**setState()**

Let’s understand the do's and don'ts with state and setState()

let us create a counter component, we will basically have a count value and a button to increment that counter value.

Create a new component called Counter.js

Class component react snippets: rce

class Counter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    increment()

    {

        this.state.count=this.state.count+1

        console.log(this.state.count)

    }

    render()

    {

        return (

        <div>

            <div>count: {this.state.count}</div>

            <button *onClick*={()=>this.increment()}>Increment

            </button>

        </div>

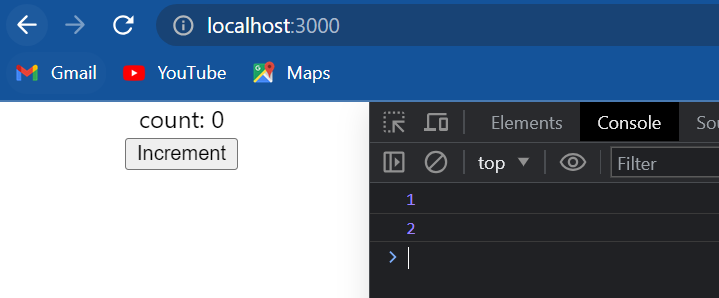
        )

    }

}

*export* default Counter

let's save this and take a look at the browser and open the console and now when I click on the increment button we can see that the value is not incremented in the UI. In the console though we can see that it has changed from 0 to 1. If I click again in the UI it is still 0 but in the console, it is now 2. Few more clicks and we can see that it's the same 0 in the UI but increments in the console.



What this means is that the UI is not rear-ending whenever the state is changing and this is the main reason **we should never modify the state directly**. The only place where we can assign **this.state** is the constructor any other time to change the state setState() method has to be used.

let's make the change and see if it helps:

    increment()

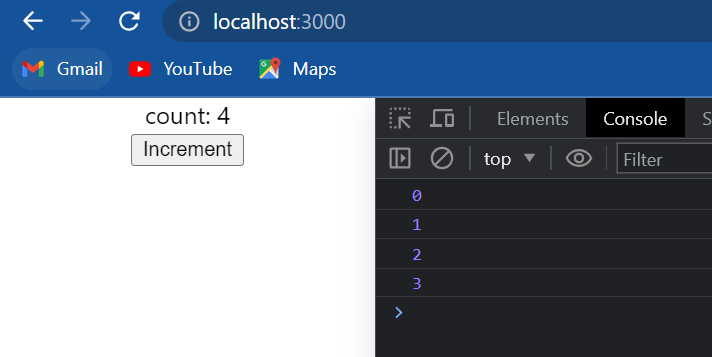
    {

        this.setState({ count : this.state.count + 1 })

        console.log(this.state.count)

    }

If we now save this and take a look at the browser and click on increment we can see that the count value in the UI increments to 1.



* Never modify the state directly instead make use of setState() when we modify the state directly, react will not re-render the component setState() on the other hand will let react now it has two re-render the component.

There is one detail to observe with said state in the browser we can see that when we click on increment the value is 1 but if we take a look at the console the value is 0.

So the console value is 1 less than the rendered value and this is because calls to setState() are asynchronous so what is happening is console.log over here is being called before the state is actually set.

Many times in our application we might want to execute some code only after the state has been updated to handle such a situation we can pass in a callback function as the second parameter to the setState() method.

So the setState() method has two parameters the **first parameter is the state object** and the **second parameter now is the callback function.** The callback function will be an arrow function.

Within the function body let's log to the console callback value it is going to be this.state.count so just formatted the code and we can see that this dot setState() first parameter is an object which sets the state value and the second parameter is the arrow function where we simply log the updated state dot count value :

 increment()

    {

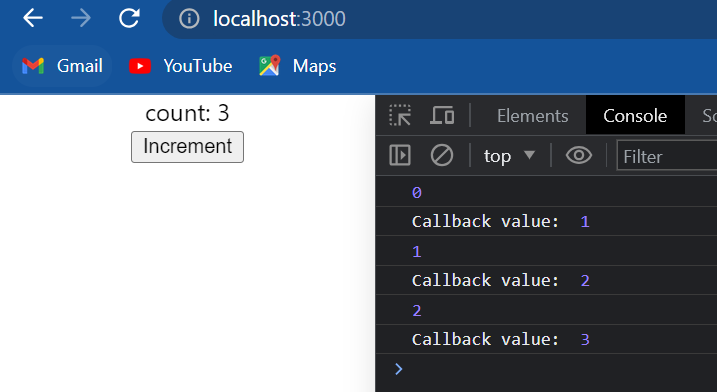
        this.setState({ count : this.state.count + 1 }, ()=>{console.log("Callback value: ",this.state.count)})

        console.log(this.state.count)

    }

So we have two console.log: one console.log value outside the setState() method and one within the callback function within the setState() method.

Now if we save this and take a look at the browser click on the button and in the console we can see that we have a value of zero and then a callback value of one the same value of 1 is also rendered in the browser.



* Whenever we need to execute some code after the state has been changed do not place that code right after the setState() method instead place that code within the callback function that is passed as a second parameter to the setState() method.

let's take a look at the next scenario and try to use the current state to calculate the new state value the code is working as expected we don't see any problem the increment is working fine now that is simply because the current scenario is pretty simple.

let's make the scenario slightly complicated I'm going to create a new method called incrementFive() and within the body I simply call the increment method five times. Also now with a click of a button, we are going to call this.incrementFive() so the count should increment from zero to five when we click on the button.

 incrementFive()

    {

        this.increment()

        this.increment()

        this.increment()

        this.increment()

        this.increment()

    }

    render()

    {

        return (

        <div>

            <div>count: {this.state.count}</div>

            <button *onClick*={()=>this.incrementFive()}>Increment

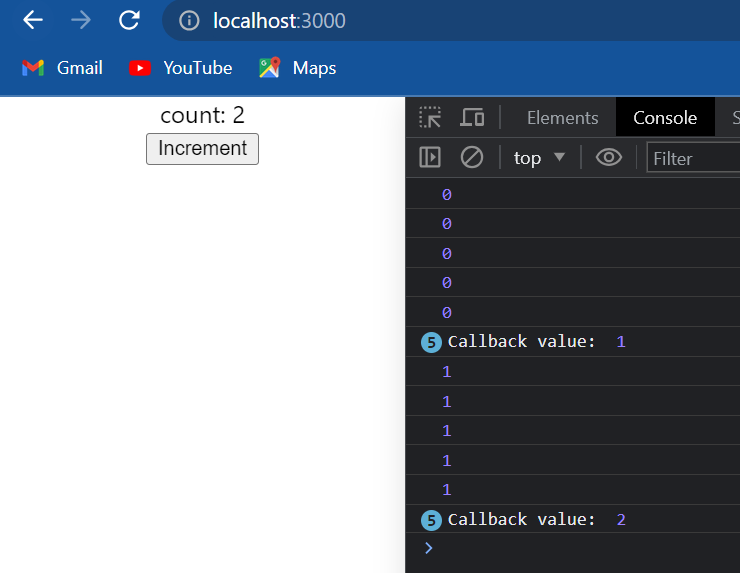
            </button>

        </div>

        )

    }

let's save this and take a look at the browser. When I click on the button we can see that the value changes to one instead of changing to five and in the console, strangely zero is logged five times, and even the callback value of one is logged five times. Now why is this ? This behavior is because **react may group multiple setState() calls into a single update for better performance**.



So what happens in our scenario is that all the five setState() calls are done in one single go and the updated value does not carry over between the different calls. **So whenever we have to update the state based on the previous state we need to pass a function as an argument to setState() method instead of passing in an object**.

increment()

    {

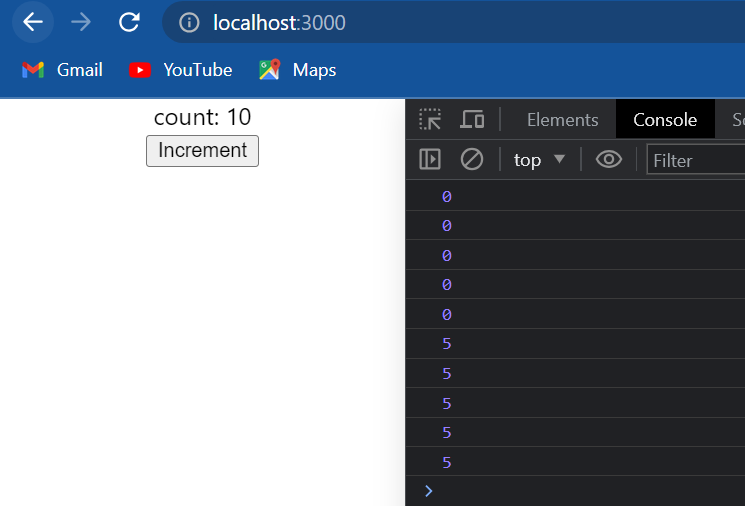
*// this.setState({ count : this.state.count + 1 }, ()=>{console.log("Callback value: ",this.state.count)})*

        this.setState(prevState=>({count:prevState.count+1}))

        console.log(this.state.count)

    }

So **this.setState** and as the argument we pass in a function. Again because we use ES6 we passing an arrow function. As a parameter to the arrow function we get the previous state of the component. so the parameter I'm going to name it as previousState and within the body we said the count state value to **previousState.count+1.** so very important we make note of this difference we are not using the current state instead we are always using the previous state this is going to give us the right results. let's save this go back to the browser click on increment.



In browser the count is displayed as 5 increment again and the count is 10. We are able to correctly render the UI based on this state. The console values logged are from the synchronous console log statement and can be ignored for now.

* When we have to update the state based on the previous state make sure to pass in a function as an argument instead of the regular object.

The function has access to the previous state which can be used to calculate the new state and as it turns out the second parameter to this function is the props object.so if our new state is dependent on props as well (props.addValue) maybe we can go with the function parameter approach and make use of props.

this.setState((prevState,props)=>({count:prevState.count+addValue}))

**Summary:**

* 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 callback function which is the second argument to the setState() method.
* When we have to update sthe tate based on the previous state value passing a function as an argu nt instead of the regular object.

**Final code: App.js**



**Final code: Counter.js******

**Destructuring props and state**

* Destructuring is an ES6 feature that makes it possible to unpack values from arrays or properties from objects into distinct variables.
* In react destructuring props and state improves code readability.

Let's start with functional components.

**App.js**

import logo from './logo.svg';

import './App.css';

import Greet from './components/Greet'

function App() {

  return (

    <div *className*="App">

      <Greet *name*="Bruce" *heroName*="batman"/>

      <Greet *name*="Clark" *heroName*="superman"/>

      <Greet *name*="Diana" *heroName*="spiderman"/>

    </div>

  );

}

*export* default App;

**Greet.js**

import React from 'react'

const Greet=(props)=>

{

    console.log(props)

    return (

        <div>

            <h1>

                Hello {props.name} aka {props.heroName}

            </h1>

        </div>

    )

}

*export* default Greet

Earlier we had created a functional component called Greet. In *App.js* , we can see that we are passing in **name** and **heroName** as props and in the Greet component we access them using **props.name** and **props.heroName** .

Now there are two ways to destructure props in a functional component.

The **first way is to destructure it in the function parameter itself**. So instead of just props(const Greet=(props)=> ) we are going to have parenthesis and within the parentheses we have the destructuring using curly braces name, heroName (const Greet=({name,heroName})=>).

we are basically extracting name and heroName from the props object. In the JSX we can now simply use **name** and **heroName** (Hello {name} aka {heroName}) instead of props.name and props.heroName

let me also remove the **console.log** statement.

import React from 'react'

const Greet=({name,heroName})=>

{

    return (

        <div>

            <h1>

                Hello {name} aka {heroName}

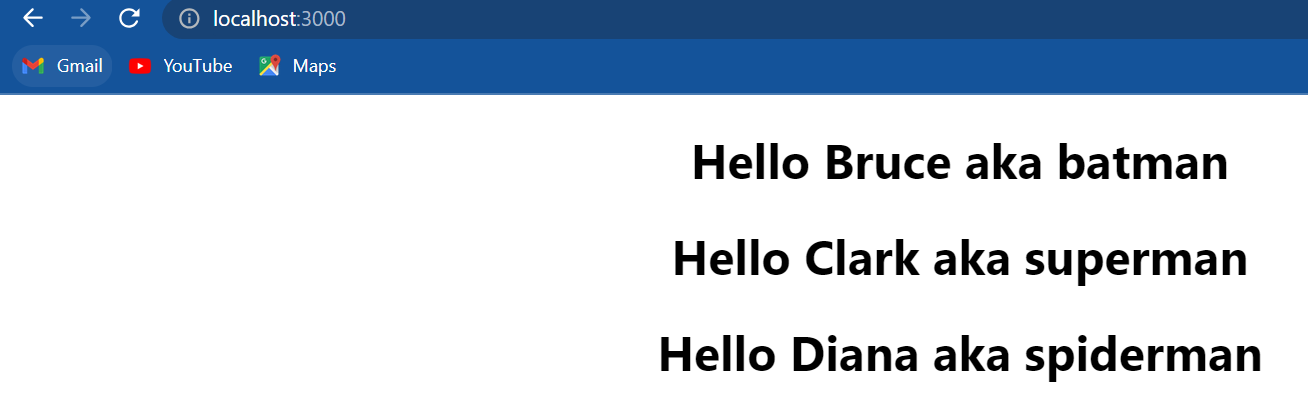
            </h1>

        </div>

    )

}

*export* default Greet



If we take a look at the browser we can see that the app still works. So that is the first way: destructuring in the parameter.

The **second way is destructure it in the function body**. So we go back to naming the parameter as props:

import React from 'react'

const Greet= props =>

{

    const {name, heroName} = props

    return (

        <div>

            <h1>

                Hello {name} aka {heroName}

            </h1>

        </div>

    )

}

*export* default Greet

In the body const curly-braces me comma heroName is equal to props so we extract name and heroName property from the props object after this we can use just name and heroName (Hello {name} aka {heroName}) in the JSX. Go back to the browser and it should still work as expected.

So these are the two ways to destructure props in functional components.

Let's take a look at class components.

I'm going to open *Welcome.js* which is our class component. This component also accepts the same two props name and heroName.

**App.js**

import logo from './logo.svg';

import './App.css';

import Welcome from './components/Welcome'

function App() {

  return (

    <div *className*="App">

      <Welcome *name*="Bruce" *heroName*="batman"/>

      <Welcome *name*="Clark" *heroName*="superman"/>

      <Welcome *name*="Diana" *heroName*="spiderman"/>

    </div>

  );

}

*export* default App;

**Welcome.js**

import React,{Component} from "react"

class Welcome *extends* Component

{

    render()

    {

        return (

            <h1>

                Welcome {this.state.name} aka {this.state.heroName}

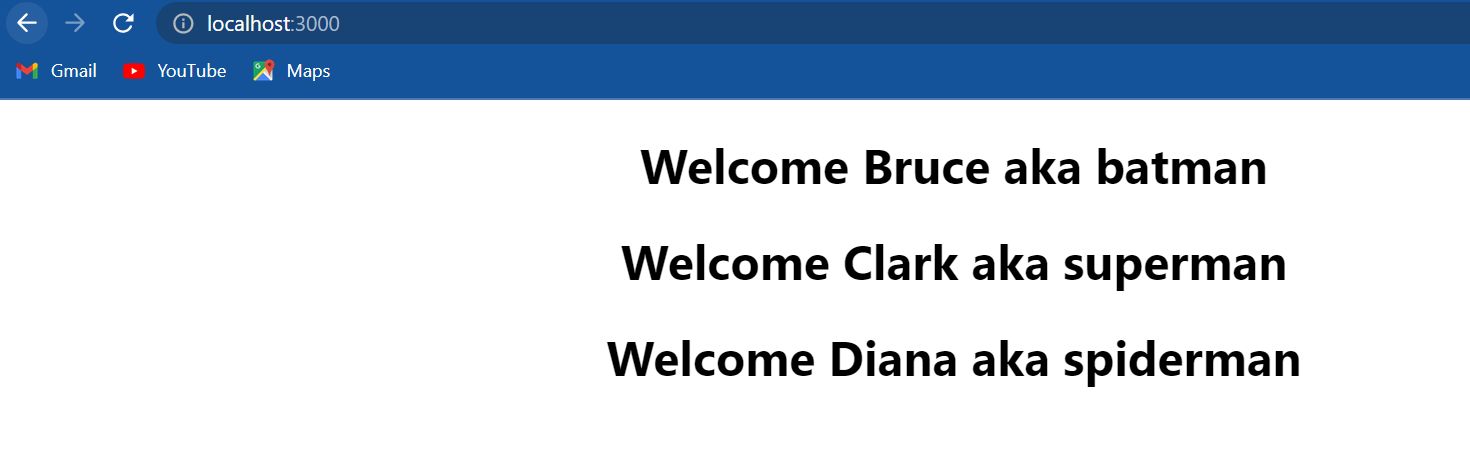
            </h1>

        )

    }

}

*export* default Welcome



Now in class components we generally tend to destructure props or state in the render method.

**Our props object could contain ten different props but we can destructure only the ones we wish to use in our component.**

import React,{Component} from "react"

class Welcome *extends* Component

{

    render()

    {

        const {name,heroName}=this.props

        return (

            <h1>

                Welcome {name} aka {heroName}

            </h1>

        )

    }

}

*export* default Welcome

If we take a look at the browser we should have same output.

Now on similar lines we might also have a number of state properties we can destructure them in a similar way (const {state1, state2} = this.state) .

**Event handling**

Any web application we create typically tends to have user interaction when the user interacts with our application events are fired for example mouse clicks mouse over key press change event and so on the application must handle such events and execute the necessary code.

let's see how to handle events in react we will focus on just the click event but the concept holds good for other events as well let's begin with functional components. I'm going to create a new file called *FunctionClick.js*

Functional component react snippets: rfce

Within the file I am going to create a new functional component and as part of the HTML or JSX I will add a button that says click. Next let add the component in the App component(App.js) so within the return statement in App.js add **<FunctionClick/>** make sure to import it as well.

If we look at the browser we should be able to see the click button when the user clicks on this button a click event is fired. Our goal is to capture that click event and execute some basic code.

The first point we have to make note of is react events are named using camel case rather than lowercase. In plain HTML and vanilla JavaScript we would specify **onclick** but in react it has to be camel cased **onClick** .

The second point is with JSX we pass a function as the event handler rather than a string. So instead of click handler within double-quotes we are going to have clickhandler within curly braces.

Now let's define the **clickHandler()** function, remember in JavaScript it is very much possible to define a function within another function. So inside the functional component we can define our **clickHandler()** so function click handler and within the body.

import React from 'react'

function FunctionClick() {

    function ClickHandler()

    {

        console.log("button clicked")

    }

  return (

    <div>

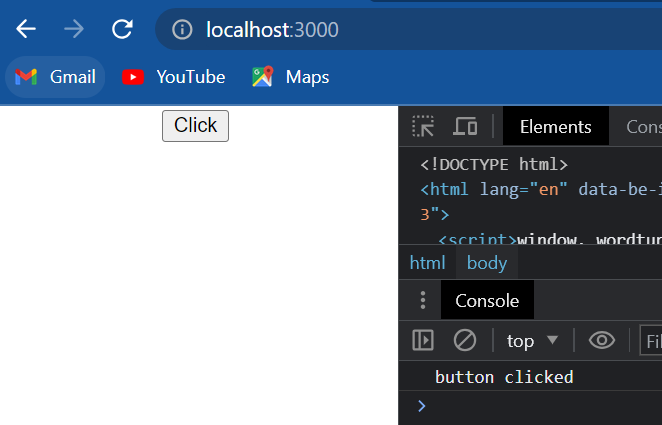
        <button *onClick*={ClickHandler}>Click</button>

    </div>

  )

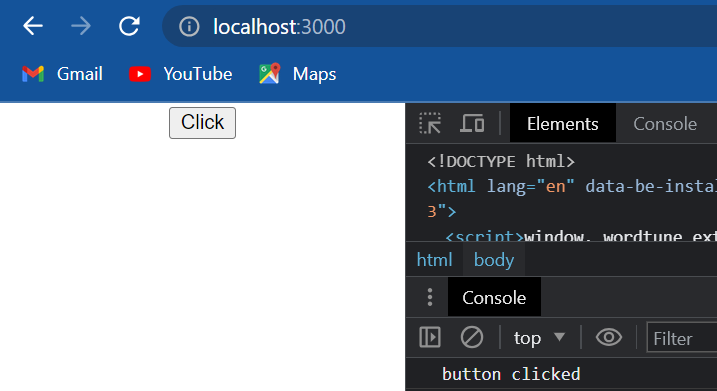
}

*export* default FunctionClick



If I open the developer tools and the console tab and click on the button we should see the message button clicked in the console our click event handling works as expected.

Now let me point out one of the more common mistakes we as beginners tend to make with event handling. In our code we can see that onClick we pass the function as the event handler pay close attention to the absence of parentheses if we do add parentheses it becomes a function call and that is not what we want. We want the handler to be a function and not a function call. let's quickly see what happens if we do leave the parentheses:



We can see that in the console the message “button clicked” is already logged. Now when I click on the button nothing happens no message is logged in the console. now the scenario becomes worse in class components when wer click handler changes the state of the component. the component constantly re-renders and we might see an infinite number of messages in the console so what I want we to keep in mind is that the event handler is a function and not a function call. (Do not add the parentheses)

let's take a look at event handling in class components.

I'm going to create a new file called *ClassClick.js*

import React, { Component } from 'react'

class ClassClick *extends* Component {

    clickHandler()

    {

        console.log("clicked the button")

    }

  render() {

    return (

      <div>

        <button *onClick*={this.clickHandler}>Click me</button>

      </div>

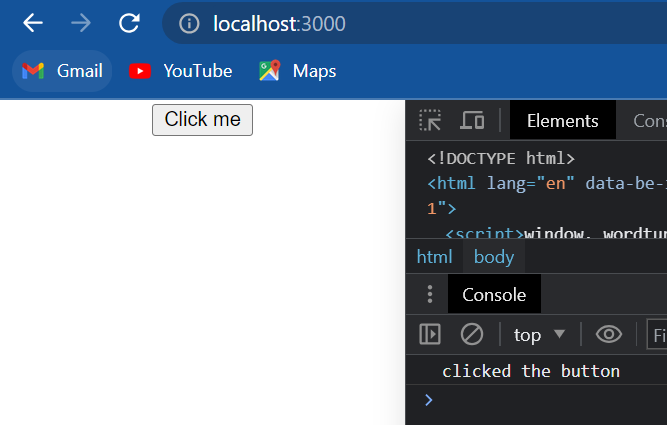
    )

  }

}

*export* default ClassClick

We have the **onClick** attribute and this is going to be equal to curly braces and within the curly braces the handler but in a class component methods will be accessed using this keyword so **this.onClick** handler.



