React

MODULE 1

React Components and where they Live

# React Componenets and Where They Live

## Why React?

React is used everywhere in the tech world. React does not use inheritance, rather it uses composition. It is very flexible and we can easily integrate third party libraries with it. React is also great if we have very complex UIs and we want to reuse code across many different pages. Compared to react, angular is a fully fledged solution for an app or website and we do not have to integrate with third party libraries. Whereas react is just a front-end library and we have to learn routing, server client communication.

## React Overview

SPA (Single page application) is a single page website where some of the page content changes depending upon the user input as opposed to the traditional web sites where the whole page needs to be reloaded when a change occurs. React is the best way to create SPAs.

### React has component-based architecture:

Component based architecture is building software based on reusable components of code. Components are the building blocks of react. Each component is composed of well-defined functionality that can be inserted into an application without requiring modification of other components. It means that the components are **independent** and **reusable.** As the components are self-contained, they have their own HTML, CSS and javascript

We can think of a react website as a collection of components.

### React uses virtual DOM:

**DOM** stands for Document Object Model. It is a programming interface for web documents. When a web page is loaded into a web browser, the browser creates a model of the page's structure, content, and properties. This model, known as the DOM, allows JavaScript to interact with the webpage dynamically.

The **Virtual DOM** is a concept used in front-end web development frameworks like React.js. It's an abstraction of the actual DOM (Document Object Model) and serves as a lightweight copy or representation of the DOM.

#### Here's how it works:

1. Representation of UI: The Virtual DOM is essentially a JavaScript representation of the actual DOM. It is a tree structure where each node represents a DOM element, such as a div, span, or input.
2. Efficient Updates: When the state of a React component changes due to user interaction or other events, React creates a new Virtual DOM representation of the UI.
3. Diffing Algorithm: React then compares this new Virtual DOM with the previous Virtual DOM snapshot (from before the state change) using a process called "reconciliation" or "diffing." This process efficiently identifies the differences (or "diffs") between the old and new Virtual DOM.
4. Minimal DOM Manipulation: After identifying the differences, React calculates the most efficient way to update the actual DOM to match the new Virtual DOM. Instead of directly updating the entire DOM, React only makes the necessary changes, minimizing the number of DOM manipulations.
5. Batch Updates: React often batches multiple updates together and applies them in a single pass to further optimize performance.
6. The Virtual DOM allows React to achieve high performance by reducing the number of costly DOM operations. By batching updates and only manipulating the parts of the DOM that have changed, React can provide a smoother user experience, especially in complex and dynamic web applications.

## Introduction to functional Components:

Functions in javascript are reusable codes that can take an input and return an output after doing some logical implementation. React components work the same way as the javascript functions. React provides:

### Two types of components:

#### Functional Components

Acts just like javascript functions. In the default react application (index.js in root folder), only one component is rendered by the name of **App** located inside App.js file in the **src** folder. Every react app must have at least one component and it’s called the root component. This component is loaded using the **import** statement.

#### Class components

Acts like javascript Classes.

### Rendering components:

The syntax to render a component is very similar to a self-closing HTML tag.

<App/>

function App (){

return (

<div className=’App’>

<h1> Hello <h1/>

</div>

)

}

export default App

### JSX

React is scripted using a special syntax called JSX (Javascript XML). In the above example the statements inside the return statement, is JSX which looks very similar to HTML. Just like a Javascript won’t work just by declaration, react components too won’t render until they are used.

### Steps to create react Components:

**Create a file (Heading.js):** Notice how the first letter is capitalized. It is important because reacts treats capitalized and non-capitalized component names. The lowercase components are treated as regular HTML elements. Capitalizing helps react to distinguish JSX elements from HTML elements.

Inside the file create a function with first letter capitalized:

function Heading () {

title =’This is some heading’

return (

<h1>{title}</h1>

)

}

### Transpiling

Interpreting a programming language and translating it to a specific target language. The above will render into an html with element <h1>. This process is called transpiling.

## Creating React Components

In VS code, create a folder and go to the folder. Once in the folder open terminal in VS code and run the following command.

npm init react-app

once the installation completes run the following command to run the app:

npm start

The above command will run the react app in localhost:3000.

After installation, a folder structure will appear with many files. We are interested in the ‘src’ folder since we will create our components in this folder.

In App.js, remove the unnecessary code and rewrite it the following way:

function Header () {

return <h1>Helllo world </h1>

}

function App () {

return <Header/>

}

export default App

## Transpiling JSX

### Introduction

Components are a nice way to build websites in React because they allow you to build more modular apps. However, how do you build components using React, JSX, and JavaScript?

A browser cannot understand JSX syntax. This means that making a browser understand React code requires a lot of supporting technologies. An example of such a technology is a transpiler. A transpiler takes a piece of code and transforms it into some other code. To understand why this is done, here is an example of an ES6 variable declaration:

const PI = 3.14

This is perfectly valid ES6 syntax. However, if you were using a very old computer, that computer will have an old browser. Perhaps that browser was built before ES6 came out in 2015. This means that the JavaScript engine that is built into your old computer's browser is likely to be an ES5 JavaScript engine. In ES5, the only way to declare a variable is the following:

var pi = 3.14

What this means is that for this old browser to understand the ES6 code, the only way to do it is by transpiling it. If you feel like it, you can try transpiling ES6 to ES5 code yourself, using the es6console website. Now, let’s move the focus to another example of transpiling. Let's say that you want to use a brand new, most modern ECMAScript syntax in an app. The only problem is that this new syntax is currently not supported by any browser; even an up-to-date browser.

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

Likely the most popular site that shows off how this works is Babel. As the heading of the website reads, "Babel is a JavaScript Compiler". This finally brings you to the point of this discussion about transpiling JavaScript code. What Babel does is this: it allows you to transpile JSX code (which cannot be understood by a browser) into plain JavaScript code (which can be understood by a browser).

This is where React and JSX come in.

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

To demonstrate how this works, let’s use the Heading component from the previous lesson. Add the JSX code into the online Babel repl. Repl stands for "read-eval-print loop" and it accepts code you write, evaluates it, and produces some result. In the specific case of the online Babel repl, that result is some transpiled code. Here's a more detailed explanation. If you've visited the above-linked URL, you'll find a web page that has two panels. On the left, there's source JSX code:

function Heading(props) {

return <h1>{props.title}</h1>

}

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

function Heading(props) {

return /\*#\_\_PURE\_\_\*/React.createElement("h1", null, props.title);

}

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

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

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

"h1"

null

props.title

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

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

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

Again, using the Babel repl, the output of the tranpilation is the following code. Ensure that you select the classic runtime for React in the left sidebar.

/\*#\_\_PURE\_\_\*/

React.createElement(Heading, {

title: "This is the heading text!"

});

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

This brings us to an interesting question: What is the minimum code that a component must have to be able to show something on the screen when rendered? You can see the answer below:

function Example() {

return <div>An element</div>

}

export default Example

## The React Project Structure

When we install react, we have three folders:

**node\_modules**

It’s a repository for all the modules in our react app. This module is automatically added when we install react app.

**public**

Contains the assets that will be displayed to the users in our app i.e., images, logos etc. This folder has an index.html file. A react app gets injected into a specific element inside the body of this file. Based on changes in the react app, they are injected into the same div of the index.html file.

**src**

Contains all the necessary components required to ensure that a react app functions.

## React Components and Modules

Advantage of Component-Based Architecture: The app is split into individual, self-contained components for building powerful, reusable UIs.

**Challenge**: Integrating and managing numerous components within the app.

### Components and Modules:

**Components**: Small, self-contained units of code in React, building blocks for UIs.

**Modules**: Standalone units of code in JavaScript, can be reused and replaced, enhancing modular programming.

**Relationship:** Components can be considered as single parts, while modules encapsulate a series of components.

### Import and Export Statements:

**Export Statement:** Used to make a module available to another module.

**Two types:** default exports and named exports.

* Default Export: When the function name is the same as the file name.
* Named Export: When the function name differs from the file name.

