**React**

* Opensource js library for building user interfaces
* Not a framework
* Focus on UI
* Reusability of code
* React is declarative-Tell react what you want and React will build the actual UI
* React will handle efficiently updating and rendering of the components
* DOM updates are handled gracefully in React.
* React can be integrated anywhere.

**Component based Architecture**

* Header
* Sidenav
* Main content
* Footer

**To create react app**

npx create-react-app my-app

cd my-app

npm start

or

npm install create-react-app -g

create-react-app<project\_name>

**Folder structure**

Package.json- contains dependencies and scripts required for the project

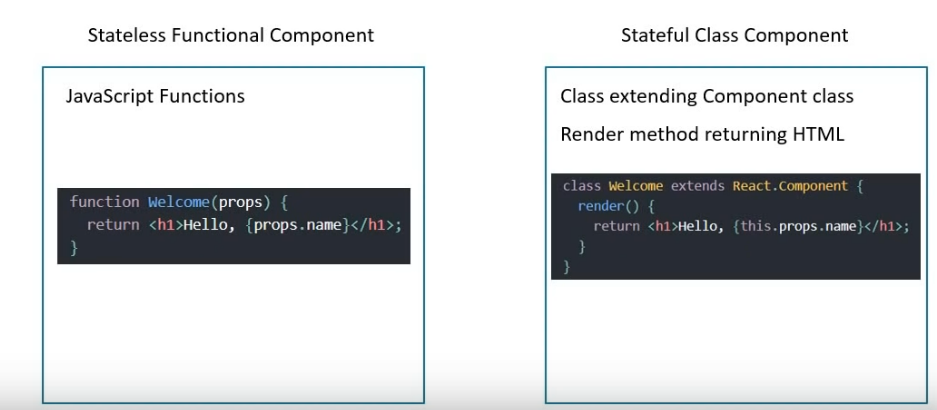
Node-module- a folder in which all the dependencies are installed

Public- index.html

Src- index.js here root component (App) and DOM element (div id=root)

is specified

**Components**

****

Components describe a part of the user interface

They are re-usable and can be nested inside other components

**Two types:**

Stateless functional components

Stateful Class Components

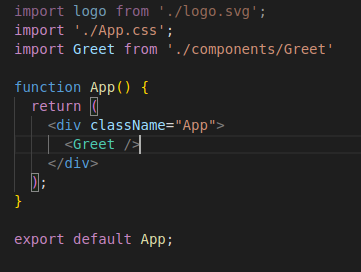
**Functional components**

Javascript function accepts the input as properties and returns the HTML that describes the UI

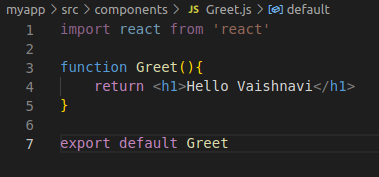
Javascript function

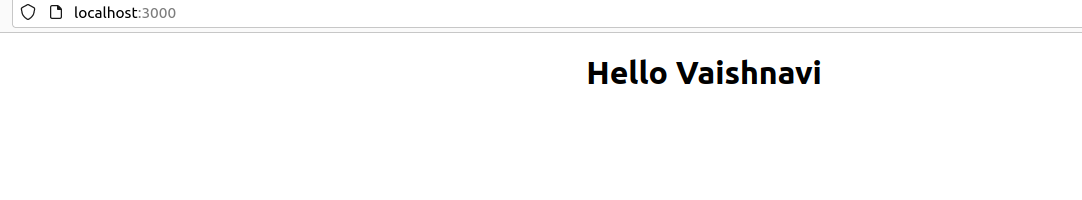
Properties HTML(JSX)

App.js



Greet.js





**Class Component**

import React, {Component} from 'react'

class Greet extends Component{

render(){

return <h1>Class Component</h1>

}

}

export default Greet

1.Import React and component class from react

2.Define the class

3.The class to become a react component,

* It should extend component class from react
* The class has to implement a render method which will return null or some html

|  |  |
| --- | --- |
| **Functional component** | **Class component** |
| Simple functions | More feature rich |
| Use func components as much as possible | Maintain their own private data-state |
| Absence of this keyword | Complex UI logic |
| Solution without using state- components with private states, maintenance and debugging app is difficult i.e class component | Provide lifecycle hooks |
| They tend to be w/o any complicated logic and mainly responsible for UI  Stateless/Dumb/Professional component | Stateful/smart/container component |

JSX

JSX- Javascript XML

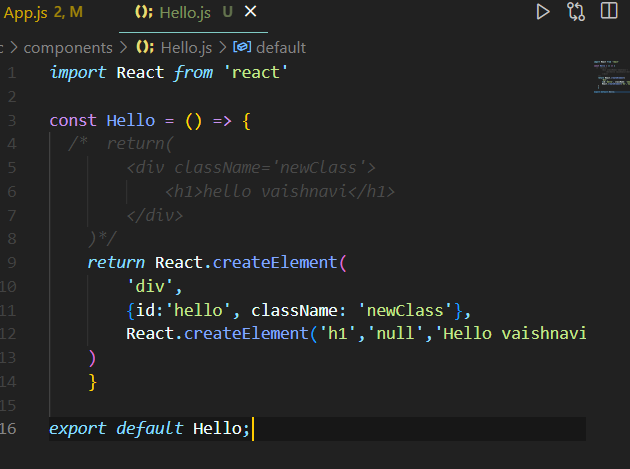
JSX allows us to write HTML in React. JSX tags have a tag name, attributes and children

JSX makes it easier to write and add HTML in React

JSX makes react code simpler and elegant

JSX ultimately transpiles to pure javascript which is understood by the browser

JSX and createElement method



JSX differences

Class -> className

For -> htmlFor

camelCase property naming convention

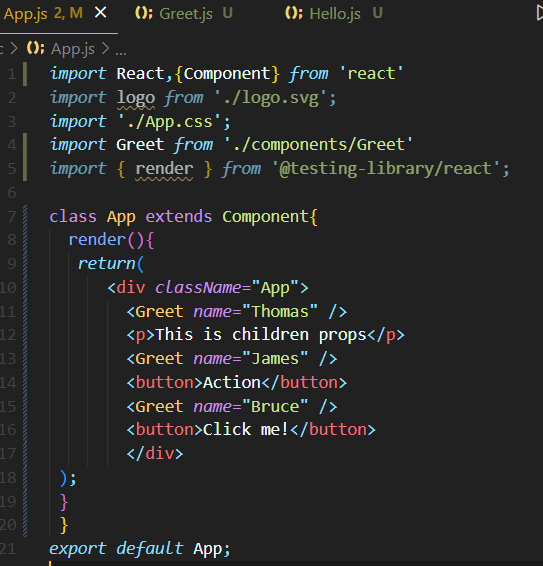
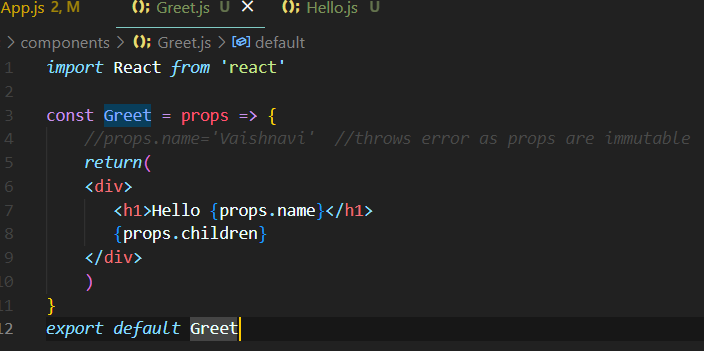
onclick -> onClick

tabindex -> tabIndex

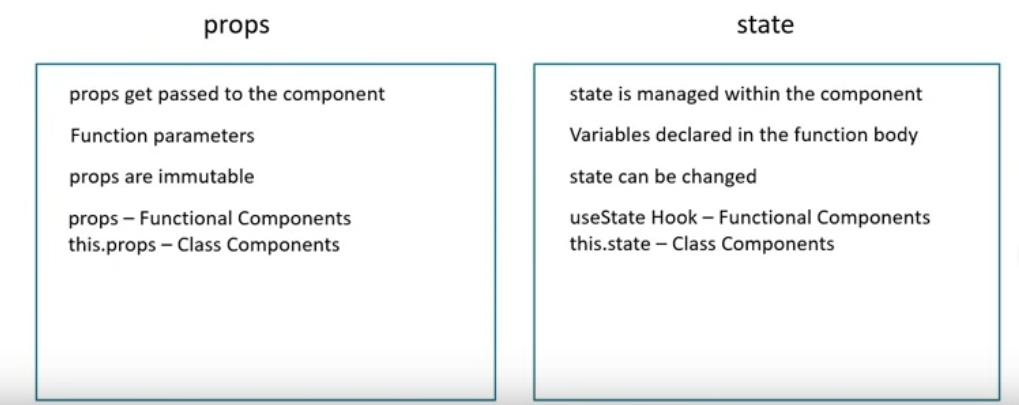
**Props**

To pass data from parent component to child component.

Parent component app.js Child component greet.js

 ****

**Props vs state**



**State**

State is an object that is privately maintained in the component. it can be changed within the component. It also influences what should be rendered in the browser.

First step create a state object and initialize it within a class constructor

Bind the state value in the render function

Message.js

****

**Setstate**

Do not modify the state directly. Instead use setState. If not, the react will not re-render the component. setState on the other hand let react know it has to re-render the component.

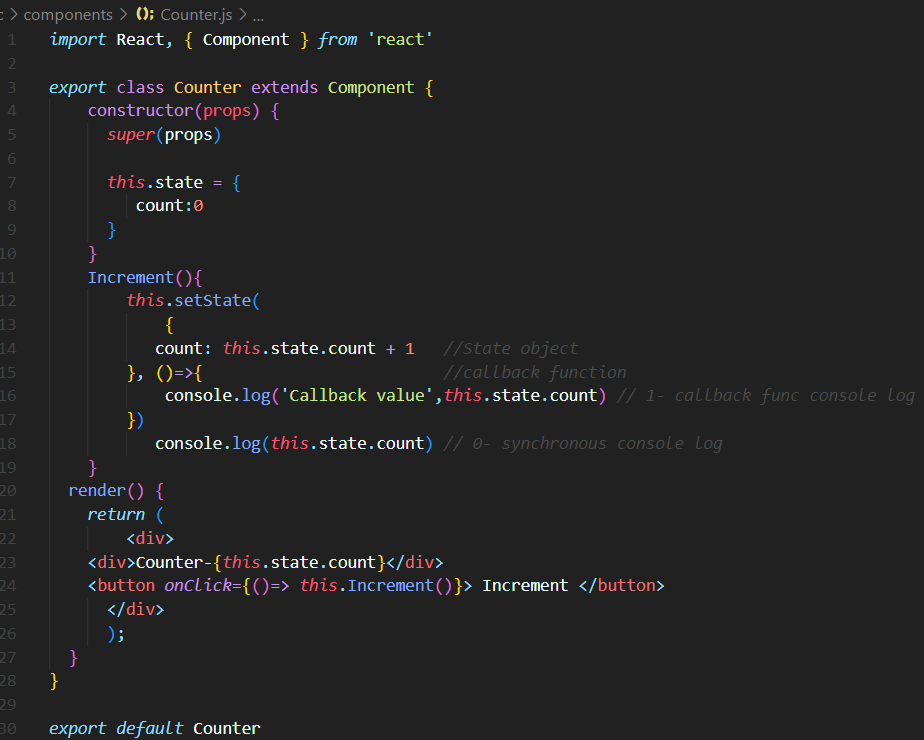
Counter.js

****

****

When the counter is 1 the console still displays 0. This is because calls to setState are asynchronous. Console.log executes before the state has been set. To resolve this issue pass a callback function to the set state method as a second parameter. First parameter is state object.





