declarative vs Imperative Programming

The imperative paradigm of programming, which is what we've been doing with JavaScript so far in this course, requires developers to explicitly define each step of the program – make this change to the DOM, then that change; if the user does this, then make a different DOM update, etc.

Let's look at a very simple example to help illustrate the difference between **imperative** and **declarative**:

Imperative Example:
Imagine you want to change the background color of a div with an ID of "myDiv" to red when a button is clicked. In imperative programming, you would explicitly specify the steps to achieve this:

```
const button = document.getElementById('myButton');
const myDiv = document.getElementById('myDiv');

button.addEventListener('click', function () {
  myDiv.style.backgroundColor = 'red';
});
```

Declarative Example:
Now let's look at the same example, implemented in a declarative way with React. In the declarative paradigm, we simply express what we want to achieve without detailing the specific steps. React will then handle the updating the DOM for us:

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

function App() {
  const [bgColor, setBgColor] = useState('');

  const handleClick = () => {
    setBgColor('red');
  };

  return (
    <div>
      <button onClick={handleClick}>Change color</button>
      <div id="myDiv" style={{ backgroundColor: bgColor
}}>Hello, World!</div>
    </div>
  );
}

export default App;
```

Here, we're using the **useState** React hook to manage the background color state (more on hooks and state later). When the button is clicked, the **handleClick** function updates the background color to red, and React takes care of updating the UI accordingly.

React follows this declarative paradigm, which is one of the reasons it has become so popular among developers. By abstracting away much of the complexity involved in updating the UI, React enables developers to focus on what they want their applications to do, rather than how they should do it.

Virtual DOM

React uses a virtual DOM to improve performance. The virtual DOM is an in-memory representation of the actual DOM (Document Object Model), which is the structure representing the elements on a web page. In traditional web development, manipulating the DOM can be slow and resource-intensive, leading to poor performance and a suboptimal user experience.

Whenever a component's state changes, React calculates the difference between the current virtual DOM and the new one. Once the differences have been determined, React updates the real DOM with the minimal set of changes required. This approach minimizes the number of DOM manipulations, leading to significant performance improvements.

Reactive Updates

React automatically updates the UI when the application state changes. This is achieved through the use of reactive updates, which efficiently update only the components affected by the state change. When a component's state or properties (also known as "props") change, React triggers a re-render of that component and its children, ensuring that the UI is always in sync with the application state.

This reactive approach simplifies the process of managing UI changes in response to data updates. React handles this automatically, allowing developers to focus on implementing the application's core functionality.

Component-based Architecture

React follows a component-based architecture, allowing developers to build complex UIs by breaking them down into reusable, isolated components. Each component is a self-contained unit with its own state and properties, encapsulating the logic and rendering for a specific part of the UI.

This modular approach promotes code reusability and maintainability, making it easier to manage large applications with numerous features and components. By composing an application from smaller, focused components, developers can create highly organized and scalable codebases that are easier to understand and debug.

One-way Data Binding

React enforces a unidirectional data flow (one-way data binding). This means that data flows from parent components to child components, making it easier to track and debug data changes in the application.

Conclusion and Takeaways

In this lesson, we have learned about the origins and purposes of React, explored its key concepts, and understood its underlying principles. We have gained an appreciation for React's focus on the declarative programming paradigm, which simplifies the process of creating and managing UI updates. We have also learned about the virtual DOM and how it improves performance by minimizing costly DOM manipulations. Additionally, we have explored React's component-based architecture, which promotes code reusability and maintainability, as well as the importance of one-way data binding and unidirectional data flow. In the next lesson, we will set up our development environment and start writing some React code!

React Basics - Part 1

Goals

By the end of this lesson, you should understand:

- How to set up a React environment using both create-react-app (CRA) and Vite
- What JSX is and its role in React components
- How to create both class components and function components

Introduction

In this lesson, we will focus on setting up a React development environment and explore the fundamentals of creating React components. We will cover different tools for setting up your React environment, such as create-react-app (CRA) and Vite, discussing their pros and cons to help you choose the best option for your projects. We'll also introduce JSX, which is a core concept in React. Finally, we'll dive into the two types of React components: class components and function components. By the end of this lesson, you'll have a solid understanding of many of React's basic concepts. So, let's get started!

Setting up a React Project with CRA

There are a few different ways to set up a React development environment. While it's possible to create a React project from scratch this involves a lot of complex configuration using Babel and Webpack; the easier way by far is to use a project scaffolding tool like Create React App (developed by the creators of React) or the new and increasingly popular Vite. We'll take a look at creating a React project boilerplate with both scaffolding tools in this section, as well as briefly explore the files they output and how they connect. Let's start with the older, more standard way of generating a React project - Create React App (CRA for short).

Setup a React project using Create React App

To generate a new React project using CRA, you would run the following command in the terminal (**don't do this just yet!**):

```
npx create-react-app my-app
```

Let's break down this command a bit, beginning with the **npx** part.

npx is a command-line tool that is included with the Node.js package manager, npm, starting from version 5.2.0. It's purpose is to make it easier to run Node.js packages and scripts from the command line without installing them in a project or globally on our computers.

create-react-app refers to the Node package developed by the React team to scaffold React projects. By running **npx create-react-app**, we're saying we want to run the **create-react-app** executable but not install it in our project or on our machine.

Lastly, **my-app** is simply the name we want to give the directory our project will be created in (this will also be the name of our app listed in the package.json file which will be generated). This name could be anything you want, technically, or if you want to simply create the React project in whatever directory you're currently in you would replace the name **my-app** with a period character:

```
npx create-react-app .
```

This is the approach we will take here, since a directory has already been created in Codio to house our react project. Go ahead and run this command in the terminal now!

Keep in mind, this will take a while to complete since a lot is happening behind the scenes. Not only is our project boilerplate being generated but **create-react-app** is also initializing a Git repository in our project and installing all of the necessary dependencies.

When the process finishes, you'll see a message in the terminal that says "Happy hacking!". Next, run the following command in the terminal to start up the development server:

```
npm start
```

This may take a moment to complete as well. Once it does, click to open a preview in the bottom left panel and see the current state of our React application. Then, go ahead and click the **Next** button below to proceed to the next section where we'll take a brief look at the files and folders which were just created for us in our project!

CRA - Boilerplate Files and Folders

After running **create-react-app** our project directory should be organized as follows:

```
node_modules/      // Contains all project dependencies, should
be in .gitignore
public/
  ├── favicon.ico     // Icon that appears in the browser tab
  ├── index.html      // Main HTML file that gets served to the
user's browser
  ├── logo192.png     // Icon that appears on user's home screen
(192px)
  ├── logo512.png     // Icon that appears on user's home screen
(512px)
  ├── manifest.json   // Provides various info about application
(mainly for PWAs)
  └── robots.txt      // Provides instructions to search engine
crawlers
src/
  ├── App.css              // CSS file for App.js
  ├── App.js               // Main file containing all
application components
  ├── App.test.js          // Test file for App.js
  ├── index.css            // CSS file for index.js
  ├── index.js             // File which renders our app to the
DOM
  ├── logo.svg             // SVG logo
  ├── reportWebVitals.js   // Contains code to measure
performance of our app
  └── setupTests.js        // Imports jest-dom package for tests
.gitignore          // Specifies which files Git should ignore
package-lock.json   // Contains project info (name, description,
scripts, dependencies, etc.)
package.json        // Contains project info (name, description,
scripts, dependencies, etc.)
README.md           // README file generated by CRA
```

Many developers often do a bit of file cleanup (largely a matter of preference), such as deleting everything in the **public** folder except **index.html**, deleting **reportWebVitals.js**, **logo.svg**, **App.test.js** and **setupTests.js** (unless testing is needed in the application), etc., but we won't worry about that here.

For now, let's just open up and examine a few files. The main ones of interest to us are: **index.html**, **index.js**, and **App.js**. Click the link below to open these files in new tabs in the top left panel:

If the link doesn't open the files, from the Codio menu bar go to **Project -> Resync File Tree** and then try the "Open Files" link again after about 5 seconds (sometimes, when creating files via the terminal, the file tree doesn't update right away).

Let's start with **index.html**, which should look like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-
scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png"
/>
    <!--
      manifest.json provides metadata used when your web app is
installed on a
      user's mobile device or desktop. See
https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder
during the build.
      Only files inside the `public` folder can be referenced
from the HTML.

      Unlike "/favicon.ico" or "favicon.ico",
"%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-
root public URL.
      Learn how to configure a non-root public URL by running
`npm run build`.
    -->
    <title>React App</title>
  </head>
```

```
  <body>
    <noscript>You need to enable JavaScript to run this app.
</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an
empty page.

      You can add webfonts, meta tags, or analytics to this
file.
      The build step will place the bundled scripts into the
<body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or
`yarn build`.
    -->
  </body>
</html>
```

As stated above in the file structure overview, **index.html** is the file that will be served to the user's browser. It contains a few things particular to React - most importantly the **div** element with the id of **root**, where our entire React application will be loaded.

Next, let's have a look at **index.js** which should look as follows:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass
a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more:
https://bit.ly/CRA-vitals
reportWebVitals();
```

This file's main purpose is to establish the app's root element for React, by passing our div with the id of **root** to the **createRoot** method of **ReactDOM**, which handles the rendering of our application.

Last, but not least, is **App.js** which should look as follows:

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

This is the file where you typically start working on your React app, replacing everything in here with your own components, routes (which we'll cover later), etc. Normally, you don't do much if anything in **index.js** except remove a couple of things during any initial file cleanup.

Now, let's move on and see how to scaffold a React project with Vite.

Setting up a React Project with Vite

To create a React project with Vite, we need to run the following command in the terminal:

```
npm create vite@latest .
```

Notice we are using the period character (**.**) again because, as with CRA, we're going to scaffold the project inside the current directory which is a directory already created in Codio to hold this project (separate from our CRA-generated project).

Follow the Vite CLI prompts: it shouldn't ask you to give your project a name, but if it does just accept the default it suggests. Then, at the prompt asking what kind of project you want to create, use the down arrow on your keyboard to select **React** and hit enter. In the same way, select **JavaScript** at the next prompt and hit enter.

Next, we need to install the project dependencies. To do so, run the following command:

```
npm install
```

Next, we need to make one small change to the **dev** script in the **package.json** file in order to be able to preview our Vite-generated React app in Codio. Click the link below to open **package.json** in a new tab in the top left panel:

. If the link doesn't open the file, refresh the file tree (**Project -> Resync File Tree**) and then try again after about 5 seconds.

In the **scripts** section of this file, modify **dev** so it reads as follows:

```
  "dev": "vite --host",
```

Next, feel free to close the file and let's fire up the development server using the following command:

```
npm run dev
```

Once the server is ready, click to open a preview in the bottom left panel. Voila! There is our starter React project generated by Vite.

Vite - Boilerplate Files and Folders

Now, as with CRA, let's look at the files and folders created:

```
node_modules/       // Contains all project dependencies, should
be in .gitignore
public/
  └── vite.svg          // Vite SVG logo
src/
  ├── assets/
        ├── react.svg   // React SVG logo
  ├── App.css              // CSS file for App.js
  ├── App.jsx              // Main file containing all
application components
  ├── index.css            // CSS file for index.js
  ├── main.jsx             // File which renders our app to the
DOM
.gitignore          // Specifies which files Git should ignore
index.html          // Main HTML file that gets served to the
user's browser
package-lock.json   // Auto-generated during "npm install";
keeps track of dependency versions
package.json        // Contains project info (name, description,
scripts, dependencies, etc.)
vite.config.js      // Config file for Vite
```

As with CRA, some developers will want to do some basic file cleanup to start fresh, such as deleting the SVGs, **index.css**, etc. Again, it's a matter of preference, but here in this lesson we'll leave everything as it is.

Now that we've had a look at both approaches to creating a React project, let's quickly talk about the pros and cons of each.

CRA vs Vite: Pros and Cons

CRA is the most popular choice for getting started with React, as it is officially supported by Facebook and provides a comprehensive setup out of the box. Some advantages of CRA include:

- A large community, which means that finding support and resources is easier
- A fully-configured environment, complete with all required dependencies, development server, and build configurations fine-tuned by the React team
- Official support from Facebook, ensures that the tool will remain up-to-date and compatible with new React features

However, CRA also has some disadvantages:

- Slower development speed compared to Vite, due to its reliance on Webpack for bundling. This can become noticeable in larger projects.
- Slower to generate project boilerplate

Vite, on the other hand, uses **esbuild** under the hood (which is written in the Go programming language and compiles to native code). Vite also uses the native ES modules feature in modern browsers (with some enhancements), which allows for faster development and better performance.

Some advantages of Vite include:

- Much faster project scaffolding
- Faster development speed
- Simpler configuration, making it easier for beginners to understand
- Improved performance during development, as it only rebuilds and reloads the changed modules (Hot Module Reloading or HMR)

A couple of downsides of using Vite are:

- Fewer resources available compared to CRA, as it is a newer tool
- Potentially less support compared to CRA, since it is not officially supported by Facebook

Whichever scaffolding tool you use in your own projects is up to you. However, going forward in these lessons all our React projects will be generated with Create React App.

A Primer on JSX

A fundamental concept to understand in React is JSX. But, what is JSX exactly?

JSX is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript code. It is an essential part of React because it helps to simplify the process of creating complex UIs. JSX expressions are written just like any HTML tag and are then compiled into JavaScript function calls by Babel, a JavaScript compiler. This compilation step is typically handled automatically by the development environment (CRA or Vite) during the development and build processes.

Here's a simple example of JSX:

```
const element = <h1>Hello, world!</h1>
```

In the example above, the JSX expression is assigned to the **element** variable. Note that you normally wouldn't write JSX like this in React components; this is just a simple example. We'll have a look at component syntax shortly.

At any rate, when our code above is compiled, it is transformed into a JavaScript function call, similar to the following:

```
const element = React.createElement('h1', null, 'Hello, world!')
```

JSX also supports embedding JavaScript expressions within the HTML-like code by wrapping them in curly braces. This allows you to create dynamic content within your components. Here's an example:

```
const name = 'John Doe';
const element = <h1>Hello, {name}!</h1>
```

In this example, the **name** variable is embedded within the JSX expression, resulting in a dynamically generated greeting. This way, if you ever wanted to change the greeting's output in every place it gets used, all you would have to do is change the value of the **name** variable.

Now that we've seen what this strange HTML-JavaScript hybrid called JSX looks like, let's move on and explore how to create actual components in React!

Syntax of Components: Classes vs Functions

In React, there are two ways to create components: using class components and using function components. Both options have their advantages and use cases, but with the introduction of hooks in React 16.8, function components have become more powerful and are now the recommended way to create components.

In fact, in React's new documentation, everything regarding class components has now been moved to a section called 'Legacy APIs' and while class components are still supported they explicitly state not to use class components in new code.

We will still take a look at class components here, because in a future job you may run into a React codebase which still uses them and it is thus helpful to know what they look like and how they work.

Class components

Class components are created using ES6 classes and extend the **React.Component** class. They require a render method that returns the JSX to be rendered.

Here is an overview of the anatomy of class components:

```
// Required React imports
import React, { Component } from 'react';

// Component Declaration
class ComponentName extends Component {

    // Component constructor (optional)
    constructor(props) {
        // 'super' method (optional)
        super(props);
        // Any initial setup for the component, including state
    }

    // Any additional methods or event handlers go here

    // Mandatory render method
    render() {
        // Return JSX
        return <div>Some content here...</div>;
    }
}

// Export the component
export default ComponentName;
```

Breakdown

- **Import Statement**: The required React imports. The basic React import is mandatory for every component.
- **Component Declaration**: The class component is declared using the class keyword followed by the name of the component. This class extends React.Component to inherit React functionality.
- **Constructor**: This is a special method for creating and initializing an object created from a class. If we need to use props in the constructor, we pass it to both the constructor and the super method.
- **Methods/Event Handlers**: Any custom methods or event handlers would go in this section.
- **Render Method**: The only required method in a class component. This is where you define the component's output. It must return a single parent element, though this parent element can contain any number of children.
- **Export Statement**: This makes the component available for use in other parts of the application.

Go ahead and click the following link to open App.js, to see an example of a class component:

If the link doesn't open the file, refresh the file tree (**Project -> Resync File Tree**) and try again.

Here in **App.js** we have a very basic class component, containing only a **render** method which in turn simply returns the message "Hello, world!" inside **h1** tags.

To get a little practice creating class components, let's create another file in our project. First we need to stop our dev server by clicking inside the terminal window to make sure it has focus and then hitting **CTRL+C** on the keyboard.

Once the dev server is stopped, run the following command in the terminal:

```
touch HelloWorld.js
```

Now, click the following link to open our newly created **HelloWorld.js** file in a new tab (make sure you've run the above command *before* clicking the link!):

.

If the link doesn't open the file, refresh the file tree (**Project -> Resync File Tree**) and try again.

Next, write the following code in the file:

```
import React, { Component } from 'react'

class HelloWorld extends Component {
  render() {
    return <h1>Hello, World!</h1>
  }
}

export default HelloWorld
```

Basically what we've done here is to abstract the "Hello, world!" message to its own component. Next, let's modify **App.js** to include the **HelloWorld** component:

```
import React, { Component } from 'react'
import HelloWorld from './HelloWorld'
import './App.css'

class App extends Component {
  render() {
    return (
      <HelloWorld />
    )
  }
}

export default App
```

Now, fire up the dev server again so we can make sure everything is still working:

```
npm start
```

Once the dev server is running, be sure to click the refresh button in the preview window.

If you still see “Hello, world!” message in the preview screen then everything is working correctly.

Great! You just created your first component in React!

Note that class components also provide what are called lifecycle methods, such as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**, which can be used to execute code at specific points during a component’s lifecycle. Here’s a brief overview of each of these methods and what they do:

- **componentDidMount**: This method is called once the component has been rendered to the DOM. It is a good place to fetch data, set up subscriptions, or perform other actions that need to be executed once the component is visible on the screen.
- **componentDidUpdate**: This method is called whenever the component’s state or props change. It receives the previous props and state as arguments, allowing you to compare the old values with the current values and perform actions based on the changes.
- **componentWillUnmount**: This method is called just before the component is removed from the DOM. It is a good place to clean up any subscriptions, timers, or other resources that were created during the component’s lifecycle.

If you want to learn more about lifecycle methods and how they're used in class components, see the React documentation.

Let's move on for now and look at function components - the now standard and recommended way of creating components in React.

Function Components

Function components are a more concise way to create React components. Instead of using a class, they are simply JavaScript functions that return JSX. With the introduction of hooks (which, as mentioned previously, we'll dive more into later), function components can now manage state and implement functionality similar to the lifecycle methods in class components, making them just as powerful.

Here is an overview of the anatomy of function components:

```
// Component Declaration
function ComponentName(props) {
    // Return JSX
    return <div>Some content here...</div>;
};

// Export the component
export default ComponentName;
```

> **NOTE**: `import React from 'react';` not required as of React v17.0

Breakdown

- **Component Declaration**: The function component is simply a JavaScript function. This function receives one argument: `props`.
- **Return Statement**: The JSX to be rendered is returned by the function. As with the class component's render method, this must return a single parent element.
- **Export Statement**: This makes the component available for use in other parts of the application.

Let's look at function components using the same example we used for class components. Click the link below to open **App.js**, now refactored as a function component:

If the link doesn't open the file, refresh the file tree (**Project -> Resync File Tree**) and try again.

As before, let's stop the dev server using **CTRL+C**, and create a new file in the **src** directory called **HelloWorld.js** by running the command below:

```
touch HelloWorld.js
```

Next, open **HelloWorld.js** by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), and then write the following code:

```
function HelloWorld() {
  return <h1>Hello, world!</h1>;
};

export default HelloWorld;
```

> **Note**: Function components can be written using either the syntax above, or arrow function syntax:

```
const HelloWorld = () => {
  return <h1>Hello, world!</h1>;
};

export default HelloWorld;
```

Now, let's add our **HelloWorld** component in **App.js**:

```
import HelloWorld from './HelloWorld';

function App() {
  return <HelloWorld />;
};

export default App;
```

Great! Now, let's restart our dev server to make sure everything is working. In the terminal, run:

```
npm start
```

Once the dev server is running, be sure to click the refresh button in the preview window.

As with our class component example, we've created a very basic component named **HelloWorld** which simply returns JSX that will display the words "Hello, world!" on the screen. As we progress further in this module, we'll be creating more and more complex components.

Knowledge Check

KNOWLEDGE CHECK

Conclusion and Takeaways

In this lesson, we explored the basics of React and learned how to set up a development environment using both create-react-app and Vite. We also discussed the pros and cons of each tool, helping to make an informed decision on which one to use for your projects.

Furthermore, we delved into JSX, an essential part of React that allows you to write HTML-like code within your JavaScript. We covered how JSX expressions are compiled into JavaScript function calls and how to embed JavaScript expressions within JSX to create dynamic content.

We also discussed the two ways to create components in React: class components and function components, exploring their differences, use cases, and structure. We also talked about how, with the introduction of hooks, function components have become the recommended way to create components in React.

With this foundation in place, we are now ready to dive a little deeper into the world of React.

React Basics - Part 2

Goals

By the end of this lesson, you should understand:

- How to utilize props to pass data to child components
- The concept of hooks and how to utilize them
- How to manage state within a React component
- Various methods of styling React components, including inline styles, CSS/SCSS modules, and styled-components

Introduction

In the previous lesson, we covered many of the basics of React, including setting up a development environment, JSX, and components. In this lesson, we'll continue learning about key React concepts that will help us understand how to build dynamic frontend applications. We'll explore the use of props to pass data to components, as well as React hooks, managing state within components, deciding which components should have state, and various methods of styling React components.

The Concept of Props

In React, props (short for “properties”) are the way to pass data from a parent component to a child component. Props are read-only, meaning that a child component should never modify the data passed to it through props. This concept is known as “unidirectional data flow” because data flows in one direction, from parent to child.

🛠 Hands-On Challenge

Step 1: Setting up our workspace

1. Start by creating two new files named **ParentComponent.js** and **ChildComponent.js** in the **src** folder. To do this, execute the following command in the terminal:

```
touch ParentComponent.js ChildComponent.js
```

2. Access your files using this link: . If it doesn’t work, resync the file tree and try again.

Step 2: Modifying App.js

1. First, clear out the default content in **App.js**.
2. Next, import `ParentComponent` from **ParentComponent.js** at the top of your file.
3. Create a function component named `App`. This function should return `ParentComponent`.
4. Don’t forget to export the `App` function as a default export at the bottom of the file!

Step 3: Creating the ParentComponent

1. In **ParentComponent.js**, start by importing `ChildComponent` from **ChildComponent.js**.
2. Create a function component named `ParentComponent`.
3. Inside this component, declare a constant called **userName** and assign it the value “John Doe”.
4. The component should return `ChildComponent`, and the **userName** constant should be passed as a prop called **name**.
5. Lastly, export `ParentComponent` as default so it can actually be accessed in **App.js**.

Step 4: Creating the ChildComponent

1. Inside **ChildComponent.js**, create a function component named

`ChildComponent` that takes in one argument, **props**.
2. Within this component, return an `h1` element containing the starting text “Hello, !”.
3. Display the prop **name** (which was passed to this component) inside opening and closing curly braces, after the word *Hello* and the comma but before the exclamation mark and closing `h1` tag. Remember, in function components, props are accessed directly via the `props` argument!
4. Export `ChildComponent` as default, so it can be accessed in **ParentComponent.js**.

Test Your Implementation

Fire up the dev server using `npm start`.

Once the dev server is running, click to open a preview in the bottom left panel. We should see the message “Hello, John Doe!” on the screen.

➡ Up Next

In the next section, we’ll take a look at React **hooks**, which, when they were introduced, effectively made function components just as powerful and versatile as class components. We’ll also start to explore another very important React concept: **state**.

Overview of Hooks

Overview of Hooks

React Hooks are a powerful feature introduced in React 16.8 that allows you to use state and other React features in functional components. Hooks are functions that let you “hook into” React state and lifecycle features from functional components.

The motivation behind introducing hooks in React was to address several issues developers faced with class components, such as complexity in reusing stateful logic, and difficulties in understanding and maintaining the lifecycle methods.

There are two main types of hooks in React:

1. **Built-in Hooks**: These are hooks provided by the React library. Some of the most commonly used built-in hooks are:

- **useState**: This hook allows you to declare a state variable in your functional component. It returns an array with the current state value and a function to update it.
- **useEffect**: This hook lets you perform side effects, such as fetching data or updating the DOM, in your functional component. It serves a similar purpose to componentDidMount, componentDidUpdate, and componentWillUnmount in class components.
- **useContext**: This hook allows you to access the value of a context without using a context consumer in the component tree. It simplifies working with context in functional components.
- **useRef**: This hook is used to create and access references to DOM elements or other objects within a functional component.
- **useCallback**: This hook returns a memoized version of a callback function that only changes if one of its dependencies has changed. It’s useful for optimizing performance in certain scenarios.
- **useMemo**: This hook helps you memoize the result of a function so that it’s only recomputed when one of its dependencies changes. It’s useful for optimizing performance in certain scenarios as well.

2. **Custom Hooks**: These are user-defined hooks that allow you to reuse stateful logic between different components. Custom hooks are essentially JavaScript functions that follow a naming convention (usually starting with “use”) and can utilize other hooks inside them. By extracting common stateful logic into custom hooks, you can make your code more modular, reusable, and easier to maintain.

To use hooks in your React project, it's essential to follow the some basic rules, such as the naming convention of starting with "use", not calling hooks inside loops, conditions, or nested functions, and only calling hooks from React functional components or custom hooks.

In this section, we'll explore some of the most common built-in hooks and their usage.

useState Hook

State: An Overview

State is an essential concept in React, as it allows components to maintain and manage their own data, which can change over time. When a component's state changes, React automatically re-renders the component, updating the UI to reflect the new state.

To use state in function components, we will make use of the **useState** hook provided by React.

First things first, however; let's create the files we'll need for our various hook example components:

```
touch StateHook.js EffectHook.js RefHook.js RefPersistedValue.js
```

Now, go ahead and open the necessary files by clicking the following link:

If the link doesn't open the files, refresh the file tree (**Project -> Resync File Tree**) and try again.

useState Hook

The **useState** hook allows you to add state to your functional components. It returns an array with two elements: the current state value and a function to update it. This hook provides a simple way to manage local state in your components without needing to refactor them into class components or rely on external state management libraries.

Add the following code to **StateHook.js**:

```
import { useState } from "react"

function Notes() {
  const [notes, setNotes] = useState([])

  const addNote = () => {
    setNotes([
      ...notes,
      {
        id: Date.now(),
        text: `New note ${Date.now()}`
      }
    ])
  }

  const deleteNote = (id) => {
    setNotes(notes.filter((note) => note.id !== id))
  }

  return (
    <div className="container">
      <h3>Creating and updating state</h3>
      <button
        onClick={addNote}
      >
        Add Note
      </button>
      <ul>
        {notes.map((note) => (
          <li key={note.id}>
            {note.text}{' '}
            <button
              onClick={() => deleteNote(note.id)}
            >
              Delete
            </button>
          </li>
        ))}
      </ul>
    </div>
  )
}

export default Notes
```

In this example, we just made a very (*very*) basic notes app, which tracks a single piece of state called **notes** as an array. An empty array is passed as an argument into the **useState** function call which is the state's initial value.

We then create two functions, one to add a note and one to delete a note. The **addNote** function, in this case, simply adds a note with predetermined text using the **setNotes** function returned from **useState**. Normally, in an app like this, you would let the user enter custom text; we're writing it like this here for the sake of simplicity.

Then the **deleteNote** function takes in an **id** as an argument, and filters out the note with that **id** from the notes list.

Let's now import this to **App.js** and add it to the return:

```
import StateHook from './StateHook'
import './App.css'

function App() {
  return (
    <div className="App">
      <section>
        <h1>useState</h1>
        {/* Include useState component here */}
        <StateHook />
      </section>
    </div>
  )
}

export default App
```

Let's start the dev server and make sure it works:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

It is also possible, and occasionally necessary, to pass state to child components. Let's see what this would look like next.

Passing State as Props

Once a state is defined in a parent component, you can pass it down to child components as a prop.

Basic Example:

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

function ParentComponent() {
  const [count, setCount] = useState(0);

  return <ChildComponent count={count} />;
}

function ChildComponent(props) {
  return <h1>{props.count}</h1>;
}
```

In the example above, the **ParentComponent** has a state called count and it's passed to the **ChildComponent** as a prop.

Passing Down a Function to Update State

Often, child components need a way to update the state of their parent component. You can pass down a function that updates the state to the child component.

Basic Example:

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

function ParentComponent() {
  const [count, setCount] = useState(0);

  return <ChildComponent count={count} increaseCount={() =>
setCount(count + 1)} />;
}

function ChildComponent(props) {
  return (
    <div>
      <h1>{props.count}</h1>
      <button onClick={props.increaseCount}>Increase
Count</button>
    </div>
  );
}
```

Here, the ParentComponent passes down the increaseCount function to the ChildComponent which allows it to update the count state.

Prop Drilling

When you have multiple levels of components, you might find yourself passing props through several components that don't necessarily use them, just so they reach a deeper nested component. This is referred to as "prop drilling".

Basic Example:

```
function ParentComponent() {
  const [message, setMessage] = useState('Hello from Parent!');

  return <ChildComponent message={message} />;
}

function ChildComponent(props) {
  return <GrandchildComponent message={props.message} />;
}

function GrandchildComponent(props) {
  return <h1>{props.message}</h1>;
}
```

The Problem with Prop Drilling:

- **Verbosity**: As the application grows, passing down props through multiple levels can become tedious and hard to manage.
- **Maintenance**: It can become challenging to track where the data is coming from or where it's being used.
- **Reusability**: Components become less reusable since they might require props that they don't directly use.

To overcome prop drilling, developers often use React's Context API or state management libraries like Redux.

Summary:

- State in function components can be maintained using the useState hook.
- State and functions to update state can be passed to child components as props.
- Prop drilling refers to the pattern where props are passed through components that don't necessarily use them, leading to verbosity, maintenance challenges, and reduced component reusability.

By understanding these concepts and using them judiciously, you can ensure your React applications are both efficient and maintainable.

Now that we've spent some time exploring how to work with state, it's important to note that not all components necessarily *need* state. Let's talk about some general guidelines for determining which components should have state and which should not.

Which Components Should Have State

Deciding which components should have state

It's important to carefully consider which components should have state. As a general rule, you should strive to keep as many components as possible stateless. Stateless components are easier to reason about, test, and maintain. When determining which components should have state, consider the following questions:

- Is the data specific to this component and not needed by any other components?
- Does the data change over time, requiring the component to re-render?

If the answer to both questions is yes, then the component likely needs state. Otherwise, and if a given component still needs to access a particular piece of state, it's likely best to consider using a shared state management solution like Redux or React Context.

Let's now move on to the next React hook we will look at: **useEffect**.

useEffect Hook

useEffect

The **useEffect** hook is used to perform side effects, such as fetching data, manipulating the DOM, or subscribing to a service. It takes two arguments: a function containing the side effect, and an array of dependencies. When the dependencies change, the effect function will run. If the array is empty, the effect will only run once, when the component mounts.

Inside **EffectHook.js**, add the following code:

```
import { useState, useEffect } from "react"

function EffectHook() {
  const [user, setUser] = useState(null)

  useEffect(() => {
    async function fetchUser() {
      let userData
      setTimeout(() => {
        userData = {
          id: Date.now(),
          name: {
            first: "John",
            last: "Doe"
          }
        }
        setUser(userData)
      }, 2000)
    }

    fetchUser()
  }, [])

  return (
    <div>{user ? <div>Name: {`${user.name.first}
        ${user.name.last}`}</div> : <p>Loading...</p>}</div>
  )
}

export default EffectHook
```

In this example, we use the **useEffect** hook with the **setTimeout** method to mimic fetching user data from an API (we'll look at using an actual API with React in a future lesson). The *async* function **fetchUser** is defined and called within the **useEffect** callback. Once the timeout elapses, the **setUser** function is called to update the component's state.

Now let's add this component to **App.js** as well:

```
// After StateHook import
import EffectHook from './EffectHook'

//  ...

      <section>
        <h1>useEffect</h1>
        {/* Include useEffect component here */}
        <EffectHook />
      </section>
```

useRef Hook

The **useRef** hook returns a mutable ref object whose current property is initialized to the passed argument. **useRef** can serve multiple purposes, including:

a. Storing a reference to a DOM element
b. Storing values that persist across re-renders but do not cause a re-render when changed

🛠 Hands-On Challenge

Step 1: Initializing `useRef` with DOM Reference

1. In **RefHook.js** import the useRef hook from React.
2. Define a functional component named RefHook.
3. Inside the component, create a ref named inputRef using the useRef hook.
4. Create a function named focusInput that uses the current property of inputRef to set the focus to an input element.
5. Return a div containing an input element and a button. Attach the inputRef to the input element using the ref attribute. The button should have an onClick event handler set to the focusInput function.
6. Export the RefHook component.

Step 2: Using `useRef` to Persist Values

1. In **RefPersistedValue.js**, import the necessary hooks: useState, useRef, and useEffect from React.
2. Define a functional component called RefPersistedValue.
3. Declare a state variable count initialized to 0.
4. Create a ref named countRef and initialize it with the value of count.
5. Set up an useEffect to update the countRef with the current value of count whenever count changes.
6. Set up another useEffect to increment the count every second using setInterval. Remember to clear the interval on cleanup.
7. Return a div displaying the value of countRef.current.
8. Export the RefPersistedValue component.

Step 3: Integrating with App.js

1. In **App.js**, import both `RefHook` and `RefPersistedValue` at the top.
2. Inside the App component, after any existing sections:
3. Find the heading “useRef” and nest the `RefHook` component inside this section.
4. Next, find the section titled “useRef (persist value)”. Place the

`RefPersistedValue` component in this section.

5. Ensure that the entire App structure is intact and correctly nested.

Test Your Implementation

Run your development server by executing npm start in your terminal. Once the server is up, access the provided preview link. You should see sections dedicated to both useRef use-cases. Try interacting with the input in the RefHook section by clicking the button. For the RefPersistedValue section, you should observe the count updating.

Once the dev server is running, click to open a preview in the bottom left panel.

➡ Up Next

In the next section, we'll dive deeper into state management in function components.

Styles

Styles in React

There are several ways to style React components, each with its own set of advantages and trade-offs. In this lesson, we'll cover four popular styling methods: inline styles, CSS modules, SCSS modules, and styled-components.

Inline styles

Inline styles are an easy way to apply styles directly to an element in your JSX. To use inline styles, you'll need to create a JavaScript object that contains the CSS properties and their corresponding values, and then pass that object to the style attribute of your element.

Lets create a new file called **InlineStyledComponent.js**:

```
touch InlineStyledComponent.js
```

Now open the new file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), then add the following code:

```
function InlineStyledComponent() {
  const buttonStyles = {
    backgroundColor: "blue",
    color: "white",
    padding: "10px",
    borderRadius: "5px",
    cursor: "pointer"
  }

  return (
    <div className="inline-styles-container">
      <h3>Inline Styled Button</h3>
      <button style={buttonStyles}>Click me!</button>
    </div>
  )
}

export default InlineStyledComponent
```

Add the new component to **App.js**:

```
import './App.css'
import InlineStyledComponent from './InlineStyledComponent'

function App() {
  return (
    <div className="App">
      <h1>Styles</h1>
      <InlineStyledComponent />
    </div>
  )
}

export default App
```

While inline styles work, and in certain circumstances where there are only a couple styles that need to be added they may be all you need, inline styles definitely have limitations such as the inability to use pseudo-selectors like **:hover**, and a lack of support for media queries.

CSS modules

CSS modules are a way to write CSS that is scoped to a specific component, avoiding global namespace conflicts. To use CSS modules, create a **styles.module.css** file as well as a component file called **CSSModuleComponent.js**:

```
touch styles.module.css CSSModuleComponent.js
```

Open the files by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**). Next, add the following styles to the CSS file:

```
/* styles.module.css */
.button {
  background-color: blue;
  color: white;
  padding: 10px;
  border-radius: 5px;
  cursor: pointer;
}
```

Now add the following code to **CSSModuleComponent**:

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

function CSSModuleComponent() {
  return (
    <div className="css-modules-container">
      <h3>CSS Module Button</h3>
      <button className={styles.button}>Click me!</button>
    </div>
  )
}

export default CSSModuleComponent
```

Add the new component to **App.js**:

```
import './App.css'
import InlineStyledComponent from './InlineStyledComponent'
import CSSModuleComponent from './CSSModuleComponent'

function App() {
  return (
    <div className="App">
      <h1>Styles</h1>
      <InlineStyledComponent />
      <CSSModuleComponent />
    </div>
  )
}

export default App
```

Now, before proceeding, let's run the dev server to make sure there are no errors:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

SCSS modules

SCSS modules

SCSS modules are similar to CSS modules but use the SCSS syntax, allowing you to leverage features like variables and mixins. Since React supports the use of SCSS out of the box, to use SCSS modules all we have to do is install **sass**, then create a **styles.module.scss** file and **SCSSModuleComponent.js** file, and follow the same steps as with CSS modules.