**Import Statement:** Used to bring components into the app.

Syntax: import componentName from './path/to/component';

### Folder Structure:

**Components Folder:** One common approach is to place all components in a folder named "components."

**Organizational Benefits:** Structuring projects by grouping similar files together for better organization.

**Example Scenario:** Building a payment page for an e-commerce app with header, main, and sidebar components.

# Component Use and Styling

## Principles of Components: props

**Functional Components Recap:** Reusable blocks of code in React, akin to JavaScript functions.

**JavaScript Flexibility:** Functions can use parameters to receive values.

**React Equivalent:** Utilizing properties (props) to pass data between React components.

### Props and JavaScript Objects:

**JavaScript Objects:** Special variables storing related data.

**Object Properties:** Name-value pairs.

**Accessing Properties:** Dot notation.

**Props in React:** Similar concept, allowing data transfer between components.

### Using Props in React:

**Props as Arguments:** JSX syntax, analogous to HTML attributes.

function App(){

return (

<Header data=’How are you’/>

)

}

**Accessing Props:** Inside the component, using the props object.

function Header(props){

return (

<h1>{props.data}<h1/>

)

}

**Props Data Types**: Versatility from simple types (strings, integers) to complex types (functions, arrays, objects).

**Dynamic Components:** props enhance flexibility for dynamic data flow.

### Component Communication:

**Parent-Child Relationship:** Components sending props data are called parents and components receiving props data are called children.

**Data Flow Direction:** One-directional, from parent to child.

**Multiple Child Components:** Parent can send the same data to multiple child components.

**Limitation**: No communication from child to parent using props.

### Limitations of Props:

**One-Directional Flow:** Child components cannot communicate back to parent components using props.

**Pure Functions in React:** Functions returning the same outputs for the same argument values.

**Pure Functions Requirement:** React components using props must not modify their own props.

## Introducing JSX

### Expressiveness of JSX:

* JSX is special due to its expressiveness.
* Developers can express React components using a syntax almost identical to HTML or XML.
* JSX allows writing HTML directly inside JavaScript, making it powerful and expressive.

### JSX Rules in the return Statement:

* return statement is where JSX expressions are used.
* If returning HTML code over multiple lines, wrap it in parentheses.
* HTML code must be wrapped in a top-level <div> or any other wrapping element, or use fragments (<> …data </>).
* Fragments are like HTML open and close tags without the tag name.

### Differences in JSX due to being a JavaScript file:

* Use ‘className’ instead of ‘class’ for CSS classes (due to class being a reserved keyword).
* ‘className’ is written in Camel Case.

### Practical Considerations:

* While JSX has some differences, in practical terms, it resembles regular HTML.
* JSX allows mixing JavaScript, HTML, and CSS, providing a familiar approach for web developers.

## Props and children:

There is also a special prop called **props.children**, which is automatically passed to every component.

### Introduction to props.children:

* props.children is a special prop automatically passed to every React component.
* It allows a component to render whatever content is provided between its opening and closing tags.

### Real-Life Analogy: Apples, Pears, and Bags:

* Imagine having components like Apples and Pears representing different content.
* A new component, Bag, is desired, acting as a container for both apples and pears.

### Using props.children in the Bag Component:

* The Bag component is created with a specific styling.
* The Bag component is used to wrap elements in the props.children property.

function Bag(props) {

const bag = {

padding: "20px",

border: "1px solid gray",

background: "#fff",

margin: "20px 0"

}

return (

<div style={bag}>

{props.children}

</div>

)

}

export default Bag

### Passing Children Syntax:

* The unusual syntax:

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

can also be used to send children to Bag.

* It is equivalent to the more familiar syntax of:

<Bag>

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

</Bag>

<Bag>

<Pears friend="Peter" />

</Bag>

### Nested Components:

The following example demonstrates how a Trunk component can wrap a Bag component, which, in turn, wraps Apples and Pears.

<Trunk>

<Bag>

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

<Pears friend="Peter" />

</Bag>

</Trunk>

### Understanding props.children:

* props.children allows components like Bag to be versatile and accept different content, just like a real bag accommodating various groceries.

## Styling JSX elements:

We’ve observed that JSX is incredibly versatile, and can accept a combination of JavaScript, HTML and CSS.

There are various ways to style JSX elements. Probably the simplest way to do this is using the **link** HTML element in the head of the **index.html** file in which your React app will mount.

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

Another way to add CSS styles to components is using inline styles. The syntax of inline styles in JSX is a bit custom:

<h1 style={{color:"tomato", fontSize:"40px", fontWeight:"bold"}}>

{props.heading}

</h1>

As explained previously, this means that whatever code you add inside these opening and closing curly braces is to be parsed as regular JavaScript. So, there's nothing special about this object, except for the fact that you’ve inlined it and placed it inside a pair of curly braces. Additionally, since it's just JavaScript, those CSS properties that would be hyphenated in plain CSS, such as, for example, font-size:40px, become camelCased, and the value is a string, making it look like this: fontSize:"40px".

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

**My Inference:** The components should have their own styles, they should not be thrown to them from parent.

## JSX syntax and the arrow function

### Components as Function Expressions

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

### Function Expressions

const Nav = function(props) {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

The component is the same for most part. The only thing that's changed is that you’re now using an anonymous (nameless) function, and assigning this anonymous function declaration to a variable declared using the const keyword, and the name Nav. The rest of the code is identical. You can also take this concept a step further, using arrow functions.

const Nav = (props) => {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

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

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

To reiterate, take the smallest possible anonymous ES5 function:

const example = function() {}

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

const example = () => {}

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

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

const Nav = props => {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

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

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

const Nav = () => {

return (

<ul>

<li>Home</li>

</ul>

)

}

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

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

const Nav = () => <ul><li>Home</li></ul>

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

### Using Arrow Functions in Other Situations

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

For example: [10, 20, 30].forEach(item => item \* 10)

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

## Embedded JSX expressions

### Key Points about JSX

* **Syntax extension for JavaScript:** It allows writing HTML-like structures within React components.
* **Flexibility in React development**: Offers a more intuitive way to create and structure UI elements.
* **Automatic React element creation**: JSX code is transformed into React elements during compilation.

### Key Feature: Embedded Expressions

* JSX allows embedding JavaScript values (variables, functions, etc.) within JSX code.
* **Syntax**: Enclose expressions within curly braces {}.

For Example:

1. const name = "John Doe";

const element = <p>Hello, {name}!</p>;

1. function formatName(firstName, lastName) {

return `${firstName} ${lastName}`;

}

const fullName = formatName("John", "Doe");

const element = <h1>Welcome, {fullName}!</h1>;

**Remember:**

JSX automatically handles quotes for attribute values, so you don't need to add them manually.

## Ternary operators and functions in JSX

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

### Ternary operators: A different way of writing an if...else conditional

A ternary operator in JavaScript uses two distinct characters: the first one is the question mark, that is, the ? character. To the left of the ? character, you put a condition that you'd like to check for.

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

name == 'Bob' ?

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

name == 'Bob' ? :

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

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

### Using function calls in JSX

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

You can use function invocation inside JSX to return a random number:

function Example2() {

return (

<div className="heading">

<h1>Here's a random number from 0 to 10:

{ Math.floor(Math.random() \* 10) + 1 }

</h1>

</div>

);

};

## Expressions as props

export default function App() {

return (

<div className="App">

<Example

**toggleBoolean={!bool}**

**math={(10 + 20) / 3}**

**str={str1 + ' another ' + 'string'}**

/>

</div>

);

};

## Embedding in attributes

**Application**: Rendering images in a React app.

**Method**: Embedding a JS expression in the src attribute of an <img> tag.

Steps:

* Import image: Import the image file (e.g., avatar.png) into your component using import.
* Create a function (optional): Create a separate function for the image component (e.g., logo) for better organization.
* Define JSX expression:
* Create a JSX element for the image (<img>).
* Use the src attribute and embed a JS expression to reference the imported image using curly braces { }.
* Example: <img src={avatarPng} alt="Avatar Image" />.
* Render the component:
* In your main component's return statement, render the function or JSX element directly.