Always make use of setState and never modify the state directly

Code has to be executed after the state has been updated. Place that code in the call back function which is the second argument to the setState method.

*this*.setState(prevState => ({

            count: prevState.count +1

        }))

           console.log(*this*.state.count) *// 0- synchronous console log*

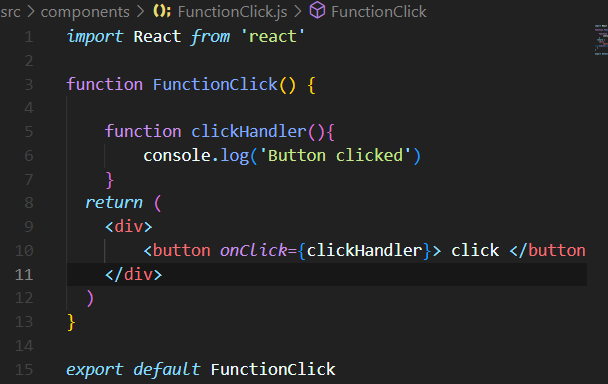
    }

When you have to update state based on the previous state value, pass in a function as an argument instead of regular object.

**Event handler**

Use Camel case for the event and specify function or method in curly braces.

**Functional Component**

****

**Note:** do not add paranthesis in the handler (onClick={clickHandler()}). This will not work. In class component methods will be accessed with this keyword.(onClick={this.clickHandler}).

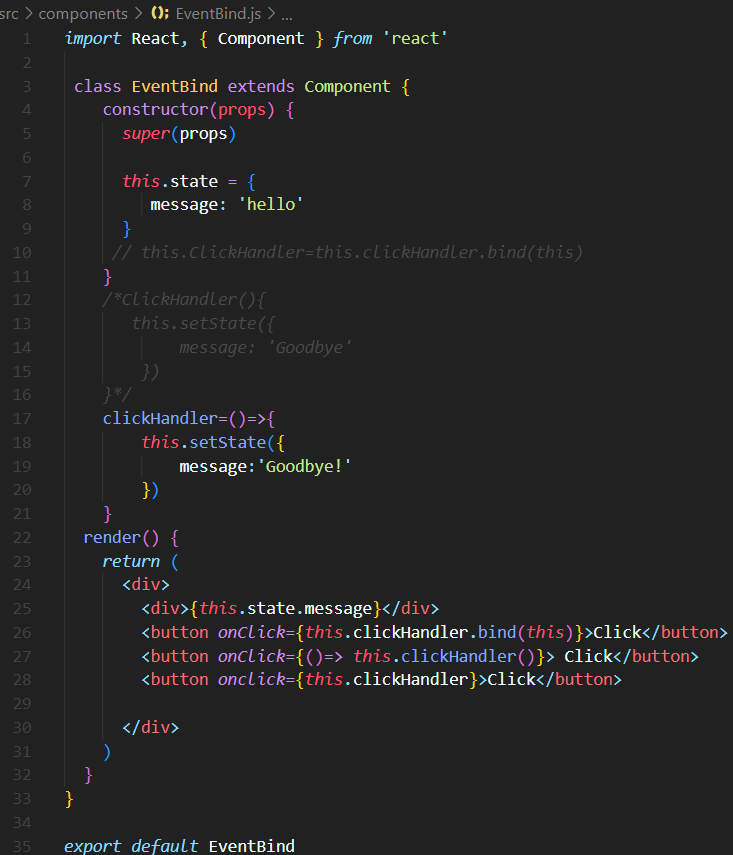
Binding Event handler

This keyword is undefined in an event handler and that is the reason that event binding is necessary in react class components.

There are 4 ways of binding

* Binding in render
* Arrow function in render
* Binding in the class constructor
* Class property as arrow function

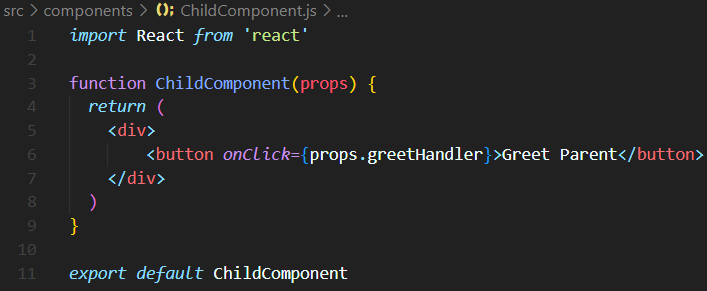
Eventbind.js

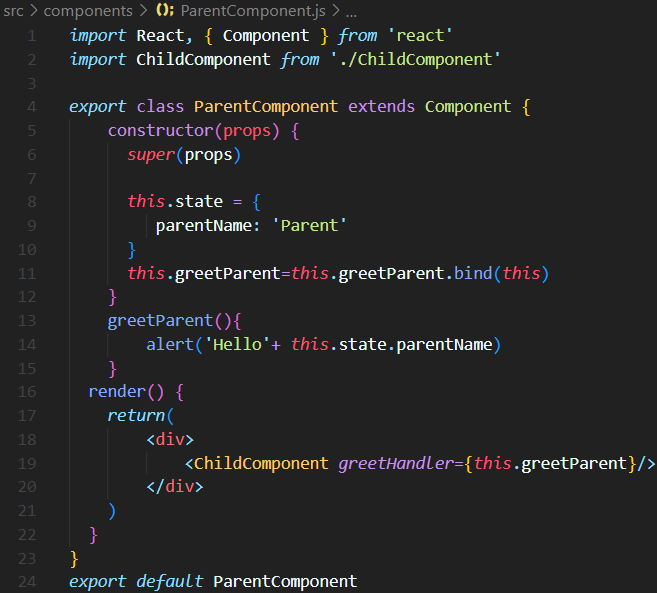


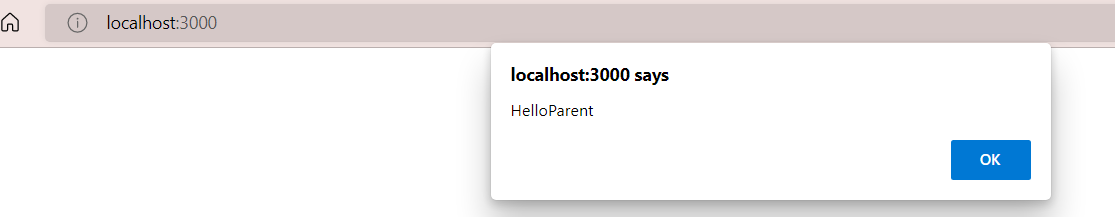
**Methods as props**

Props: to pass data from parent to child component.

Methods as props is used for child to communicate with parent component. Here method itself is passed as a prop to the child component.



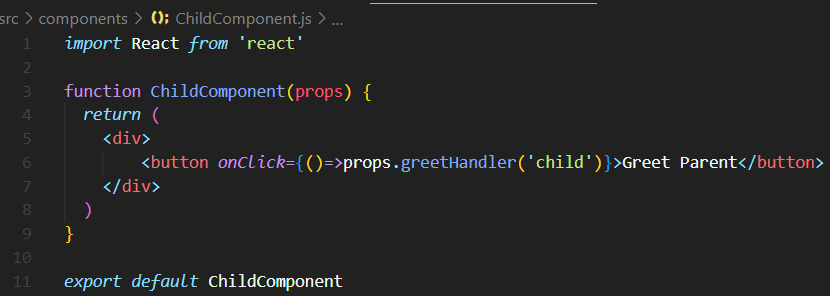




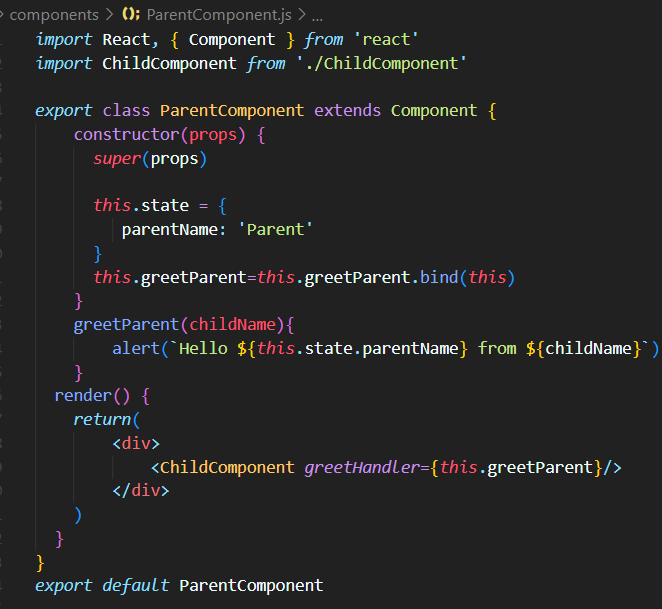
1. In the parent component define the method
2. On the child component tag pass the method as a prop
3. In the child component, access the method using props object

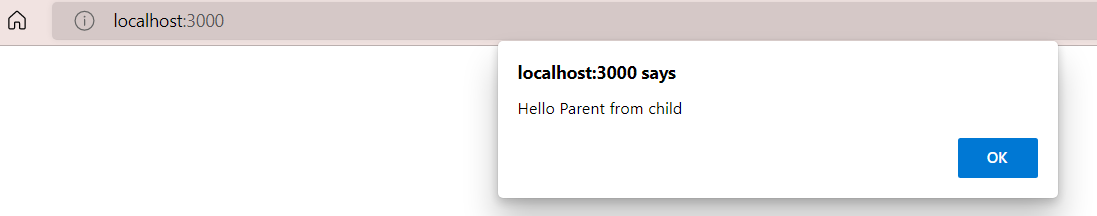
A method in parent component is called with a button in the child component by passing method as props to the child component.