Take a look at the browser when I click on the “click me” button we should see the message “clicked the button” in the console.

**Binding event handlers**

The reason we bind event handlers in react is purely because of the way **this** keyword works in JavaScript it is not because of how react works.

What I want to achieve is on the click of a button we simply change a message which is part of the component state. So first I am going to create a new file called *EventBind.js* within the file I will use the react snippet rce to create a class component. I will include the component in the app component(*App.js*).

Constructor react snippet: rconst

import React, { Component } from 'react'

class EventBind *extends* Component {

    constructor(props) {

      super(props)

      this.state = {

         message:'Hello'

      }

    }

    clickHandler()

    {

        this.setState({message:"good bye!"})

    }

  render() {

    return (

      <div>

        <div>{this.state.message}</div>

        <button *onClick*={this.clickHandler}>

            Click

        </button>

      </div>

    )

  }

}

*export* default EventBind



I save this and go back to the browser and click on the button we can see that our application breaks and if we take a closer look at the error it says “cannot read property setState of undefined”.

So go back and comment out the set state method and instead simply log to the console that this keyword now.

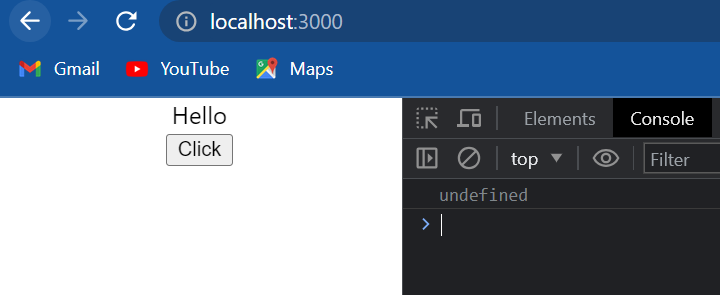
 clickHandler()

    {

*// this.setState({message:"good bye!"})*

        console.log(this)

    }



if we go back to the browser and click on the button we can see that undefined is logged in the console. So this keyword within our event handler is undefined and let me tell we this is the typical behavior in JavaScript.

this keyword is undefined in an event handler and that is the reason event binding is necessary in react class components. now there are a number of ways to bind event handlers in react.

The first option we have is to use the **bind** keyword and bind the handler in the render() method so on click **this.clickHandler.bind(this)** and we pass in this keyword and also uncomment setState().

import React, { Component } from 'react'

class EventBind *extends* Component {

    constructor(props) {

      super(props)

      this.state = {

         message:'Hello'

      }

    }

    clickHandler()

    {

        this.setState({message:"good bye!"})

        console.log(this)

    }

  render() {

    return (

      <div>

        <div>{this.state.message}</div>

        <button *onClick*={this.clickHandler.bind(this)}>

            Click

        </button>

      </div>

    )

  }

}

*export* default EventBind



let's save the file and go back to the browser now when I click on the button we can see that the setState() method works without any errors the message now reads “goodbye” and this keyword as we can see in the console is not undefined anymore. it refers to the event bind component instance.

Although this option works perfectly fine every update to the state will cause the component to rerender. this in turn will generate a brand new event handler on every render. although the impact on performance is not severe in small applications, it could be troublesome in large applications and components that contain nested children components. So approach one binding in the render method.

The second approach is to use arrow functions in the render() method. So the second one the arrow function approach is simply calling the event handler in the arrow function body so on click and within curly braces we are going to have an arrow function.

render() {

    return (

      <div>

        <div>{this.state.message}</div>

        <button *onClick*={()=>this.clickHandler()}>Click</button>

      </div>

    )

  }

Notice that we don't need curly braces or the return keyword for the arrow function body and that is because the function body is a single statement. also we can notice that we are calling the eventHandler() and returning that value that is why parentheses is required in this approach. if I now save the file and take a look at the browser and click on the button we can see that the message changes from hello to goodbye.



so the second approach also works as expected but similar to the first approach this also has performance implications in some scenarios. so second approach is using arrow functions in the render method.

Next let's discuss approach number three and this is the approach we are going to see in most of the cases and it is also the approach in the official react documentation. this approach deals with binding the event handler in the constructor as opposed to binding in the render() method.

import React, { Component } from 'react'

class EventBind *extends* Component {

    constructor(props)

    {

        super(props)

        this.state = {message:'Hello'}

        this.clickHandler=this.clickHandler.bind(this)

    }

    clickHandler()

    {

        this.setState({message:"good bye!"})

        console.log(this)

    }

    render()

    {

    return (

        <div>

            <div>{this.state.message}</div>

            <button *onClick*={this.clickHandler}>Click</button>

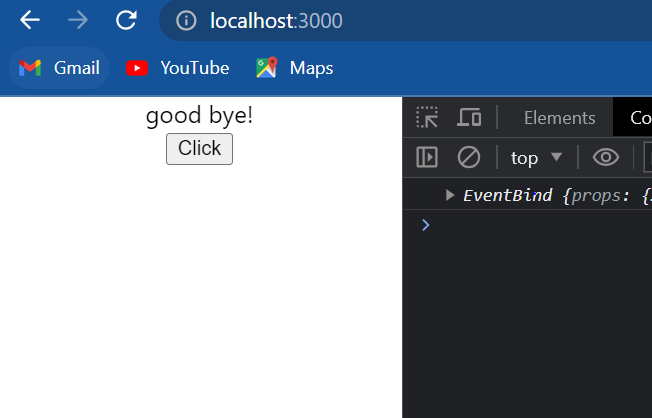
        </div>

    )

    }

}

*export* default EventBind



Go back to the browser here the console and click on the button we should be able to see the message change from hello to goodbye. So that is the third approach binding in the class constructor.

Because the **binding happens once in the constructor this is better compared to binding in the render method**. so approach number three binding in the constructor.

Final approach is to use an arrow function as a class property. Basically change the way we define our method in the class.

import React, { Component } from 'react'

class EventBind *extends* Component {

    constructor(props)

    {

        super(props)

        this.state = {message:'Hello'}

    }

    clickHandler=()=>

    {

        this.setState({message:'good bye!'})

    }

    render()

    {

    return (

        <div>

            <div>{this.state.message}</div>

            <button *onClick*={this.clickHandler}>Click</button>

        </div>

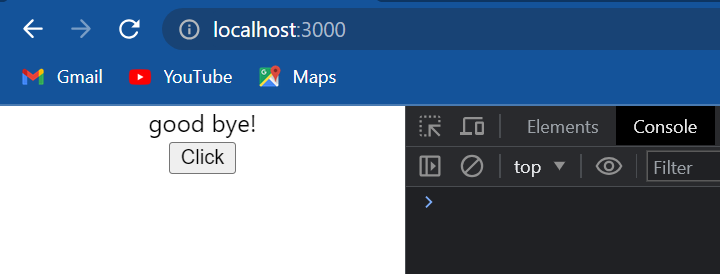
    )

    }

}

*export* default EventBind

In the new approach clickHandler is going to be equal to an arrow function and within the other function this dot setState() message set to goodbye. if we save the file and take a look at the browser we can see that the message changes on button click. this keyword has the expected value within the handler method.



So those are the four approaches: **binding in render**, **arrow function in render**, **binding in the class constructor** and finally **class property as arrow functions.**

* The first approach is something we might not want to use because of performance implications.
* The second approach although is similar to approach number one is probably the easiest way to pass parameters. if our code doesn't involve re-rendering nested children components this approach is still a viable option.

**React documentation suggests either approach number three or approach number four.**

* As approach number four is still an experimental feature.
* Approach number three which is binding in the constructor is our best option right now.

However create-react-app does support the class property approach so there is nothing wrong with using approach number four. In fact when the class property approach becomes an accepted feature it would probably be the go-to approach.

I will try to stick to approach number three or four for most parts in this series and when we need to pass parameters I might resort to approach number two for its simplicity.

**Method as props**

We have seen how a parent component can pass down props to its children components in the data. Any data in the parent component when passed as props is available in the children components.

Now what if a child component wanted to communicate with the parent component. Strangely we still use props, but this time we pass in a reference to a method as props to the child component.

I am going to create a new file called *ParentComponent.js*

alert("Hello"+this.state.parentName)

This is fine but since we are using ES6 I would like to use “**template literals**”. So instead of the regular concatenation (alert("Hello"+this.state.parentName)) we are going to have **backticks**(key right below Escape key) and within backticks ‘hello’ followed by the dollar sign and curly braces and within the curly braces this.state.parentName.

Now please understand **backticks and dollar curly braces is a feature in ES6 and is not a feature specific to react**. Also since we are using this keyword in the method we need to bind this method.

import React, { Component } from 'react'

class ParentComponent *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {parentName:'Parent'}

        this.greetParent=this.greetParent.bind(this)

    }

    greetParent()

    {

*// alert("Hello"+this.state.parentName)*

        alert(`Hello ${this.state.parentName}`)

    }

    render()

    {

        return (

            <div/>

        )

    }

}

*export* default ParentComponent

I am going to leave the JSX empty for now and get back.

Next I will create another file called *ChildComponent.js* and within the file I will create a functional component. we don't make use of state in this component and that is the reason I am sticking to the more simpler functional component. In the JSX I will add a button that says greet parent

Now back in the parent component (*ParentComponent.js)* , I will include the child component in the render() method (<ChildComponent/>). Make sure to import the child component at the top(*ParentComponent.js*). Next in app component(*App.js*) I will include the parent component in the render() method <ParentComponent/> and again make sure to import the component at the top.

***ParentComponent.js***

import React, { Component } from 'react'

import ChildComponent from './ChildComponent'

class ParentComponent *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {parentName:'Parent'}

        this.greetParent=this.greetParent.bind(this)

    }

    greetParent()

    {

*// alert("Hello"+this.state.parentName)*

        alert(`Hello ${this.state.parentName}`)

    }

    render()

    {

        return (

            <div>

                <ChildComponent/>

            </div>

        )

    }

}

*export* default ParentComponent

***ChildComponent.js***

import React from 'react'

function ChildComponent()

{

    return (

    <div>

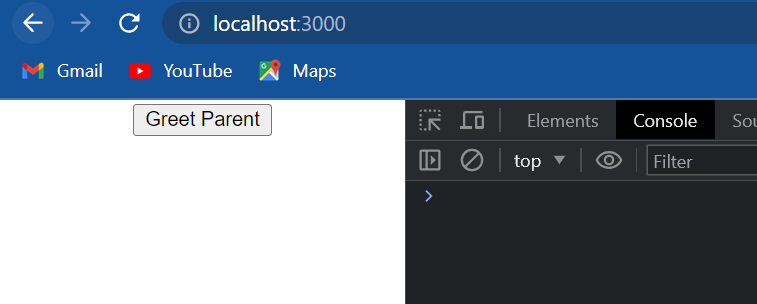
        <button>Greet Parent</button>

    </div>

    )

}

*export* default ChildComponent



Now if I take a look at the browser we should be able to see the button greet parent if I click on the button though nothing happens right now but what I want is when we click on the button in the child component I want to execute the method defined in the parent component. Basically the child component calls a parent component method and as I mentioned already this is achieved using props. The only difference this time is we passed the method itself as a prop to the child component.

Go back to *ParentComponent.js* and in parent component JSX we add an attribute to the child component and let's call the attribute **greetHandler** and to this we assign the greetParent() method so this is going to be equal to **this.greetParent** again make sure we don't add the parentheses. we are passing a reference to the greetParent() method as a prop called greetHandler. we can now grab hold of this reference in our child component. In the child component(*ChildComponent.js*) on click of this button we simply call **props.greetHandler**. Make sure to add props as a parameter.

***ParentComponent.js***

import React, { Component } from 'react'

import ChildComponent from './ChildComponent'

class ParentComponent *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {parentName:'Parent'}

        this.greetParent=this.greetParent.bind(this)

    }

    greetParent()

    {

*// alert("Hello"+this.state.parentName)*

        alert(`Hello ${this.state.parentName}`)

    }

    render()

    {

        return (

            <div>

                <ChildComponent *greetHandler*={this.greetParent}/>

            </div>

        )

    }

}

*export* default ParentComponent

***ChildComponent.js***

import React from 'react'

function ChildComponent(props)

{

    return (

    <div>

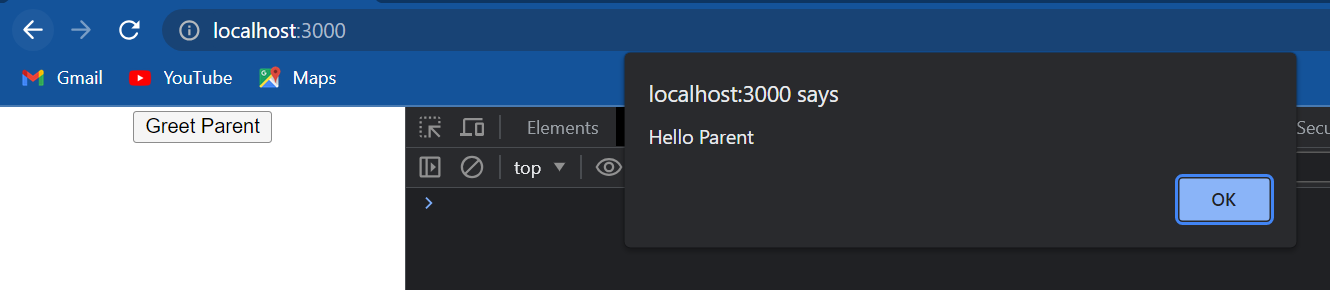
        <button *onClick*={props.greetHandler}>Greet Parent</button>

    </div>

    )

}

*export* default ChildComponent



let's go back to the browser click on the button and we can see the alert hello parent.

So we have successfully called a method in the parent component from a button in the child component by passing the method as props to the child component. This is something we're going to do quite often in our projects so please do make a note of it.

Now let's see how to pass a parameter when calling the parent method from the child component and this is where an arrow function in the return statement is really useful. Arrow function syntax is the simplest way to pass parameters from the child component to the parent component. So let me convert the onclick handler into an arrow function.

<button *onClick*={()=>props.greetHandler()}>Greet Parent</button>

Now we can pass in any number of parameters to the greetHandler(). To keep it simple let's pass one parameter which is the string “child “. Next let's incorporate this parameter in the greetParent() method. In greetParent() method we are going to receive a parameter **childName**.

***ParentComponent.js***

import React, { Component } from 'react'

import ChildComponent from './ChildComponent'

class ParentComponent *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {parentName:'Parent'}

        this.greetParent=this.greetParent.bind(this)

    }

    greetParent(childName)

    {

*// alert("Hello"+this.state.parentName)*

        alert(`Hello ${this.state.parentName} from ${childName}`)

    }

    render()

    {

        return (

            <div>

                <ChildComponent *greetHandler*={this.greetParent}/>

            </div>

        )

    }

}

*export* default ParentComponent

***ChildComponent.js***

import React from 'react'

function ChildComponent(props)

{

    return (

    <div>

        <button *onClick*={()=>props.greetHandler('child')}>Greet Parent</button>

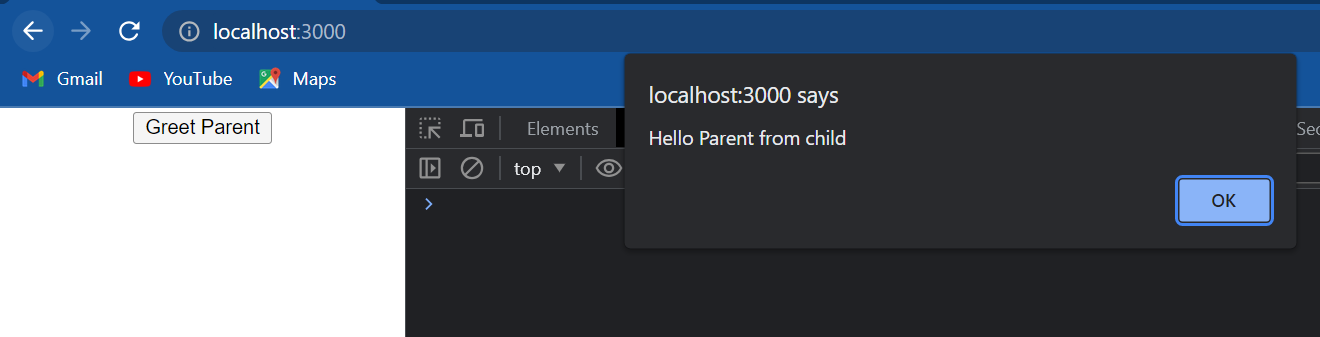
    </div>

    )

}

*export* default ChildComponent

if we save the files go to the browser and click on the greet pattern button we should see the alert “hello parent from child“.



We have successfully passed a parameter from the child to the parent. So this is pretty much how we pass methods as props in react components.

* In the parent component define the method. On the child component tag(<ChildComponent>) past the method as a prop.
* In the child component access the method using the props object. if at all we have to pass a parameter use the arrow functions syntax.

By the way we can destructure the props object in this functional component but since we have just this one prop I'm going to leave it as it is.

**Conditional rendering**

When we are building react applications we may often need to show or hide some HTML based on a certain condition. Luckily conditional rendering in react works the same way conditions work in JavaScript. we have four different approaches and we will take a detailed look at each one of them. we have if/else, element variables, ternary conditional operator and short-circuit operator.

let's begin with the if-else approach:

first I am going to create a new file called *UserGreeting.js* within the file I'm going to create a class component and in the JSX I will simply return welcome vishwas in the app component I will include the UserGreeting component again make sure to import it at the top now if I save the files and take a look at the browser we should be able to see the message “welcome vishwas”. let's go back to the UserGreeting component and make some changes.

I'm going to begin by adding a constructor with the snippet our Const within the constructor call super and then define the state I am going to create one state property called **isLoggedIn** and initialize it to false; in the JSX i am going to add another message that says “welcome guests”.

import React, { Component } from 'react'

class UserGreeting *extends* Component

{

    constructor(props) {

      super(props)

      this.state = {

         isLoggedIn:false

      }

    }

    render()

    {

    return (

        <div>

            <div>welcome vishwas</div>

            <div>welcome Guest</div>

        </div>

    )

    }

}

*export* default UserGreeting

now what I want is the message to be conditionally rendered based on the **isLoggedIn** state. if I am logged in the message “welcome vishwas” should be displayed and if I am not logged in the message “welcome guests” should be displayed.

let's see how to achieve that with the **first approach** that is using the if-else condition. In the render method let's add the if-else condition.

import React, { Component } from 'react'

class UserGreeting *extends* Component

{

    constructor(props) {

      super(props)

      this.state = {

         isLoggedIn:false

      }

    }

    render()

    {

        if(this.state.isLoggedIn)

        {

            return <div>welcome vishwash</div>

        }

        else

        {

            return <div>welcome guest</div>

        }

*// return (*

*//     <div>*

*//         <div>welcome vishwas</div>*

*//         <div>welcome Guest</div>*

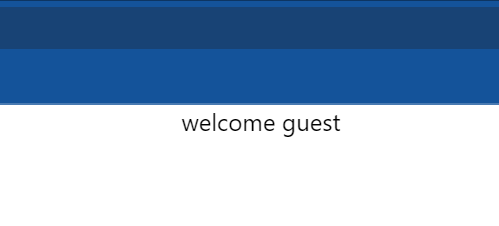
*//     </div>*

*// )*

    }

}

*export* default UserGreeting



if we take a look at the browser we can see the message “welcome guest” is displayed and that is because **isLoggedIn** is set to false. if I now change it to true and take a look at the browser we can see that the message “welcome vishwas” is displayed.

now we might be thinking there's a lot of repetition and the render method looks crowded can we not simply use the f/else condition on the message being displayed. the answer is no; if/else statements don't work inside the JSX that is because JSX is just syntactic sugar for function calls and object construction adding if else statements within the JSX is not valid; that is the reason we have if else statements outside the JSX and the entire return statement containing the JSX is placed inside the if or else block.

Now a better approach is the **second approach** of using element variables. in this approach we use JavaScript variables to store elements. this will also help we conditionally render the entire component or only a part of the component as well. let's see how.

I'm going to begin by commenting the if-else blocks. next I'm going to declare a variable inside the render method.

import React, { Component } from 'react'

class UserGreeting *extends* Component

{

    constructor(props) {

      super(props)

      this.state = {

         isLoggedIn:true

      }

    }

    render()

    {

*// approach: second*

        let message

        if(this.state.isLoggedIn)

        {

            message=<div>welcome vishwas</div>

        }

        else

        {

            message=<div>welcome guest</div>

        }

        return <div>{message}</div>

*// approach: first*

*// if(this.state.isLoggedIn)*

*// {*

*//     return <div>welcome vishwash</div>*

*// }*

*// else*

*// {*

*//     return <div>welcome guest</div>*

*// }*

*// return (*

*//     <div>*

*//         <div>welcome vishwas</div>*

*//         <div>welcome Guest</div>*

*//     </div>*

*// )*

    }

}

*export* default UserGreeting



if we now save the file and take a look at the browser we can see that we have the same result “welcome vishwas” is displayed change is logged in to false and the message “welcome guest” is displayed. So message is the variable which stores the element to be rendered and hence this is the element variable approach.

Now although this approach looks much better the **third approach** is even more simpler and this approach uses the ternary conditional operator the benefit of this approach is that it can be used inside the JSX.

so I'm going to comment out what we have done with the second approach and now within the render method I'm going to add the return statement and within parentheses we use the conditional operator.

import React, { Component } from 'react'

class UserGreeting *extends* Component

{

    constructor(props) {

      super(props)

      this.state = {

         isLoggedIn:false

      }

    }

    render()

    {

*// approach: third*

        return (

            this.state.isLoggedIn ? (<div>welcome vishwas</div>):(<div>welcome guest</div>)

        )

    }

}

*export* default UserGreeting

how does this work the first operator this dot state dot is logged in is evaluated to either true or false if it is true the second operator is returned in our case the div tag “welcome Vishwas” if the first operator turns out to be false then the third operator is returned in our case a div tag that says “welcome guest”.

if I go back to the browser we should be able to see the text “welcome guest” I change it to true and we should be able to see “welcome vishwas”. so this is the ternary conditional operator approach and is probably the approach we might want to follow most of the time keeps the code simple and readable.

Final or **fourth approach** is the short-circuit operator approach and this approach is just a specific case of this ternary operator approach that we have just learned. when we want to render either something or nothing we make use of the short-circuit operator. for example right now we return either “welcome vishwas” or “welcome guest” based on the **isLoggedIn** value.

now let's say if the user is logged in I want to display “welcome vishwas” and if the user is not logged in I want to render nothing onto the screen. so based on **isloggedIn** render either “welcome vishwas” or nothing. To do that we simply return:

import React, { Component } from 'react'

class UserGreeting *extends* Component

{

    constructor(props) {

      super(props)

      this.state = {

         isLoggedIn:false

      }

    }

    render()

    {

*// approach: fourth*

        return this.state.isLoggedIn && <div>welcome vishwas</div>

    }

}

*export* default UserGreeting

what happens here is the expression first evaluates the left hand side of the operator **this.state.isLoggedIn** if it is true it also evaluates the right hand side which in our case is the JSX that will be rendered in the browser. however if at all the left hand side evaluates to false the right hand side will never be evaluated as it doesn't affect the final value of the whole expression.