First, navigate up one directory by running the following command:

```
cd ..
```

Now install **sass**:

```
npm install sass
```

Go ahead and navigate back to the **src** directory:

```
cd src/
```

Next, let's create the files we'll need:

```
touch styles.module.scss SCSSModuleComponent.js
```

Open the files by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**).

Add the following to the **styles.module.scss** file:

```
/* styles.module.scss */
$bg-color: blue;
$border-radius: 5px;

.button {
  background-color: $bg-color;
  color: white;
  padding: 10px;
  border-radius: $border-radius;
  cursor: pointer;
}
```

Add the following to component file:

```
import styles from './styles.module.scss'

function SCSSModuleComponent() {
  return (
    <div className="scss-modules-container">
      <h3>SCSS Module Button</h3>
      <button className={styles.button}>Click me!</button>
    </div>
  )
}

export default SCSSModuleComponent
```

Next, update **App.js**:

```
import './App.css'
import InlineStyledComponent from './InlineStyledComponent'
import CSSModuleComponent from './CSSModuleComponent'
import SCSSModuleComponent from './SCSSModuleComponent'

function App() {
  return (
    <div className="App">
      <h1>Styles</h1>
      <InlineStyledComponent />
      <CSSModuleComponent />
      <SCSSModuleComponent />
    </div>
  )
}

export default App
```

Styled-components

Styled-components is a popular library for styling React components using tagged template literals. It allows you to write CSS directly in your JavaScript, and it automatically generates unique class names for your styles. To use **styled-components**, you'll need to change directories to the project root and install the library.

First, change directories:

```
cd ..
```

Then, install **styled-components**:

```
npm install styled-components
```

Next, change directories back to the **src** folder:

```
cd src/
```

Now, create a file called **StyledComponent.js**:

```
touch StyledComponent.js
```

Open the file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**).

Next, add the following code in your component file:

```
import styled from "styled-components"

const StyledButton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px;
  border-radius: 5px;
  cursor: pointer;

  &:hover {
    background-color: white;
    color: blue;
  }
`

function StyledComponent() {
  return (
    <div>
      <h3>Styled Component button</h3>
      <StyledButton>Click me!</StyledButton>
    </div>
  )
}

export default StyledComponent
```

Lastly, let's update **App.js** as we have done before:

```
import './App.css'
import InlineStyledComponent from './InlineStyledComponent'
import CSSModuleComponent from './CSSModuleComponent'
import SCSSModuleComponent from './SCSSModuleComponent'
import StyledComponent from './StyledComponent'

function App() {
  return (
    <div className="App">
      <h1>Styles</h1>
      <InlineStyledComponent />
      <CSSModuleComponent />
      <SCSSModuleComponent />
      <StyledComponent />
    </div>
  )
}

export default App
```

Now, to make sure there are no errors, let's start the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

The advantage to using CSS/SCSS modules or styled-components is you can apply styles to pseudo-classes and selectors, as well as define media queries. In CSS/SCSS you would do this like normal, but in **styled-components** you can add the **&** character which stands for the current element to which the styles are being applied - similar to how you would write it if you were nesting styles in SCSS.

Knowledge Check

KNOWLEDGE CHECK

Conclusion & Takeaway

Conclusion & Takeaway

In this lesson, we covered some more essential concepts for building dynamic and interactive React applications. We learned how to pass data using props, manage state with the **useState** hook, and decide which components should have state. We also explored different styling methods, including inline styles, CSS modules, SCSS modules, and styled-components.

In the next two lessons, we'll take a deep dive into more complex topics in React and begin building some simple web apps.

React Deep Dive - Part 1

Goals

By the end of this lesson, you should understand:

- How to use conditional rendering in React to display content based on specific conditions, and how to iterate over lists to display content
- How to handle user input and create forms in React applications

Introduction

In this lesson, we'll dive into some of the most important React concepts, such as conditional rendering, lists, and forms. By the end of this lesson, you'll have a deeper understanding of how React works and how to build more advanced components for your applications. This comprehensive guide will help you master the fundamental aspects of React and lay the groundwork for future lessons, which will explore more advanced topics.

Conditional Rendering - Overview

In React, you can render content conditionally based on certain conditions. This is a powerful feature that allows you to create dynamic components that respond to user interaction, state, or other factors. You can use JavaScript operators, such as **if**/**else** statements or ternary expressions, to achieve this.

Syntax

In JSX, the most common approach to conditional rendering is using the ternary operator:

```
{ condition ? <ComponentIfTrue /> : <ComponentIfFalse /> }
```

In React, when we want to include JavaScript expressions within JSX, we wrap them in curly braces `{}`. This signifies a dynamic value or expression that React should evaluate.

For a concrete example, consider the following component called **DisplayGreeting**:

```
function DisplayGreeting({ isMorning }) {
  return (
    <div>
      { isMorning ? <h1>Good morning!</h1> : <h1>Good evening!
</h1> }
    </div>
  );
}
```

Let's dissect this example, piece by piece.

1. Function Component Declaration:

```
function DisplayGreeting({ isMorning }) {
```

Here, a function component named **DisplayGreeting** is declared. It accepts a prop called **isMorning**.

2. JSX Return:

```
return (
   ...
);
```

Every React component must return JSX (or `null`).

3. JavaScript Expression in JSX:

```
{ isMorning ? ... : ... }
```

The content inside `{}` is a JavaScript expression (in this case, a ternary expression), which will be evaluated to determine which JSX element to return.

4. Ternary Expression:

```
isMorning ? <h1>Good morning!</h1> : <h1>Good evening!</h1>
```

This is a standard JavaScript ternary expression. It first evaluates the prop **isMorning**. If it's true, the expression after the `?` is executed and it'll show the morning greeting (`<h1>Good morning!</h1>`), otherwise the expression after the `:` is executed and it'll display the evening greeting (`<h1>Good evening!</h1>`).

The key takeaway: whenever we need React to evaluate a dynamic value or a JavaScript expression within JSX, we wrap that expression inside curly braces `{}`. This is fundamental to JSX syntax and is used frequently, not only for conditional rendering but anytime we want to output dynamic values.

Conditional Rendering - Practice

First run the following:

```
npm install
```

That sets up our dependencies. Then navigate to the src folder

```
cd src
```

Now let's create a component called **WelcomeMessage.js**:

```
touch WelcomeMessage.js
```

Now open the file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), and add the following code:

```
function WelcomeMessage({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? (
        <p>Welcome back, user!</p>
      ) : (
        <p>Please sign in to continue.</p>
      )}
    </div>
  )
}

export default WelcomeMessage
```

In this example, we use the ternary operator to conditionally render a welcome message based on the **isLoggedIn** prop. If **isLoggedIn** evaluates to **true**, we display a welcome message; otherwise, we prompt the user to sign in. By using conditional rendering, we can create components that adapt to different situations and provide a more interactive user experience.

Now, let's add this component to **App.js** and create state to keep track of a user's logged-in status so we can pass that state as a prop to our **WelcomeMessage** component. We'll also create two buttons that will toggle the **isLoggedIn** state and conditionally render those as well:

```
import { useState } from 'react'
import WelcomeMessage from './WelcomeMessage'
import './App.css'


function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false)

  function toggleLoggedIn() {
    setIsLoggedIn(!isLoggedIn)
  }

  return (
    <div>
      <h1>Conditional Rendering</h1>
      {isLoggedIn ? (
        <button onClick={toggleLoggedIn}>Log Out</button>
      ) : (
        <button onClick={toggleLoggedIn}>Log In</button>
      )}
      <WelcomeMessage isLoggedIn={isLoggedIn} />
    </div>
  )
}

export default App
```

Now, let's run our app and see if everything is working as expected:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Lists, Iterating, & Keys

Lists, iterating, and keys

Rendering lists in React is straightforward using the map() function to iterate over an array and render each item as a component. When rendering a list, you need to provide a unique key for each item to help React identify which items have changed, been added, or removed. Keys are essential for efficient and accurate updates when the list changes, as they enable React to track and reorder items in the list more effectively.

Run the following:

```
npm install
```

and then navigate to the src folder

```
cd src
```

To get started, let's create a component called **ItemList.js**:

```
touch ItemList.js
```

Now open the file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), and add the following code:

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li className="list-item" key={item.id}>
          <h3 className="list-item__name">{item.name}</h3>
          <p className="list-item__description">
        {item.description}</p>
          <div className="list-item__price">${item.price}</div>
        </li>
      ))}
    </ul>
  )
}

export default ItemList
```

In the example above, we use the **map()** function to iterate over the **items** array (which is imported in **App.js** from a separate file called **items.js** and will be passed as a prop to **ItemList**), rendering a list item for each as shown. We provide a unique key for each list item based on the item's id. This key helps React optimize rendering performance when the list is updated.

> **Pro Tip**
>
> As a general rule, you should avoid using array indexes as keys if the order of items may potentially change, as this can negatively impact performance and cause issues with the component's state. Instead, it's always best to use a unique identifier from the data itself, such as an id, as the key.

Now, let's add our **ItemList** component to **App.js** and pass an **items** array as a prop:

```
import items from './items'
import './App.css'
import ItemList from './ItemList'


function App() {

  return (
    <div>
      <h1>Lists, Iterating & Keys</h1>
      <ItemList items={items}/>
    </div>
  )
}

export default App
```

Lastly, let's run our app and see if everything is working as expected:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Forms

Forms

Handling user input and creating forms in React is simple, thanks to the concept of controlled components. A controlled component is a form element, such as an **input** or a **textarea**, whose value is controlled by React. To create a controlled component, you need to set the element's value to a state variable and update that state variable when the user inputs data. This approach provides better control over the form data and simplifies form validation and submission.

Run the following:

```
npm install
```

and then navigate to the src folder

```
cd src
```

To begin, let's create a new file called **ContactForm.js**:

```
touch ContactForm.js
```

Now, open the new file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), and add the following code:

```
import { useState, useRef } from "react"

function ContactForm() {
  const [name, setName] = useState("")
  const [email, setEmail] = useState("")
  const [message, setMessage] = useState("")

  const submitMsg = useRef()

  function handleSubmit(e) {
    e.preventDefault()
    // Process the form data, e.g., send it to a server.