**Passing parameter from child to parent**

**Use arrow function while passing parameter**

**Parent component**

****

****

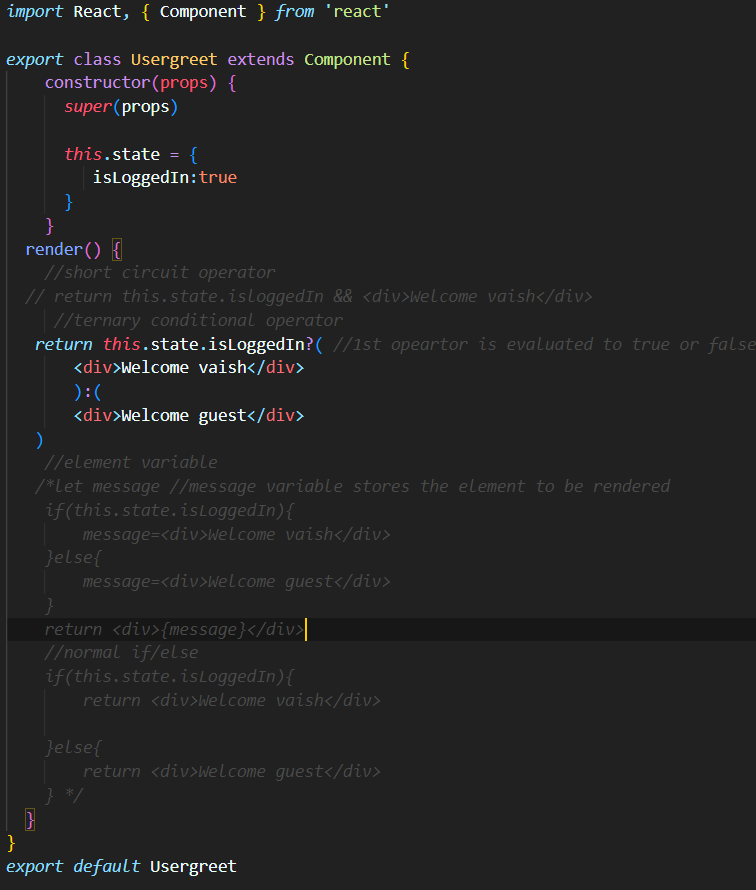
**Conditional rendering**

**1.if/else**

**2.Element variables**

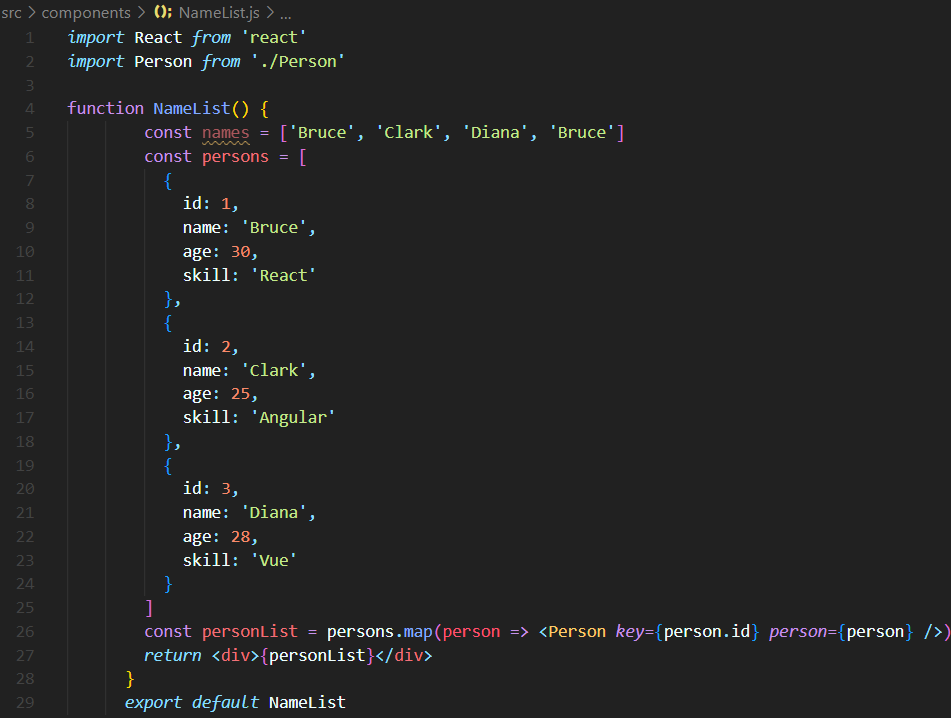
**3.Ternary conditional operator**

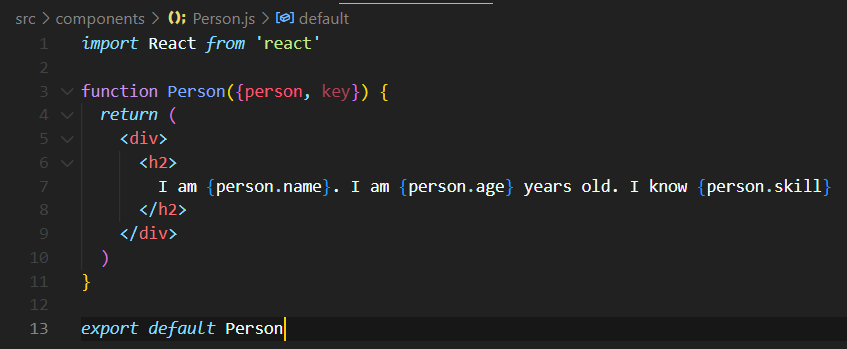
**4.short circuit operator**

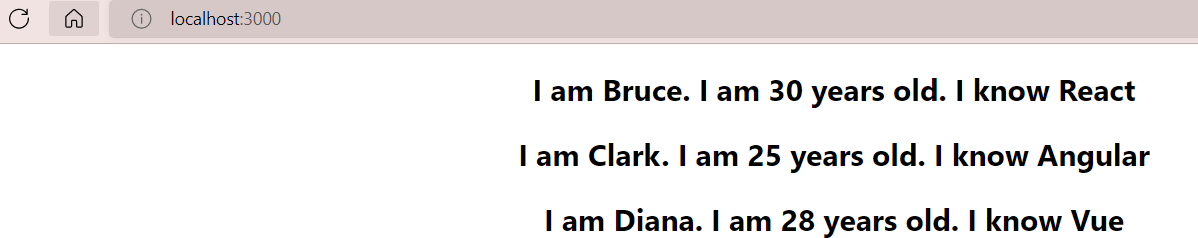
****

**List rendering**

****

****

****

****

****

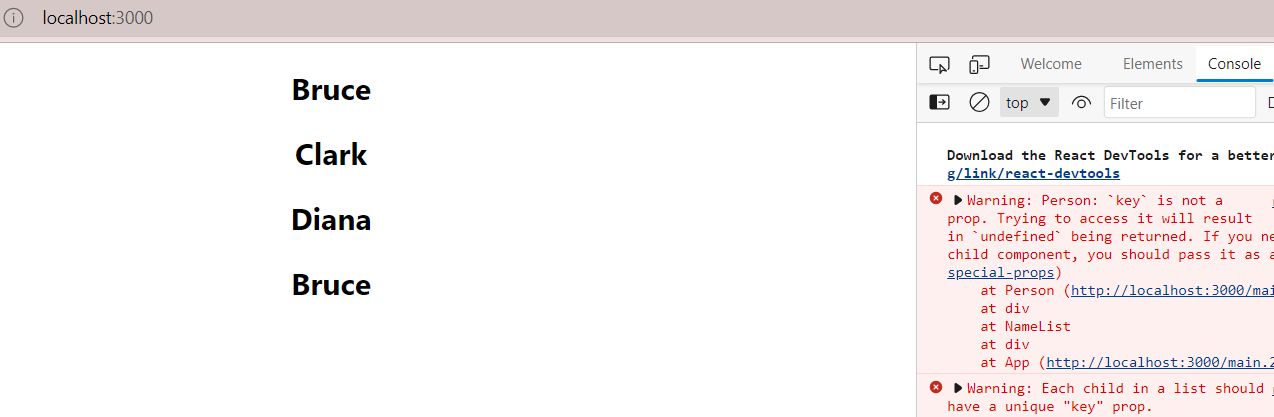
A “key” is a special string attribute you need to include when creating lists of items.

Keys give the elements a stable identity

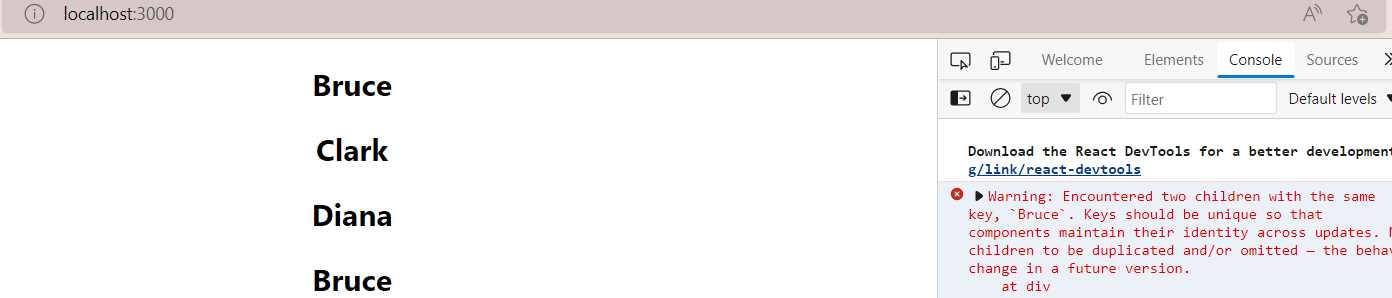
Keys help React identify which items have changed , added or removed

Help in efficient update of the UI.,

****

****

 const nameList = names.map(name => <h2 *key*={name}> {name} </h2>)

****

 const nameList = names.map((name,index) => <h2 *key*={index}> {index}{name} </h2>)

****

When to use index as a key?

The items in the list do not have a unique id

The list is a static list and will not change

The list will never be reordered or filtered.

**Controlled component**

[Controlled components](https://itnext.io/controlled-vs-uncontrolled-components-in-react-5cd13b2075f9) in React are those in which form data is handled by the component’s state.