**Additional notes:**

This approach is ideal for simple cases. For complex scenarios, consider importing images as standalone components.

# Dynamic Events and how to handle them:

## Types of Events

### What Are Events?

Events are occurrences that happen in the system you’re programming. When an event occurs, the system produces a signal, and your code can react to it.

#### Examples of events include:

* User clicks a button.
* Cursor hovers over an element.
* Keyboard key is pressed.
* Web page finishes loading.
* Video starts or ends.
* Error occurs.

Events are attached to specific elements in the browser, such as buttons, forms, or the entire window.

### Event Handlers (Listeners)

* To react to an event, you attach an event handler (or listener) to it.
* An event handler is a block of JavaScript code that runs when the event fires.
* For instance, when a button is clicked, its click event triggers the attached event handler.

### Example: Handling a Click Event

Suppose we have an HTML button:

<button>Change color</button>

We can add a JavaScript event handler to the button’s “click” event:

const btn = document.querySelector("button");

function random(number) {

return Math.floor(Math.random() \* (number + 1));

}

btn.addEventListener("click", () => {

const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;

document.body.style.backgroundColor = rndCol;

});

In this example, clicking the button changes the page background color to a random shade.

### Event Types

There are various event types, including:

* Mouse events (click, hover, etc.)
* Keyboard events (key presses)
* Window events (resize, close)
* Form events (submit)
* Media events (video playback)
* Custom events (created by developers)

### React and Events

* In React, event handling is slightly different due to its component-based architecture.
* You’ll encounter events like onClick, onChange, and onSubmit when building React applications.

## Eventful issues

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

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

### Event Errors

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

Sometimes it's just about writing the wrong syntax. Other times it's about not thinking of all the possible scenarios and all the possible ways that things can go wrong in your code. Regardless of what causes them, errors are a part of everyday life for a developer.

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

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

(5).toUpperCase()

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

Uncaught TypeError: 5.toUpperCase is not a function

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

try {

(5).toUpperCase();

}

catch(e) {

console.log(`Oops, you can't uppercase a number.

Trying to do it resulted in the following`, e);

}

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

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

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

function NumBillboard(props) {

return (

<>

<h1>{prop.num}</h1>

</>

)

}

export default NumBillboard;

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

In this specific example, the error would be:

ReferenceError

prop is not defined

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

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

## Common event Handling

We will learn how to handle events in react.

### Understanding Events and JSX Attributes:

* **JSX Attributes:** We define event handlers in React using special JSX attributes attached to specific elements. These attributes start with "on" followed by the camelCase version of the event name (e.g., onClick, onMouseOver, onSubmit).

### Setting Up Event Handlers:

1. **Define an Event Handler Function:** Inside your component, create a function specifically designed to handle the chosen event. Give it a descriptive name (e.g., handleClick, handleMouseOver).
2. **Assign the Function to the JSX Attribute:** In your JSX, utilize the appropriate event attribute and assign the defined function name to it. For example:

<button onClick={handleClick}>Click Me</button>

1. **Access Event Information:** The event handler function receives an event object as an argument. This object contains details about the event, such as the clicked element, mouse position, and keyboard key pressed. You can access these details to modify your component's behavior based on the event.

## Syntax For Handlers

### Key Differences from HTML:

* No addEventListener: React avoids direct DOM manipulation. Use JSX event attributes instead.
* CamelCase Event Names: Event attributes start with *‘on’* followed by *camelCase* event names (e.g., onClick, onMouseOver).
* Function References, Not Invocations: Pass a reference to the handler function, not its invocation.
* JSX Expression Delimiters: Enclose handler names in curly braces {}.
* Function Props: Pass handler functions as props to child components for communication.

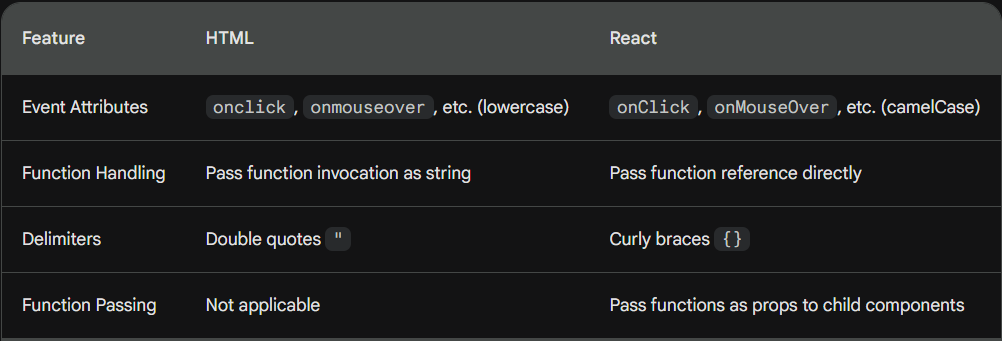
### Example: HTML vs. React Click Handler:

**HTML:**

<button onclick="clickHandler()">Click Me</button>

**React:**

<button onClick={clickHandler}>Click Me</button>



## Event handling and embedded expressions

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

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

*You may find this reading useful as a reference sheet.*

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

### Handling events using inline anonymous ES5 functions

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

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

An inline anonymous ES5 function event handler

</button>

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

### Handling events using inline anonymous ES6 functions (arrow functions)

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

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

An inline anonymous ES6 function event handler

</button>

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

### Handling events using separate function declarations

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

function App() {

function thirdExample() {

console.log('third example');

};

return (

<div className="thirdExample">

<button onClick={thirdExample}>

using a separate function declaration

</button>

</div>

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

### Handling events using separate function expressions

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

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

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

function App() {

const fourthExample = () => console.log('fourth example');

return (

<div className="fourthExample">

<button onClick={fourthExample}>

using a separate function expression

</button>

</div>

);

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

## User events

We will explore how we can toggle the value of a Boolean state variable using *user triggered* events and how to handle multiple events on a single element.

### Toggling a Boolean State Variable with Events:

function ModeToggler(){

return (

<div>

{darkModeOn ? darkMode: lightMode}

<button onClick={handleClick} > Toggle Mode </button>

</div>

)

}

1. In the above function, create a Boolean State Variable as shown in the following code:

const darkModeOn=false;

const darkMode=<h1>Dark Mode is On<h1/>

const lightMode=<h1>Light Mode is On<h1/>

1. Define the Event Handler Function:

Create a function handleClick to toggle the state:

function handleClick() {

darkModeOn=!darkModeOn;

if (darModeOn){

console.log(‘Dark Mode is On’)

}

else{

console.log(‘Light Mode is On’)

}

}

When we press the *button*, then the header element will not change, rather we will see console.log output. We will learn the reason in the state section

### Handling Multiple Events on a Single Element:

Separate event handler names with a space: <button onClick={handleClick} onMouseMove={handleMouseMove}>...</button>

### Key Insights:

* **Conditional Rendering:** Use conditional expressions (e.g., {darkModeOn ? <h1>Dark mode is on</h1> : <h1>Light mode is on</h1>}) to render different content based on state.
* **State Updates Aren't Automatically Reflected**: React needs to re-render components to reflect state changes in the UI.
* **Data Flow in React:** Understanding how data flows between components is crucial for managing state and updates effectively.

# Data and Events

## Parent and Child Data Flow

### Key Concepts:

* **Unidirectional Data Flow:** Data flows from parent to child components, not vice versa.
* **Components as Building Blocks:** React applications are built by nesting components, creating hierarchies.
* **DRY Principle:** Avoid code duplication by managing data in one place (parent) and passing it down.

### Example:

**Create Components:**

* Promo: Renders content from PromoHeading.
* PromoHeading: Displays discount text (h1 and h2).

**Single Source of Truth:**

* Create a data object in Promo to store discount messages.

**Pass Data with Props:**

* Pass data.heading and data.callToAction as props to PromoHeading.
* Access props in PromoHeading using props.heading and props.callToAction.

**Extend to Other Components:**

* Use the same approach for Sidebar and Footer components.

### Benefits:

* **Maintainability**: Easier to update data in one place.
* **Flexibility**: Child components can dynamically adapt to different data.
* **Scalability**: Complex UIs can be built with clear data flow.