So these are the four approaches to conditionally render UI in react. For most of the cases I recommend we stick to either **conditional operator** or the **short-circuit operator** they tend to be much more clean and readable.

**List rendering**

when we build web applications a common scenario is to display a list of items. for example a list of names, a list of products, a list of courses, and so on. so what we want is to repeat some HTML for each item in the list. let's see how to do that in react.

One of the best things about react is that it relies heavily on the JavaScript language itself and not introduce new methods to manipulate and render data. if we are an expert in JavaScript we probably know that we can use the map method to quickly iterate over an array and return a new array with the desired changes applied.

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

so what the map method does is go over each element in the array and apply a transformation specified in the function body. so we have a list of numbers and we are able to return a list of numbers after applying some transformation using the map method this is pretty much the concept behind list rendering in react the only difference is instead of a transformation like multiplication we will be transforming into JSX. let's understand how that works with a simple example:

I am going to create a new file called *Namelist.js* and within the file I am going to use functional component within the component I am going to create an array of names and in the return statement I will add an <h2> tag to display each name using the array index.

import React from 'react'

function NameList()

{

    const names=['bruce','clark','diana']

    return (

    <div>

        <h2>{names[0]}</h2>

        <h2>{names[1]}</h2>

        <h2>{names[2]}</h2>

    </div>

    )

}

*export* default NameList

And in app component I will include the name list component make sure to import it at the top. now if I say both the files and take a look at the browser we should be able to see the list of names.



Although this works fine we know that this really is not a proper solution so let's see how to use the map method to render the list of names.

I'm going to go back in NameList component let me remove the <h2> elements and instead begin with curly braces. Remember the map method is JavaScript code which needs to be evaluated, in curly braces at the way to do that in JSX. So within curly braces **array.map()** is the syntax. So **names.map()** the map method takes in a function as an argument we will be using an arrow function the syntax is parameter followed by the fat arrow followed by the function body.

The parameter I am going to call it as **name** ; now the fat arrow symbol followed by the function body which will be the transformation. what we want is for every **name** return and <h2> tag with the **name** as the inner HTML so it's the tag with the **name** as the inner HTML. now once we start writing HTML we need to reuse curly braces if we have to evaluate the JavaScript expression. so even though we do already have a pair of outer curly braces we need another pair for the name parameter.

import React from 'react'

function NameList()

{

    const names=['bruce','clark','diana']

    return (

    <div>

        {

            names.map(name => <h2>{name}</h2>)

        }

    </div>

    )

}

*export* default NameList



if we now save the file and take a look at the browser we should still be able to see the list of names rendered correctly. one change we can do though if we wish to do this is keeping the return statement simple by moving out the list rendering logic.

so I'm going to declare a new constant const nameList and assign the result of the map operation. then in the return statement I will simply include the nameList in the JSX.

import React from 'react'

function NameList()

{

    const names=['bruce','clark','diana']

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

    return (

    <div>

        {nameList}

    </div>

    )

}

*export* default NameList

I format the code we can see that it becomes much more concise just a personal preference which we can follow as well.

The other thing is how simple are JSX is right now. Typically we're going to have a list of objects with a few properties that have to be rendered. In such cases it is always a good idea to refactor the JSX into a separate component and then use the component in the map method JSX. That sounded more complicated than it should be so let me help to understand with an example:

I am going to replace the **names** array with an array of **persons.** So for each person in the list we now need to render the **name**, **age**, and **skill.** we could simply render them in the map method with additional HTML.

let's start with that change **nameList** to **personList**, **names.map** to **person.map** and **name** to **person.** In the JSX we are going to have:

import React from 'react'

function NameList()

{

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

    const personList=persons.map(person => (<h2>

                                                I am {person.name}.

                                                I am {person.age} years old.

                                                I know {person.skill}.

                                            </h2>))

    return (

    <div>

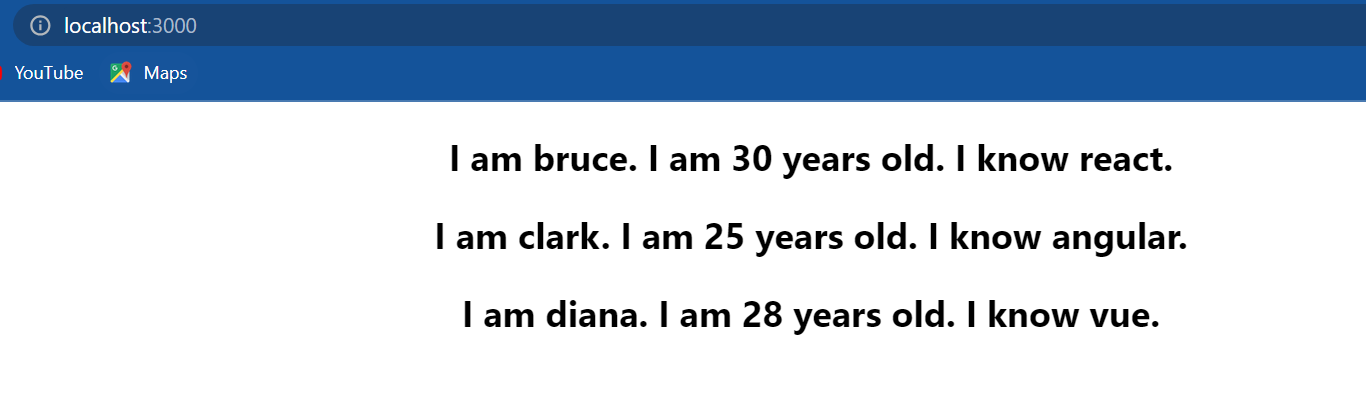
        {personList}

    </div>

    )

}

*export* default NameList



Take a look at the browser and it works as expected. Again there is nothing wrong with the code but the recommended way is to refactor the JSX into a separate component and that is really simple. I'm going to create a new file called *Person.js* within the file create a functional component remove the JSX from the list component and include it in *person.js* . So the <h2> tag I'm going to remove it from **personList** and include it in *Person.js* but how does this component know what the person data is. it doesn't right now. So let's pass the data down as props from the list component. So in the list component, **map()** method include the person component(<Person/>) in passing the person as a prop.

**Person.js**

import React from 'react'

function Person({person}) {

  return (

    <div>

        <h2>

            I am {person.name}.

            I am {person.age} years old.

            I know {person.skill}.

        </h2>

    </div>

  )

}

*export* default Person

**NameList.js**

import React from 'react'

import Person from './Person'

function NameList()

{

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

    const personList=persons.map(person => <Person *person*={person}/>)

    return (

    <div>

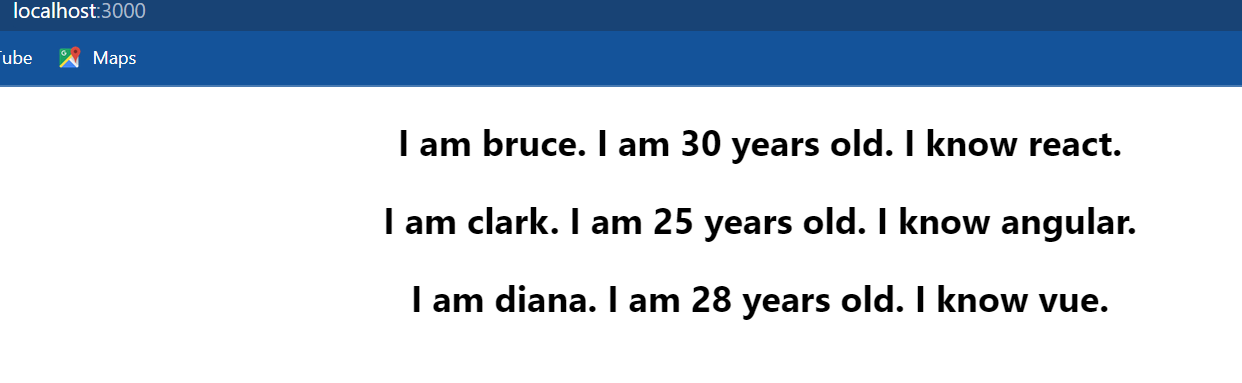
        {personList}

    </div>

    )

}

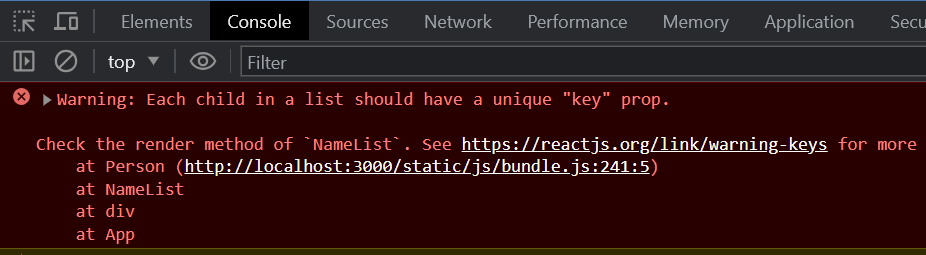
*export* default NameList



so after formatting we have person list is equal to persons dot map person as a parameter which is passed as the prop to the person component. finally in the person component we can destructure the prop right in the parameter so curly braces person if we now save the files and take a look at the browser we should still see the same output but this time we are writing better code.

The list component is only responsible for rendering the list and the person component is only responsible for rendering the person HTML and let me tell we this is also the pattern we commonly see when building applications that render lists of data. all right we are able to render a list of items in the browser without any problem at least that's what we think.

if I open the console we see the dreadful red text we have a warning in the console “each child in an array or iterator should have a unique key prop”.



what exactly is this unique key prop and what does this warning really mean let's understand it next.

**lists and keys**

We saw that when we render a list of items react through a waring in the console “each child in an array or iterator should have a unique key prop” or in simpler words react is telling us hey each item in the list rendered using the map operator should have a prop called key and the value to that prop should be unique within the list. so let's add that prop to each item in our list.

Go back to the NameList component within the map method I am going to add the **key** prop to this prop we need to assign a value that will be unique within the list. if I were to assign a string “unique” as the value(key=”unique”), what happens is every item rendered will have the same value for the **key** prop so the value is not unique within the list. Typically the ID of the item is a great choice for the **key** prop value. we can see that every person has an **id** property whose value is unique within the list so we can assign **person.id** as the value to the **key** prop. so key is going to be equal to within curly braces **person.id .**

import React from 'react'

import Person from './Person'

function NameList()

{

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

    const personList=persons.map(person => <Person *key*={person.id} *person*={person}/>)

    return (

    <div>

        {personList}

    </div>

    )

}

*export* default NameList

if we now save the file and take a look at the browser we should not see any warnings in the console and the list is still rendered as expected.



Now the key prop value need not be ID all the time. it could even be the name. it really can be anything as long as we know for sure it is unique within the list. for example I know for sure the name property for each person is unique within the list so I could have the key props value person.name (key=peron.name).

import React from 'react'

import Person from './Person'

function NameList()

{

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

    const personList=persons.map(person => <Person *key*={person.name} *person*={person}/>)

    return (

    <div>

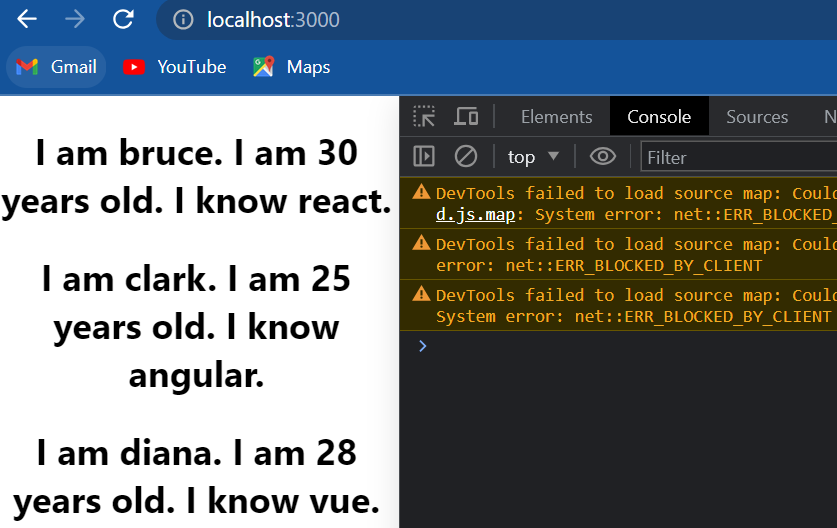
        {personList}

    </div>

    )

}

*export* default NameList



Now if we take a look at the browser it still works perfectly fine.

So the key prop is a special attribute we need to include when creating lists of elements. now an important point to keep in mind about key prop is **that they are not accessible in the child components.** for example if I go back to *Person.js* and destructure the key prop and try to render it in the JSX we can see that it is not rendered in the UI.

import React from 'react'

function Person({person,key}) {

  return (

    <div>

        <h2>

            {key}

            I am {person.name}.

            I am {person.age} years old.

            I know {person.skill}.

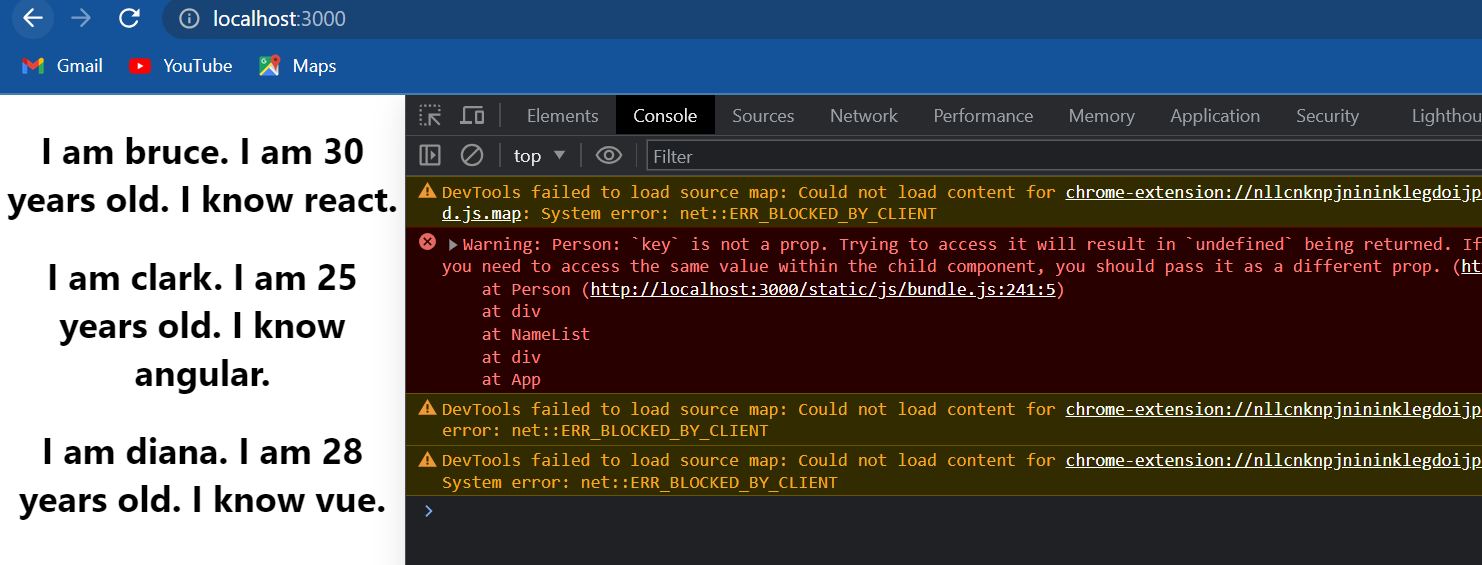
        </h2>

    </div>

  )

}

*export* default Person

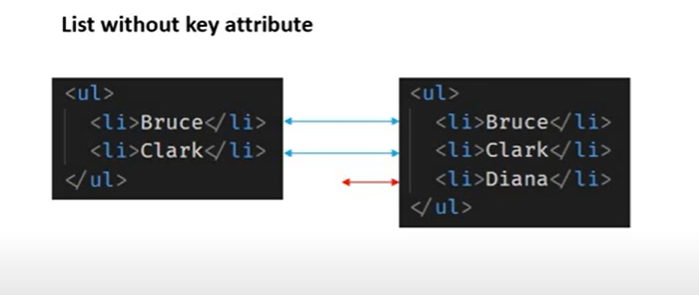


if we take a look at the console react throws a warning yet again in the “person component key is not a prop trying to access it will result in undefined being returned if we need to access the same value within the child component we should pass it as a different prop”.

So what react is telling us is hey key prop is something I need to render the list efficiently. if we are trying to pass down a value to be used in the child component pass it as a different prop; key prop is reserved. so in any of our components do not try to use the key prop to render any data.

Now we have understood what exactly is the key prop. now let us ask the question why do we need ? it well it turns out keys help react identify which items in a list have changed or added or removed and place a crucial role in handling UI updates efficiently.

let me show we how: consider a list of items we have two items to begin with Bruce and Clark now let's say we need to add another item at the end of the list.



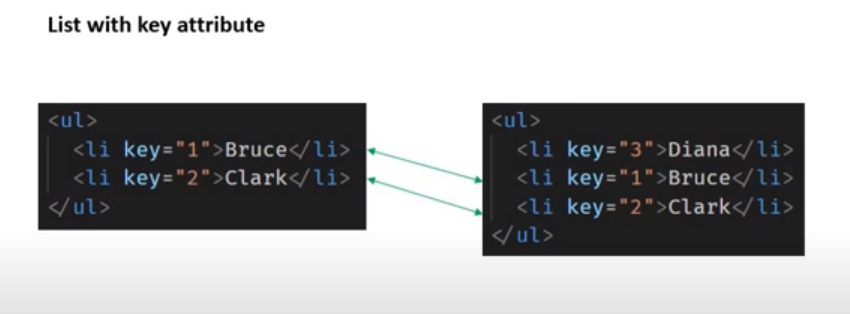
how react handles this update is react just iterates over both the lists at the same time and generates a mutation whenever there is a difference. in our example react will match the first items in each list and cease there is no difference it then matches the second two items and sees no difference and when it tries to match the third item it says that there is a difference and simply inserts the third item into the DOM tree. this way instead of tearing down the old tree and building the new tree from scratch, react simply updates the tree by inserting the item at the end of the list nothing else has to change.

now consider the same initial list of two items Bruce and Clark.



Again let's say we need to insert a new item to the list but this time we need to insert it at the beginning of the list. what happens in this situation is that when react iterates over both the lists and makes a comparison it finds out that every item is different. react will end up mutating every child instead of realizing it can keep the Bruce and Clark sub trees intact. this inefficiency can be a problem and in order to solve this issue react supports the key attribute.

when children have keys react uses the key to match the children in the original tree with the children in the subsequent tree.



now react knows that the element with key 3 is the new one and the elements with the keys 1 & 2 have just moved. so react will preserve the sub tree and simply insert the new item at the top of the list. this as it turns out is much more efficient.

Summarize:

* a key is a special string attribute we need to include when creating lists of elements
* keys give the elements a stable identity
* keys help react identify which items have changed or added or are removed this results in a much more efficient update of the user interface

Now everything works perfectly fine in our code but many a times the list to be rendered could be a simple array which doesn't have an **id** property that uniquely identifies the item what do we do in such a scenario let's understand it next.

**Index as key anti-pattern**

when reading articles are going through react tutorials we're often going to see a lot of examples where developers use the index of an element as the key for list rendering. it keeps the code simple and elegant and of course gets rid of the warning we see in the console.

Now we will see how to use the index as a key when rendering lists and also the problems we might face when used in the wrong scenario.