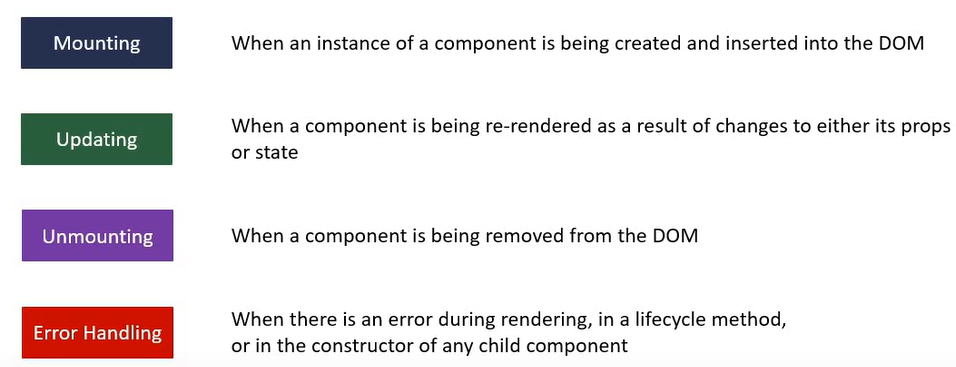
**3 steps**

* Add the HTML element
* Assign the component state to the element value
* Assign an onchange handler that updates the state

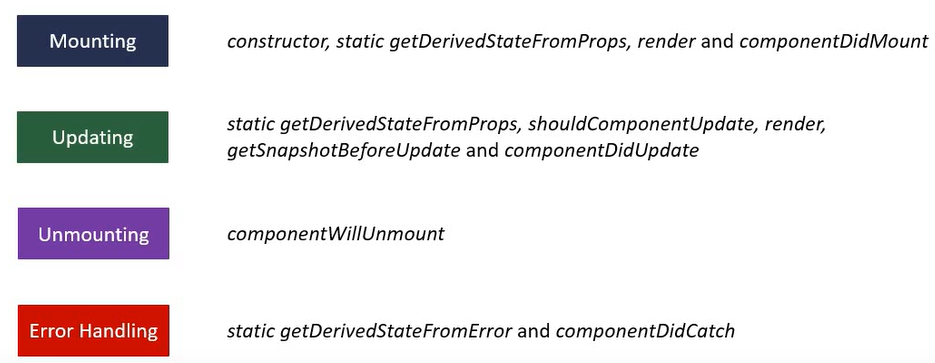
**Lifecycle methods**

A component goes through several stages in its lifecycle. React provides built-in methods that we can overwrite at particular stages in the lifecycle.

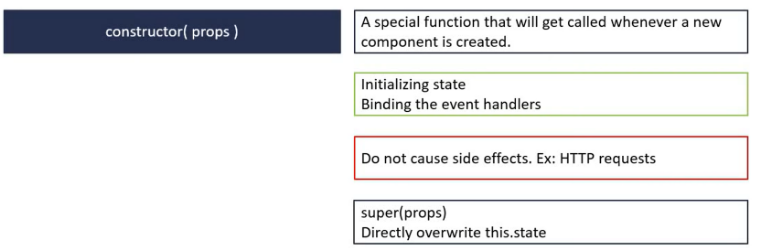
**There are 4 phases in a class component**

****

**Methods of each phase**

****

**Mounting Lifecycle method**

****

* Call the special function called super. This will call the base class constructor.
* In our component, we have access to this.props only after we have initially called super passing in props as an argument.



* When the state of the component depends on changes in props over time
* This method is used to set the state when the initial state of the component depends on the props being passed
* Since it is a static method, it does not have access to this keyword
* Do not cause side effects. eg : fetching data from an endpoint



* Only required method
* Read props and state and return JSX
* Do not change state or interact with the DOM or make AJAX calls
* Children component lifecycle methods are also executed

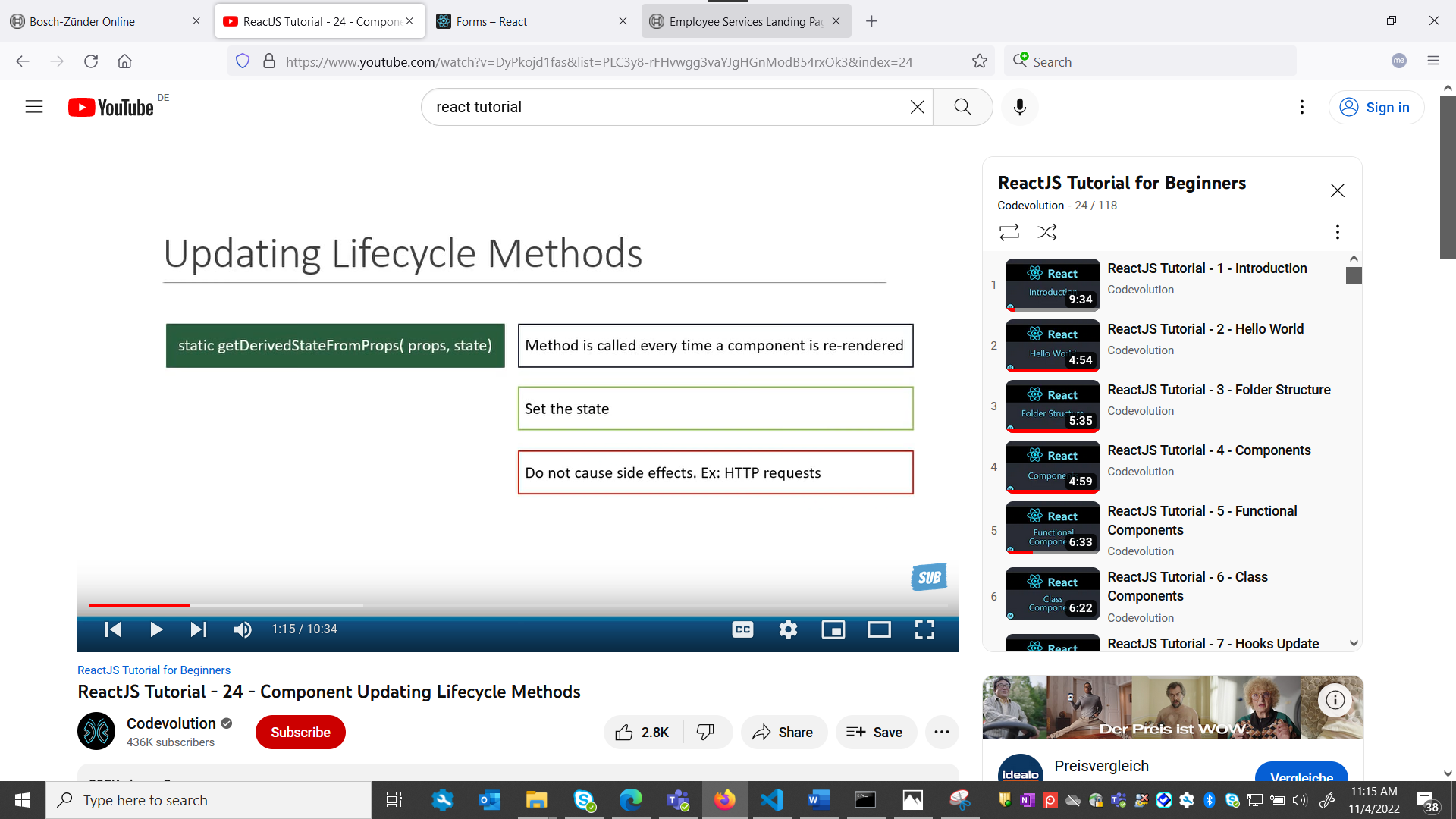


* Called only once in the lifecycle of the given component
* Invoked immediately after a component and all its children components have been rendered to the DOM
* Perfect method to cause side effects. Ex: Interact with the DOM or perform any ajax calls to load data.

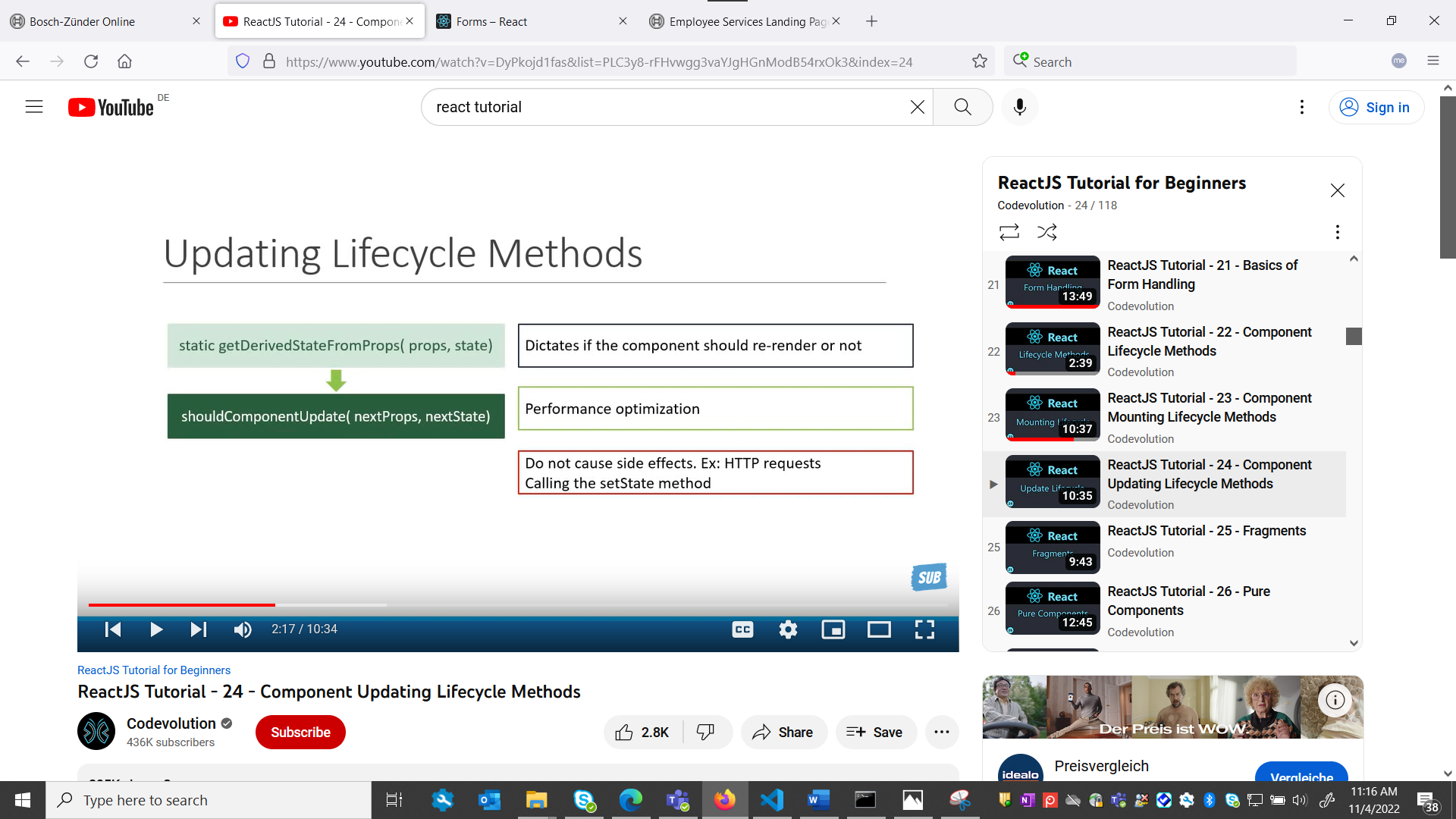
**Updating lifecycle methods**

Methods that are called when a component is being re-rendered because of changes in either props or state.