    // Reset the state values
    resetForm()
```

```
    submitMsg.current.style.display = "block"
    submitMsg.current.innerText = "Submitted successfully!"

    setTimeout(() => {
      submitMsg.current.innerText = ""
      submitMsg.current.style.display = "none"
    }, 3000)
  }

  function resetForm() {
    setName("")
    setEmail("")
    setMessage("")
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <div className="form-control">
          <label>Name:</label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
          />
        </div>
        <div className="form-control">
          <label>Email:</label>
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
        </div>
        <div className="form-control">
          <label>Message:</label>
          <textarea
            value={message}
            onChange={(e) => setMessage(e.target.value)}
          />
        </div>
        <div className="btn-group">
          <button type="reset" onClick=
        {resetForm}>Reset</button>
          <button type="submit">Submit</button>
        </div>
      </form>
```

```
      <div ref={submitMsg} className="submit-msg"></div>
    </div>
  )
}

export default ContactForm
```

In this example, we create a controlled contact form with three input fields: **name**, **email**, and **message**. We use **useState** hooks to manage the state of each input field and update their values using the **onChange** event handler. When the form is submitted, the **handleSubmit** function is called to process the form data, reset the form, and display a success message. When the reset button is clicked, the values of **name**, **email**, and **message** each get reset back to their initial value (an empty string)

Let's update **App.js** with our new component:

```
import './App.css'
import ContactForm from './ContactForm'

function App() {
  return (
    <div>
      <h1>Forms</h1>
      <ContactForm />
    </div>
  )
}

export default App
```

Now run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel. Also, you may want to resize the panel with the preview window to make it big enough to see the entire form.

Knowledge Check

KNOWLEDGE CHECK

Conclusion

Conclusion & Takeaway

In this comprehensive lesson, we've covered essential React concepts, such as conditional rendering, lists, and forms. With a deeper understanding of conditional rendering, you can dynamically display content based on specific conditions. Understanding how to render lists and work with keys will help you manage large datasets efficiently. Finally, knowing how to create controlled components and handle user input will enable you to build interactive forms and capture user data.

As you continue your journey as a React developer, these foundational concepts will serve as the basis for more advanced topics and techniques. Keep practicing in your free time and building more complex components to reinforce your understanding of React and to become more proficient in developing user interfaces.

React Deep Dive - Part 2

Goals

By the end of this lesson, you should understand:

- The basics of routing and navigation in React using React Router 6
- How to fetch and manage data from APIs in your React app
- How to use environment variables and build a React app for production

Introduction

In this lesson, we'll explore some additional fundamental topics, like routing and navigation, fetching data from APIs, handling environment variables, and deploying a React app. This knowledge will enable you to build more complex and sophisticated applications using React. By the end of this lesson, you'll have a solid understanding of React Router 6, how to fetch and manage data from APIs, handle environment variables, and deploy your React app for the world to see.

Routing

Routing

Note: if at any time the terminal doesn't open, press the next or > button to go to the next page, then press the back button < and the terminal should refresh. Do not open a new terminal.

React Router 6 is a popular library for handling routing and navigation in React applications. To get started, install React Router in your project:

```
npm install react-router-dom@6
```

Next, create a new folder called **pages** inside the **src** folder and add three files: **Home.js**, **About.js**, and **Contact.js**:

```
mkdir src/pages
```

```
touch src/pages/Home.js src/pages/About.js src/pages/Contact.js
```

Open these newly created files by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**). These files represent different pages in your application.

In **Home.js**, add the following code:

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

function Home() {
  return (
    <div>
      <h1>Home</h1>
      <p>Welcome to the Home page!</p>
      <ul>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
    </div>
  )
}

export default Home
```

In **About.js**, add the following:

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

function About() {
  return (
    <div>
      <h1>About</h1>
      <p>Welcome to the About page!</p>
      <Link to="/">Home</Link>
    </div>
  )
}

export default About
```

In **Contact.js**, add the following:

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


function Contact() {
  return (
    <div>
      <h1>Contact</h1>
      <p>Welcome to the Contact page!</p>
      <Link to="/">Home</Link>
    </div>
  )
}

export default Contact
```

Notice that we are using the special **Link** element from **react-router-dom** to create links to our different pages. Using a regular **a** tag wouldn't work!

Now, let's set up the main routing code. In **App.js**, import the necessary components from **react-router-dom** and your pages:

```
import { BrowserRouter, Route, Routes } from 'react-router-dom'
import Home from './pages/Home'
import About from './pages/About'
import Contact from './pages/Contact'
```

Next, use the **BrowserRouter** component as a wrapper around the **Routes** component, which in turn will wrap all of the **Route** components:

```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </BrowserRouter>
  )
}

export default App
```

Now, let's start the dev server and make sure everything is working so far:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Route Params

Route Params, Nested Routes, and Protected Routes

Route params are used to capture values from the URL. To illustrate this, let's create a new file called **User.js** inside the pages folder:

```
touch pages/User.js
```

Next, open the new file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**).

```
import { useParams, Link } from 'react-router-dom'

function User() {
  const { id } = useParams()

  return (
    <div>
      <h1>User</h1>
      <p>User ID: {id}</p>
      <Link to="/">Home</Link>
    </div>
  )
}

export default User
```

Here, we are making use of the **useParams** hook from **react-router-dom** to access the value of the **id** param.

Now, add a new route for the User component in **App.js**:

```
import User from './pages/User'

// ...

// inside the <Routes> component
<Route path="/user/:id" element={<User />} />
```

The **:id** syntax in the route path is how you define a route param.

Nested routes

Nested routes allow you to create hierarchical routing structures. For example, let's create a **Dashboard** component which will hold nested routes for **Profile** and **Settings** components.

Create the necessary files:

```
touch pages/Dashboard.js pages/Profile.js pages/Settings.js
```

Open the new files by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), then, in **Dashboard.js**, add the following code:

```
// src/pages/Dashboard.js
import { Outlet, Link } from 'react-router-dom'

function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <ul>
        <li>
          <Link to="/dashboard/profile">Profile</Link>
        </li>
        <li>
          <Link to="/dashboard/settings">Settings</Link>
        </li>
      </ul>
      <Outlet />
      <Link to="/">Home</Link>
    </div>
  )
}

export default Dashboard
```

Notice we are including another component from React Router 6 called **Outlet**. This is a special component which acts as a placeholder for whichever child routes we will nest inside the **Dashboard** route.

Now add the following code to the **Profile.js** and **Settings.js** components, respectively:

```
// src/pages/Profile.js

function Profile() {
  return (
    <div>
      <h2>Profile</h2>
      <p>Nested profile component</p>
    </div>
  )
}

export default Profile
```

```
// src/pages/Settings.js

function Settings() {
  return (
    <div>
      <h2>Settings</h2>
      <p>Nested settings component</p>
    </div>
  )
}

export default Settings
```

Lastly, let's add the nested routes in **App.js**:

```
import Dashboard from './pages/Dashboard'
import Profile from './pages/Profile'
import Settings from './pages/Settings'

// ...

// inside the <Routes> component
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<Profile />} />
  <Route path="settings" element={<Settings />} />
</Route>
```

Our **App.js** file should now look as follows:

```
import { BrowserRouter, Route, Routes } from 'react-router-dom'
import Home from './pages/Home'
import About from './pages/About'
import Contact from './pages/Contact'
import User from './pages/User'
import Dashboard from './pages/Dashboard'
import Profile from './pages/Profile'
import Settings from './pages/Settings'

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="/user/:id" element={<User />} />
        <Route path='/dashboard' element={<Dashboard />}>
          <Route path='profile' element={<Profile />}/>
          <Route path='settings' element={<Settings />}/>
        </Route>
      </Routes>
    </BrowserRouter>
  )
}

export default App
```

Also, be sure to add a couple more links to **Home.js** so we can navigate to our new routes:

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

function Home() {
  return (
    <div>
      <h1>Home</h1>
      <p>Welcome to the Home page!</p>
      <ul>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
        <li>
          <Link to="/user/1">User 1 (route params)</Link>
        </li>
        <li>
          <Link to="/dashboard">Dashboard</Link>
        </li>
      </ul>
    </div>
  )
}

export default Home
```

Now, let's start the dev server and make sure everything is working:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Protected routes

Protected routes are another feature you can implement with React Router, which allow you to restrict access to certain parts of your application behind some sort of authentication/authorization system. How to set up protected routes goes a little beyond the scope of this lesson, but if you want to learn more about it, you can check out the following article:

Creating Protected Routes With React Router V6

Using APIs

Using APIs

To fetch data from APIs, you can use the **fetch** method or other libraries like **Axios**. Let's create a new component called Posts that fetches data from a sample API using **fetch**.

First, run

```
npm install
```

Next, navigate to the folder

```
cd src
```

Now, create the file:

```
touch Posts.js
```

Then open the new file by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**), and add the following code:

```
// src/pages/Posts.js
import { useEffect, useState } from "react"

const Posts = () => {
  const [posts, setPosts] = useState([])

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts?
        _limit=10&_expand=user")
      .then((response) => response.json())
      .then((data) => {
        data.forEach((post) => {
          post.title = post.title.split(' ').slice(0,3).join('
        ')
        })

        setPosts(data)
      })
  }, [])

  return (
    <div>
      <h2>Posts</h2>
      <ul className="posts">
        {posts.map((post) => (
          <li key={post.id}>
            <div className="card">
              <div className="card__title">
                {post.title}
              </div>
              <div className="card__author">
                by: {post.user.name}
              </div>
              <div className="card__body">
                {post.body}
              </div>
            </div>
          </li>
        ))}
      </ul>
    </div>
  )
}

export default Posts
```

In this example, inside **useEffect**, we fetch data from the JSONPlaceholder API, which is a fake REST API useful for testing out frontend code. The

endpoint we're using returns an array of posts with some additional information about the user who created them. As the second argument of the **useEffect** hook we pass an empty array, so that the **fetchUser** method will only be executed once, when the component mounts.

Once the data is fetched, the **response.json()** method is used to parse the response body as JSON. Then, we loop through each post object in the array and modify the **title** property to only contain the first three words of the original title (which is much longer). Finally, the modified array is set as the new state of the **posts** state variable.

In the component's return statement, we have an unordered list in which we loop through the **posts** array creating **li** elements each containing a "card" div that displays the post's modified title, author's name, and body.

Let's now add the **Posts** component in **App.js**:

```
import Posts from './Posts'
import './App.css'

function App() {
  return (
    <div className="App">
      <h1>Using APIs</h1>
      <Posts />
    </div>
  )
}

export default App
```

To test everything out, as always, let's run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Environment Variables

Environment variables

Note: if at any time the terminal doesn't open, press the next or > button to go to the next page, then press the back button < and the terminal should refresh. Do not open a new terminal.

To get started, install the dependencies:

```
npm install
```

Environment variables allow you to manage configuration settings and other information used throughout in your application. In React, environment variables are stored in a .env file in the root of your project. These variables should be prefixed with REACT_APP_.

For example, in the **.env** file open in the top left panel, add the following:

```
REACT_APP_MY_NAME="Darth Vader"
```

To access the environment variable in your code, use **process.env.REACT_APP_MY_NAME**:

```
// App.js

function App() {
  const name = process.env.REACT_APP_MY_NAME

  return (
    <div className="App">
      <h1>Environment variables</h1>
      {/* Add code below */}
      <div>Hello, my name is {name}</div>
    </div>
  )
}

export default App
```

Now start the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel. You should see the message “Hello, my name is Darth Vader” below the heading.

Git-ignoring .env files

While environment variables can be very useful, it is crucial to handle them securely to prevent security vulnerabilities.

One essential step is to add your .env files to your .gitignore file to prevent accidental commits – especially to a public repository.

The following is an example .gitignore file in a React app:

```
# See https://help.github.com/articles/ignoring-files/ for more
about ignoring files.