Go back to NameList component (*NameList.js*) and add back the **names** array (const names=['bruce','clark','diana']). let's render this list of names instead of the **persons** array. const name list is equal to names dot map and the map method takes in an arrow function let's call the parameter as **name** which represents the each item in each iteration within the function body. let's return an <h2> tag with the **name** as inner text ( const nameList=names.map(name => <h2>{name}</h2>) ) and the **nameList** component now returns name list instead of personList.

import React from 'react'

import Person from './Person'

function NameList()

{

    const names=['bruce','clark','diana']

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

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

    return (

    <div>

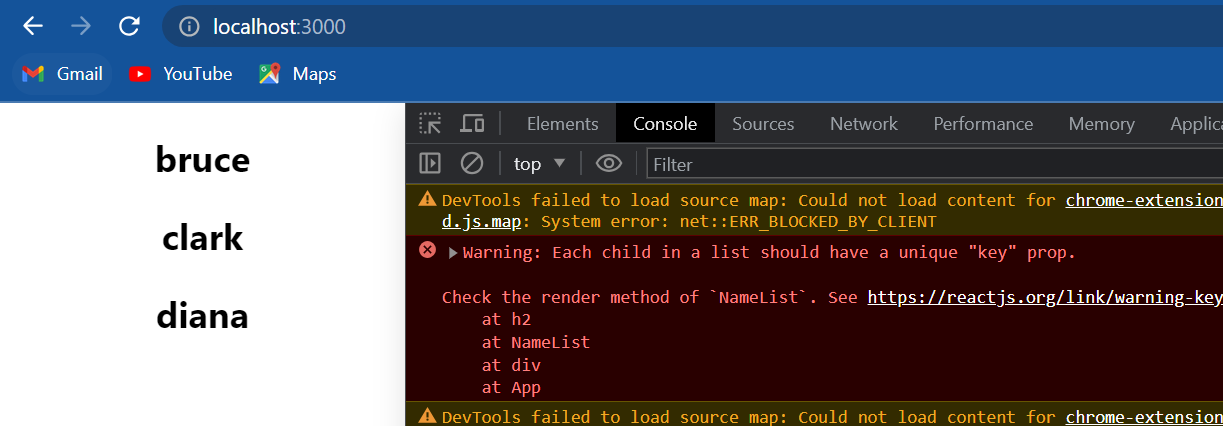
        {nameList}

    </div>

    )

}

*export* default NameList



If take a look at the browser we can see that we are rendering the list of names. if we open the console we of course have the error. but we do know how to fix this error, we need to add the **key** prop. There is one problem though which we did not have before we don't have an **id** that uniquely identifies each item in the array. we could use the **name** itself as the key. so in the **map**() method on the h2 element the **key** prop and this is going to be equal to name.

import React from 'react'

import Person from './Person'

function NameList()

{

    const names=['bruce','clark','diana']

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

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

    return (

    <div>

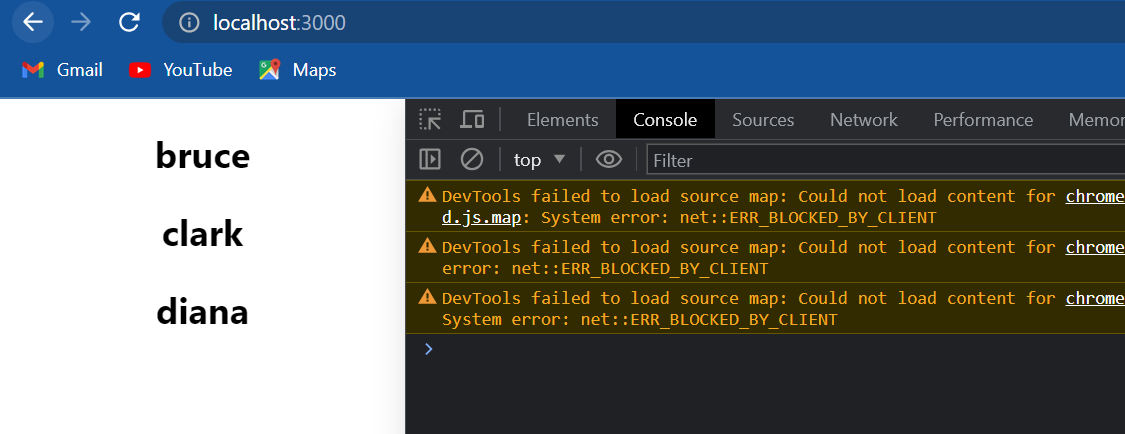
        {nameList}

    </div>

    )

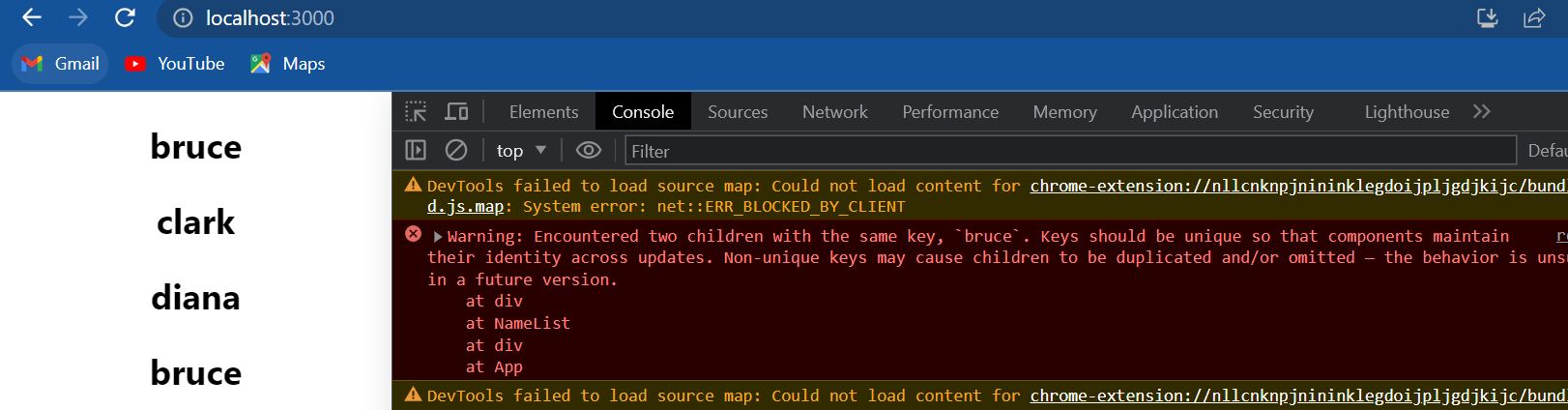
}

*export* default NameList



if I save the file now and take a look at the browser we can see that the error is gone.

But what if there is duplication of names. let's add Bruce again to the end of the list. Now the **name** is not unique anymore and if we remember from the warning the key prop has to be unique. if we take a look at the browser we should be able to see a new warning “encountered two children with the same key Bruce”



well it turns out the arrow function that is passed to the map method receives a second parameter which is the index of the element in the current iteration. let me show that to we the arrow function will now accept two parameters so we need parenthesis **name** and **index** map((name,index)) we can now use this index as the key when rendering the list so **key** is equal to **index**. let me also display the index ( {index} ) so as to get a better picture of how it works.

import React from 'react'

import Person from './Person'

function NameList()

{

    const names=['bruce','clark','diana','bruce']

    const persons=[{id:1,name:'bruce',age:30,skill:'react'},

                    {id:2,name:'clark',age:25,skill:'angular'},

                    {id:3,name:'diana',age:28,skill:'vue'}]

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

    return (

    <div>

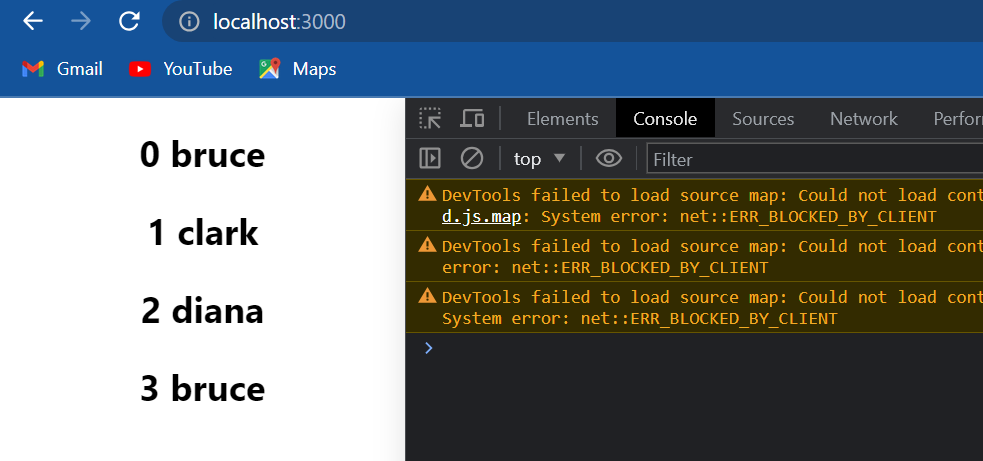
        {nameList}

    </div>

    )

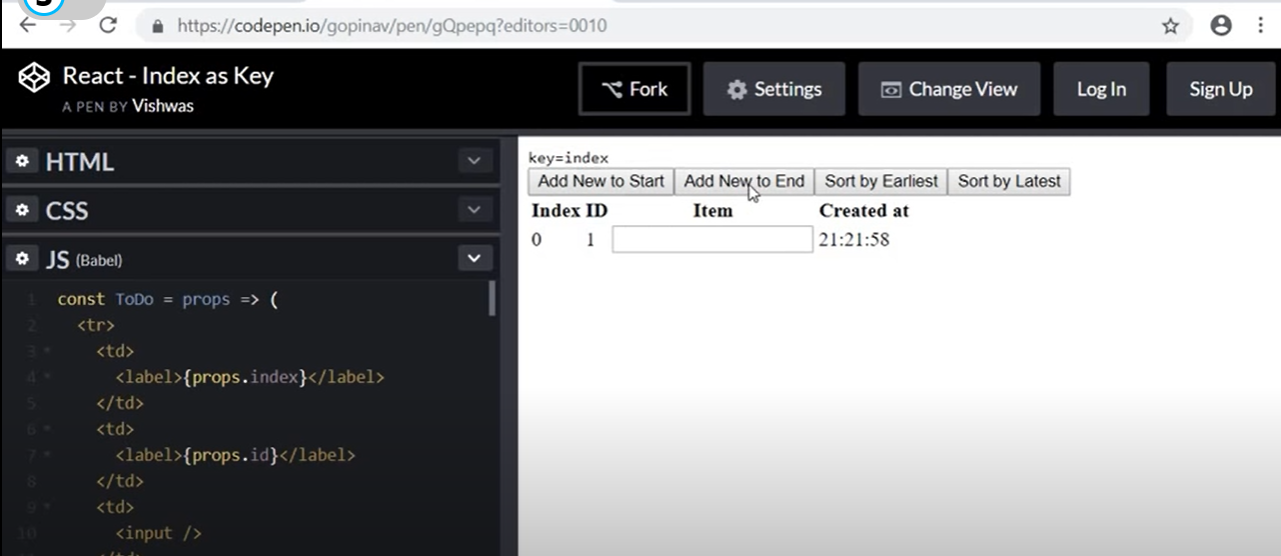
}

*export* default NameList



if we now save the file and take a look at the browser we can see the list displayed along with the index we can see that the index starts at zero and keeps incrementing for every element in the array and that way every item receives a unique key value. so this is how we can use the index value as a key for list rendering. the index is passed as a second parameter to the arrow function within the map method and that index is used as a value to the **key** prop.

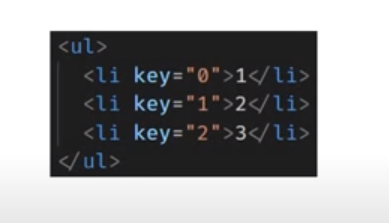
let me tell we though **using the index as a key will cause some serious UI issues in certain scenarios** and to demonstrate such scenarios I have modified a [codepen](https://codepen.io/gopinav/pen/gQpepq) example from the react docs the link will be in the description



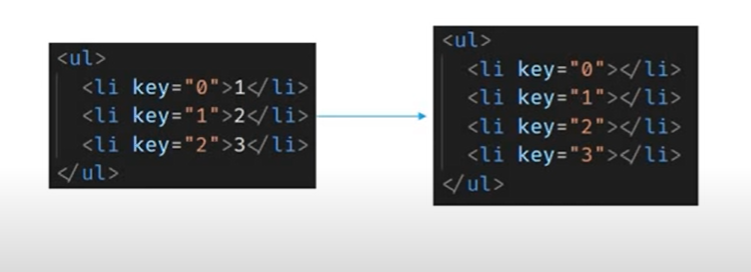
At the very top we have four buttons: add a new item to the start of the list, add a new item to the end of the list, sort the list by earliest created date, and finally sort by latest created date. Below that we have a table with the headings **index** which specifies the position of the item in the list and **ID** which is self incrementing, **item** which is an input where we can enter a value and the **created at** timestamp. on the list we are using the **map** operator the arrow function accepts two parameters to do which represents the item in iteration and the index of the item in the list the same index is used as a value to the key attribute for rendering the list. Now back in the UI for the first item I am going to add a value of 1. next I'm going to add a new item to the end of the list (click Add new to end button) we can see that a new row is inserted into the list. the index is now 1, ID is incremented to 2, the timestamp is also present. let me add a value of 2. similarly if I add another item to the end of the list we see the same behavior. everything works as expected.

Now what should happen if I add a new item to the start of the list for the new item I would expect the index to be 0 because it is at the beginning, the ID however should be incremented so it should be 4, the value should be empty and there should be a new timestamp. pay real close attention when I click on **add new to start** we can see that a new row is inserted but it doesn't quite work as we expected it to. we expected index to be 0; so that works we expected ID to be incremented to 4; so that works as well, the timestamp is new so that also works, but we expected the value to be blank or empty instead it has a value of 1; and the last item in the list which previously had a value of 3 is now empty. this is the problem of using the key as index.

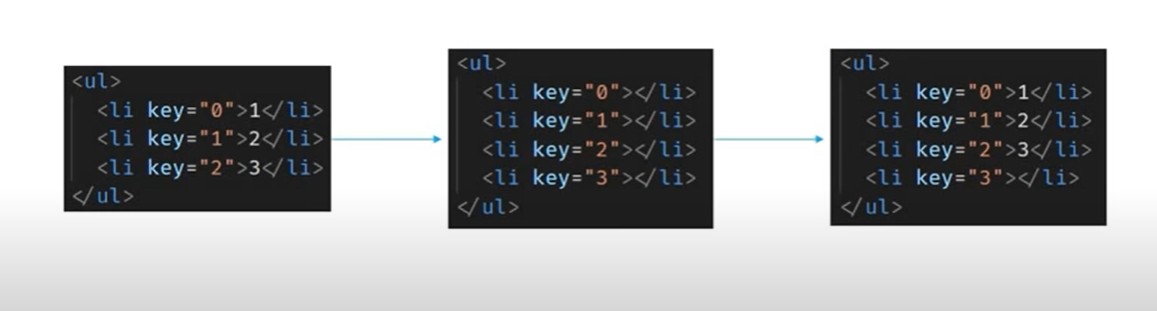
let me show we how react interprets the update. initially we have the three items with the index as keys.



For key zero there is a value of 1 for key 1 there is a value of 2 and similarly for key too there is a value of 3.



now when we insert a new item at the beginning, the new item gets an index of 0 which effectively creates a key with a value of 0 and that pushes the key value by 1 for the remaining elements.



when updating the UI react thinks that hey I already have the three elements with key equal to 0,1 and 2 let me reuse them and then insert that new item with key value of 4 oh and as it turns out the first three items had some values let me put them back. this is how even though we inserted a new item at the top it appears as though it was inserted at the bottom and the values are all messed up; and this is also the problem when trying to sort the list.

if I go back to [codepen](https://codepen.io/gopinav/pen/gQpepq) and refresh we should have a single item. I will add two more items to the end of the list so three items with value of 1,2 and 3. each of them with different time stamps. if I click on sort buy earliest the timestamp should be in ascending order and we can see that the list is sorted when I click on sort by latest the timestamp is in descending order but item values don't seem to be in the right order. irrespective of me sorting by earliest or latest the item values never change. this again is because of using the index as key when rendering the list.

now we might have a question if using the index as key results in such issues why do so many articles or videos do that. well it turns out we can use the index as a key provided our list satisfies certain conditions:

* **the first condition is that the items in wer list do not have a unique ID.** if the items do have a unique ID always go with that.
* **the second condition is that the list is a static list and will not change.** for example we are never adding items to the list or removing items from the list.
* **finally the list will never be reordered or filter.** for example sorting based on a column value or filtering based on user inputs.

when our list satisfies not just one but all the three conditions we can safely use the index as a key. most of the times we end up having a unique ID or the list will be satisfying the three conditions. if not we can try one of the NPM packages that generate a unique ID for a list or also try hashing out a unique value from one of the existing properties but the bottom line is try avoiding the use of index as a key and let that be our last resort. in fact react uses the index as the key if at all we don't specify the key prop and of course we have just learned how that can be a disaster in certain scenarios.

**Styling and basic CSS**

There are a couple of options to style react components: the **first one is regular CSS stylesheets,** the **second one is inline styling**, **third we have CSS modules,** **fourth we can use CSS in J's libraries** which work really well with react. As of now we will see only first three approaches.

let's start with the **first approach using regular CSS stylesheets**:

I'm going to create a new file called *Stylesheet.js* and create a functional component within the return statement I will simply add an h1 tag that says style sheets. I will also include this file in the app component, make sure to import the component at the top. To specify the CSS for this component I am going to create a new file called *myStyles.css* (inside components folder) and within the file I am going to create a class **primary** and set the **color** property to orange. now to be able to use this class in our component we will have to import it. so in Stylesheet component **import ‘./myStles.css’** . Now on the h1 tag we can specify the class using the **className** attribute class name is equal to **primary.**

import React from 'react'

import './myStyles.css'

function Stylesheet()

{

    return (

    <div>

        <h1 *className*='primary'>stylesheets</h1>

    </div>

    )

}

*export* default Stylesheet

**myStyles.css**

.primary{

    color:orange;

}



if we save the file and take a look at the browser the text should appear in orange.

**We can also conditionally apply a class based on props or state of the component.** for example: let me pass down a prop called **primary** and set it false.

**App.js**

import logo from './logo.svg';

import './App.css';

import Stylesheet from './components/Stylesheet'

function App() {

  return (

    <div *className*="App">

      <Stylesheet *primary*={false} />

    </div>

  );

}

*export* default App;

now I can use the props in the component has props as parameter and within the body let **className** is equal to **props.primary** then the conditional operator and if it is true we set class name to the string primary if not we set it to an empty string and we are going to use this **className** variable as the value to our **className** attribute.

**Stylesheet.js**

import React from 'react'

import './myStyles.css'

function Stylesheet(props)

{

    let className=props.primary ? 'primary':''

    return (

    <div>

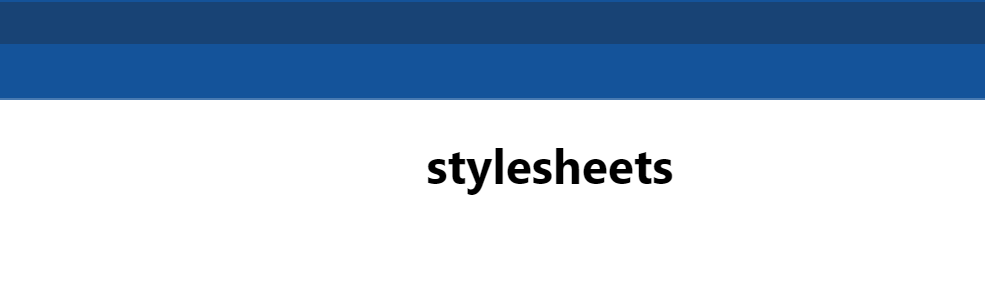
        <h1 *className*={className}>stylesheets</h1>

    </div>

    )

}

*export* default Stylesheet



if we now take a look at the browser we should see the text appear in black. if I go back to app component(*App.js*) pass the prop primary as true the text will be displayed in orange. So basically what we are doing is reading the value of the primary prop and if it is true we are setting a value of primary and if it is false we set it to an empty string and that className variable is assigned as a value to the className attribute.

Now if we want to specify multiple classes the simplest option is to use template literals. so in the CSS file (*myStyles.css*) I'm going to create a new class font-xl and I'm going to set the font-size to 72 pixels.

.primary{

    color:orange;

}

.font-xl{

    font-size: 72px;

}

Now back in the component we change the value of the **className** attribute to a template literal using backticks so this is going to be backticks and because className is a variable value we include it within dollar curly braces and then we can specify our second class font-xl.

import React from 'react'

import './myStyles.css'

function Stylesheet(props)

{

    let className=props.primary ? 'primary':''

    return (

    <div>

        <h1 *className*={`${className} font-xl`}>stylesheets</h1>

    </div>

    )

}

*export* default Stylesheet



if we now take a look at the browser we can see that the text is in orange and also the font size is larger both the classes have been applied. as an alternative to template literals there is also a library called class names which we can make use of. it tends to be a bit more cleaner but this is pretty much how we apply classes using regular style sheets and if necessary we can also conditionally apply the class based on props or state of the component.

Next up we have **inline styling**:

I'm going to create a new file called *Inline.js* so within the components folder and within the file I'm going to create a functional component within the return statement I will add a new heading that says “Inline”. Now let's style this heading. In react inline styles are not specified as a string, instead they are specified with an object whose key is the camel cased version of the style name and the value is usually a string. for example I can create a new object called heading ii steding is equal to an object we are going to set font size as 72 pixels. the key is the CSS property name but as we notice it has to be camel cased and the value is specified as a string. if we want to specify additional CSS properties simply add a comma and then list the property. let's go with color set to blue. now to apply this style inline we make use of the **style** attribute; on the h1 tag style attribute is going to be equal to the object which is heading. now let's include this component in app component (<Inline/>)

import React from 'react'

const heading={fontSize:'72px', color:'blue'}

function Inline()

{

    return (

    <div>

        <h1 *style*={heading}>Inline</h1>

    </div>

    )

}

*export* default Inline



and if we take a look at the browser we should see the text in line with the styling applied. so create an object and apply it to the style attribute, inline styling is pretty straightforward.

finally let's talk about **CSS modules.** CSS modules feature is available with react scripts version 2 or higher. so make sure we have updated our create react app package if it is below major version 2.

There is a file naming convention to be used for CSS modules with create react app. The file name must be suffixed with dot module dot CSS for example let me create two style sheets in the source folder so within the source (src) folder if **appStyles.css** is a regular stylesheet then **appStyles.module.css** is a CSS module stylesheet. In the regular style sheet that is appStyles.css I'm going to add a class **error** this is going to be colored red, in the module stylesheet I'm going to add a class **success** which has a color of green. Now in the app component I am going to import both the files **import ’.appStyles.css’** and for the module file **import styles from ’./appStyles.module.css’** . we can see that there is a difference in how we import a module stylesheet. now to use the error class from the regular style sheet I'm going to add an h1 tag that says error and set the class name attribute to error, to use the class from the module stylesheet I'm going to add another h1 tag success and the class name is going to be styles dot success so we access the class using the import statement variable name so error is our class name in the regular style sheet and success is a class name from the module style sheet.

**appStyles.css**

.error {

    color:red;

}

**appStyles.module.css**

.success {

    color:green;

}

**App.js**

import logo from './logo.svg';

import './App.css';

import Inline from './components/Inline';

import './appStyles.css'

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

function App() {

  return (

    <div *className*="App">

      <h1 *className*='error'>Error</h1>

      <h1 *className*={styles.success}>Success</h1>

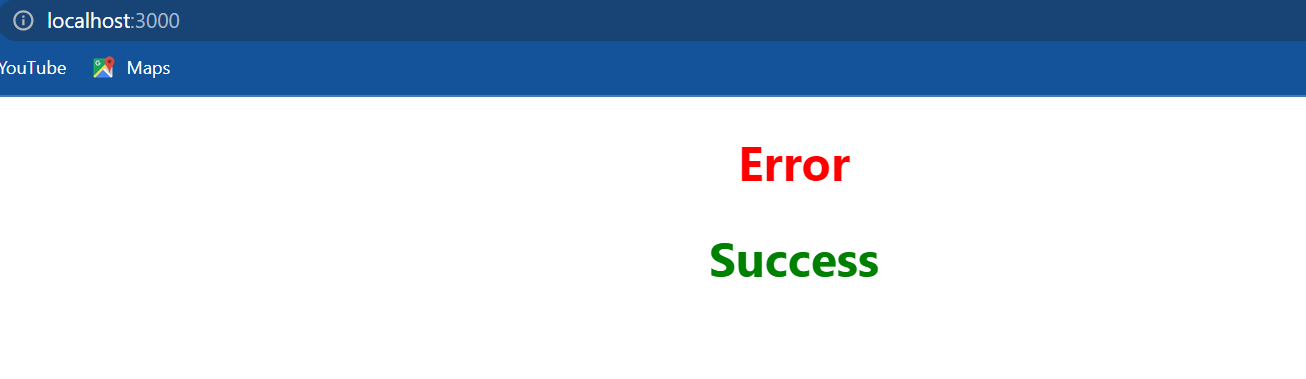
      {*/\* <Inline/> \*/*}

    </div>

  );

}

*export* default App;



if we now save all the files and take a look at the browser we should see both the classes being applied; error in red and success in green. now one advantage of using CSS modules is that the **classes are locally scoped by default.** for example if I copy the error heading ( <h1 className='error'>Error</h1> ) into the Inline component and uncomment the Inline component (<inline/>) we can see that it still works.