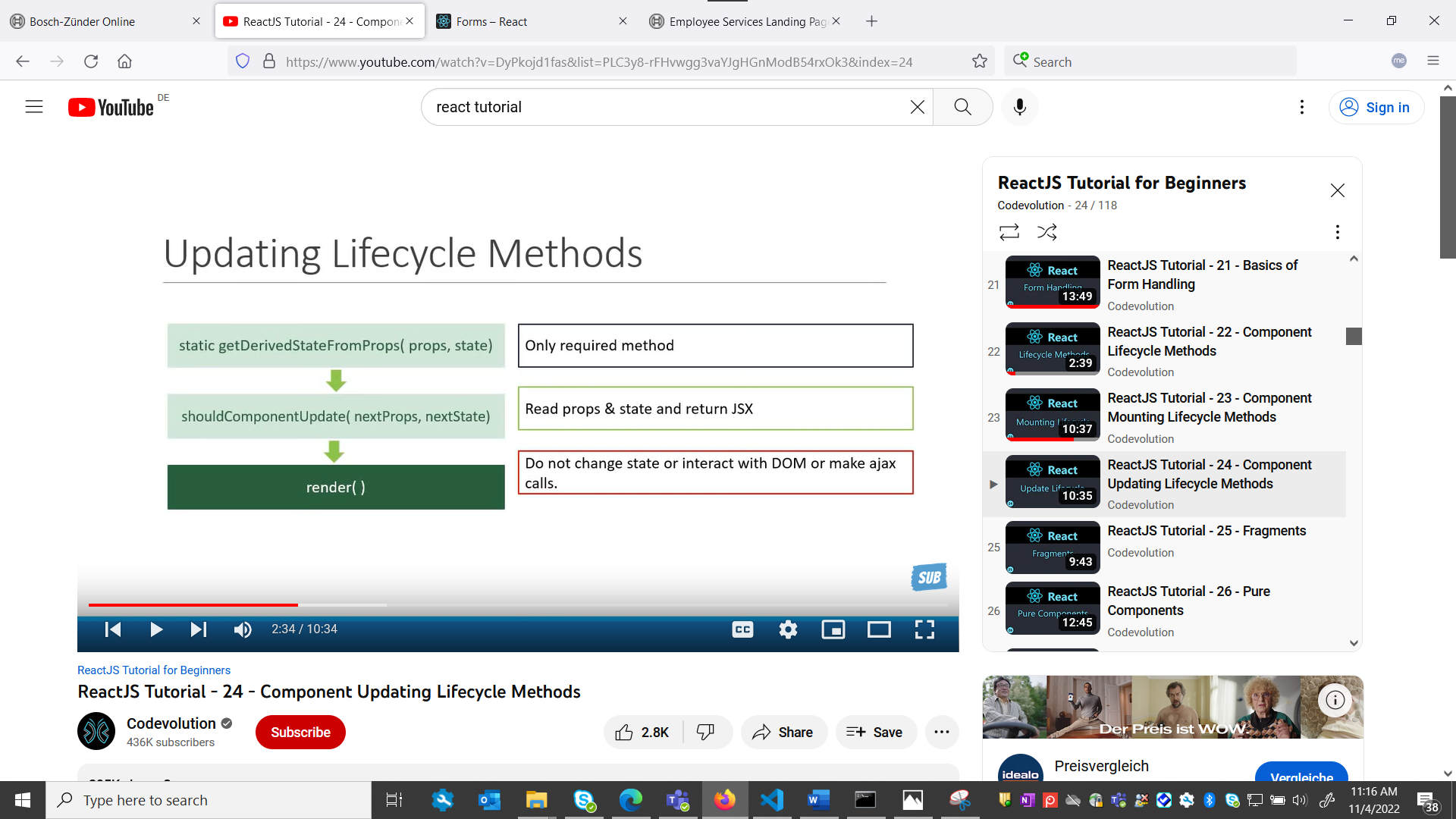
1



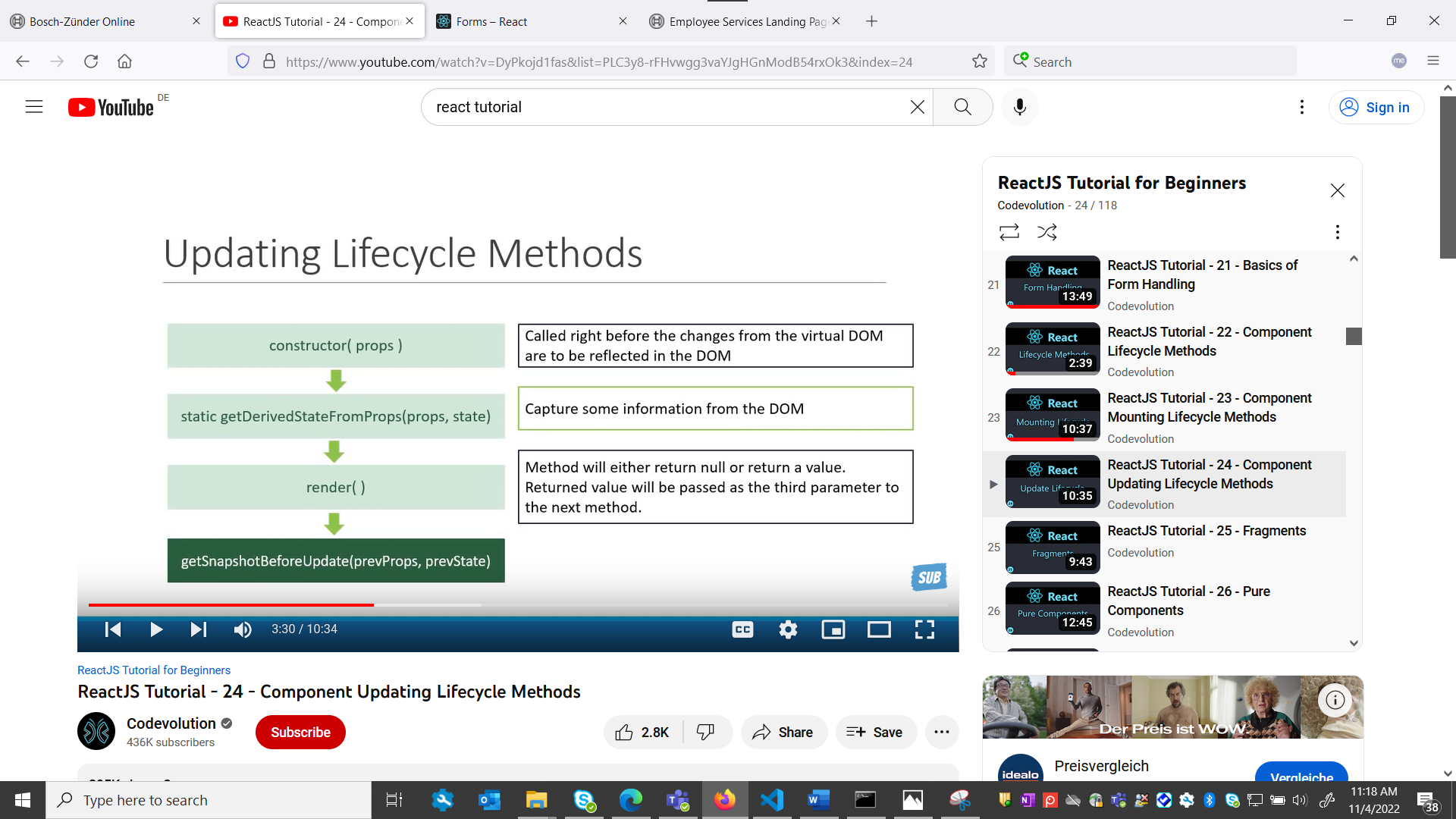
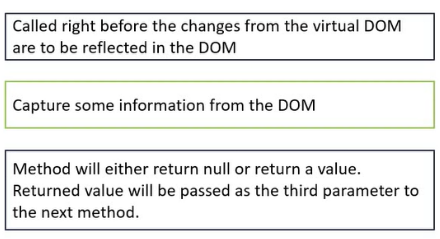
2



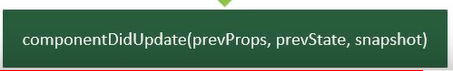
3



4.

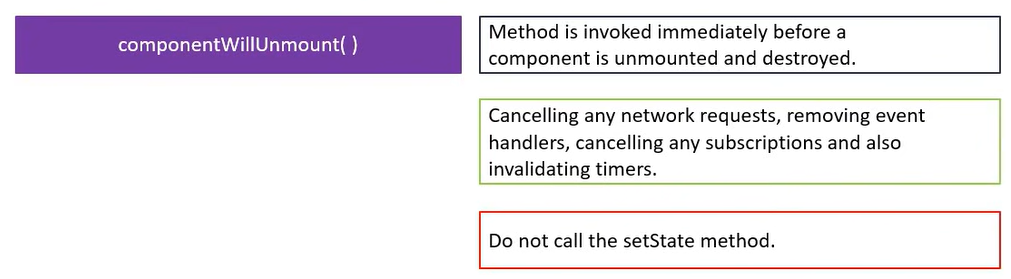
5.



Called after the render is finished in the re-render cycle

Cause side effects i.e,, make ajax calls.

**Unmounting**



**React fragments**

Fragments let you group a list of children elements without adding extra nodes to the dom.

function Fragment() {

*return* (

   <React.Fragment>

    <h1>Fragment</h1>

        <p>thid describe the fragment component</p>

   </React.Fragment>

  )

}

Before adding fragment there is a extra node in the DOM i.e additional div tag



By adding React.fragment we can remove that extra node. Short hand property for React.fragment is <> </>.



**Pure Component**

A React component can be considered pure if it renders the same output for the same state and props.

We can convert component to pure component as below:

* For class components react provides React.PureComponent base class.
* For Functional component react provides React.memo HOC (Higher Order Component).

**React.PureComponent**

* When a class component extends React.PureComponent base class then it is a Pure component.
* Pure component **avoids unnecessary re-renders**
* It does not implement **shouldComponentUpdate() method.**
* In React.Component, shouldComponentUpdate() enables re-rendering of components to the DOM. But a PureComponent will compare(SC) the current state and props with new state and props and checks if re-render is actually necessary by improving performance.