# dependencies
/node_modules
/.pnp
.pnp.js

# testing
/coverage

# production
/build

# misc
.DS_Store
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

npm-debug.log*
yarn-debug.log*
yarn-error.log*
```

Note that while there are a few .env file variations included in a React project’s .gitignore file by default, simply .env is not and it is therefore a good idea to add it. Taking this step ensures that Git will not track the file.

Sensitive Information in Environment Variables

While environment variables can help you manage sensitive information, it's essential to understand their limitations. Since React is a client-side library, any sensitive information stored in environment variables will be exposed to users when the application is built. For example, secret API keys stored in environment variables can be discovered through browser DevTools or by inspecting the built files.

To securely handle sensitive information like secret API keys, you should use a proxy server or a serverless function. This way, your sensitive data is stored on the server-side, ensuring it never gets exposed to users through the client-side code. Here are a few ways to implement a proxy server or serverless function:

- **Express.js Proxy**: Create an Express.js server that acts as a proxy for your API requests. Instead of making requests directly to the API from your React app, send requests to your Express server, which then makes requests to the API with the secret key. This keeps the secret key on the server-side, away from prying eyes.
- **Serverless Functions**: Use serverless functions (such as AWS Lambda, Netlify Functions, or Vercel Functions) to handle API requests with sensitive information. Serverless functions allow you to run code on-demand without the need for a dedicated server. Your sensitive information is securely stored in the serverless function's environment variables.

By using a proxy server or serverless functions, you can keep your sensitive information secure while still providing the necessary functionality to your React application.

Deploying a React app

Deploying a React app

As you begin building more and more projects in React, eventually you'll want to deploy those projects live on the internet.

There are many options for deploying a React app, but two very popular choices are **Netlify** and **Vercel**. There are several others and whichever platform you use is entirely up to you, just be sure to double-check that the platform supports React apps.

In any case, when deploying, the first step you will always take is to create a production build of your application. Remember the **scripts** section in **package.json**? The **react-scripts** package provides a command already to accomplish this very task:

```
"scripts": {
  "build": "react-scripts build"
}
```

So, in order to run the build script to create a production-ready version of your application, you would simply need to run the following command:

```
npm run build
```

Then, you would just need to follow the instructions for whichever platform you're deploying to. Here are a links to deployment instructions for both Netlify and Vercel:

- **Netlify**: https://create-react-app.dev/docs/deployment#netlify
- **Vercel**: https://create-react-app.dev/docs/deployment#vercel

Knowledge Check

KNOWLEDGE CHECK

Conclusion

Conclusion and Takeaway

In this lesson, you learned about routing and navigation using React Router 6 - including route params, nested routes, and protected routes. You also learned how to fetch and manage data from APIs in React and how to use environment variables, as well as some options for deploying React apps.

These skills are essential for building complex and feature-rich React applications. As you continue to practice and explore the React ecosystem, you'll be well-equipped to create amazing web applications.

In the next lesson, we'll take a look at performance optimization and some available options for advanced state management.

Advanced React

Goals

By the end of this lesson, you will:

- Have learned various techniques to improve performance in React applications
- Understand available options for advanced state management and the basics of how they work (both built-in tools in React, as well as a couple of external libraries)

Introduction

In this lesson, we will dive deeper into React's advanced features and best practices to help you build scalable, maintainable, and high-performance applications. We will explore performance optimization techniques, more advanced methods of state management using React Context, the useReducer hook, and external libraries such as Redux Toolkit and MobX. By the end of this lesson, you will have a strong understanding of the tools and techniques necessary for building complex and efficient React applications.

Performance Optimization

Performance Optimization

Performance optimization is essential in ensuring that your React application runs smoothly and provides a seamless user experience. Here are some key techniques to optimize your React application:

- **React.memo**: Wrap your functional components with React.memo to prevent unnecessary re-renders when the component's props have not changed. This is particularly useful for components that are computationally expensive or frequently updated.
- **Lazy-load components**: Use **React.lazy** and **React.Suspense** to load components only when needed, reducing the initial load time of your application.
- **useCallback**: Use the **useCallback** hook to memoize callback functions passed as props to child components, preventing unnecessary re-renders when the parent component updates.
- **useMemo**: Use the **useMemo** hook to memoize expensive calculations, ensuring that they are only recomputed when their dependencies change.
- **useTransition**: Use the **useTransition** hook (React version 18) to improve the perceived performance of your application by coordinating updates to multiple pieces of state according to priority.

Let's see some examples of each of these.

Lazy-Load Components

Lazy-load components

We'll actually start with **React.lazy** to simplify things for the other examples.

Note: if at any time the terminal doesn't open, press the next or > button to go to the next page, then press the back button < and the terminal should refresh. Do not open a new terminal.

First let's create the files we'll need:

```
touch pages/Home.js pages/ReactMemo.js pages/CallbackHook.js
        pages/MemoHook.js pages/TransitionHook.js
```

Now open the new files by clicking (if needed, refresh the file tree: **Project -> Resync File Tree**).

For this and the following optimization examples, we'll use routing. Go ahead and change directories to the project root:

```
cd ..
```

Now let's install **react-router-dom**

```
npm install react-router-dom@6
```

Next, in **Home.js**, add the following code:

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

function Home() {
  return (
    <div>
      <h1>React Optimization Examples</h1>
      <ul>
        <li>
          <Link to="/react-memo">React.memo</Link>
        </li>
        <li>
          <Link to="/use-callback">useCallback</Link>
        </li>
        <li>
          <Link to="/use-memo">useMemo</Link>
        </li>
        <li>
          <Link to="/use-transition">useTransition</Link>
        </li>
      </ul>
    </div>
  )
}

export default Home
```

Next, in all the other page files we'll add some boilerplate for now.

ReactMemo.js

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

function ReactMemo() {
  return (
    <div>
      <h1>React.memo</h1>
      <Link to="/">Home</Link>
    </div>
  )
}

export default ReactMemo
```

CallbackHook.js

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

function CallbackHook() {
  return (
    <div>
      <h1>useCallback</h1>
      <Link to="/">Home</Link>
    </div>
  )
}

export default CallbackHook
```

MemoHook.js

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

function MemoHook() {
  return (
    <div>
      <h1>useMemo</h1>
      <Link to="/">Home</Link>
    </div>
  )
}

export default MemoHook
```

TransitionHook.js

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

function TransitionHook() {
  return (
    <div>
      <h1>useTransition</h1>
      <Link to="/">Home</Link>
    </div>
  )
}

export default TransitionHook
```

Once this is done, feel free to close all the files except **App.js**.

Now, to demonstrate the **React.Lazy** optimization, we will update **App.js** to include routing and use **React.lazy** to lazy-load the route components:

```
import React, { Suspense, lazy } from "react"
import { BrowserRouter, Route, Routes } from "react-router-dom"
import Home from "./pages/Home"
import './App.css'

const ReactMemo = lazy(() => import("./pages/ReactMemo"))
const CallbackHook = lazy(() => import("./pages/CallbackHook"))
const MemoHook = lazy(() => import("./pages/MemoHook"))
const TransitionHook = lazy(() =>
        import("./pages/TransitionHook"))

function App() {
  return (
    <div className="container">
      <BrowserRouter>
        <Suspense fallback={<div>Loading...</div>}>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/react-memo" element={<ReactMemo />} />
            <Route path="/use-callback" element={<CallbackHook
        />} />
            <Route path="/use-memo" element={<MemoHook />} />
            <Route path="/use-transition" element=
        {<TransitionHook />} />
          </Routes>
        </Suspense>
      </BrowserRouter>
    </div>
  )
}

export default App
```

In this example, we use **React.lazy** to import the components for each route. The **lazy** function takes a function that must return a Promise resolving to the desired component. This allows Webpack to split the code and load the components only when they are needed.

We wrap the **Routes** component with the **Suspense** component, which allows us to provide a fallback UI while the components are being lazy-loaded. In this case, the fallback is a simple “Loading...” message.

With this setup, the app will load the components for each route lazily, reducing the initial bundle size and improving the app’s performance.

Let’s run the dev server and try it out:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

We should see a list of links to all our pages.

React.memo

React.memo

For React.memo, let's consider a real-world scenario where you're building an e-commerce application, and you have a shopping cart component that displays a list of items along with a summary containing the total cost. When you update the cart, you don't want to re-render the entire list of items if the only change is to the summary.

Now, let's create a couple components we'll need, **Cart.js** and **CartItem.js**, by running the following command in the terminal:

```
touch components/Cart.js components/CartItem.js
```

After running this command, click the link below to open **ReactMemo.js**, as well as the two components we just created, in new tabs:

(if needed, refresh the file tree: **Project -> Resync File Tree**)

We'll start with the **CartItem** component, which will display an individual item in the shopping cart. We'll wrap this component with **React.memo** to prevent unnecessary re-renders when the cart updates but the item's data remains the same:

```
import React from "react"

const CartItem = React.memo(({ item }) => {
  return (
    <div>
      <span>{item.name}</span> - <span>${item.price.toFixed(2)}
        </span>
    </div>
  )
})

export default CartItem
```

Now, let's code the **Cart** component which will display the list of items and the summary with the total cost. When the total cost is updated, the **CartItem** components will not re-render, thanks to **React.memo**.

```
import { useState } from "react"
import CartItem from "./CartItem"

function Cart({ initialItems }) {
  const [items, setItems] = useState(initialItems)
  const [total, setTotal] = useState(
    initialItems.reduce((sum, item) => sum + item.price, 0)
  )

  // Update the total cost and add a new item to the cart
  const addItem = (item) => {
    setTotal((prevTotal) => prevTotal + item.price)
    setItems((prevItems) => [...prevItems, item])
  };

  return (
    <div>
      <h3>Shopping Cart</h3>
      <div>
        {items.map((item) => (
          <CartItem key={item.id} item={item} />
        ))}
      </div>
      <div id="total">Total: ${total.toFixed(2)}</div><br /><br
        />
      <button onClick={() => addItem({ id: Date.now(), name:
        "New Item", price: 9.99 })}>
        Add Item
      </button>
    </div>
  )
}

export default Cart
```

Now let's bring the **Cart** component into **ReactMemo.js**, and modify our return statement to render it:

```
import React from 'react'
import { Link } from 'react-router-dom'
import Cart from '../components/Cart'
import items from '../items'

function ReactMemo() {
  return (
    <div>
      <h1>React.memo</h1>
      <Link to="/">Home</Link>
      <Cart initialItems={items} />
    </div>
  )
}

export default ReactMemo
```

In this example, the **CartItem** components are wrapped with **React.memo**, which means they will only re-render when their item prop changes. When adding a new item to the cart using the “Add Item” button, only the total cost updates, and the existing **CartItem** components do not re-render. This optimization is particularly useful in scenarios like this, where a parent component’s state update does not necessarily require updating all of its child components.

Now run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel, then click the link to go to the **React.memo** page, to make sure everything is working.

useCallback

useCallback

Suppose you have a list of numbers and a button to increment a count. When you increment the count, you want to filter out even numbers from the list without re-rendering the whole list.

Click to open **CallbackHook.js** and modify it as follows:

```
import React, { useState, useCallback } from 'react'
import { Link } from 'react-router-dom'

const List = React.memo(({ list }) => {
  return (
    <ul>
      {list.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  )
})

function CallbackHook() {
  const [count, setCount] = useState(0)
  const increment = () => setCount(count + 1)
  const getEvenNumbers = useCallback(
    () => {
      const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      return numbers.filter((number) => number % 2 === 0)
    },
    []
  )
  return (
    <div>
      <h1>useCallback</h1>
      <Link to="/">Home</Link><br /><br />
      <button onClick={increment}>Increment count: {count}
        </button>
      <List list={getEvenNumbers()} />
    </div>
  )
}

export default CallbackHook
```

In this example, **useCallback** memoizes the **getEvenNumbers** function, so it's not recreated on every render. This prevents the List component from re-rendering when the count increments, as the memoized function reference remains the same.

Now run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel. Then, click the link to go to the **useCallback** page to make sure everything is working.

useMemo

useMemo

Note: if at any time the terminal doesn't open, press the next or > button to go to the next page, then press the back button < and the terminal should refresh. Do not open a new terminal.

Consider an app that calculates the factorial of a number. The calculation is expensive, so you would want to memoize the result. For that, we can use the **useMemo** hook.