**Inline.js**

import React from 'react'

const heading={fontSize:'72px', color:'blue'}

function Inline()

{

    return (

    <div>

        <h1 *className*='error'>Error</h1>

        <h1 *style*={heading}>Inline</h1>

    </div>

    )

}

*export* default Inline

**App.js**

import logo from './logo.svg';

import './App.css';

import Inline from './components/Inline';

import './appStyles.css'

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

function App() {

  return (

    <div *className*="App">

      <h1 *className*='error'>Error</h1>

      <h1 *className*={styles.success}>Success</h1>

      <Inline/>

    </div>

  );

}

*export* default App;



the heading error is in red. so the CSS kind of applies to every child component as well. in our code it applies to app component and inline component which is the child component. this might lead to CSS conflicts. CSS modules on the other hand because **we reference the class names using the Styles variable it cannot be used in the children component** if I copy the success heading and paste it in the inline component:

import React from 'react'

const heading={fontSize:'72px', color:'blue'}

function Inline()

{

    return (

    <div>

        <h1 *className*={styles.success}>Success</h1>

        <h1 *className*='error'>Error</h1>

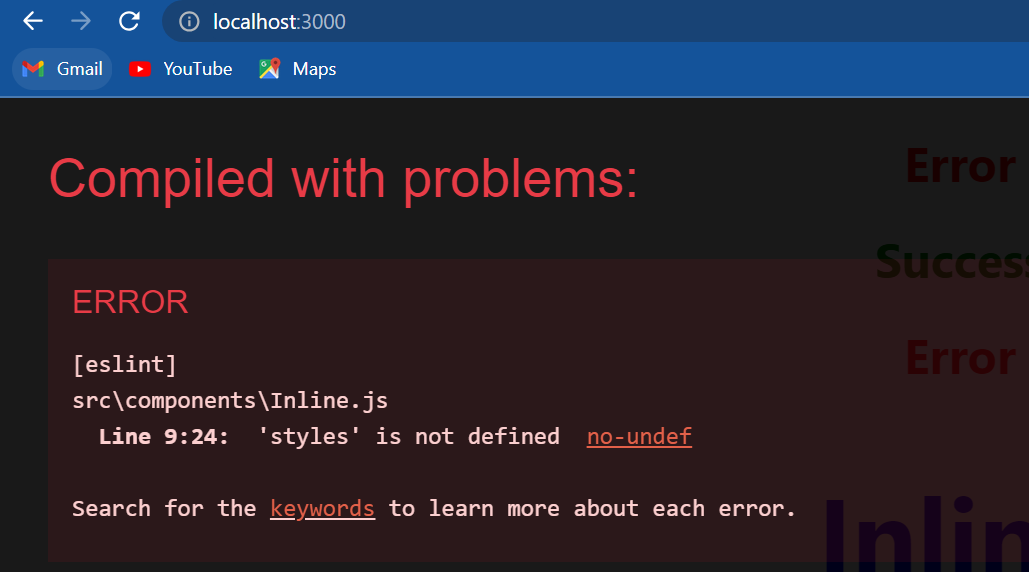
        <h1 *style*={heading}>Inline</h1>

    </div>

    )

}

*export* default Inline



we can see that the application doesn't compile it doesn't know what Styles is. So **we can't accidentally use a class that is defined for some other component.** so that is the CSS module approach.

**Basic of form handling**

We will see how to capture input from form elements like the input tag, text area tag, and also a select tag and have the data available for form submission. In regular HTML form elements like input text area and so on are responsible on their own to handle the user input and update their respective values. But what we want is for react to control the form elements instead. **Such form elements whose value is controlled by react is called a controlled component.**

Consider an input element, the input element can have a value; the input elements value can also change based on user interaction for example a user typing in their email address. now how do we deal with values that can change within the component viewState and setState. so in a controlled component the value of the input field is set to the **state** property. Next we have an unchanged event fired whenever there is a change in the input fields value, in the unchanged handler we use the **setState** method to update the state. when the state gets updated the render method is called and the new state is assigned as a value to the input element. so there is this cycle of setting the initial value from the **state**, propagating the changed value to the **state** and then back to the input field. React will always have access to the component state which reflects the updated values of the form element that state object can then be used to submit the form data when needed. so let's see how all of this translates the code.

I am going to create a new file called *Form.js* and within the file I'm going to create a new class component and add some text to the return statement “form component”, now in that component I'm going to include the form component. if we save the files and take a look at the browser we should be able to see the text “form component”.

now let's add our very first form control in the form component. I'm going to replace the existing JSX with a <form> tag within the form tag a label that says user name followed by the corresponding input.

import React, { Component } from 'react'

class Form *extends* Component

{

    render()

    {

        return (

        <form>

            <div>

                <label>Username</label>

                <input *type*='text'/>

            </div>

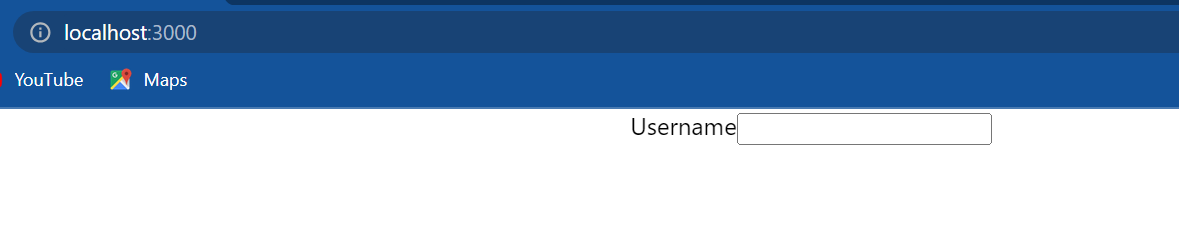
        </form>

        )

    }

}

*export* default Form



if we take a look at the browser we should be able to see the label and the input field. right now though the form is regular HTML, it is not a controlled react component. To convert this into a controlled component we need to follow two steps. The **first step is to create a component state that we control the value of the input element.** So within the component I am going to create a constructor, I'm going to add a new state property called **username** and initialize it to an empty string. Next I am going to assign this state property as the value of the input element. so input type is equal to ‘text’ and value is going to be equal to **this.state.username** .

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:''}

    }

    render()

    {

        return (

        <form>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={this.state.username} />

            </div>

        </form>

        )

    }

}

*export* default Form

Now if we go back to the browser and try to type in a username we're going to find out that the input doesn't reflect the changes, which brings us to the **second step handling the on change event**. So back in a component I'm going to create an **onChange** event handler. so on the input element let's listen to the on change event and assign a handler called **handleUsernameChange**. now let's define the method as a class property so **handleUsernameChange()** and the property is going to be equal to an arrow function. it just so happens that when we assign a handler to the unchanged event the event itself is passed as a parameter to the handler. So we have one parameter called **event,** from this event we can extract the value of the input element using **event.target.value** , so anytime we change the input value that value is captured using **event.target.value** . all we have to do is simply assign this captured value back to the state property and to update the state we will be using the **setState()** method. so within the function body **this.setState()** username set to e**event.target.value .**

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:''}

    }

    handleUsernameChange = (event)=>

    {

        this.setState({username:event.target.value})

    }

    render()

    {

        return (

        <form>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={this.state.username} *onChange*={this.handleUsernameChange} />

            </div>

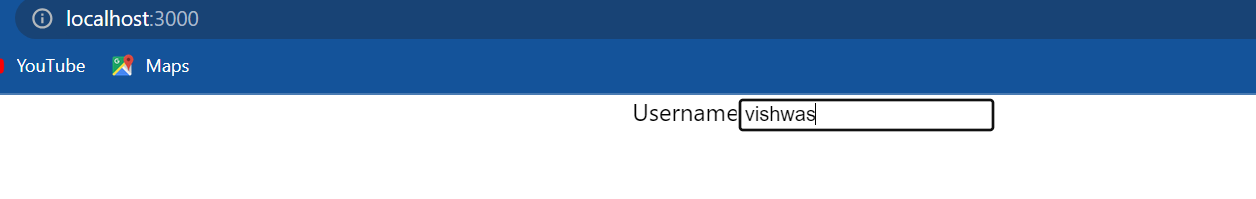
        </form>

        )

    }

}

*export* default Form



Now test it out in the browser. if I type in my name we can see that it works. the difference now though is that we are working with a controlled component and react state is the single source of truth for this input element. we can see that we have username as a state property which is supplied as a value to the value attribute of the input element. whenever there is a change that new value is propagated to handle user name change which sets back the state property username to the updated value and when the state is said the render method is called again and the new value is available to the value property and that is how we have a controlled component.

Next let's try to create controlled components for a text area as well as a select tag; this will also help we get used to the controlled component way of working with form elements. There are three simple steps: **add the element HTML**, **assign the component state to the element value** and **assign an unchanged handler that updates the state**.

Let's begin with **step 1 adding the HTML** ,so within the form I'm going to add a new div tag that has a label called “comments” and a text area. **Step 2 assign the component state to the element value.** so I'm going to create a new state **comments** which is initialized to an empty string and on the text area element value is equal to **this.state.comments** and final **step 3 is to assign the change handler that updates the state** , **onChange** is equal to **this.handleCommentsChange** . **handleCommentsChange**() is going to be a class property so is equal to an arrow function which takes event as its argument and we are going to call **this.setState** passing in **comments** with **event.target.value**

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:'', comments:''}

    }

    handleUsernameChange = (event)=>

    {

        this.setState({username:event.target.value})

    }

    handleCommentsChange = (event) =>

    {

        this.setState({comments:event.target.value})

    }

    render()

    {

        return (

        <form>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={this.state.username} *onChange*={this.handleUsernameChange} />

            </div>

            <div>

                <label>Comments</label>

                <textarea *value*={this.state.comments} *onChange*={this.handleCommentsChange}></textarea>

            </div>

        </form>

        )

    }

}

*export* default Form



if we save the file and take a look at the browser we should be able to see the text area type in something and it should work as expected.

now let's quickly take a look at the Select tag. Again we have three simple steps **step 1 add the HTML**  we are going to have a label that says topic and then the Select tag and as options we are going to have react angular and view so three options “react”, “angular” and “view” and the respective values in lowercase. **step 2 assign the component state to the element value**, so I'm going to create a new state property this is going to be **topic** and initially let's set it to “react”. on the select tag we can add the value attribute and this is going to be said to **this.state.topic** . **step 3 assign the change handler that updates the state** , so **onChange** is going to be equal to **this.handleTopicChange** which is going to be a class property similar to the previous handlers this.setState , topic is going to be equal to **event.target.value**

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:'', comments:'',topic:'react'}

    }

    handleUsernameChange = (event)=>

    {

        this.setState({username:event.target.value})

    }

    handleCommentsChange = (event) =>

    {

        this.setState({comments:event.target.value})

    }

    handleTopicChange =(event)=>

    {

        this.setState({topic:event.target.value})

    }

    render()

    {

        return (

        <form>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={this.state.username} *onChange*={this.handleUsernameChange} />

            </div>

            <div>

                <label>Comments</label>

                <textarea *value*={this.state.comments} *onChange*={this.handleCommentsChange}></textarea>

            </div>

            <div>

                <label>Topic</label>

                <select *value*={this.state.topic} *onChange*={this.handleTopicChange}>

                    <option *value*="react">React</option>

                    <option *value*="angular">Angular</option>

                    <option *value*="vue">Vue</option>

                </select>

            </div>

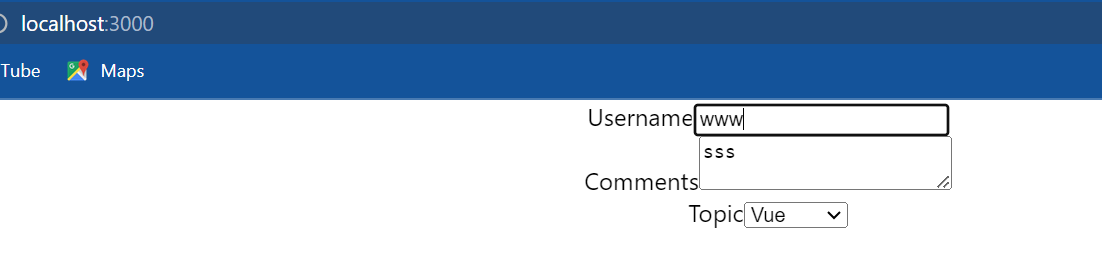
        </form>

        )

    }

}

*export* default Form



if we save the file and take a look at the browser we should be able to see the Select field by default react should be selected as the topic and we should also be able to select any of the other options. so there we go an input element a text area and a select tag all of them as controlled components in react. let's bind up this by understanding how we can submit this form data.

Go back to code and in the JSX I'm going to add a submit button. if we go back to the browser and click on the submit button we can see that the page refreshes. So right now the form has the default HTML form behavior of browsing to a new page when the submit button is clicked. A common approach however is to have a JavaScript function that handles the submission of the form and that method will also have access to the data that the user entered into the form. So back in code on the form tag we are going to assign a handler to the **onSubmit** event **this.handledSubmit** . Now we can define the handler **handleSubmit()** and this is going to be equal to an arrow function which is going to receive event as an argument and to retrieve the form data that the user has entered into the form, we use the component state for now. let's simply alert those values alert within backticks dollar curly braces this dot state dot username, this dot state dot comments and this dot state dot topic

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:'', comments:'',topic:'react'}

    }

    handleUsernameChange = (event)=>

    {

        this.setState({username:event.target.value})

    }

    handleCommentsChange = (event) =>

    {

        this.setState({comments:event.target.value})

    }

    handleTopicChange =(event)=>

    {

        this.setState({topic:event.target.value})

    }

    handleSubmit= (event)=>

    {

        alert(`${this.state.username} ${this.state.comments} ${this.state.topic}`)

    }

    render()

    {

        return (

        <form *onSubmit*={this.handleSubmit}>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={this.state.username} *onChange*={this.handleUsernameChange} />

            </div>

            <div>

                <label>Comments</label>

                <textarea *value*={this.state.comments} *onChange*={this.handleCommentsChange}></textarea>

            </div>

            <div>

                <label>Topic</label>

                <select *value*={this.state.topic} *onChange*={this.handleTopicChange}>

                    <option *value*="react">React</option>

                    <option *value*="angular">Angular</option>

                    <option *value*="vue">Vue</option>

                </select>

            </div>

            <button *type*="submit">Submit</button>

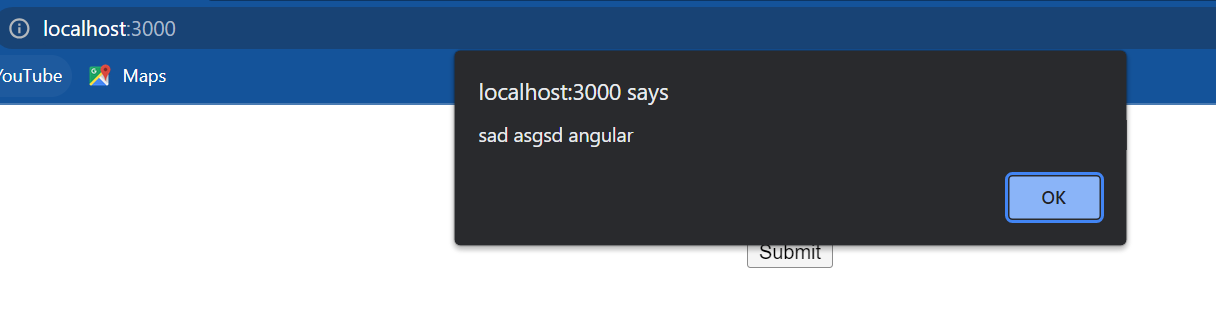
        </form>

        )

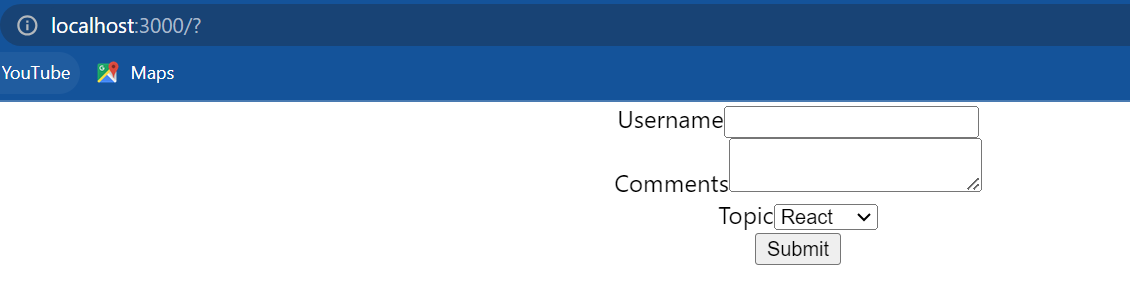
    }

}

*export* default Form



if we now go back to the browser fill in the details and click on submit we should be able to see the alert message with the form data. when I dismiss the alert though we can see that the page refreshes and the filled in data is lost.



To avoid this we simply add an **event.preventDefault().** this will prevent the default behavior of form submission so back in **handleSubmit()** after the alert statement **event.preventDefault().**

handleSubmit= (event)=>

    {

        alert(`${this.state.username} ${this.state.comments} ${this.state.topic}`)

        event.preventDefault()

    }

if I now go back to the browser fill in the details and click on submit we can see that we have the alert message when I click on OK the page doesn't refresh.

Now suppose we don't really have a form tag with which we can listen to the **onSubmit** event what we can do is create a button create an **onClick** event handler on the button and have the exact same body as the **onSubmit** event. now again type is equal to submit will give the user the ability to submit forms by hitting the enter key which is always good. so please do consider user experience when developing forms.

I will destructure the state properties in the render method so const username comments and topic from this dot state now I can remove this dot state from all three occurrences:

import React, { Component } from 'react'

class Form *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state ={username:'', comments:'',topic:'react'}

    }

    handleUsernameChange = (event)=>

    {

        this.setState({username:event.target.value})

    }

    handleCommentsChange = (event) =>

    {

        this.setState({comments:event.target.value})

    }

    handleTopicChange =(event)=>

    {

        this.setState({topic:event.target.value})

    }

    handleSubmit= (event)=>

    {

        alert(`${this.state.username} ${this.state.comments} ${this.state.topic}`)

        event.preventDefault()

    }

    render()

    {

        const {username,comments,topic}=this.state

        return (

        <form *onSubmit*={this.handleSubmit}>

            <div>

                <label>Username</label>

                <input *type*='text' *value*={username} *onChange*={this.handleUsernameChange} />

            </div>

            <div>

                <label>Comments</label>

                <textarea *value*={comments} *onChange*={this.handleCommentsChange}></textarea>

            </div>

            <div>

                <label>Topic</label>

                <select *value*={topic} *onChange*={this.handleTopicChange}>

                    <option *value*="react">React</option>

                    <option *value*="angular">Angular</option>

                    <option *value*="vue">Vue</option>

                </select>

            </div>

            <button *type*="submit">Submit</button>

        </form>

        )

    }

}

*export* default Form

**lifecycle method**

when we create a react component the component goes through several stages in its lifecycle. react provides us with built in methods that we can override at particular stages in the life cycle.

let's take a look at the lifecycle methods available for a class component. these methods do not exist on a functional component. With the new feature proposal of hooks there is the use effect hook which partially relates to the lifecycle hooks but for now our focus is going to be only on the lifecycle methods in a class component. the lifecycle methods have had their fair share of changes over the years the methods we will be discussing now are for react version 16.4 and above.

we can merely classify the methods into four phases: **mounting**, **updating,** and **unmounting** and **error handling.**

* The **mounting** lifecycle methods are called when an instance of a component is being created and inserted into the DOM.
* The **updating** lifecycle methods are called when a component is being rendered as a result of changes to either its props or state.
* The **unmounting** lifecycle method is called when a component is being removed from the DOM.
* The **error handling** methods are called when there is an error during rendering in a lifecycle method or in the constructor of any child component.
* During the mounting phase we have four methods: **constructor,** **static getDerivedStateFromProps, render** and **componentDidMount**.
* During the updating phase we have five methods: **static getDerivedStateFromProps**, **shouldComponentUpdate**, r**ender**, **getSnapshotBeforeUpdate** and **componentDidUpdate**
* During unmounting we just have one method: **componentWillUnmount** and
* For error handling we have two methods: **static methodGetDerivedStateFromError** and **componentDidCatch**

Knowing when to use which lifecycle method is crucial to properly understanding how to work with react.

**Component mounting lifecycle method**

Let's take a look at the mounting lifecycle methods that is methods which are called when an instance of a component is being created and inserted into the DOM and we will be going through them in the order that they are invoked.

So **first we have the constructor(props)**, it is a special function that will get called whenever a new component is created. now what is the constructor used for; well the constructor is perfect for initializing state or binding the event handlers to the class instance. what we should do in a constructor is cause side effects for example we should never make HTTP requests from within a constructor. there are also two important points to keep in mind when it comes to defining our own constructor: the **first one is that we have to call a special function called super.** this will call the base class constructor. in our component we have access to this dot props only after we have initially called super passing in the props as an argument. the **second point is that constructor is also the only place where we are expected to change or set the state by directly overwriting this dot state fields.** In all other scenarios we have to use this dot setState. So constructor set initial state, find event handlers and don't cause any side effects like making Ajax calls for example.

The **second method we have is a static getDerivedStateFromProps(props,state)**. The react documentation classifies this method as a rarely used lifecycle method. this method is basically used when the state of the component depends on changes in props over time. so let's say we have a component but the initial state of the component depends on the props being passed to the component in such a scenario we can use this method to set the state. since this method is a static method it does not have access to the **this** keyword so we cannot call this dot setState within this particular method instead we simply have to return an object that represents the new state of the component. again what we shouldn't do is cause side effects for example fetching data from an endpoint. so getDerivedStatefromPropsuse it when state depends on changes in props over time and also did not cause any side effects.

**The third method is the render() method** which by now we are quite familiar with. the render method is the only required method in a class component. in the render method we simply read this dot props and this dot state and return the JSX which describes the UI. the render function is a pure function for the given props and state it should always render the same UI. what we should not do here is changing the state of the component or interacting with the DOM or making any Ajax calls. since it is the render method JSX which also contains the other children components right after the parent render method the children components lifecycle methods are also executed. so render method read props in state and return the JSX.

the final and **fourth method which is part of the mounting phase is componentDidMount().** this method will be called only once in the whole lifecycle of a given component and it is invoked immediately after a component and all its children components have been rendered to the DOM. this method is the perfect place to cause side-effects; we can interact with the DOM or perform any Ajax calls to load data. so **componentDidMount** is a good place to perform initialization that requires DOM nodes and also load data by making Network requests.

Let's head over to the code and see their order of execution in code. I'm going to start off by creating a new file the file name is going to be *LifecycleA.js* and within the file I'm going to create a class component and include the component in app component. Now let's add the different lifecycle methods. **First we have the constructor()**. I'm going to create a new constructor; within the state I will simply add a property **name** initialized to “vishwas”. we don't really need the state here but we will make use of it in the next. what we do need to do is a **console.log** statement to understand the order of execution so within the constructor console.log “lifecycleA constructor”.