## Data flow in React

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

### Parent-child data flow

In React, data flow is a one-way street. *The data flow starts at the root and can flow to multiple levels of nesting, from the root component (parent component) to the child component, then the grandchild component, and further down the hierarchy.*

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

*Props are immutable (cannot be changed).*

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

1. Comprehend the logic of React apps more quickly and
2. Simplify the data flow.

### Here’s a practical example of this:

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

**Parent component:**

function Dog() {

return (

<Puppy name="Max" bowlShape="square" bowlStatus="full" />

);

};

**Child component:**

function Puppy(props) {

return (

<div>

{props.name} has <Bowl bowlShape="square" bowlStatus="full" />

</div>

);

};

**Grandchild component:**

function Bowl(props) {

return (

<span>

{props.bowlShape}-shaped bowl, and it's currently {props.bowlStatus}

</span>

);

};

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

## Children and Data

### Props vs. State:

**Props:**

* Passed down from parent components.
* Read-only in child components.
* Represent external data the child works with.

**State:**

* Controlled by individual components.
* Can be mutated within the component.
* Represents internal data that affects the component's behavior.

**Example:**

* Parent component (App) manages state (current date/time).
* Child component receives this data as a prop (message).
* Child component displays the received data (date/time) in an h1 element.

### Benefits of One-Way Data Flow:

* **Maintainability**: Easier to track data changes and reason about application behavior.
* **Predictability**: Child components react predictably to parent data changes.
* **Testability**: Easier to test components in isolation.

### Additional Notes:

* Stateless components only use props, while stateful components manage their own state.
* Complex applications might require state management libraries for larger data flows.

## What are hooks

In React, **hooks** are functions that allow developers to use state and other React features in functional components, which were traditionally stateless. Hooks provide a way to reuse stateful logic across different components without having to use class components or higher-order components.

### Benefits of Hooks:

* **Improved Readability:** Simplify complex component logic.
* **Code Reuse:** Create reusable functions for common stateful logic.
* **Flexibility:** Use state and lifecycle features without class components.

### useState Hook:

* Most common hook.
* Used to manage state within a component.
* Its built directly into React.
* Following is how you can use it.

1. First import the {useReact} from react, so that its available for use.
2. Declare state variable within a component. We can provide any name to the state variable and the setState function.

const [showMenu, setShowMenu] =useState(false) //Array Destructuring

1. The useState(false) returns an array with two items, the state variable and the set state function.
2. Calling the useState() hook does two things.
3. Creates a state variable with an initial value that represents the current state i.e., showMenu
4. Creates a function to set the state variable’s value i.e., setShowMenu

The function setShowMenu is used to update the value of showMenu by passing the Boolean value to it.

1. The useState() hook should be called at the top level of your component.
2. We can use the useState() hook to keep track of any type of data.

### Example:

import React, { useState } from 'react';

function MyComponent() {

const [showMenu, setShowMenu] = useState(false);

const handleClick = () => {

setShowMenu(!showMenu);

};

return (

<div>

<button onClick={handleClick}>Toggle Menu</button>

{showMenu && <div>Menu content</div>}

</div>

);

}

### Additional Notes:

* Hooks can be used for various purposes, not just state management.
* Custom hooks allow for even more flexibility and code reuse.
* Understanding hooks is essential for building modern React applications.

## Using hooks

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

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

Let's discuss an example:

import { useState } from 'react';

export default function InputComponent() {

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

  function handleChange(e) {

    setText(e.target.value);

  }

  return (

    <>

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

      <p>You typed: {inputText}</p>

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

        Reset

      </button>

    </>

  );

}

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

An input text field

1. Any text that has been entered into the field
2. A Reset button to set the field back to its default state
3. As the user starts typing within the text field, the current text that was typed is also displayed.



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

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

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

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

function handleChange(e) {

    setText(e.target.value);

};

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

Isn’t this neat?

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

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

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

Following is example of state variable with an object assigned to it:

const[form, setForm] =useState({

firstName:'Luke',

lastName:'Jones',

email:'lukeJones@sculpture.com',

});

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

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

### The useRef hook

useRef is a React Hook that lets you reference a value that’s not needed for rendering.

#### Usage

1. Referencing a value with a ref

* Call useRef at the top level of your component to declare one or more refs.

import { useRef } from 'react';

function Stopwatch() {

const intervalRef = useRef(0);

// ...

* useRef returns a ref object with a single current property initially set to the initial value you provided.
* On the next renders, useRef will return the same object. You can change its current property to store information and read it later. This might remind you of state, but there is an important difference.
* Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn’t affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its current property:

function handleStartClick() {

const intervalId = setInterval(() => {

// ...

}, 1000);

intervalRef.current = intervalId;

}

* Later, you can call that interval ID from the ref so that you can clear that interval.

function handleStopClick() {

const intervalId = intervalRef.current;

clearInterval(intervalId);

}

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

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

function TextInputWithFocusButton() {

  const inputEl = useRef(null);

  const onButtonClick = () => {

    // `current` points to the mounted text input element

    inputEl.current.focus();

  };

  return (

    <>

      <input ref={inputEl} type="text" />

      <button onClick={onButtonClick}>Focus the input</button>

    </>

  );

}

Using the ref attribute on the **input** element, we can access it using inputEl.current value and invoke the focus() method on it, thereby focusing the input field.

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

## What is state (A Component's Memory)

### What is State?

Think of state as a component’s internal data that determines the current behaviour of the component. It is often used to store data that effects the behavior of the component. Used to represent dynamic elements that change based on user interactions or other events.

State ensures components stay in sync and update automatically when State changes. For example:

If one component updates its state, all other components that depend on that state will automatically update too. This means that a component sends its state to its children through props.

If the child components have their own grandchild components, then the child components might have their own state that they send as props to those grandchild components.

In react, state is kept in variables and to change state, we have to update the variable.

When a component is created, it gets an initial state. The state is used to initialize the component’s props.

### Stateful vs. Stateless Components:

* **Stateful**: Manage their own State using useState hook.
* **Stateless**: Don't have State, only rely on props for data.

### Example:

* Stateless component renders fixed text "Hello".
* Stateful component renders text from its State variable ("Greet"), initially set to "Hello".

### Benefits of State:

* Creates dynamic and interactive UIs.
* Simplifies complex data management.
* Enables components to react to changes effectively.

### Additional Notes:

* State variables are updated using setter functions (e.g., setGreet).
* Components can be both stateful and stateless depending on their needs.
* Understanding State is crucial for building well-behaved React applications.

## Observing state

### What is state in React?

* Internal data of a component that affects its behavior.
* Used to represent dynamic elements that change based on user interactions or other events.
* Allows components to re-render and update the UI when state changes.

### Why use state?

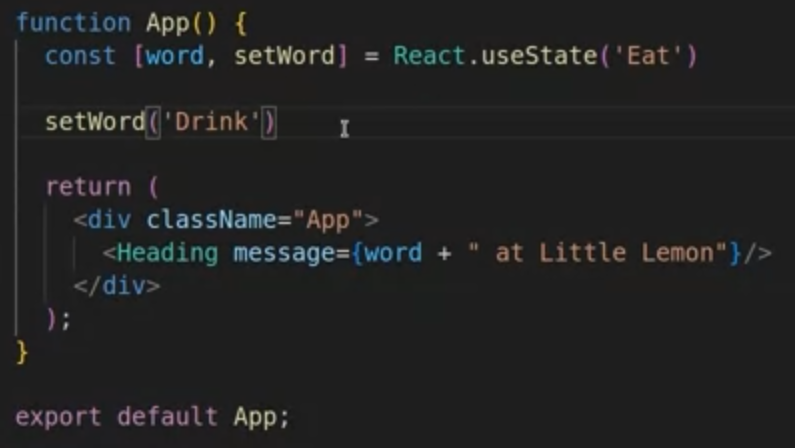
* Manage data that is likely to change within a component.
* Create dynamic and interactive user interfaces.
* Simplify complex data management by keeping track of changes within components.