Click to open **MemoHook.js** and modify it as follows:

```
import { useState, useMemo } from 'react'
import { Link } from 'react-router-dom'

function factorial(n) {
  return n === 0 ? 1 : n * factorial(n - 1)
}

function MemoHook() {
  const [number, setNumber] = useState('1')

  const factorialResult = useMemo(() => {
    if (number || number === 0 || number === '0') {
      return factorial(number)
    } else {
      return
    }
  }, [number])

  const handleChange = (value) => {
    if (!value) {
      setNumber('')
    } else {
      setNumber(parseInt(value, 10))
    }
  }

  return (
    <div>
      <h1>useMemo</h1>
      <Link to="/">Home</Link> <br /><br />
      <input type="text" value={number} onChange={(e) =>
        handleChange(e.target.value)} />
      <p>Factorial: {factorialResult}</p>
    </div>
  )
}

export default MemoHook
```

In this example, **useMemo** memoizes the result of the factorial function. When the number input changes, the memoized value is recalculated only if the number dependency changes.

Let's run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Click the link to go to the **useMemo** page to make sure everything is working.

useTransition

useTransition

To demonstrate this optimization, let's say we have an input for the user to type something in, and a long list of data that will render as a result of what the user types. Normally, if using state as we will here, React would try to combine both state updates and render everything at once. However, if the list of data is long, this may create a very noticeable lag in the input, where whatever the user types take a long time to appear.

This can be perceived in a very negative light and the user may very likely leave your site. However, if the input updates immediately, while the list shows a loading component until the data is finally rendered, this creates a much more positive user experience.

We can achieve this using the **useTransition** hook made available in the React library as of version 18, which returns two things we can access using array destructuring: **isPending**, a boolean that keeps track of whether the low-priority state update is still pending or not, and the **startTransition** function which takes an anonymous function as an argument, in which we include all the code related to our low-priority state update.

Let's see what this looks like. Click to open our **TransitionHook** page, and modify it as follows:

```
import { useState, useTransition } from 'react'
import { Link } from 'react-router-dom'

function TransitionHook() {
  const [isPending, startTransition] = useTransition()
  const [inputValue, setInputValue] = useState('')
  const [list, setList] = useState([])

  const listLength = 3000

  function handleChange(e) {
    setInputValue(e.target.value)
    startTransition(() => {
      const largeList = []
      for (let i = 0; i < listLength; i++) {
        largeList.push(e.target.value)
      }
      setList(largeList)
    })
  }

  return (
    <div>
      <h1>useTransition</h1>
      <Link to="/">Home</Link> <br /><br />
      <input type="text" value={inputValue} onChange=
        {handleChange} /><br />
      {isPending ? (
        'Loading....'
      ) : (
        list.map((item, index) => {
          return <div key={index}>{item}</div>;
        })
      )}
    </div>
  )
}

export default TransitionHook
```

Using this optimization, when the user types in the input that state update should occur immediately, while the **setList** state update will render only afterwards. As mentioned before, this creates a much better experience for the user.

Let's run the dev server:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Click the link to go to the **useTransition** page to make sure everything is working.

Advanced State Management

Advanced State Management

As discussed earlier this week when we went over the basics of using state in React, there are some more advanced tools available to manage state in React applications which are the preferred option if state needs to be shared by multiple components - rather than resorting to prop-drilling.

While any in-depth look at these options would go beyond the scope of these lessons, at least in any way that does them justice, we will still go over what these options are and briefly explain how they work.

These advanced stated management options are:

- **useReducer**: A built-in React hook used for managing complex state
- **useContext**: Another built-in React hook, used to manage global state
- **MobX**: An external state management library
- **Redux (Toolkit)**: A comprehensive external state management library, capable of managing very complex state

Let's explore each of these in a little more detail.

useReducer

The **useReducer** hook helps manage state that has multiple values. **useReducer** works by accepting a reducer function and an initial state, then returning the current state and a **dispatch** function. The reducer function takes the current state and an action as arguments and returns a new state based on the action type.

For more information on **useReducer** and how to implement it, check out the documentation.

useContext

React Context is a built-in feature that allows you to create global state and share it across components without having to pass props manually. **useContext** is a hook that allows you to access the context value in a functional component. To use context, you create a context using **React.createContext**, provide a value using a **Provider**, and consume it in any child component using the **useContext** hook.

For more information on **useContext** and how to implement it, check out the documentation.

MobX

MobX is a state management library for React applications that emphasizes simplicity and scalability. It is based on reactive programming principles and uses observables, actions, and computed values to manage state. Observables represent the state of your application, while actions modify the state, and computed values derive new data from observables. MobX automatically tracks dependencies between observables and components and updates the UI when needed, optimizing re-renders.

For more information on **MobX** and how to use it in your React applications, check out the documentation.

Redux (Toolkit)

Redux is a very popular state management library for React that focuses on predictability, flexibility, and easy debugging. It is built on the principles of functional programming and utilizes a single, immutable store to manage application state. To update the state, you dispatch actions, which are processed by reducer functions that return a new state based on the action type. Redux promotes the separation of concerns, testability, and maintainability of your app.

Redux Toolkit, specifically, is a new and simpler way to implement Redux, and has in fact become the official and recommended implementation. It simplifies Redux development by providing utility functions and middleware that streamline common tasks, such as creating actions, reducers, and configuring the store. Redux Toolkit includes **configureStore** for easy store setup, **createSlice** for generating actions and reducers, and **createAsyncThunk** for handling asynchronous actions. It also comes with Immer built in, which is a library that allows you to work with immutable state in a more intuitive way.

For more information on **Redux** and **Redux Toolkit**, as well as how to use it in React, check out the documentation.

Knowledge Check

KNOWLEDGE CHECK

Conclusion

Conclusion & Takeaway

In this lesson, we have explored several advanced React topics and techniques to help you build scalable and high-performance applications. We have covered performance optimization techniques, such as React.memo, lazy loading, useCallback, useMemo, and the useTransition hook provided in React 18. In addition, we went over advanced solutions for state management, including useReducer, useContext, and external libraries like Redux Toolkit and MobX. By applying what you've learned in this lesson, along with all the other tools now in your React arsenal, you can create more maintainable, efficient, and professional frontend applications.

Other Frameworks - Angular

Goals

In this lesson, you will:

- Generate a new Angular project using the Angular CLI
- Learn some common Angular CLI commands
- Get a high-level overview of the structure of a basic Angular project

Introduction

Angular is a powerful and popular web application framework developed by Google. It helps developers build dynamic, responsive, and complex web applications using modern web development technologies like TypeScript and HTML. In this beginner lesson, we'll walk you through the process of generating a new Angular project, introduce you to common Angular CLI commands, and explain how different files in an Angular project connect together.

Generating a New Angular Project

Generating a New Angular Project

To generate a new Angular project, we'll need to have the Angular CLI installed. To install it, we can run the following command:

```
sudo npm install -g @angular/cli
```

In the above command, notice we are including **sudo** at the beginning. Since the virtual machine provided by Codio is a Linux-based environment, and since we are installing something "globally" (the **-g** flag indicates this) which requires admin or **root** level privileges, we need to include **sudo** at the beginning of the command otherwise we will get an error. This is also often the case if developing on a Mac, the operating system of which is similar in many respects to Linux.

Once the CLI finishes installing, we can generate a new project. To do this, run the following command:

```
ng new my-app
```

Here, **my-app** would be whatever we want to call our application. Go ahead and use this name for now.

As the CLI starts, you should see the following prompts (at least in the Codio environment; may differ slightly otherwise):

- **Would you like to enable autocompletion? This will set up your terminal so pressing TAB while typing Angular CLI commands will show possible options and autocomplete arguments. (Enabling autocompletion will modify configuration files in your home directory.)**
- **Would you like to share pseudonymous usage data about this project with the Angular Team at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more details and how to change this setting, see https://angular.io/analytics**
- **Would you like to add Angular routing?**
- **Which stylesheet format would you like to use?**

For the first three prompts go ahead and hit **n** on your keyboard and then press enter. For the stylesheet prompt, select **CSS** and press enter.

This should now generate all the starting boilerplate for a basic Angular project.

Common Angular CLI Commands

Common Angular CLI Commands

Here are some common Angular CLI commands we would use throughout the development process of our Angular app:

- **ng serve**: Starts a local development server and opens your app in the browser.
- **ng generate component [name]**: Creates a new Angular component.
- **ng generate service [name]**: Creates a new Angular service.
- **ng generate module [name]**: Creates a new Angular module.
- **ng build**: Builds your Angular app for production.

The command which is of particular interest to us right now is **ng serve**. Similar to how we had to modify the **vite** command when learning how to create a React project with Vite, we need to modify the corresponding command here for the preview of our Angular app to work in the Codio environment.

Specifically, we need to modify **start** in the scripts section of **package.json**. Click to open **package.json** in a new tab in the top left panel, and modify **start** as follows:

```
"start": "ng serve --host 0.0.0.0 --disable-host-check",
```

Basically, with **–host 0.0.0.0** we're telling **webpack-dev-server** (which Angular uses under the hood) to accept connections from any IP address (**localhost** is normally the only one allowed in development). In addition, we have to add the **–disable-host-check** flag to prevent an "Invalid host header" error.

It's important to note that in adding these two flags to the **ng serve** command we are bypassing a security feature of **webpack-dev-server**. In the Codio environment's virtual machine, we don't need to worry about this too much, but in any other context you should NEVER do this.

The reason is that it can expose your development server to security risks - in particular, the DNS rebinding attack in which an attacker can use a malicious website to send requests to your development server, bypassing CORS and potentially accessing sensitive information or executing malicious code.

Since we're in Codio, however, adding these two flags is fine and we can now go ahead and start up the dev server by running the **start** script:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

Next, we'll have a look at the code in some of the main files, and go over what those files are responsible for.

Understanding the Angular Project Structure

Understanding the Angular Project Structure

An Angular project consists of various files and folders. Here's a brief overview of the most important ones:

- **src/app**: The main application folder, containing your components, services, and modules.
- ****: The HTML template, including styles, for the main component of your app. Go ahead and click the link to open the file.
- ****: The TypeScript file containing the logic for the main component. Go ahead and click the link to open the file.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'my-first-app';
}
```

- ****: The main module that imports and declares all components, services, and other modules in your app. Go ahead and click the link to open the file.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- **src/styles.css**: The global stylesheet for your app (nothing in here by default, but we could add global styles if we wish).

Note that Angular uses TypeScript by default (the file extensions are **.ts** instead of **.js**). TypeScript is a statically-typed superset of JavaScript developed by Microsoft. It adds optional static typing to JavaScript, which can help catch errors early during development and improve code quality. TypeScript code is transpiled into JavaScript before being executed in the browser. Angular is built using TypeScript, and it is the recommended language for developing Angular applications due to its strong typing, object-oriented programming features, and compatibility with modern JavaScript features.

React also supports using TypeScript, it just doesn't use it by default. If you want to learn more about TypeScript, you can click here to check out the documentation.

That's pretty much it for our very high-level overview of Angular. If you wish to learn more about Angular and how to build applications with it, the best place to start (of course) is the documentation, which you can find here. You can also check out in-depth tutorials on YouTube to get you started.

Knowledge Check

KNOWLEDGE CHECK

Conclusion

Conclusion & Takeaway

In this beginner lesson, you've learned how to generate a new Angular project using the Angular CLI, explored common Angular CLI commands, and gained an understanding of the structure of an Angular project. With this foundational knowledge, you're now ready to dive deeper into Angular development and start creating your own web applications with this powerful framework. In the next lesson, we'll have a look at another popular frontend framework: Vue.

Other Frameworks: Vue

Goals

In this lesson, you will:

- Understand the benefits of Vue.js
- Learn how to generate a new Vue project
- Discover how the different files in a Vue project connect together

Introduction

Vue.js is another very popular progressive JavaScript framework for building user interfaces. In this brief lesson, we will cover the basics of generating a new Vue.js project, and as we did with Angular we'll explore the roles of the different files and how they connect together.

Understandig Vue

Understanding Vue.js and its benefits

Vue.js is a JavaScript framework designed to make it easy for developers to build web applications with a focus on the view layer. Some key benefits of Vue.js include its simplicity, flexibility, and excellent documentation. Vue.js is also known for its high performance and small size, making it a popular choice for modern web development projects.

Generating a new Vue.js project

To begin working with Vue.js, you'll need to run the following command in the terminal:

```
npm init vue@latest
```

You will be presented with several prompts. If you are asked if it's okay to install the **create-vue** package hit **y** and press enter. As for the rest of the prompts, go ahead and just give your project the name **my-app**, and select **No** for all the others.

Once the project scaffolding is complete, we have a couple more housekeeping item to take care of.

First, change directories into our new project:

```
cd my-app
```

Next, install the project's dependencies:

```
npm install
```

Since, at least right now, Vue apps are scaffolded with Vite by default, to be able to preview our app in Codio we need to make a slight modification to **package.json**. Open the file in a new tab in the top left panel by clicking (if needed, refresh the file tree: **Project** -> **Resync File Tree**).

Modify the **dev** script as follows:

```
"dev": "vite --host",
```

Having done that, go ahead and close the file and then run the dev server:

```
npm run dev
```

Once the dev server is running, click to open a preview in the bottom left panel. We should then see our Vue app up and running!

Understanding how the different files connect in a Vue.js project

After creating a new Vue.js project, you'll find several files and folders in the project directory. Let's briefly discuss the purpose of some key files:

- ****: This is the main HTML file that gets loaded when your app is run in the browser.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-
        scale=1.0">
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

- ****: This file is the entry point of your Vue.js application. It imports the Vue library and initializes the root Vue instance. Click on the name of the file above to open it (if needed, refresh the file tree: **Project -> Resync File Tree**).