The **second method we have static method that is** **static getDerivedStateFromProps()** and this method gets access to props and state as parameters and has to return the new state or null. To keep it simple let's return **null** and to track the execution order I will add a **console.log** statement which says “lifecycleA getDerivedStateFromProps“. Make sure to include the static keyword or else the method will be ignored.

The **third method we have is the render()** method which is already present in the JSX. I will simply add the text lifecycleA and I will add the console.log statement to track the execution order “lifecycleA render”.

Now the final or **fourth method is componentDidMount().** I'm going to add it before the render() method componentDidMount() and within the body we will simply log to the console “lifecycleA componentDidMount”.

import React, { Component } from 'react'

class LifecycleA *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vishwas'}

        console.log("LifecycleA constructor")

    }

*static* getDerivedStateFromProps(props,state)

    {

        console.log("LifecycleA getDerivedStateFromProps")

        return null

    }

    componentDidMount()

    {

        console.log("LifecycleA componentDidMount")

    }

    render()

    {

        console.log("LifecycleA render")

        return (

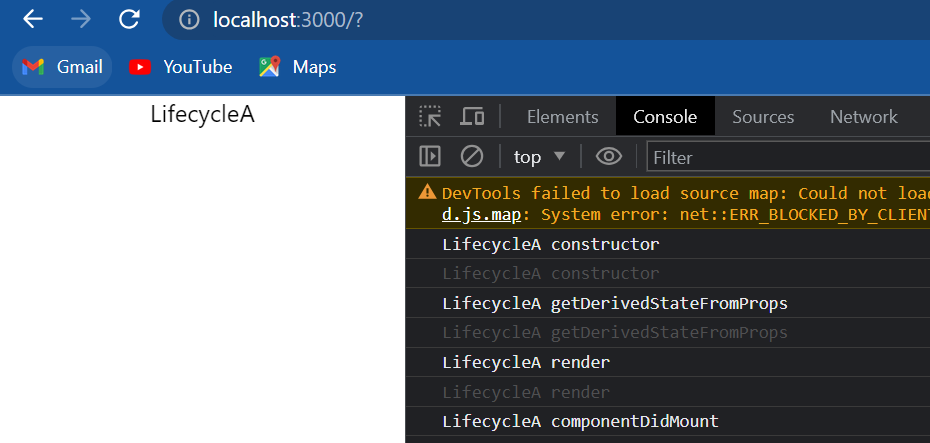
        <div>LifecycleA</div>

        )

    }

}

*export* default LifecycleA



Now let's save the file and head to the browser if I open the console we can see the order of execution first we have the constructor then we have get derived state from probes after that the render method and finally component did mount this is the order of execution during the mounting phase.

now let's see what happens when the component has a child component. I'm going to create a new file *LifecycleB.js* and I'm going to copy the content from *LifecycleA.js* and paste it in *LifecycleB.js* after that i will replace all occurrences of LifecycleA with LifecycleB. and in the LifecycleA component I will include LifecycleB component so parenthesis and closing div tag move this inside and then add LifecycleB component. So we have LifecycleA as the parent component and LifecycleB as the child component.

**LifecycleB.js**

import React, { Component } from 'react'

class LifecycleB *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vsishwas'}

        console.log("LifecycleB constructor")

    }

*static* getDerivedStateFromProps(props,state)

    {

        console.log("LifecycleB getDerivedStateFromProps")

        return null

    }

    componentDidMount()

    {

        console.log("LifecycleB componentDidMount")

    }

    render()

    {

        console.log("LifecycleB render")

        return (

        <div>LifecycleB</div>

        )

    }

}

*export* default LifecycleB

**LifecycleA.js**

import React, { Component } from 'react'

import LifecycleB from './LifecycleB'

class LifecycleA *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vsishwas'}

        console.log("LifecycleA constructor")

    }

*static* getDerivedStateFromProps(props,state)

    {

        console.log("LifecycleA getDerivedStateFromProps")

        return null

    }

    componentDidMount()

    {

        console.log("LifecycleA componentDidMount")

    }

    render()

    {

        console.log("LifecycleA render")

        return (

        <div>

            <div>LifecycleA</div>

            <LifecycleB/>

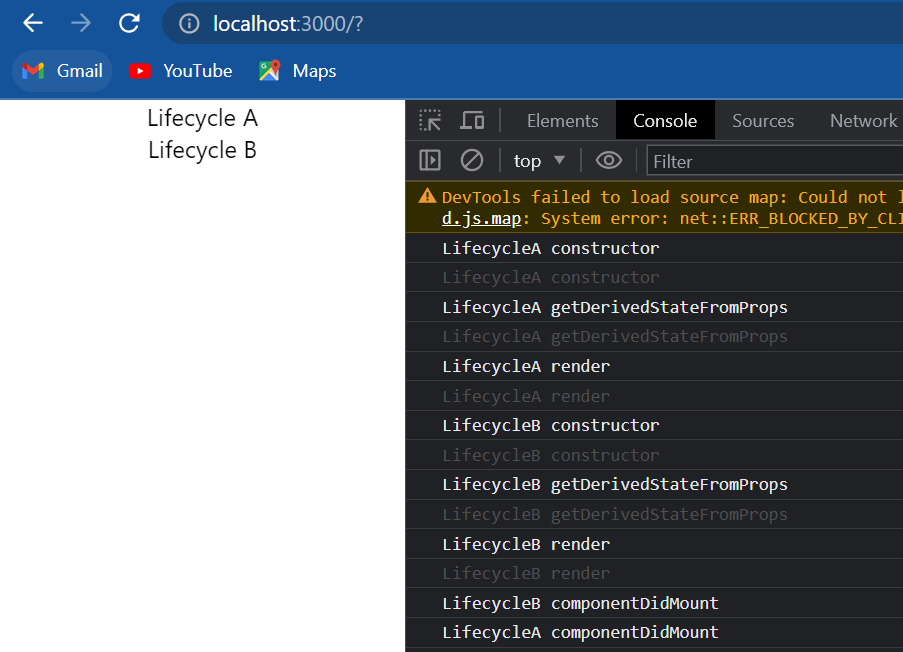
        </div>

        )

    }

}

*export* default LifecycleA



if we take a look at the console we can see that first we have “lifecycleA constructor” then “LifecycleA getDerivedStateFromProps’” that is followed by the render method of LifecycleA but after that we have the lifecycle methods of the child component which is LifecycleB. the constructor get derive state from props, render, and componentDidMount. Because the render method knows about the children components the execution passes on to the children after the parent component render method. the child component methods are then executed in order. Constructor, getDerivedStateFromProps, render and componentDidMount. after the child component has completely rendered the parent component componentDidMount is executed.

it is really important to know the order of execution because it helps we avoid writing code that might have unexpected behavior. So this is the order of execution during the mounting phase.

**Component updating lifecycle method**

let's take a look at the updating life cycle methods that is methods that are called when a component is being prerendered because of changes to either props or state. again we will be going through them in the order that they are invoked. we have a total of five methods but out of the five three fall into the category of rarely used methods. we will have ever go through all five of them.

The **first method** is a method we are now familiar with **getDerivedStatefromProps(props,state)**. this is a static method which receives props and state as its parameter and has to return either null or an object that represents the updated state of the component. this method is called every time a component is re-render and as we already discussed this method is used when the state depends on the props of the component. we should not cause any side effects and getDerivedStateFromProps. this method is one of the more rarely used methods in the updating phase.

The **second method** in the updating phase is the **shouldComponentUpdate(nextProps,nextState)** method. this method receives the updated props and state and the purpose of this method is clear from its name. it dictates if at all the component should re-render or not. **By default all class components will re-render whenever the props they receive or their state changes**. This method can prevent that default behavior by returning false. what we can do in this method is compare the existing props and state values with the next props and state values and return true or false to let react know whether the component should update or not. so **this method is basically for performance optimization**. what we should avoid is causing any side effects or calling the setState method. this lifecycle method again is classified as a rarely used lifecycle method in the official react documentation.

The **third method** is the all-too-familiar **render()** method we read this dot props and this dot state and return the JSX which describes the UI. avoid changing the state or interacting with the DOM in the render method.

The **fourth method** is called **getSnapshotBeforeUpdate(prevProps,prevState).** this method accepts previous props and previous state as its parameters and is called right before the changes from the virtual DOM are to be reflected in the DOM. this again is a method react documentation classifies as a rarely used method. **we would use this method to capture some information from the DOM** for example we can read the user's scroll position and after the update maintain that scroll position by performing some calculations. this method will either return null or return a value. the returned value will be passed as the third parameter to the next method we will be talking about. so **getSnapshotBeforeUpdate() used to read the current DOM state and returns a value or not.**

The **fifth method** in the update lifecycle is **componentDidUpdate(prevprops,prevState,snapshot)** . this method will be called after the render is finished in the re-render cycles. this means that we can be sure that the component and all its sub components have properly rendered itself after the update. this method accepts three parameters previous props, previous state and the snapshot value returned from getSnapshotBeforeUpdate() method. this method is guaranteed to be called only once in each re-render cycle. so what we can do is cause side-effects that is we can make Ajax calls, but before making the call we need to compare the previous props with the new props and then decide whether to make the Ajax call or not. if we are not comparing we are making unwanted requests which is not really a good idea. so componentDidUpdate() called once after the component has re-rendered and they suitable to make Ajax calls based on the previous and current props value.

All right now that we have gone through all the five methods and what can or cannot be done in each of the methods let's head over to the code and see that order of execution.

For this demo I will be reusing the components we created previously (*LifcycleA.js* & *LifecycleB.js*).let's begin by adding the update lifecycle methods in the LifecycleA component.

The first method is the **getDerivedStatefromProps** method which as we can see is already defined.

The second method is **shouldComponentUpdate()**. let's add that right before render shouldComponentUpdate method let's log it to the console and make sure to return true.

The third method is the **render()** method which again we can see has already been defined.

The fourth method is **getSnapshotBeforeUpdate().** so let's add that console dot log LifecycleA get a snapshot before update finally we have the component did update method console.log “LifecycleA componentDidUpdate” and return null.

now the same three methods I will also include in the child component which is LifecycleB so copy the three methods paste it in LifecycleB and change the text LifecycleA to LifecycleB. Now to trigger an update lifecycle we need to change either the props or the state. Back in LifecycleA component let's change the state on a button click, so in the JSX add a button tag the text is going to be “Change state” and onClick the handler is going to be this.changeState. now let's define the method as a class property this dot setState name as ‘codevolution’.

**LifecycleB.js**

import React, { Component } from 'react'

class LifecycleB *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vsishwas'}

        console.log("LifecycleB constructor")

    }

*static* getDerivedStateFromProps(props,state)

    {

        console.log("LifecycleB getDerivedStateFromProps")

        return null

    }

    componentDidMount()

    {

        console.log("LifecycleB componentDidMount")

    }

    shouldComponentUpdate()

    {

        console.log("LifecycleB shouldComponentUpdate")

        return true

    }

    getSnapshotBeforeUpdate()

    {

        console.log("LifecycleB getSnapshotBeforeUpdate")

        return null

    }

    componentDidUpdate()

    {

        console.log("LifecycleB componentDidUpdate")

    }

    render()

    {

        console.log("LifecycleB render")

        return (

        <div>Lifecycle B</div>

        )

    }

}

*export* default LifecycleB

**LifecycleA.js**

import React, { Component } from 'react'

import LifecycleB from './LifecycleB'

class LifecycleA *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vsishwas'}

        console.log("LifecycleA constructor")

    }

*static* getDerivedStateFromProps(props,state)

    {

        console.log("LifecycleA getDerivedStateFromProps")

        return null

    }

    componentDidMount()

    {

        console.log("LifecycleA componentDidMount")

    }

    shouldComponentUpdate()

    {

        console.log("LifecycleA shouldComponentUpdate")

        return true

    }

    getSnapshotBeforeUpdate()

    {

        console.log("LifecycleA getSnapshotBeforeUpdate")

        return null

    }

    componentDidUpdate()

    {

        console.log("LifecycleA componentDidUpdate")

    }

    changeState=()=>

    {

        this.setState({name:'Codevolution'})

    }

    render()

    {

        console.log("LifecycleA render")

        return (

        <div>

            <div>Lifecycle A</div>

            <button *onClick*={this.changeState}>Change state</button>

            <LifecycleB/>

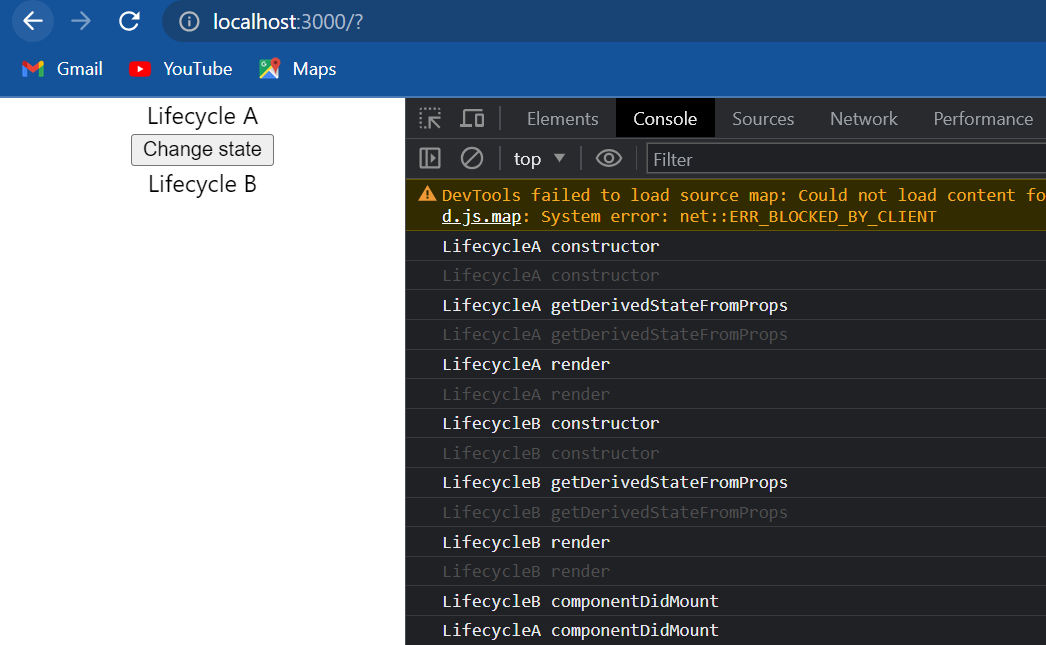
        </div>

        )

    }

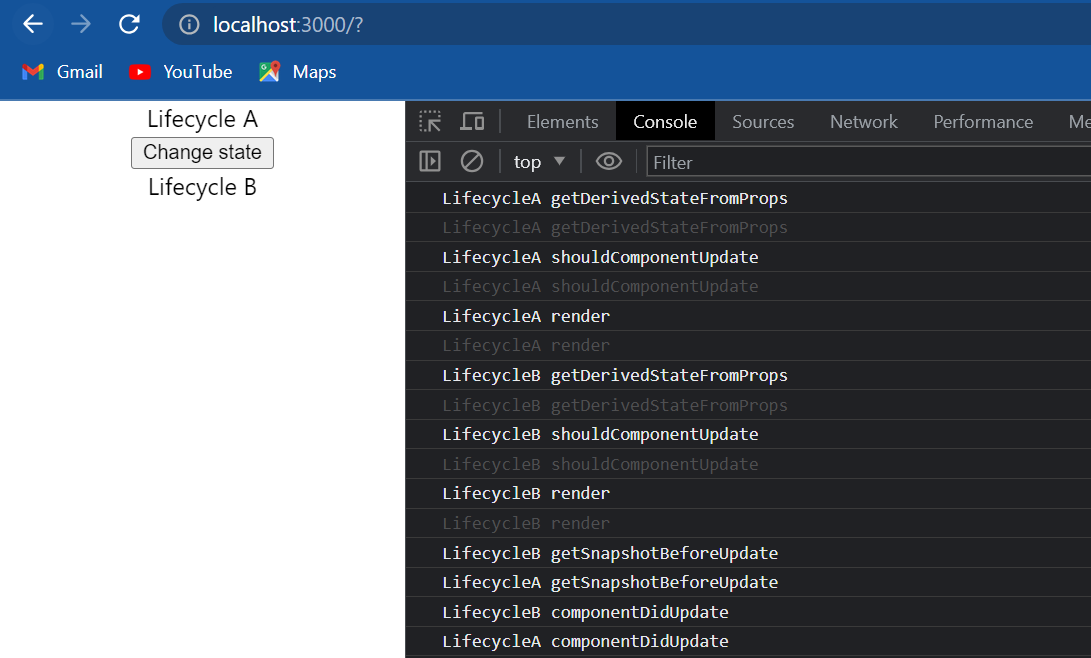
}

*export* default LifecycleA



Now let's test this, when I go back to the browser we can see that all the methods pertaining to the mounting phase are already executed.

Now I'm going to clear the console and click on the change state button we can see that the update life cycle methods are now logged.



we have lifecycleA getDerivedStateFromProps, shouldComponentUpdate and the render method after that the execution passes on to the child component. we have lifecycle B getDerivedStateFromProps, shouldComponentUpdate and the render method. Once both the child and the parent components have been rendered we have a slightly different order for getSnapshotBeforeUpdate and componentDidUpdate. first the child component method is executed and then the corresponding parent component method is executed.

But let me point out again that **render() and componentDidUpdate() are the more commonly used methods during the update lifecycle**. the remaining three methods exist for special cases and should be used sparingly. if at all we do decide to define them make sure we know what we're doing.

Lets quickly discuss the remaining two phases a **unmounting phase** and the **error-handling phase** .

The **unmounting phase** has just one method **componentWillUnmount()**. this method is invoked immediately before a component is unmounted and destroyed. in this method we can perform some cleanup tasks like canceling any network requests, removing event handlers, cancelling any subscriptions and also invalidating timers from set timeout or set interval. what we shouldn't do is called the setState() method that is simply because the component is never re-rendered after it has been unmounted. So **component will unmount perform necessary cleanup and don't call setState** .

The final phase we need to discuss about is the **error handling phase**. this phase has two methods **static getDerivedStateFromError(error)** and **componentDidCatch(error,info)**. These two methods are called when there is an error either during rendering in a life cycle method or in the constructor of any child component. we will discuss in detail about both these methods when we talk about a concept called error boundaries in react but for now that is pretty much what I have on component lifecycle methods for a class component in react.

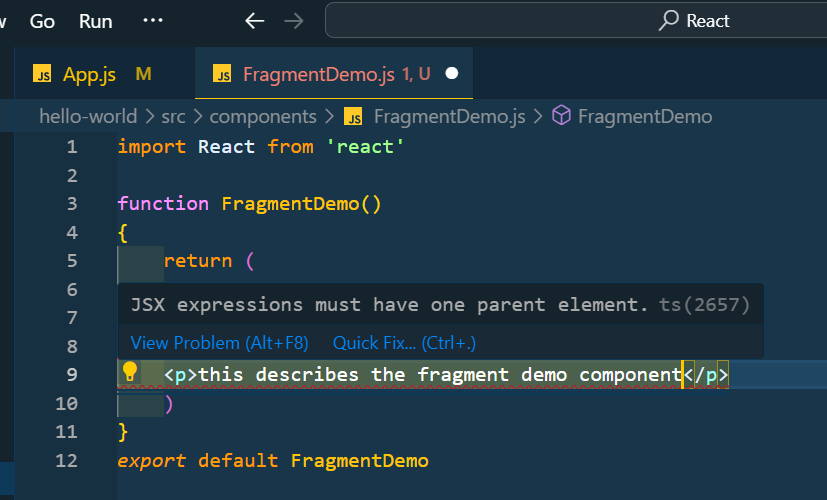
**Fragments**

we have the basics of react behind us it's time to focus on some of the topics mentioned under the Advanced section in the official react documentation. let's start with one of the more easier topics to understand.