### useState Hook:

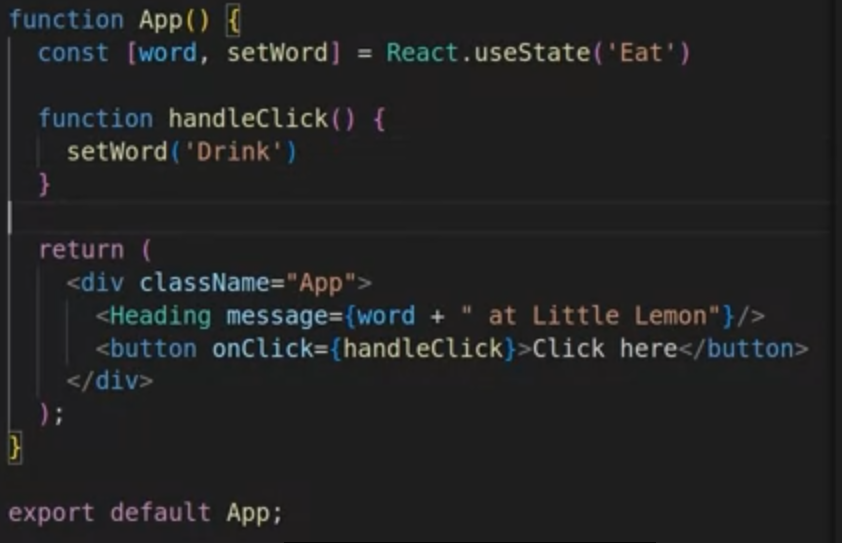
* A core React hook for managing state.
* Takes two arguments: initial state value and a setter function to update it.
* Example: const [count, setCount] = useState(0);

### Updating State:

* Cannot directly modify state variables.
* Use setter functions provided by useState (e.g., setCount).



The code above will give error because we cannot use the state setting variable (setWord) from our state directly. Instead of updating it directly, we can update it based on a click event or any other event or some other approaches which we will learn later.



### Key Takeaway:

* State is a powerful tool for managing dynamic data in React applications. By understanding how to use state effectively, you can create responsive and interactive user interfaces.

### Additional Notes:

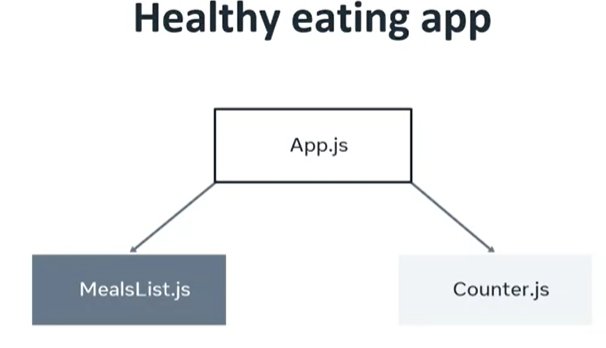
* Stateless components don't use state, relying solely on props.
* Complex applications might require additional state management libraries.
* Understanding state management is crucial for building scalable and maintainable React apps.

## Managing state

Managing state in React applications can become complex as the application grows. Developers need a way to manage this state.

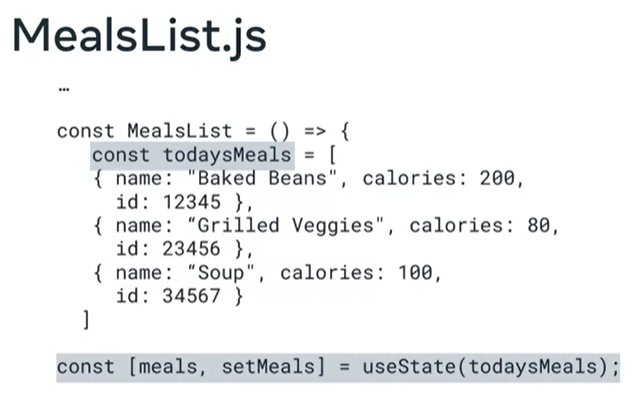
### Example Scenerio

Consider a small React app that promotes a healthier lifestyle by allowing the user to monitor their food intake. This app tracks a daily meal plan and updates to show how many meals are left to be consumed for the day. The app consists of three components: a root component called App.js, and two child components, MealsList and Counter.



The App component imports the MealsList and Counter components and renders them on the screen.



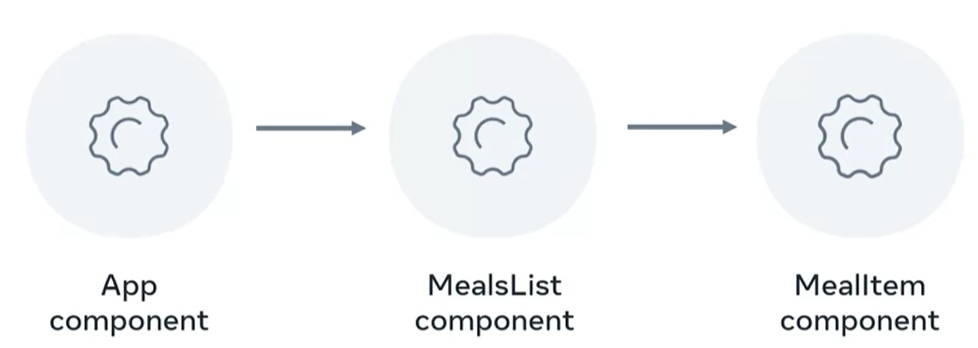
The MealsList component uses the useState hook to list the day's meals, which are stored in an array. The array elements are saved inside the todaysMeals variable, and the meals state variable is initialized to hold this value. The Counter component tracks the number of meals that a user is allowed to eat today. 

A problem arises when the Counter component needs to get state information from the MealsList component, but both components are rendered by the App component. The MealsList and the Counter components are siblings and not in a parent-child relationship.

### Lifting up state

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as “lifting state up”, and it’s one of the most common things you will do writing React code.

The practice known as **lifting state up** can be used, which means moving the state from MealsList up to the App components. Then, state can be passed through props using the MealsList component as a bridge to the MealItem component.



However, this approach relies on two practices, **lifting state up** and **prop drilling**.

### Prop drilling:

It is a term used to describe the process of passing state through props in several layers of components, from the parent to the child, to the grandchild, and so on. If the source data changes, those changes have to be transferred across the entire prop drilled structure. This complicates things because state updates go to all the child components and their siblings, which then need to be updated to reflect this state change.

As the App grows, a large amount of state may be kept in the App component. Most of that state is about components such as the MealItem component.

### Global state

The above problem can be viewed through the viewpoint of global state. Whenever states might have to be used in various places in the app, that's a global state issue.

### React’s context API

An elegant solution for this issue is React's Context API. The context API cuts out the middleman. There's no need for prop drilling and lifting state up. Instead, the component that needs the data simply gets it from the context API. This is achieved by extracting the state into a separate file that holds the state in context, then any file that needs it simply imports it and uses it. This should provide a good understanding of managing states in React.



## prop Drilling

It is like taking a bus to reach your destination. The problem with the bus is that it visits every stop before reaching your location. The context api is like teleportation.

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

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

Here is the code for the app:

function Main(props) {

  return <Header msg={props.msg} />;

};

function Header(props) {

  return (

    <div style={{ border: "10px solid whitesmoke" }}>

      <h1>Header here</h1>

      <Wrapper msg={props.msg} />

    </div>

  );

};

function Wrapper(props) {

  return (

    <div style={{ border: "10px solid lightgray" }}>

      <h2>Wrapper here</h2>

      <Button msg={props.msg} />

    </div>

  );

};

function Button(props) {

  return (

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

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

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

    </div>

  );

};

function App() {

  return (

    <Main

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

    />

  );

};

export default App;

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

## React State Management

### Context API for State Management:

The Context API in React provides a way to share data between components without having to explicitly pass props through every level of the component tree. It's especially useful for passing down global data, such as themes, user authentication, or language preferences.