```
import { createApp } from 'vue'
import App from './App.vue'

import './assets/main.css'

createApp(App).mount('#app')
```

- ****: This is the root component of your application. It contains the template, script, and style sections that define the layout, behavior, and appearance of your app. Click on the name of the file above to open it (if

needed, refresh the file tree: **Project -> Resync File Tree**).

```
<script setup>
import HelloWorld from './components/HelloWorld.vue'
import TheWelcome from './components/TheWelcome.vue'
</script>

<template>
  <header>
    <img alt="Vue logo" class="logo" src="./assets/logo.svg"
        width="125" height="125" />

    <div class="wrapper">
      <HelloWorld msg="You did it!" />
    </div>
  </header>

  <main>
    <TheWelcome />
  </main>
</template>

<style scoped>
header {
  line-height: 1.5;
}

.logo {
  display: block;
  margin: 0 auto 2rem;
}

@media (min-width: 1024px) {
  header {
    display: flex;
    place-items: center;
    padding-right: calc(var(--section-gap) / 2);
  }

  .logo {
    margin: 0 2rem 0 0;
  }

  header .wrapper {
    display: flex;
    place-items: flex-start;
    flex-wrap: wrap;
  }
```

```
}
</style>
```

- **src/components**: This folder contains your Vue components, which are reusable pieces of your application's UI.

And that's about it for our overview of Vue!

Conclusion

Conclusion & Takeaway

In this beginner lesson on Vue.js, we learned about the benefits of using the framework, how to generate a new project, and also explored the connections between different files in a Vue.js project. With this knowledge, you are ready to continue diving deeper into the Vue framework and start building your first Vue.js application!

Code-Along and Project

Goals

In this lesson, you will:

- Follow a code-along to create a simple to-do application in React with full CRUD functionality
- Work on a project to practice what you've learned

Introduction

To wrap things up for this week, we will first work together on creating a simple to-do app in React, step by step, then you will have time to work solo on further practicing the concepts covered either by adding to your personal website or by creating an entirely React project of your choosing.

To-Do App

Code-Along: To-do App

In this tutorial, we will create a simple to-do application in React with full CRUD functionality. CRUD stands for: **C**reate, **R**ead, **U**pdate and **D**elete. We will use the **useState** hook to manage the application's state. The final application will allow you to create, read, update, and delete to-do items.

You should be in the folder `code-along`. Start by running:

```
npm install
```

Open the **App.js** file using the following link:

(if needed, refresh the file tree: **Project -> Resync File Tree**)

Inside **App.js**, you should see the following:

```
import { useState } from 'react'
import './App.css'

function App() {
  // Your app code will go here
}

export default App
```

Now, to begin with, let's add the state for the to-do items, a new to-do input, and whether the to-do items are being updated:

```
function App() {
  const [todos, setTodos] = useState([])
  const [newTodo, setNewTodo] = useState('')
  const [updating, setUpdating] = useState(false)

  // Your app code will go here
}
```

Next, create a function to handle adding a new to-do item (this can be either inside or outside our **App** component, but it's more organized to include it *inside*):

```
function addTodo() {
  if (newTodo.trim()) {
    setTodos([...todos, { id: Date.now(), text: newTodo,
        completed: false, updating: false }])
    setNewTodo('')
  }
}
```

Now add a function to handle updating a to-do item's text:

```
function updateTodo(id, newText) {
  setTodos(todos.map(todo => (todo.id === id ? { ...todo, text:
        newText } : todo)))
}
```

We'll also need a function to toggle a to-do item's completion status:

```
function toggleTodo(id) {
  setTodos(todos.map(todo => (todo.id === id ? { ...todo,
        completed: !todo.completed } : todo)));
}
```

Next, create a function to delete a to-do item:

```
function deleteTodo(id) {
  setTodos(todos.filter(todo => todo.id !== id));
}
```

Lastly, let's add a function to toggle the **updating** state:

```
function toggleUpdating(id) {
  setUpdating(!updating)
  setTodos(todos.map(todo => (todo.id === id ? { ...todo,
        updating: !todo.updating } : todo)))
}
```

Note that we are toggling not only the **updating** state, but also the individual to-do property **updating**. This is so that later when we conditionally render an input field for the to-do item we wish to update (and a div with the to-do text otherwise), we can make sure this happens ONLY for the to-do we're updating and not for all the to-do items in the list!

Now that we've written all our functions to handle different to-do operations, let's finish by adding the JSX code in the return statement in order to render our to-do app:

```
  return (
    <div className="App">
      <h1>React To-Do App</h1>
      <input
        type="text"
        placeholder="Add a new to-do"
        value={newTodo}
        onChange={e => setNewTodo(e.target.value)}
        onKeyDown={e => (e.key === 'Enter' ? addTodo() : null)}
      />
      <ul>
        {todos.map(todo => (
          <li className="todo-item"  key={todo.id} >
            <div className="input-group">
            <input
              type="checkbox"
              disabled={todo.updating && updating ? true :
      false}
              checked={todo.completed}
              onChange={() => toggleTodo(todo.id)}
            />
            {todo.updating && updating ? (
              <input
                type="text"
                value={todo.text}
                onChange={e => updateTodo(todo.id,
      e.target.value)}
                onKeyDown={e => (e.key === 'Enter' ?
      toggleUpdating(todo.id) : null)}
              />
            ) : (
              <div
                className='todo-item__text'
                style={{ textDecoration: todo.completed ? 'line-
      through' : 'none' }}
              >
                {todo.text}
              </div>
            )
            }
            </div>
            <div className="btn-group">
              <button
                disabled={(todo.updating && updating) ||
      todo.completed ? true : false}
                onClick={() => toggleUpdating(todo.id)}
              >
                Edit
              </button>
              <button
```

```
                onClick={() => deleteTodo(todo.id)}
              >
                Delete
              </button>
            </div>
          </li>
        ))}
      </ul>
    </div>
  )
```

On the next page, we'll take a look at our final **App.js** code and preview our application.

Final App.js

Our final **App.js** should look as follows:

```
import { useState } from 'react'
import './App.css'

function App() {
  const [todos, setTodos] = useState([])
  const [newTodo, setNewTodo] = useState('')
  const [updating, setUpdating] = useState(false)

  function addTodo() {
    if (newTodo.trim()) {
      setTodos([...todos, { id: Date.now(), text: newTodo,
        completed: false, updating: false }])
      setNewTodo('')
    }
  }

  function updateTodo(id, newText) {
    setTodos(todos.map(todo => (todo.id === id ? { ...todo,
        text: newText } : todo)))
  }

  function toggleTodo(id) {
    setTodos(todos.map(todo => (todo.id === id ? { ...todo,
        completed: !todo.completed } : todo)));
  }

  function deleteTodo(id) {
    setTodos(todos.filter(todo => todo.id !== id));
  }

  function toggleUpdating(id) {
    setUpdating(!updating)
    setTodos(todos.map(todo => (todo.id === id ? { ...todo,
        updating: !todo.updating } : todo)))
  }

  return (
    <div className="App">
      <h1>React To-Do App</h1>
      <input
        type="text"
        placeholder="Add a new to-do"
```

```
        value={newTodo}
        onChange={e => setNewTodo(e.target.value)}
        onKeyDown={e => (e.key === 'Enter' ? addTodo() : null)}
      />
      <ul>
        {todos.map(todo => (
          <li className="todo-item"  key={todo.id} >
            <div className="input-group">
            <input
              type="checkbox"
              disabled={todo.updating && updating ? true :
      false}
              checked={todo.completed}
              onChange={() => toggleTodo(todo.id)}
            />
            {todo.updating && updating ? (
              <input
                type="text"
                value={todo.text}
                onChange={e => updateTodo(todo.id,
      e.target.value)}
                onKeyDown={e => (e.key === 'Enter' ?
      toggleUpdating(todo.id) : null)}
              />
            ) : (
              <div
                className='todo-item__text'
                style={{ textDecoration: todo.completed ? 'line-
      through' : 'none' }}
              >
                {todo.text}
              </div>
            )
            }
            </div>
            <div className="btn-group">
              <button
                disabled={(todo.updating && updating) ||
      todo.completed ? true : false}
                onClick={() => toggleUpdating(todo.id)}
              >
                Edit
              </button>
              <button
                onClick={() => deleteTodo(todo.id)}
              >
                Delete
              </button>
            </div>
          </li>
```

```
        ))}
      </ul>
    </div>
  )
}

export default App
```

Now let's run the dev server to check if everything is working:

```
npm start
```

Once the dev server is running, click to open a preview in the bottom left panel.

That's it! You now have a simple to-do application with full CRUD functionality using React. This application allows you to create, read, update, and delete to-do items. The state is managed using the **useState** hook, keeping track of the to-do items, the new to-do input, and whether or not we are updating to-do items.

Note that this application does not persist data, so any changes made to the to-do list will be lost when the page is reloaded. You can add data persistence by using a backend API, local storage, or any other storage solution.

Conclusion

Project: Add to Personal Website

As with other modules, we've once again covered a *lot* this week. Being exposed to frontend frameworks for the first time constitutes a significant shift of mentality in terms of the way we build web applications, and can definitely be difficult and confusing at first. If you struggled with this module, don't beat yourself up! Keep practicing, practicing, practicing, and you'll be a master of React soon enough. For now, take a moment and truly appreciate how far you've come since the beginning of this course!

Speaking of practice... use this project time now to try and implement what you learned throughout this week.

One suggestion would be to try and refactor your personal website in React. Alternatively, you could try searching Google for some React project ideas and attempt to build one of them. Or, maybe you've got your own idea for a React project already and could try building that.

As always, whatever project you choose to work on, happy coding!

Submission steps:

Before you click on "Mark as Completed":

You need to do one of the following: either upload all of your files to Codio or deploy a GitHub Page for this project.

Also, if you mark this project as complete but any of the boxes are blank, your TA will be unable to grade your project.

Codio upload:

- Make sure all of your project code has been uploaded to Codio.
  - If you did not write your code in Codio, you will need to import all of the required files into your workspace file tree.
  - You can do this by going to File => Upload Files, and either manually importing each file, or dragging and dropping your project folder.
  - Please refer to this video if you are unsure of what to do.

GitHub Pages:

- If you would rather upload your project to GitHub, please make sure to have the project deployed as a GitHub page so we can thoroughly test it.

- If you are unsure of how to do this, please follow these instructions.
- It is important to understand you will need to make a separate repository for every project. You **cannot** deploy multiple pages from the same repository, even with different branches.

Also, no matter whether you uploaded your files from your computer or not, **make sure to thoroughly test your code**! This only takes a few minutes, but will prevent the amount of resubmissions because you missed something.