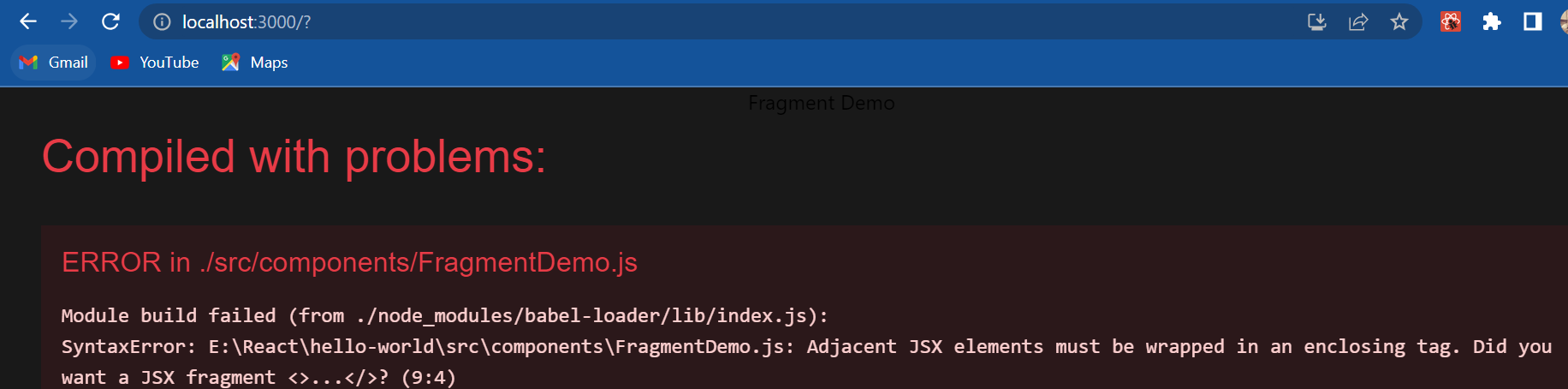
Fragments basically lets we group a list of children elements without adding extra nodes to the DOM. let us understand what that means with an example:

We'll start off by creating a new file *FragmentDemo.js* and within the file I'm going to create a functional component and in the JSX I will simply specify the text “fragment demo”, in the app component I will include the <FragmentDemo /> component.

if we save the files and take a look at the browser we should be able to see the text “fragment demo”. now let's go back to the fragment demo component and add a few more elements in the JSX. Now I want to convert the text fragment demo into a heading so I will replace the <div> tag with an h1 tag. Right after the heading I want to add a simple description using a paragraph tag “this describes the fragment demo component” but when I do this we get a red squiggly line at the closing parenthesis of the return statement and when I hover on that we can see that it says “JSX expressions must have one parent element”:



if we save the file and take a look at the browser we can see that the application is broken as well and the error message points to the same thing “adjacent JSX elements must be wrapped in an enclosing tag”:



So anytime our component has to return multiple elements we have to enclose them in a single parent element. so let's add that in closing div tag when I format it we can see that we have an enclosing dip tag and the each one and the paragraph elements are contained within this enclosing div tag.

import React from 'react'

function FragmentDemo()

{

    return (

    <div>

        <h1>Fragment Demo</h1>

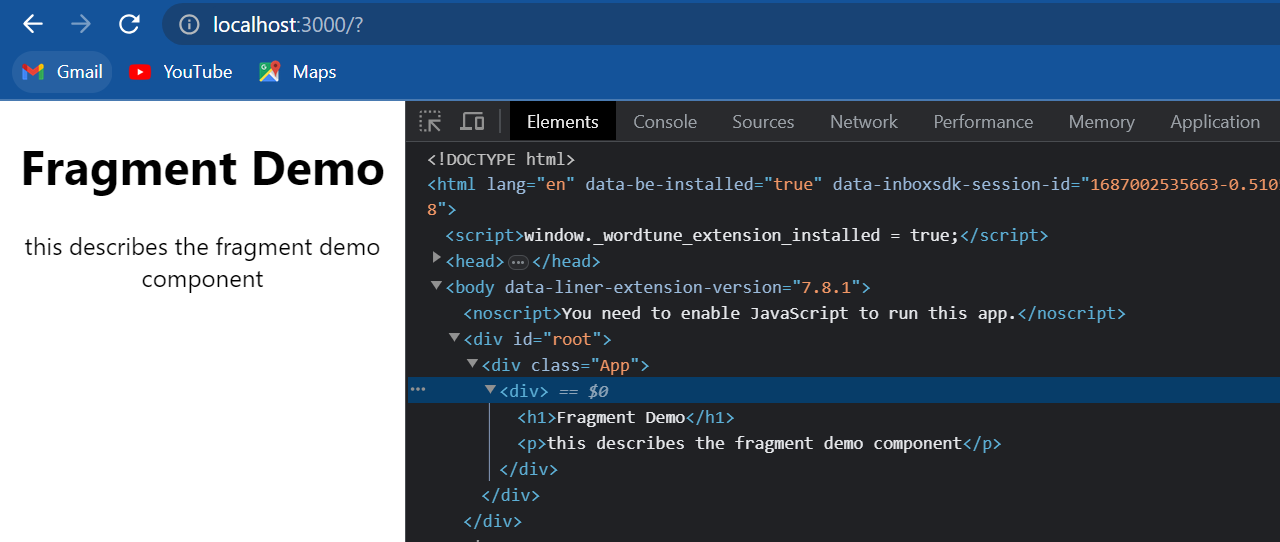
        <p>this describes the fragment demo component</p>

    </div>

    )

}

*export* default FragmentDemo



if we now save the file and take a look at the browser everything works fine. but if we inspect the element we can notice that we have the enclosing div tag included in the DOM tree. so between the div tag from app component and the h1 tag in fragment demo component we have an additional div tag. This is where react fragments come into picture. we can replace the enclosing div tag with react fragment and that will prevent the extra node from being added to the DOM. so all we have to do is in fragment demo component replace this existing <div> tag with react dot fragment (<React.Fragment>) make sure to change the closing tag as well now if we save the file and go back to the browser inspect the element.

import React from 'react'

function FragmentDemo()

{

    return (

    <React.Fragment>

        <h1>Fragment Demo</h1>

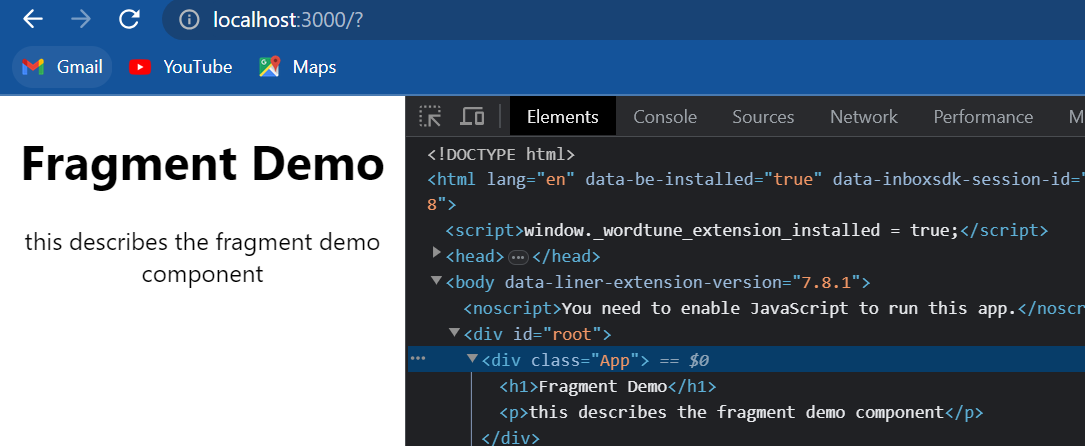
        <p>this describes the fragment demo component</p>

    </React.Fragment>

    )

}

*export* default FragmentDemo



we can see that we no longer have the div tag between the app component div tag and the h1 tag and if we go back to code we can see that we are still returning multiple elements in the JSX.

Now let's take a look at another example where react fragments seems much more appropriate. I am going to create two new files so within the components folder *Table.js* and *Columns.js* and within the *Table.js* file, i'm going to create a functional component for the JSX i'm going to replace the div tag with a table tag and within the table I'm going to add a <tbody> tag and then a <tr> tag to create a row of data. Within the row I want to render columns and the columns will be maintained in a separate component which is our Columns component. So let's go to *Columns.js* and create another functional component. Within the JSX I'm going to add two columns so <td> name<td/> and another <td>vishwas<td/> because we are returning multiple elements we need the enclosing div tag. Now back in the table component I will include the columns component and back in app component I will include the table component.

**Table.js**

import React from 'react'

import Columns from './Columns'

function Table()

{

    return (

        <table>

            <tbody>

                <tr>

                    <Columns/>

                </tr>

            </tbody>

        </table>

    )

}

*export* default Table

**Columns.js**

import React from 'react'

function Columns()

{

    return (

    <div>

        <td>Name</td>

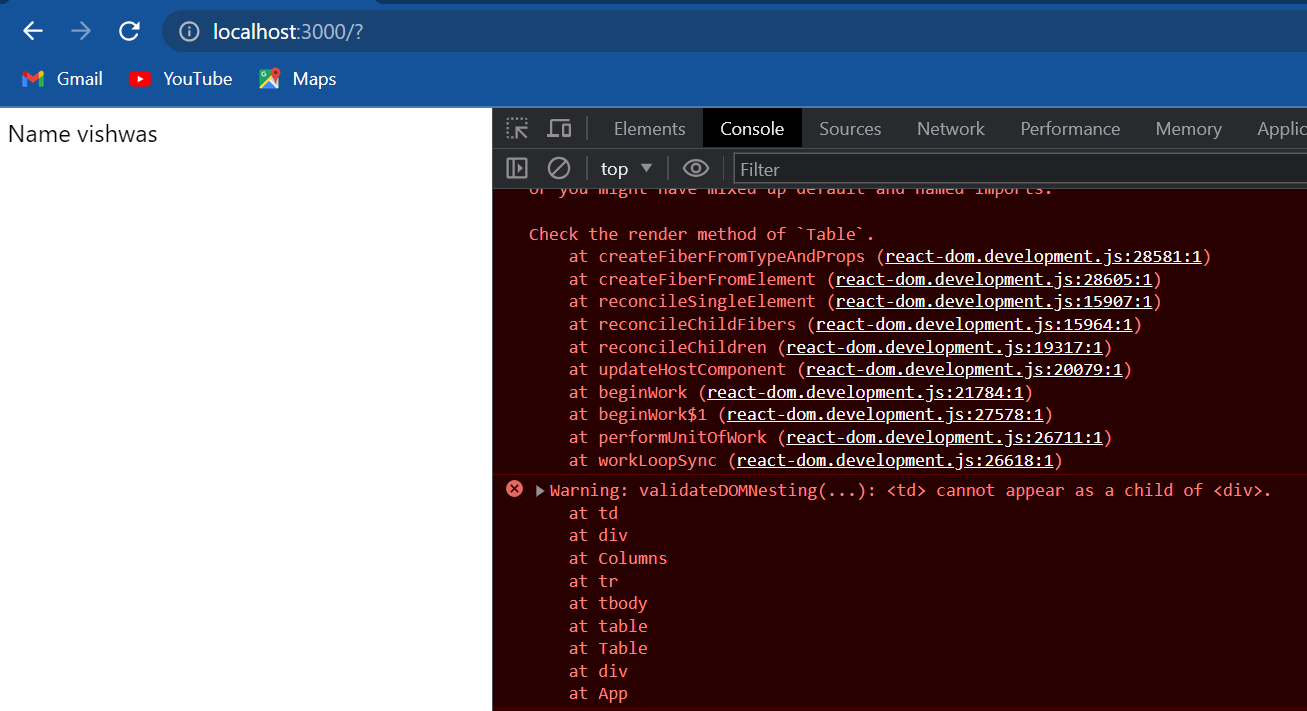
        <td>vishwas</td>

    </div>

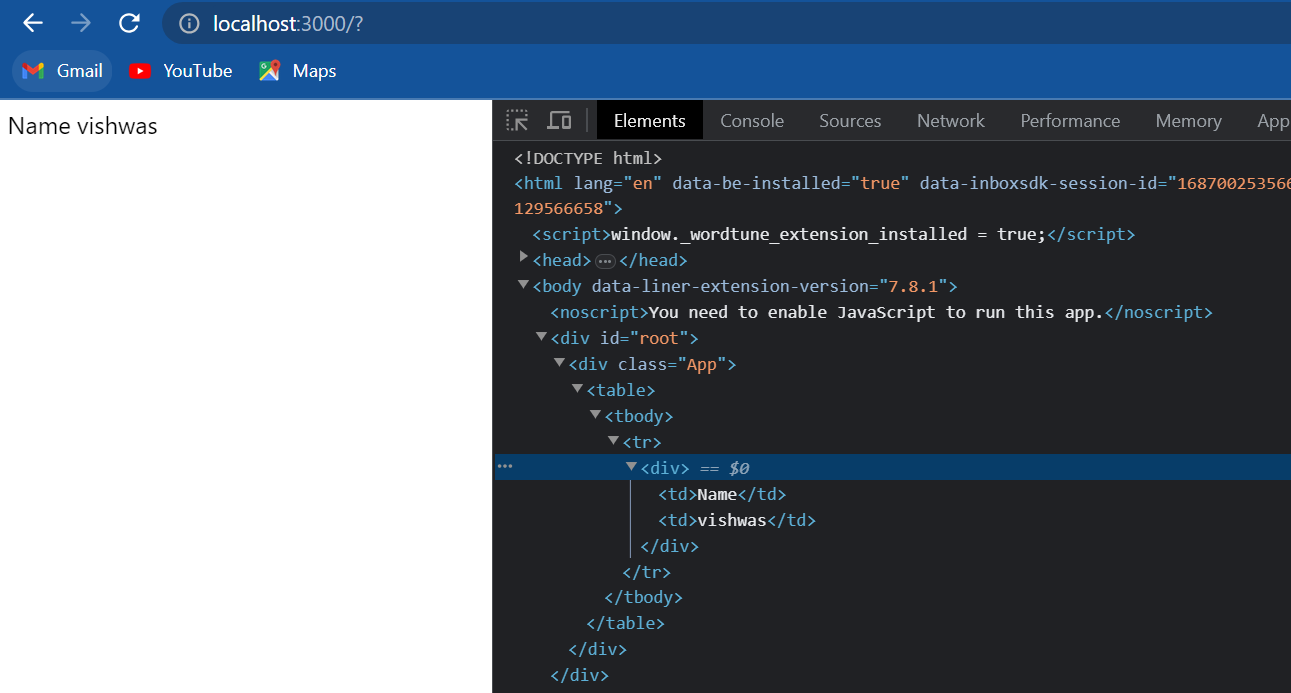
    )

}

*export* default Columns



if we know save all the files and take a look at the browser we should be able to see the two columns name and vishwas. if we take a look at the console though we have warnings and the warning is “validate DOM nesting <td> cannot appear as a child of the <div>. Basically it is telling us that it is wrong to have a <td> element as a child of a div tag. if I inspect the element:



We can pretty much see that in the DOM tree td within the div tag and this div tag was necessary because we were returning multiple elements in the JSX of Columns component.

But now we have the better alternative we can replace this <div> tag with <React.Fragment>. So I'm going to go back to code and in Columns component I'm going to replace the div tag with <React.Fragment>.

import React from 'react'

function Columns()

{

    return (

    <React.Fragment>

        <td>Name</td>

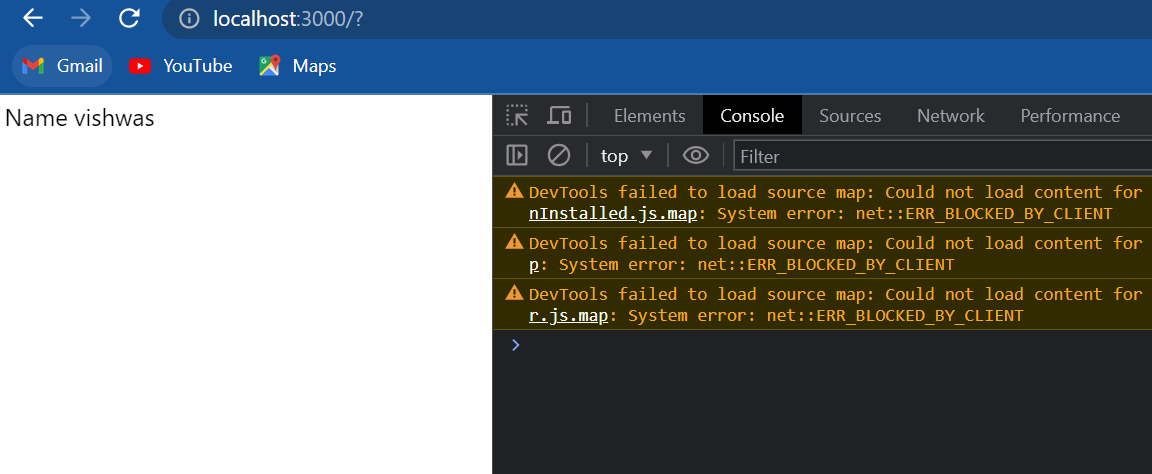
        <td>vishwas</td>

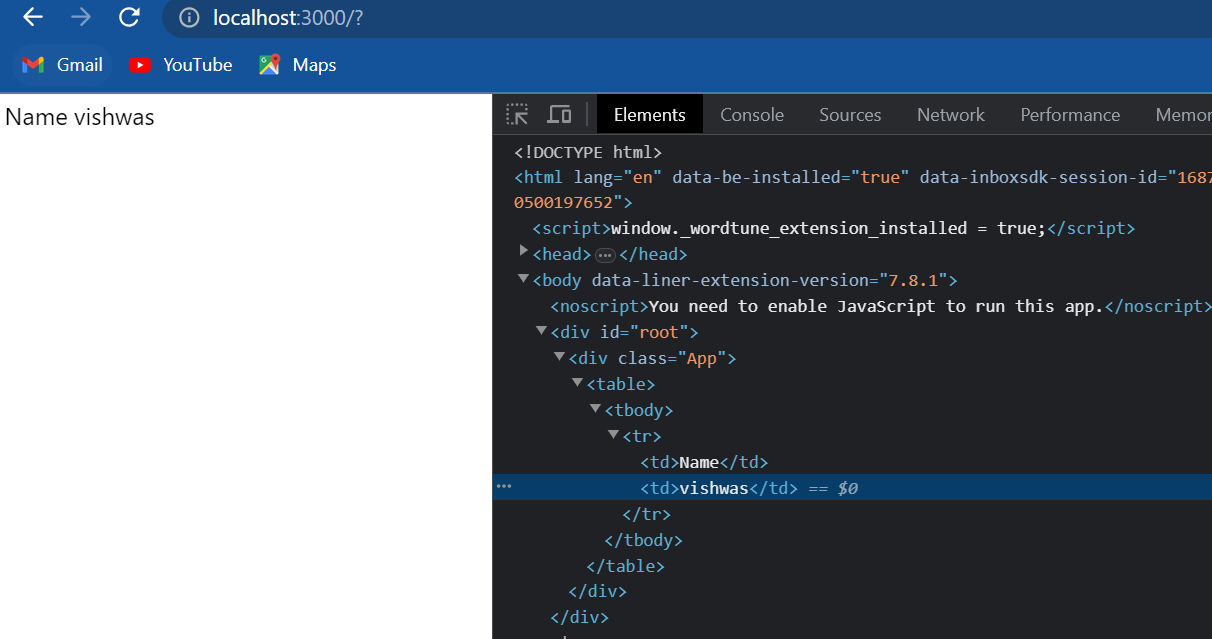
    </React.Fragment>

    )

}

*export* default Columns





if we now go back to the browser we can see that all console warnings have disappeared. if we inspect the element we can see that we have the <table> tag, <tbody>, <tr> and then <td> there is no div tag in between.

The final point on fragments is that **it can accept the key attribute when rendering lists of items.** for example let's assume we have an array of items stored in a variable called **items,** then we could have items dot map and then we have an arrow function the parameter is **item** and the function can return a react fragment. Because items in the list need the key prop though we can specify the key attribute on react fragment. key is equal to let's say item dot ID and within the JSX we can have multiple elements being returned let's say a heading that says “title” and then a paragraph tag that renders item dot title.

import React from 'react'

function Columns()

{

    const items=[]

    return (

    <React.Fragment>

        {

            items.map(item =>(

                <React.Fragment *key*={item.id}>

                    <h1>Title</h1>

                    <p>{item.title}</p>

                </React.Fragment>

            ))

        }

        <td>Name</td>

        <td>vishwas</td>

    </React.Fragment>

    )

}

*export* default Columns

this is completely possible as of this recording key attribute is the only attribute that can be passed to a reactive fragment. the reactive hopes to add additional attributes in the future but for now keep in mind to pass in only the key attribute.

Now there is also a shorthand syntax that we can use for react fragments. instead of react dot fragment we can use an empty opening tag and an empty closing tag. I can replace <React.Fragment> </React.Fragment> with <> </> (an empty opening tag and an empty closing tag).

import React from 'react'

function Columns()

{

    const items=[]

    return (

    <>

        <td>Name</td>

        <td>vishwas</td>

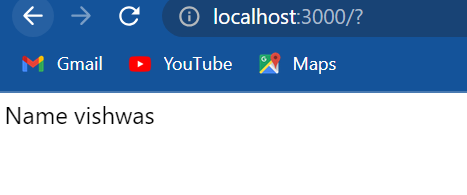
    </>

    )

}

*export* default Columns

This basically represents the idea that it won't add an actual element to the DOM. **If we do use this shorthand syntax though there is one limitation we cannot pass in the key attribute.** so let's save this and take a look at the browser and we should still see name and vishwas, the two columns we have specified.



So that is about react fragments: fragments let us group a list of children elements without adding extra nodes to the DOM.

**Pure components**

So far we have been creating class components by extending the component class, from react we can see that even the app component extends from react dot component. well it turns out there is a second way to create a class component and that is by extending the pure component class from react. let's take a look at an example:

Pure component react snippets: rpce

I'm going to create a new file called *PureComp.js* , this time though I am going to use the react snippet our rpce to create a pure component. so straight away we can notice the difference instead of importing **Component** from react we are now importing **PureComponent** and extending the same. now I will get rid of the named export and for the JSX simply add the text “pure component” back in app component; I will include the pure component.

import React, { PureComponent } from 'react'

class PureComp *extends* PureComponent {

    render()

    {

        return (

            <div>Pure Component</div>

        )

    }

}

*export* default PureComp

if we now save the file and take a look at the browser we should be able to see the text “pure component”. Nothing new or different when compared to the regular component class which brings us to the question what is the difference between the component class and the pure component class and also when should we use one over the other. I want to help we understand the answer to that question by walking we through a simple demo that involves both the classes. for this demo I need a regular component, a pure component and a containing parent component that is capable of changing its state. by now this should be pretty straightforward so let's quickly get it done.

we already have a pure component so let's create a regular component, called *RegComp.js* and within the file create a class component and for the JSX add the text “regular component”. now let's create the parent component again a new file called *ParentComp.js* and this will also be a class component and for the JSX I'm going to have “parent component”. now we also need to be able to change the state of the component so let's add a bit more code. I will add a constructor and I will create a new state property called **name** and set it over “vishwas”, next I will add the **componentDidMount**  lifecycle method and within the body I will create a timer; **setInterval()** accepts two arguments the first one is a function which will be an arrow function and the second argument is the interval, I will set it to two seconds. Within the arrow function body I will call **setState** but what I will do is set the **name** property to Vishwas again no change from the previous value whatsoever. next I will include the regular and pure components in the JSX passing in **name** as a prop regular component name is equal to this dot state dot name, pure component **name** is equal to again this dot state dot name. now let's include this name prop in the regular and pure component, this dot props dot name (in regular component) and again this dot props dot name(in pure component). finally in app component let's replace pure component with parent component.

**ParentComp.js**

import React, { Component } from 'react'

import RegComp from './RegComp'

import PureComp from './PureComp'

class ParentComp *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vishwas'}

    }

    componentDidMount()

    {

        setInterval(()=>{this.setState({name:'vishwas'})},2000)

    }

    render()

    {

        return (

            <div>

                Parent Component

                <RegComp *name*={this.state.name} />

                <PureComp *name*={this.state.name} />

            </div>

        )

    }

}

*export* default ParentComp

**RegComp.js**

import React, { Component } from 'react'

class RegComp *extends* Component

{

    render()

        {

        return (

            <div>

                Regular Component {this.props.name}

            </div>

        )

    }

}

*export* default RegComp

**PureComp.js**

import React, { PureComponent } from 'react'

class PureComp *extends* PureComponent

{

    render()

    {

        return (

            <div>

                Pure Component {this.props.name}

            </div>

        )

    }

}

*export* default PureComp



If we now take a look at the browser we should be able to see the expected output “Parent component”, “regular component vishwas” and “pure component vishwas”. What we are really concerned about in this example though is when the render method is called in each of the components. so let's go back to code and add a console log statement in all the three components. In parent component within render() at a console.log “parent component render”, within the regular component we are going to have “regular component render” and similarly within the pure component “pure component render”.