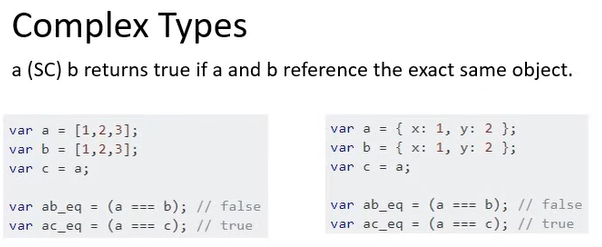
**Shallow comparison (SC)**

**Primitive types**

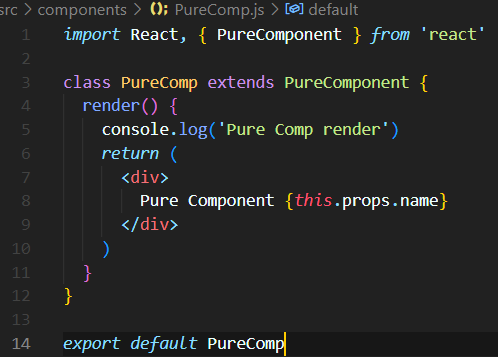
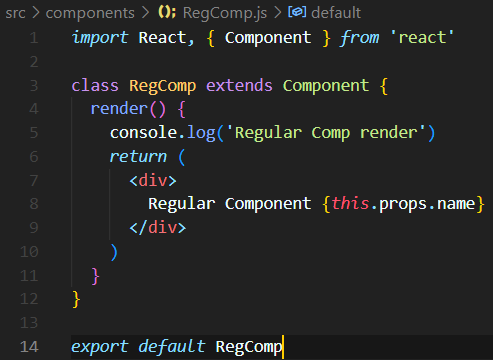
a (SC) b returns true if a and b have the same value and are of same type.

Eg: a= string(xyz) , b=string(xyz)

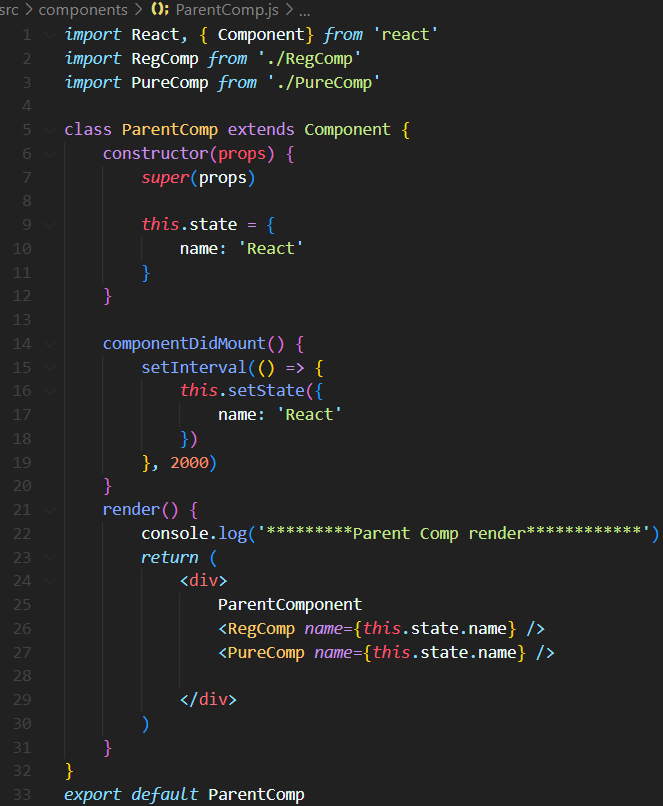
**Complex types**



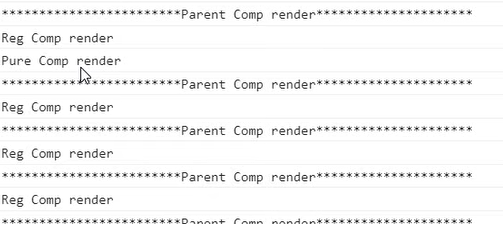
PureComp.js RegComp.js

ParentComp.js

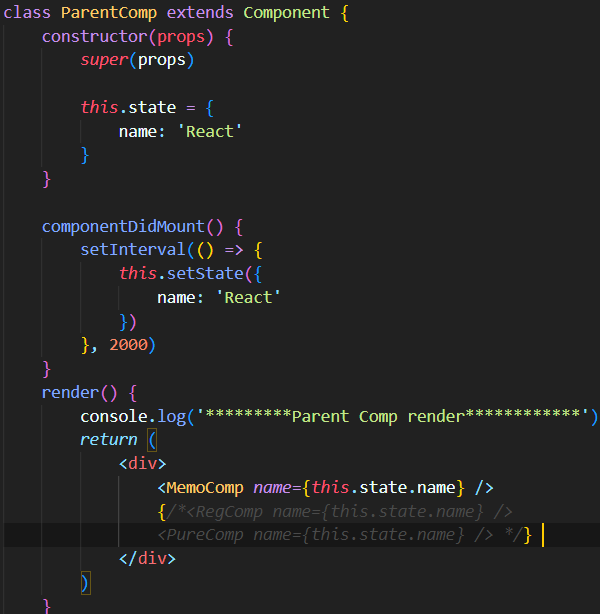
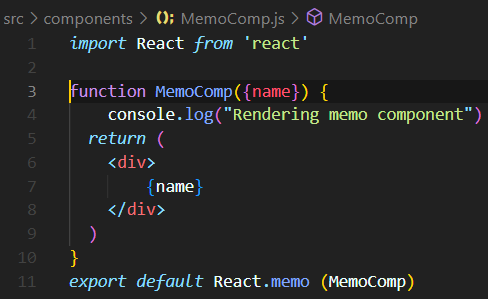


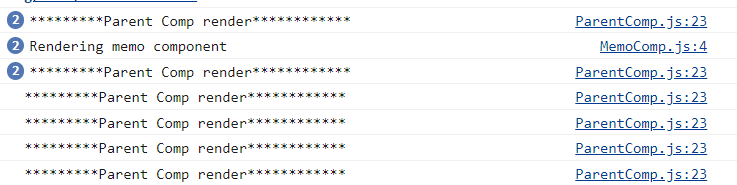
Console output



Every 2 seconds parent and regular components are re-rendered but pure component is never re-rendered.

**Functional component (React.memo)**

** **

****

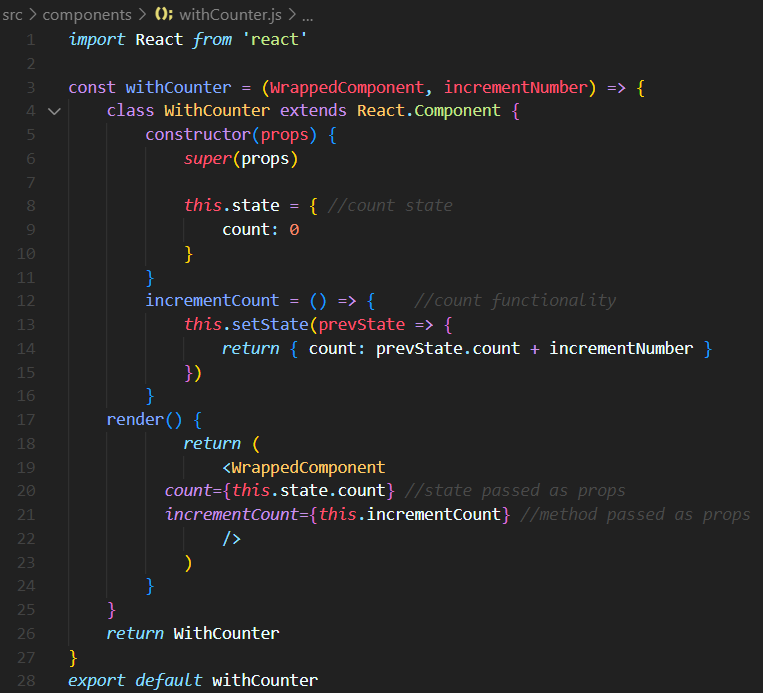
**Higher Order Component(HOC)**

A pattern where a function takes a component as an argument and returns a new component.

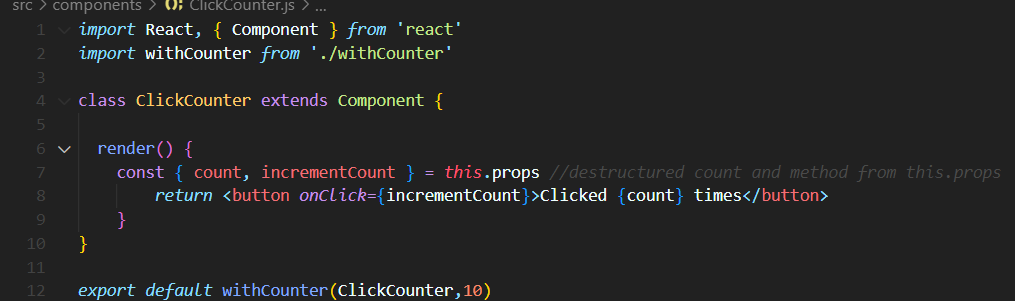
If two or more components have common functionality, instead of duplicating the code we can use HOC for reusing the code.