#### 1. Setting Up Context:

First, you need to create a context using React.createContext():

const MyContext = React.createContext(defaultValue);

Here, defaultValue is optional and serves as a fallback value when a component does not find a matching Provider above it.

#### 2. Provider Component:

The Provider component is used to wrap the part of the component tree where you want to make the context available.

<MyContext.Provider value={/\* some value \*/}>

{/\* Components inside here can access the context \*/}

</MyContext.Provider>

#### 3. Consuming Context:

To consume the context within a component, you use the useContext hook:

const value = useContext(MyContext);

This allows you to access the value provided by the nearest matching Provider in the component tree.

### Using Context API in an App

The App component includes import statements for MealsProvider and MealsList. The MealsProvider provides context state data and gives it to all the components it wraps inside the App component. The MealsProvider component holds all the states, which is organized with the help of the context API.

### UseReducer Hook

<https://react.dev/learn/managing-state#extracting-state-logic-into-a-reducer>

It is superpower to useState. While the useState hook starts with an initial state, the useReducer also gets a **reducer** function in addition to the initial state.

## Stateful vs. stateless

### Stateful Components:

Stateful components hold internal state data, which can change over time based on user interactions or other factors.

Usage: They are utilized when a component needs to maintain its own state to function properly.

#### Characteristics:

* Can store and update state using hooks like useState or useReducer.
* Typically, changes in state lead to re-rendering of the component.
* Often used for managing complex UI behaviors, forms, or interactive elements.

#### Example:

A form component that needs to track user input and update its state accordingly.

### Stateless Components:

Stateless components do not hold internal state; instead, they receive all necessary data via props.

Usage: They are suitable when a component does not need to maintain its own state and can rely entirely on the data passed down to it via props.

#### Characteristics:

* Do not manage state internally; all data is received through props.
* Renders based solely on the props it receives.
* Often used for presentational components or reusable UI elements.

#### Example:

A button component that receives its label and onClick function as props.

### Choosing Between Stateful and Stateless Components:

#### Stateful Component:

* Use when the component needs to manage its own state, such as handling user input or maintaining UI state.
* Useful for containing logic that requires data manipulation or business logic.

#### Stateless Component:

* Use when the component simply needs to render UI based on the data it receives.
* Ideal for presentational components or UI elements that do not require internal state management.

### React Component Structure:

* A common pattern in React is to have a hierarchy of components, with a stateful component at the top managing the overall state of the application.
* Stateful components pass down data and event handlers to stateless child components via props.
* Child components receive data through props and render UI based on that data.

# Linking and Routing

## Basic Types of Navigation

### Modern Website Navigation:

* Website navigation refers to the components that allow users to browse through different pages or links on a website.

#### Common navigation components include:

* Horizontal navigation bar (navbar)
* Vertical navigation menu (sidebar navigation)
* Menu hiding behind a button (burger menu)
* Mega menu (drop-down navigation menu)
* Footer navigation menu (displayed as columns containing links)
* These navigation patterns can be used simultaneously on the same page and may include multiple approaches within a single component.

### Implementation in React:

* In React applications, navigation may visually resemble traditional HTML and CSS navigation.
* However, React navigation works differently in the code due to the single-page application (SPA) architecture.
* In an SPA, the entire application is loaded inside a single div, and navigation is controlled by React based on changes to the virtual DOM.
* Unlike traditional HTML navigation with hyperlinks and separate HTML files, React navigation updates the existing view or loads a new view dynamically, giving the impression of visiting different URLs.
* React Router is a library commonly used to manage navigation in React applications, providing functionality similar to traditional multi-page websites.

### Analogy to an Elevator:

* We can use analogy of an elevator to explain React navigation.
* In a traditional elevator (analogous to HTML navigation), pressing a button takes you to a different floor.
* In a React elevator (analogous to React navigation), pressing a button doesn't physically move the elevator but dynamically changes the contents of the current floor.

### Using React Router:

* React does not provide the routing technalogy directly from the ‘react’ library. Instead we have to use the react-router library.
* React Router allows for declarative routing, enabling developers to define routes and navigate between different components within the application.

### Single-Page Apps

* You’re using many Single Page Applications every day. Think of your favorite social network, or online email provider, or the map application you use to find local businesses. Their excellent user experiences are driven by Single Page Applications.
* A Single Page Application allows the user to interact with the website without downloading entire new webpages. Instead, it rewrites the current webpage as the user interacts with it. The outcome is that the application will feel faster and more responsive to the user.

### How Does a Single-Page App Work?

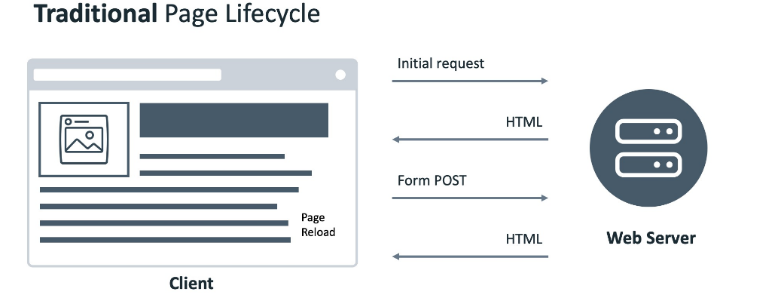
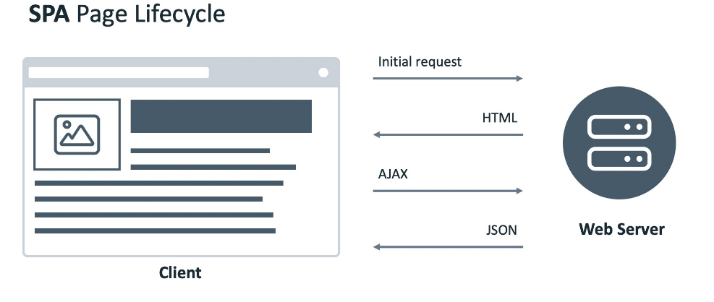
* When the user navigates to the web application in the browser, the Web Server will return the necessary resources to run the application. There are two approaches to serving code and resources in Single Page Applications.

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

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

### An Example of a Single-Page App

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

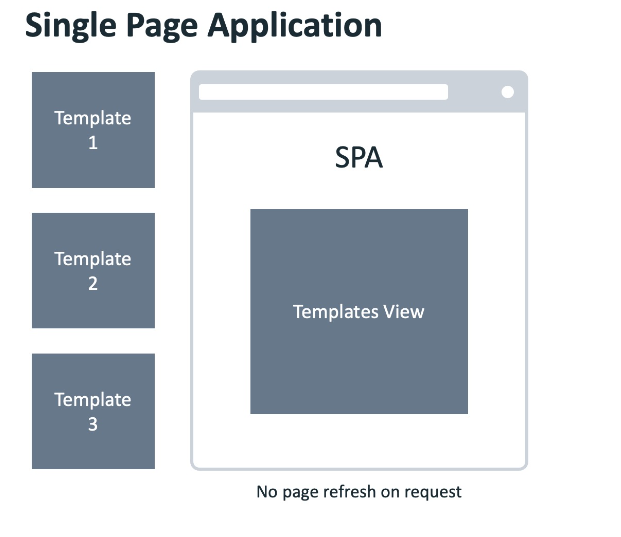
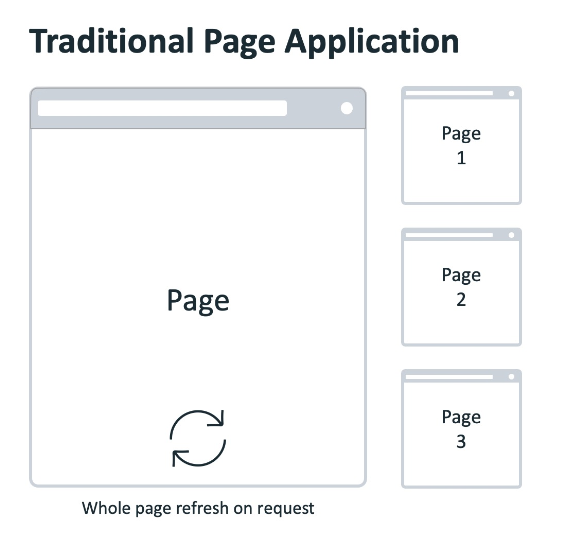
* In a traditional website, when the button is clicked, the browser will send a POST request to the web server. The web server will return a new web page containing the button and movie name, and the web browser renders the new page.
* In a Single Page Application, when the button is clicked, the browser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