**Const NewComponent= higherOrderComponent(originalComponent)**

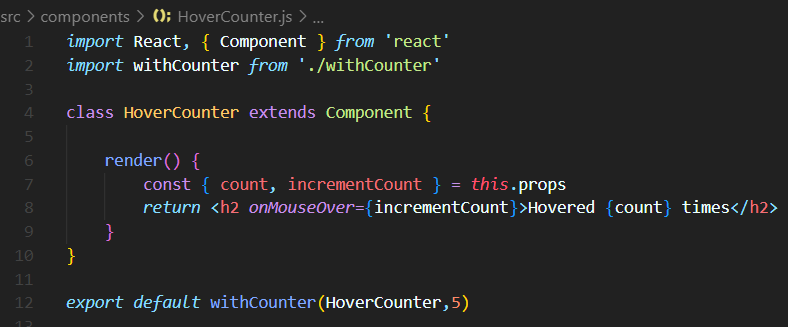
**HOC**

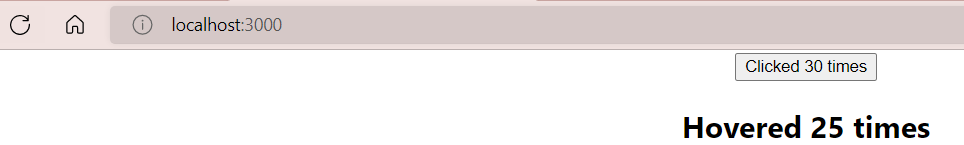
****

**ClickCounter.js**

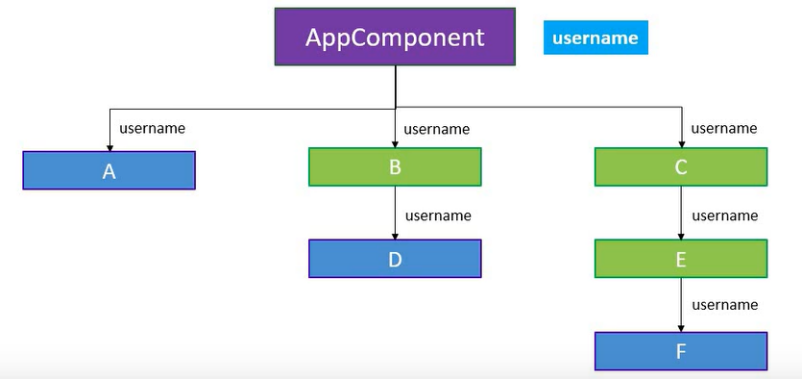
****

**HoverCounter.js**

****

****

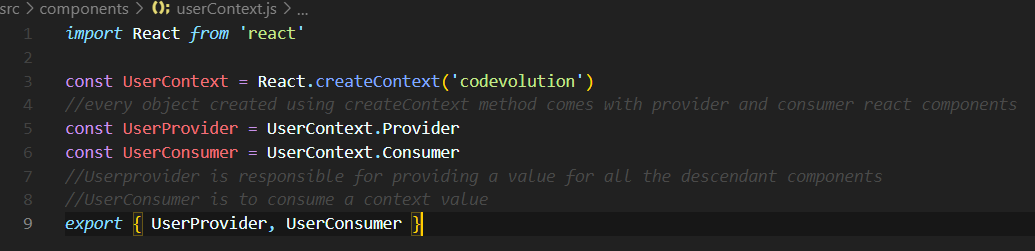
**Context**

****

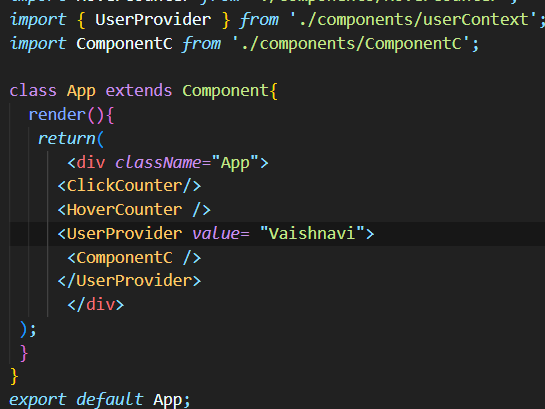
Context provides a way to pass data through the component tree without having to pass props down manually at every level

Our goal is to pass the username from the app component and read it in the component F using context

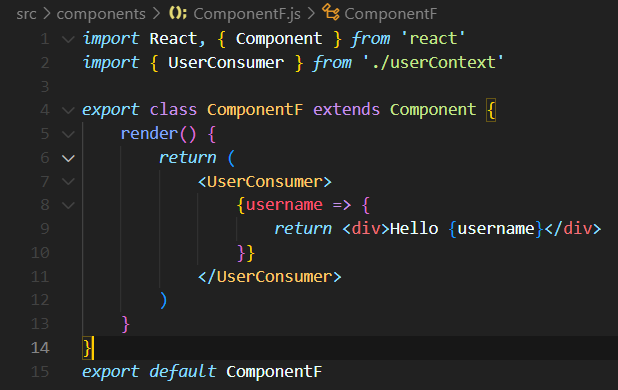
**Step1:** Create the context using React.createContext() method from react. Export the provider and consumer components**.**

****

**Step2:** At app component, include the provider component and provide a value using value attribute. This value can be consumed in any of the descendant components.

****

**Step3:** Where the username is required use consumer component and pass in a function as its child. The function receives the context value as parameter which can be used to return the desired JSX.

****

**React Hooks**

Hooks are a new feature addition in React version 16.8 which allow to use react features without having to write a class

Ex. State of a component

Hooks don’t work inside classes

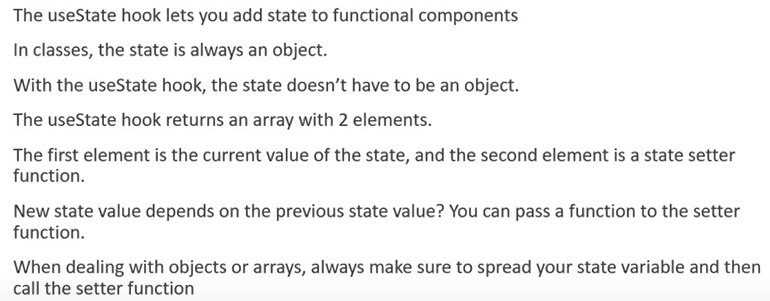
**Why hooks?**

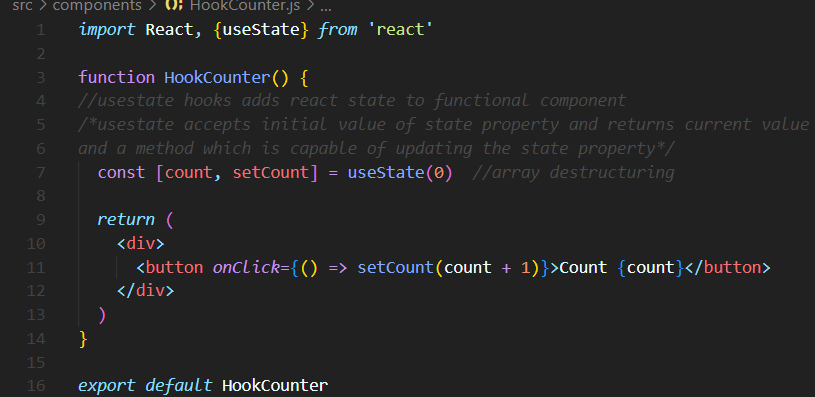
Avoid the whole confusion with ‘this’ keyword

Allow you to reuse the stateful logic

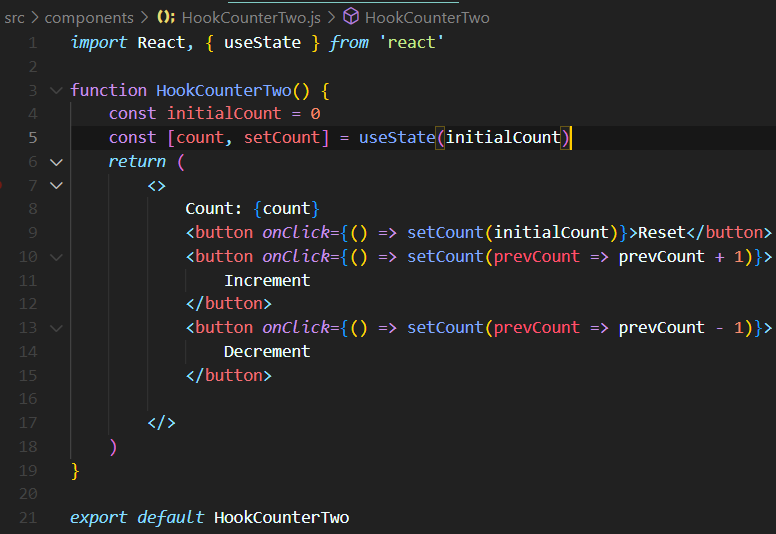
Hooks let us organize the logic inside a component into reusable isolated units.

**Usestate hook**

****

****

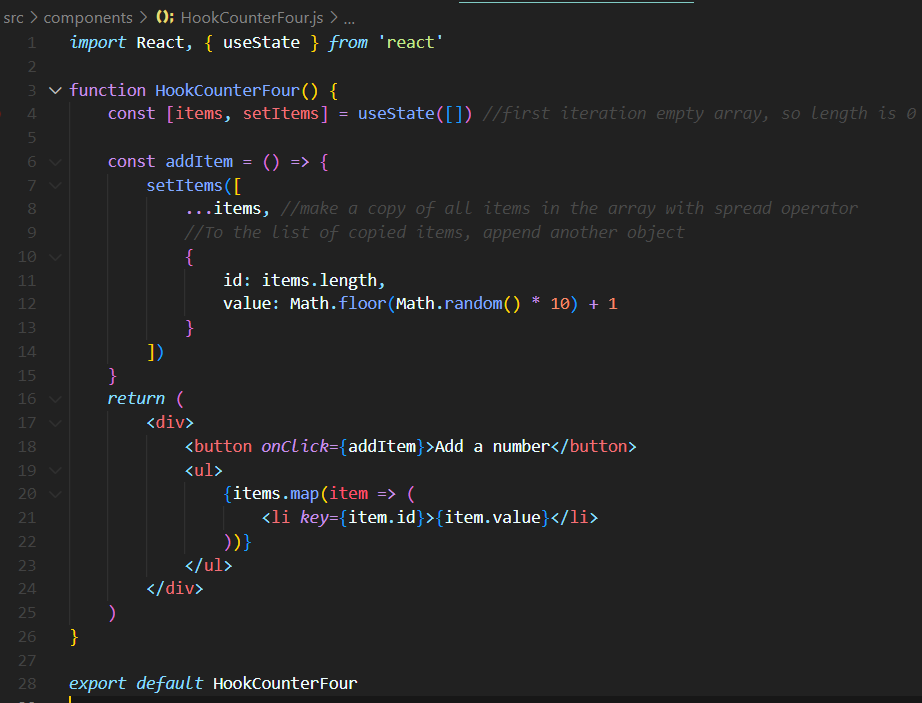
Usestate previous state



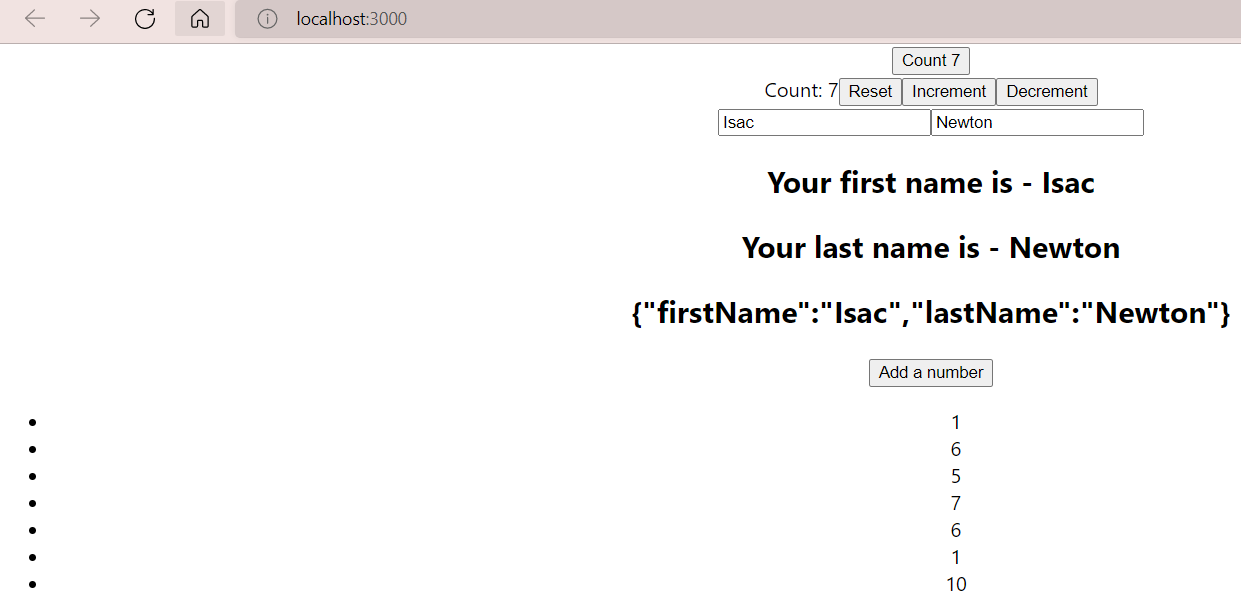
Usestate with object



Usestate with array



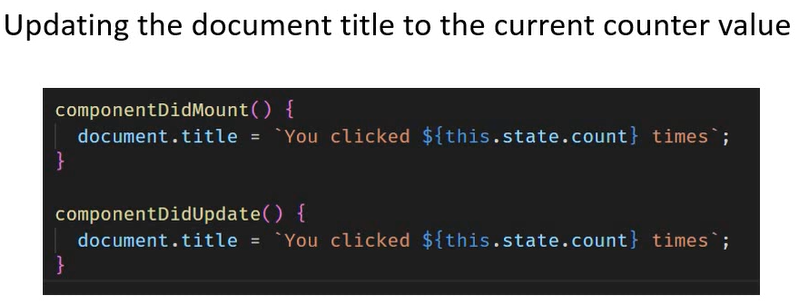
Output:



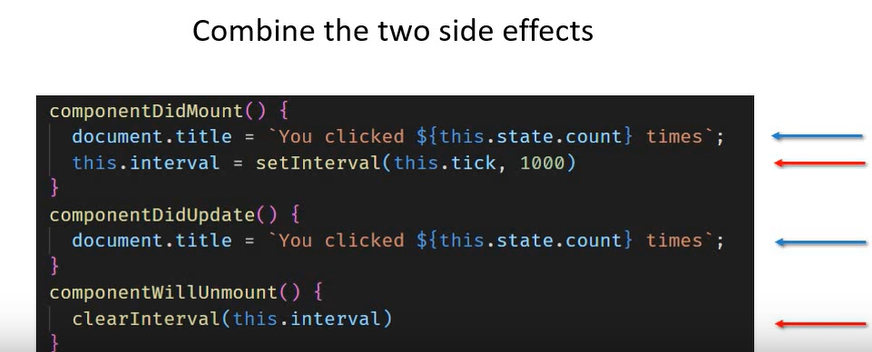
**UseEffect**

* The Effect Hook lets you perform side effects in function components
* Some examples of side effects are: fetching data, directly updating the DOM, and timers.
* useEffect accepts two arguments. The second argument is optional.
* useEffect(<function>, <dependency>)

**Why useEffect?**

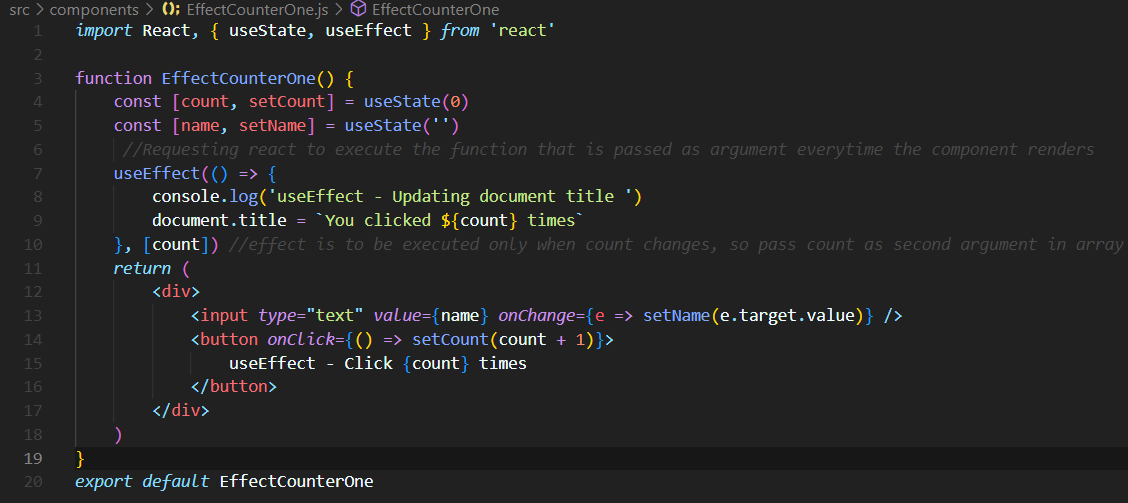






Exact same code is repeated twice. The code setInterval and clearInterval are put in separate methods. The code to update the DOM and set time interval which are completely unrelated are put together. Therefore, To avoid code repetition and to group the related code together useEffect comes into picture.

The effect Hook is a close replacement of componentDidMount, componentDidUpdate and componentWillUnmount.

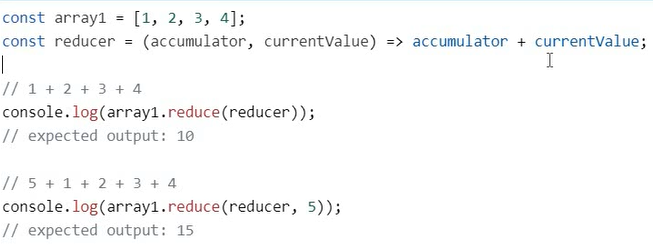


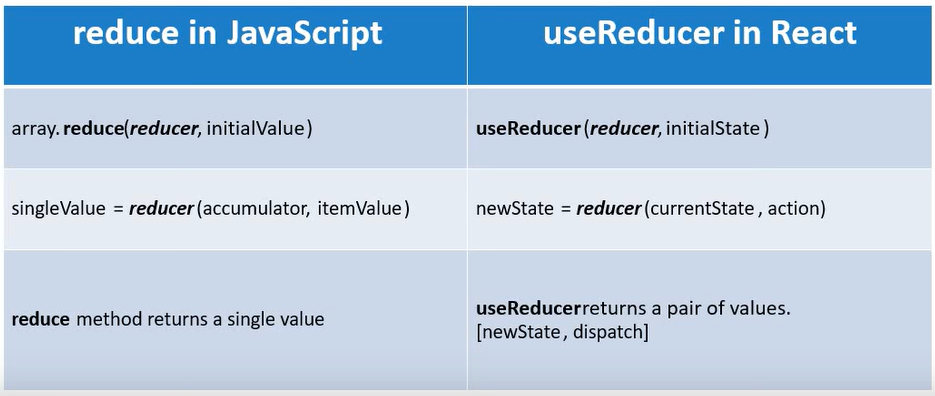
**useReducer**

useReduce is a hook that is used for state management.

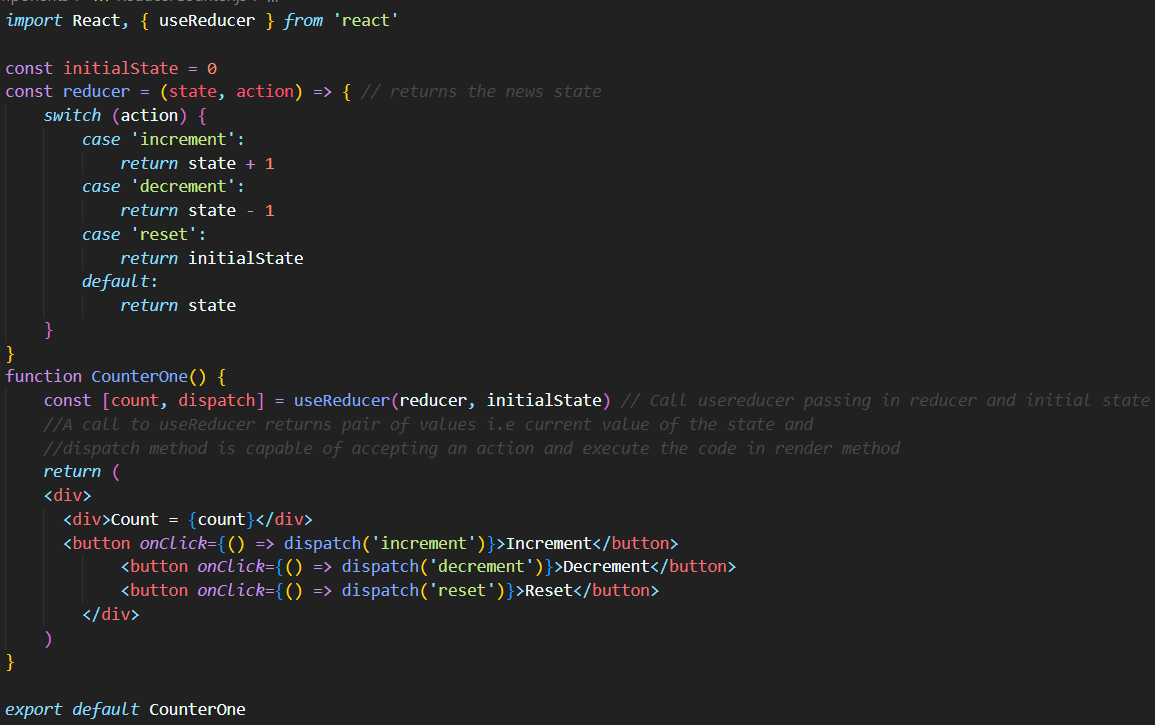
It is an alternative to use state

**Javascript reduce**



****

**useReducer**

****

**useCallback**

* The React useCallback Hook returns a memoized callback function.
* memoization - caching a value so that it does not need to be recalculated.
* This allows us to isolate resource intensive functions so that they will not automatically run on every render.
* The useCallback Hook only runs when one of its dependencies update.
* This can improve performance.
* The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function. One reason to use useCallback is to prevent a component from re-rendering unless its props have changed.

****

When clicking on one button everything gets re-rendered. This causes performance issues in the case where we have more number of components.

This can be solved by using React.memo which is a higher order component prevents the components re-render until the props or state doesn’t change.

****

After using React.memo we can still see , when we increment age the salary is also re-rendered.

**Title.js**

*import* React *from* 'react'

function Title() {

  console.log('Rendering Title')

*return* (

    <h2>

      useCallback Hook

    </h2>

  )

}*export* *default* React.memo(Title)

**Count.js**

*Import* React *from* 'react'

function Count({ text, count }) {

    console.log(`Rendering ${text}`)

*return* <div>{text} - {count}</div>

}

*export* *default* React.memo(Count)

**Button.js**

*import* React *from* 'react'

function Button({ handleClick, children }) {

  console.log('Rendering button - ', children)

*return* (

    <button *onClick*={handleClick}>

      {children}

    </button>

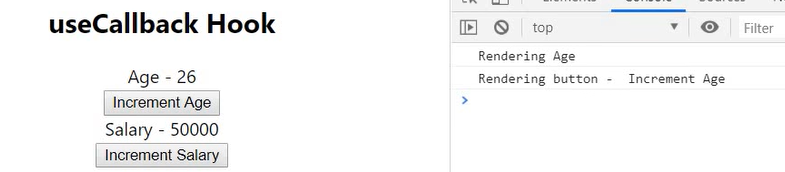
  )

}

*export* *default* React.memo(Button)

**ParentComponent.js**

****

****

When using useCallback, we are able to avoid unnecessary re-render of salary when clicking on increment age button.