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

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

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

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



### Anchor Tag Element in Single-Page Aplications

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

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

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

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

## The Navbar

### Installation of React Router:

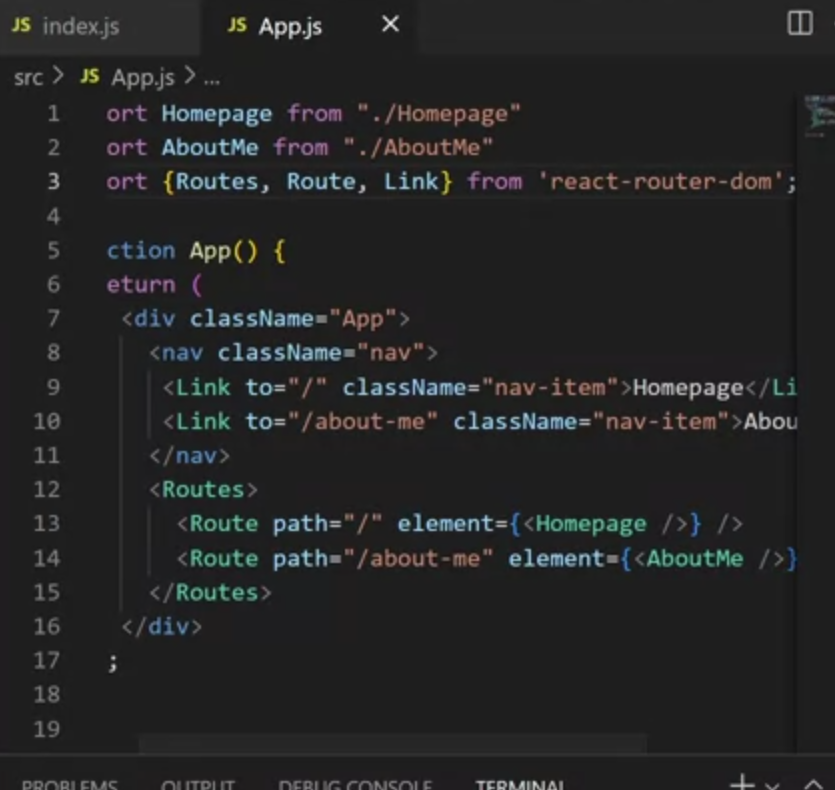
* Install the React Router library using npm: npm i react-router-dom@6.
* Confirm the installation by checking the package.json file for the entry of react-router-dom with the installed version number.

### Configuring React-router:

1. Import BrowserRouter from react-router-dom.
2. Wrap the root component (App) with <BrowserRouter> in the index.js file to set up routing within the application.

### Implementing Routes:

* Import Routes and Route: In app.js, import these components from react-router-dom.
* Create Routes: Replace child components with Route elements:
* For Homepage, use path="/" element={<Homepage />}.
* For Aboutme, use path="/about-me" element={<Aboutme />}.
* Nest Routes: Wrap all Route components within a parent <Routes> element.



### Using React Router Links:

* Replace anchor tags (<a>) with Link components from react-router-dom.
* Link components facilitate navigation within the application without causing a full page refresh, enhancing user experience.

### Testing Navigation:

* Verify navigation functionality by clicking on the navbar links to ensure that the correct components are loaded based on the URL path.

## Conditional Rendering

React updates content without full page reloads by rendering different components based on data.

Conditional Rendering: Developers control what to render based on specific conditions.

Ternary Operator is a shorthand way to write if-else statements that we can use to implement conditional rendering.

function CurrentMessage(props) {

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

        return <Workdays />

    }

    return <Weekends />

}

### Example Scenario:

A productivity app displays different messages for weekdays and weekends.

#### Implementation Steps:

1. Create CurrentMessage Component:

* Gets the day of the week (0-6, 0 being Sunday).
* Creates child components Workdays and Weekends for messages.

1. Use Ternary Operator in CurrentMessage:

* Condition: day >= 1 && day <= 5 (checks for weekdays).
* If true: Renders Workdays component.
* If false: Renders Weekends component.

### Simplified Example:

* Component IsItSummerYet with boolean variable summer set to true.
* Ternary operator checks summer value.
* If true: Returns "Let's go to the beach!".
* If false: Returns nothing.

### Benefits of Ternary Operator:

* Concise way to write simple if-else statements.
* Common pattern in React code.

### Remember:

* While powerful, ternary operators can become complex for beginners.
* Consider alternative approaches for intricate logic.

## Conditional Components

function LogInOutButton(props) {

const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn) {

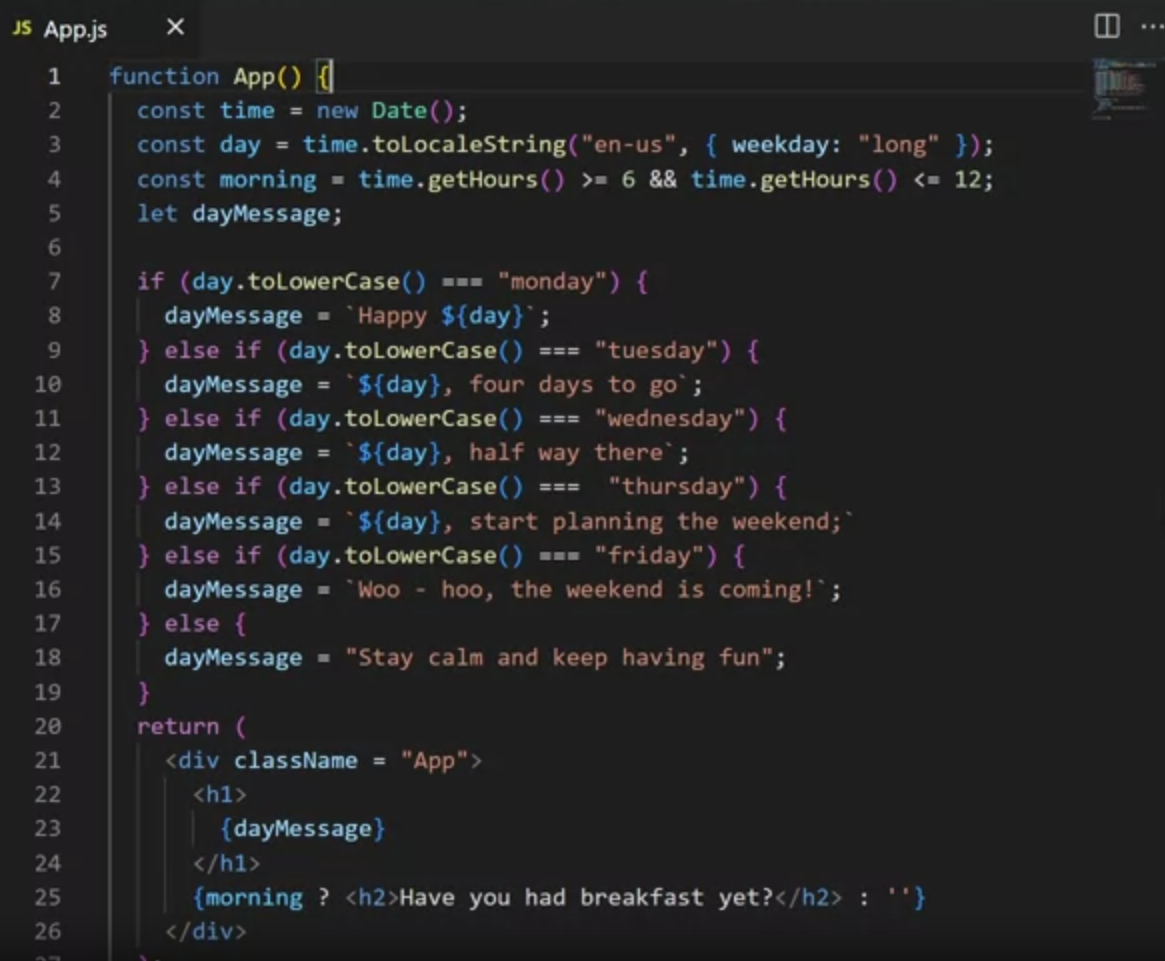
    return <LogoutButton />;

  } else {

  return <LoginButton />;

}

## Single View Conditional Updates



# Using Assets in React

## What is an asset and where does it live?

### What are assets?

In React, assets refer to any files your application needs at runtime to function correctly. These can include:

* Images (.jpg, .png, .gif)
* Style sheets (.css)
* Fonts (.ttf, .woff)
* Media files (.mp3, .mp4)
* Any file crucial for your app's functionality

### Why are assets important?

* Essential for displaying various elements within your app (e.g., images, custom fonts).
* Enhance user experience by providing visual and auditory elements.

### Where to store assets:

#### assets folder:

* Place assets required by your React components (e.g., images used within specific components).
* Recommended practice for most assets.

#### public folder:

* Contains files directly accessible by the browser without React processing (e.g., favicon).

### General rule:

* If your app can function without the file during compilation (refers to the process of transforming your React code written in JavaScript and JSX into a format that can be understood and executed by a web browser.), store it in the public folder.
* If a component depends on the file, store it in the assets folder.

### Importing assets:

#### Import statement:

* Use import followed by a descriptive name for your asset (e.g., import cat from './assets/cat.jpg').
* Specify the path to the asset from the component's location.

#### require keyword (alternative):

* Use directly within JSX expression for the src attribute of an element (e.g., <img src={require('./assets/cat.jpg')}/>).
* No separate import statement needed.

## Bundling Assets

We will learn about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. We will also learn about the trade-offs inherent in using asset-heavy apps.

The app’s files will likely be bundled when working with a React app. Bundling is a process that takes all the imported files in an app and joins them into a single file, referred to as a bundle. Several tools can perform this bundling. Since, in this course, you have used the create-react-app to build various React apps, you will focus on **webpack**. This is because webpack is the built-in tool for the create-react-app.

Let’s start by explaining what webpack is and why you need it.

Simply put, webpack is a module bundler.

Practically, this means that it will take various kinds of files, such as SVG and image files, CSS and SCSS files, JavaScript files, and TypeScript files, and it will bundle them together so that a browser can understand that bundle and work with it.

### Why is this important?

We can probably build wibsites without webpack since our project's structure might be straightforward: we may have a single CSS library, such as Bootstrap, loaded from a CDN (content delivery network). We might also have a single JavaScript file in your static HTML document. If that is all there is to it, you do not need to use webpack in such a scenario.

However, modern web development can get complex.

Here is an example of the first few lines of code in a single file of a React application:

import React from 'react';

import '@atlaskit/css-reset';

import styled from 'styled-components';

import './index.css';

import { ThemeProvider } from './contexts/theme';

import { DragDropContext } from 'react-beautiful-dnd';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Nav from './components/Nav';

import data from './data';

import Loading from './components/Loading';

The imports here are from fictional libraries and resources because the specific libraries are not necessary. All these different imports can be of various file types: .js, .svg, .css, and so on.

In turn, all the imported files might have their own imported files, and even those might have their imports.

This means that depending on other files, all of these files can create a dependency graph. The order in which all these files are loading is essential. That dependency graph can get so complex that it becomes almost impossible for a human to structure a complex project and bundle all those dependencies properly.

This is the reason you need tools like webpack.

So, webpack builds a dependency graph and bundles modules into one or more files that a browser can consume.

While it is doing that, it also does the following:

* It converts modern JS code - which can only be understood by modern browsers - into older versions of JavaScript so that older browsers can understand your code. This process is known as transpiling. For example, you can transpile ES7 code to ES5 code using webpack.
* It optimizes your code to load as quickly as possible when a user visits your web pages.
* It can process your SCSS code into the regular CSS, which browsers can understand.
* It can build source maps of the bundle's building blocks
* It can produce various kinds of files based on rules and templates. This includes HTML files, among others.

Another significant characteristic of webpack is that it helps developers create modern web apps.

It helps you achieve this using two modes: **production mode or development mode.**

In **development mode**, webpack bundles your files and optimizes your bundles for updates - so that any updates to any of the files in your locally developed app are quickly re-bundled. It also builds source maps so you can inspect the original file included in the bundled code.

In **production mode**, webpack bundles your files so that they are optimized for speed. This means the files are minified and organized to take up the least amount of memory. So, they are optimized for speed because these bundles are fast to download when a user visits the website online.

Once all the source files of your app have been bundled into a single bundle file, then that single bundle file gets served to a visitor browsing the live version of your app online, and the entire app’s contents get served at once.

This works great for smaller apps, but if you have a more extensive app, this approach is likely to affect your site’s speed. The longer it takes for a web app to load, the more likely the visitor will leave and move on to another unrelated website. There are several ways to tackle this issue of a large bundle.

One such approach is **code-splitting**, a practice where a module bundler like webpack splits the single bundle file into multiple bundles, which are then loaded on an as-needed basis. With the help of code-splitting, you can lazy load only the parts that the visitor to the app needs to have at any given time. This approach significantly reduces the download times and allows React-powered apps to get much better speeds.

There are other ways to tackle these problems.

An example of a viable alternative is SSR (Server-side rendering).

With SSR, React components are rendered to HTML on the server, and the visitor downloads the finished HTML code. An alternative to SSR is client-side rendering, which downloads the index.html file and then lets React inject its own code into a dedicated HTML element (the root element in create-react-app). In this course, you’ve only worked with client-side rendering.

Sometimes, you can combine client-side rendering and server-side rendering. This approach results in what’s referred to as **isomorphic apps**.

In this reading, you learned about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You also learned about the trade-offs inherent in the use of asset-heavy apps.

## Using Embeded assets

We can embed an image in three different ways:

1. Using import statement
2. Using require
3. Using a URL

As shown on right side:

## Media packages

In this reading, you’ll learn how to install the react-media npm package.

You can find this package on the npmjs.org website at the following URL:

[react-media](https://www.npmjs.com/package/react-player)

To install this package you'll need to use the following command in the terminal:

npm install react-player

There are a few ways that you can import and use the installed package. For example, to get the entire package's functionality, use the following import:

import ReactPlayer from "react-player";

If you are, for example, only planning to use videos from a site like YouTube, to reduce bundle size, you can use the following import:

import ReactPlayer from "react-player/youtube";

Here’s an example of using the react-player packaged in a small React app:

import React from "react";

import ReactPlayer from "react-player/youtube";

const App = () => {

return (

<div>

<MyVideo />

</div>

);

};

const MyVideo = () => {

return (

<ReactPlayer url='https://www.youtube.com/watch?v=ysz5S6PUM-U' />

);

};

export default App;

In this reading, you learned how to install and use the react-player npm package.

## Create an audio\video Component

### Introduction:

We will learn installing and utilizing the react-player package within React applications. react-player facilitates the integration of media players into React-based projects, offering customizable settings for enhanced user experience.

### Installation Process:

1. Installation Command: Execute npm install react-player in the terminal to install the React Player module.
2. Importing Module: Import the React Player module into the desired component using import ReactPlayer from 'react-player'.

### Integration and Configuration:

Adding React Player Component: Incorporate the React Player component within the application's render method to enable media playback.

#### Customizing Settings:

* Disable Automatic Playback: Set playing={false} to prevent automatic playback upon page load.
* Adjust Starting Volume: Define volume={0.5} to specify the initial volume level, with 0.5 representing 50% of the maximum volume.
* Explore Additional Settings: Refer to the React Player documentation for a comprehensive list of configurable attributes.

### Setting Video URL:

Assign the URL of the desired video to the url attribute of the React Player component, ensuring proper linking to the video file hosted online.