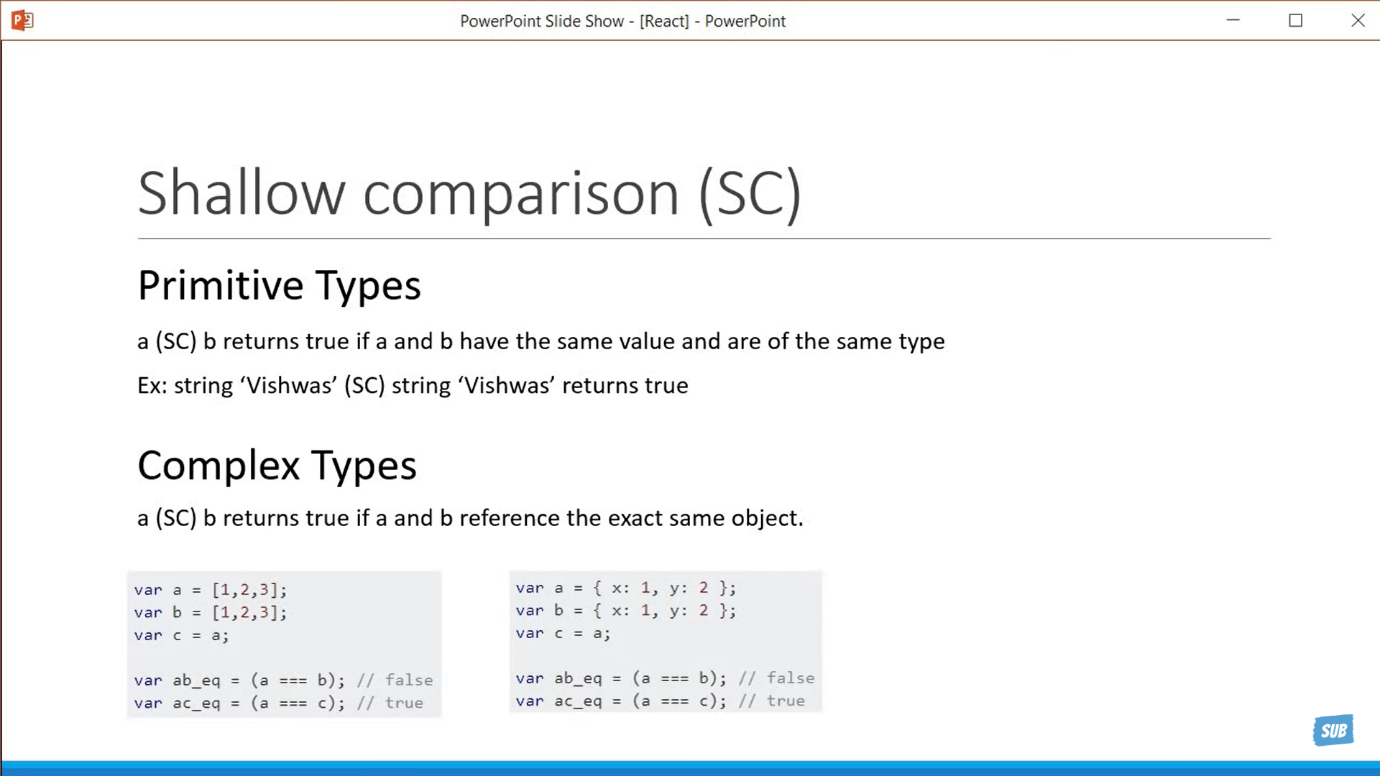
let's head back to the browser and take a look at the console I will refresh the page and now we can see the render method logs initially we have “parent component render”, “regular component render” and “pure component render” but after that every two seconds the set state method is called which will re-render the parent component and if the parent component re-renders the child components will also re-render unless of course we return false from **shouldComponentUpdate.** so we should see the render method from both regular and pure components being logged. however we see that it is not the case, every two seconds parent and the regular component are rear-ended but the pure component is never rear-ended. **this is the difference between the regular component class and the pure component class**.

* A regular component does not implement they **shouldComponentUpdate** method it always returns true by default.
* A pure component on the other hand implements **shouldComponentUpdate** with a shallow prop and state comparison.

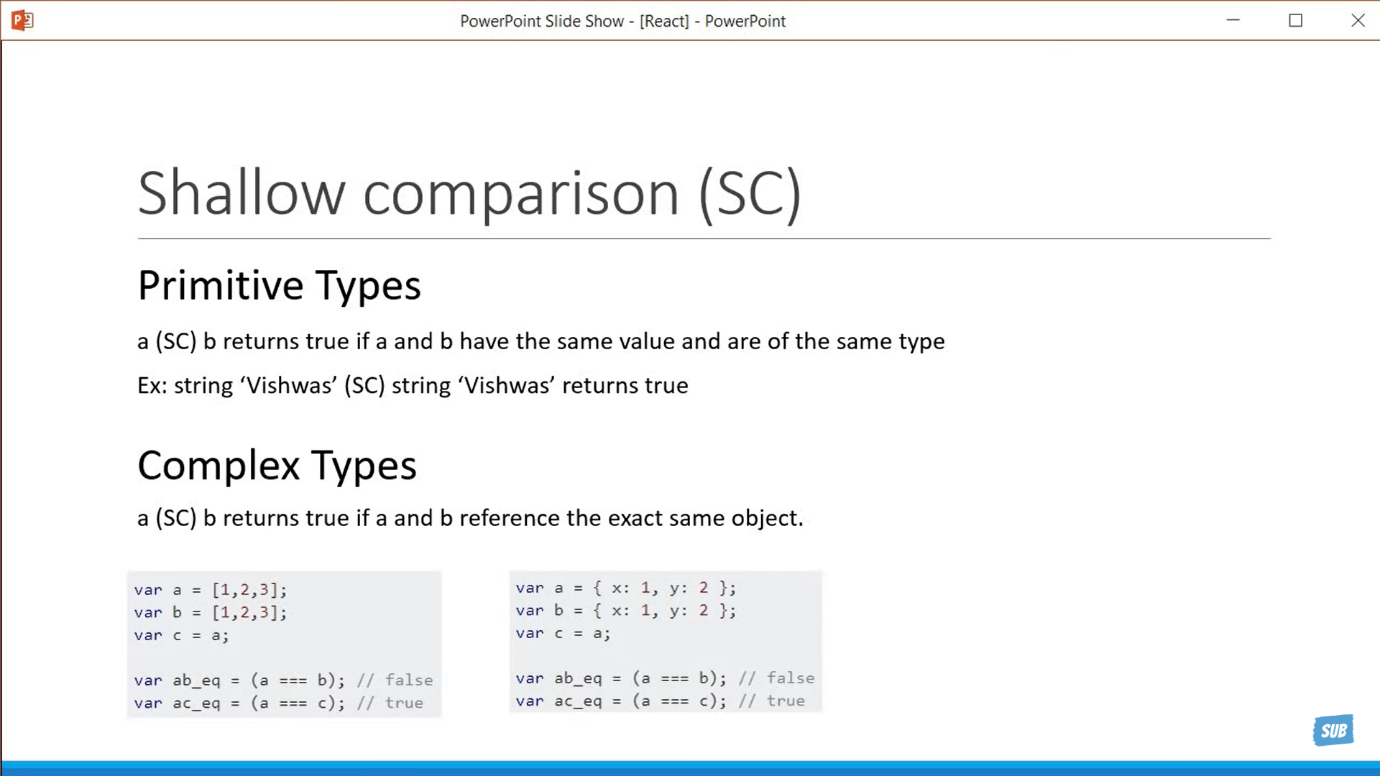
Now what exactly is shallow props and state comparison. let's take a look we can study about shallow comparison with respect to primitive and complex types.

for two primitive types **a** and **b** like numbers, strings, and boolean **a** shallow comparison **b** returns true if **a** and **b** have the same value and are of the same type. for example if **a** and **b** are both strings with the value “Vishwas”, the shallow comparison returns true.

for complex types like objects and arrays **a** shallow comparison **b** returns true if **a** and **b** reference the exact same object. for example consider three variables **a** is an array [1,2,3], **b** is also an array [1,2,3] and then we have **c** which is assigned to **a**.



In this scenario **a** shallow comparison **b** will return false, even though the items in both the areas are the same; those two arrays do not reference the same object. **a** shallow comparison **c** on the other hand will return true because they point to the same object. and this is the case with objects as well so let's consider three variables again **a**, **b** and **c**.



**a** is an object, **b** is also an object, they have the same properties but **a** shallow comparison **b** will return false. we then have variable **c** which is assigned to **a**. So **a** shallow comparison **c** will return true, both of them reference the exact same object.

Now that we understand shallow comparison the pure component definition will make much more sense. A pure component implements shouldComponentUpdate with a shallow prop and state comparison. it does a shallow comparison of previous state with current state and previous crops with current crops and only when the shallow comparison indicates there is a difference the component will re-render.

In our example we are dealing with primitive types and props comparison. we are sending a prop which is of type **string** and has a value of “Vishwas”. every two seconds the pure component sees that the previous prop value of Vishwas is no different from the current prop value of the vishwas and hence does not re-render. And because a regular component does not implement this check it will always re-render. and what we have seen is a shallow comparison of props but it's pretty much the same for state comparison as well.

I will now change the parent component to extend **PureComponent** instead of the regular component.

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

import RegComp from './RegComp'

import PureComp from './PureComp'

class ParentComp *extends* PureComponent

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vishwas'}

    }

    componentDidMount()

    {

        setInterval(()=>{this.setState({name:'vishwas'})},2000)

    }

    render()

    {

        console.log("\*\*\*\*\*\*\*\*\*\*\*\*Parent component render\*\*\*\*\*\*\*\*\*\*\*\*\*")

        return (

            <div>

                Parent Component

                <RegComp *name*={this.state.name} />

                <PureComp *name*={this.state.name} />

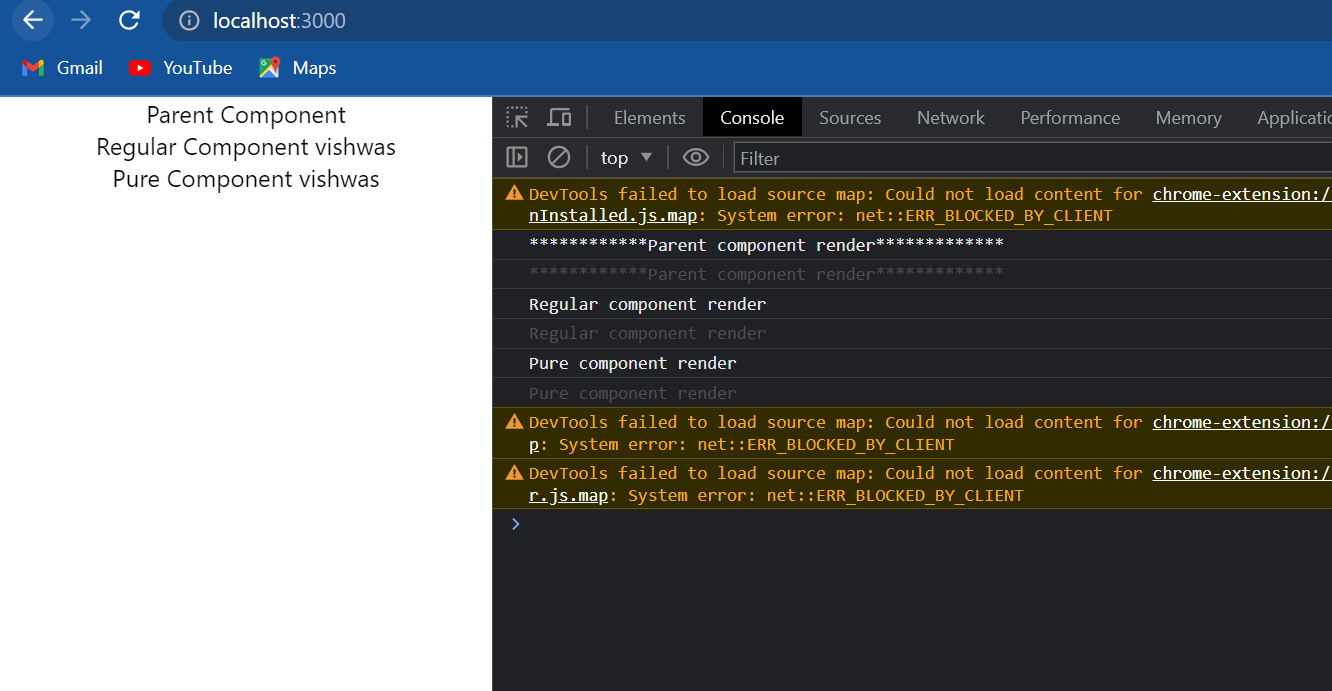
            </div>

        )

    }

}

*export* default ParentComp



if we now take a look at the browser and refresh we can see that there is the initial rendering of all the components parent, regular, and pure and two seconds later there is no re-render. the parent component checks previous state with current state and sees that there is no difference and hence there is no re-render. But what we should keep in mind is that if the parent doesn't re-render the children also will never re-render and that is why we don't see any statement logged in the two-second intervals. so we can answer the question why to use pure components. **Pure components by preventing unnecessary renders can give we a performance boost certain scenarios.** for example let's say we're rendering a list of 50 items. By not rear-ending them when it is not required they're going to have a good performance boost. A key point to keep in mind is that **we should not mutate object or arrays in props or state**. for example if we need to add a new item to the list don't mutate it by pushing the item into the list. the reference to the area never changes and because pure components only check for that the component will not re-render even if there is a difference. Always return a new object or array when dealing with pure components.

Summary:

* In react we can create a component by extending the **PureComponent** class.
* A pure component implements the **shouldComponentUpdate** lifecycle method by performing a shallow comparison on the props and state of the component.
* If there is no difference the component is not re-render thereby providing a performance boost.
* When we create a pure component, it is a good idea to ensure that all the children components are also pure to avoid unexpected behaviour.
* When we are using pure components never mutate the state. Always return a new object that reflects the new state.

It's probably safe to use regular components most of the time unless of course we are seeing a performance head in some components.

**Memo**

pure components only re-rendered the class components when there is a difference in the shallow comparison of props and state, this of course results in a performance improvement. pure components though only work with class based components. it would be nice if we could achieve the same in functional components as well, and that is where react dot memo comes into picture. what pure components is to class based components memo is two functional components. let's take a look at an example:

I'm going to create a new file called *MemoComp.js* and within the file I'm going to create a functional component, within the function parentheses I'm going to destructure the name prop and then in the JSX let's render that name property I will also add a console.log statement which says “rendering memo component”. Back in parent component which we created in the previously, I will comment out the regular component and the pure component and instead include the **<MemoComp/>.** if we remember we have a **setInterval** in **componentDidMount** which basically calls set state every two seconds resulting in a re-render of the parent component, of course extended from Component class. now since the **name** prop is not really changing every two seconds we can gain performance improvement by not re-rendering the memo component, name is equal to this dot state dot name so when the parent renders memo component also re-renders but the name prop is going to stay the same with every re-render. so we need a way to avoid re-rendering this memo functional component and we can do that using react dot memo. In memo component all we have to do is export default React dot memo and then pass in the memo component as its argument. now before we take a look at the browser I have to mention one thing react dot memo was introduced in version 16.6 so in our package JSON make sure react and react DOM are at least 16.6 or above otherwise when we head back to the browser we're going to see an error which says react dot memo is not a function. so please to make sure we are using 16.6 and above if we want to use the memo feature.

**ParentComp.js**

import React, { Component } from 'react'

import RegComp from './RegComp'

import PureComp from './PureComp'

import MemoComp from './MemoComp'

class ParentComp *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {name:'vishwas'}

    }

    componentDidMount()

    {

        setInterval(()=>{this.setState({name:'vishwas'})},2000)

    }

    render()

    {

        console.log("\*\*\*\*\*\*\*\*\*\*\*\*Parent component render\*\*\*\*\*\*\*\*\*\*\*\*\*")

        return (

            <div>

                Parent Component

                <MemoComp *name*={this.state.name}/>

                {*/\* <RegComp name={this.state.name} />*

*<PureComp name={this.state.name} /> \*/*}

            </div>

        )

    }

}

*export* default ParentComp

**MemoComp.js**

import React from 'react'

function MemoComp({name})

{

    console.log('rendering memo component')

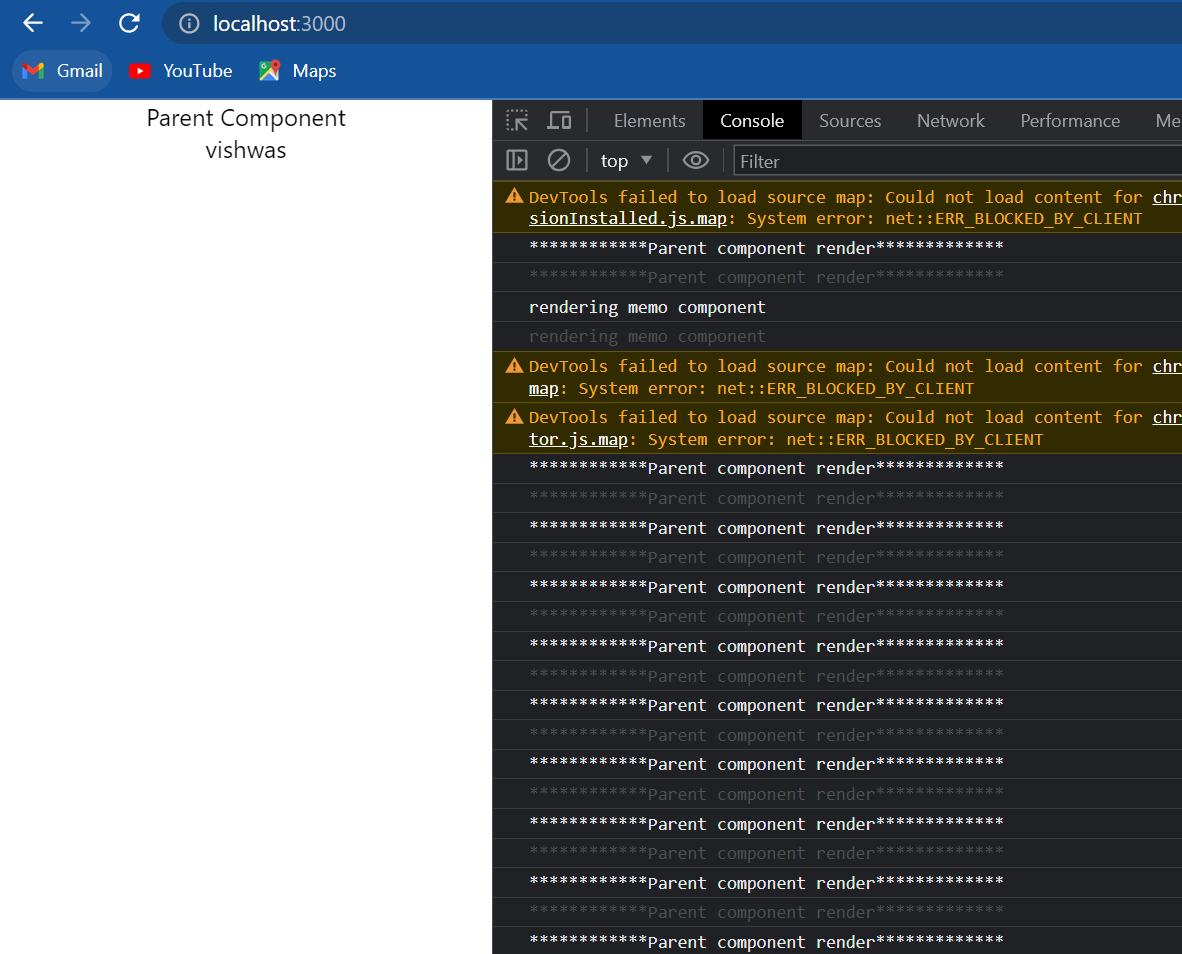
    return (

    <div>{name}</div>

    )

}

*export* default React.memo(MemoComp)



so now if we go back to the browser click on refresh we can see that we have the render statements from the initial render “parent component” and “rendering memo component” but after that for the two-second intervals only the parent is rerendered and the memo component is not re-rendered. the props are not changing and hence it's better to avoid re-rendering the memo component. now what we see here is something called a higher-order component, **React.memo** accepts a component add some things to that component and returns a new enhanced component. in our case a component capable of avoiding re-renders when there is no changes in props.

This is how we use react dot memo, similar to pure components but it works with functional components instead.

**Refs**

let's learn about the concept of refs in react. basically refs make it possible to access DOM nodes directly within react. for our understanding we will take a look at one of the more common use cases which is focusing a text input. for example: Suppose we have a login form as soon as the page is loaded let's say by default we want the user name input field to be focused. let's see how to achieve that with refs. I'm going to create a new file called *RefsDemo.js* and within the file I am going to create a class component and within the JSX I am going to add an input element. And then import the component in App component.

import React, { Component } from 'react'

class RefsDemo *extends* Component

{

    render()

    {

        return (

            <div>

                <input *type*="text"/>

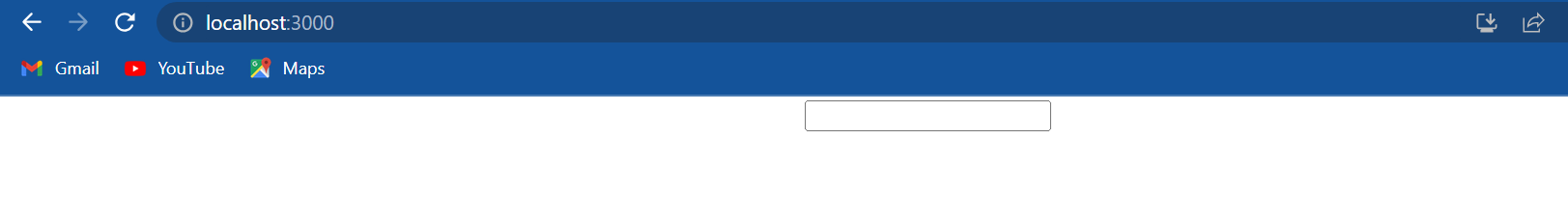
            </div>

        )

    }

}

*export* default RefsDemo



now if we save the files and take a look at the browser we should be able to see an input field. What we want to achieve is as soon as the page loads we want this input field to be focused.

Using refs we can do this in three simple steps:

1. The first step is to create a ref. we do that using react dot **createRef** method. so in the RefsDemo component I'm going to add a constructor and within the constructor we create a new property this dot **inputRef** and to this property we assign react dot **createRef(),** it is common to createRefs in the constructor so that they can be referenced throughout the component.
2. The second step is to attach this ref to our input element in the render method, and to attach a ref we make use of the reserved **ref** attribute so on the input element ref is equal to within curly braces the ref we have created which is this dot inputRef. we now have a reference to the input element.
3. The third and final step is to call the **focus()** method on this input element. But first let us find out what exactly does the property this dot inputRef hold after a reference is created. so I am going to add the **componentDidMount** lifecycle hook and within the hook I will simply log to the console this dot inputRef.

import React, { Component } from 'react'

class RefsDemo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef = React.createRef()

    }

    componentDidMount()

    {

        console.log(this.inputRef)

    }

    render()

    {

        return (

            <div>

                <input *type*="text" *ref*={this.inputRef} />

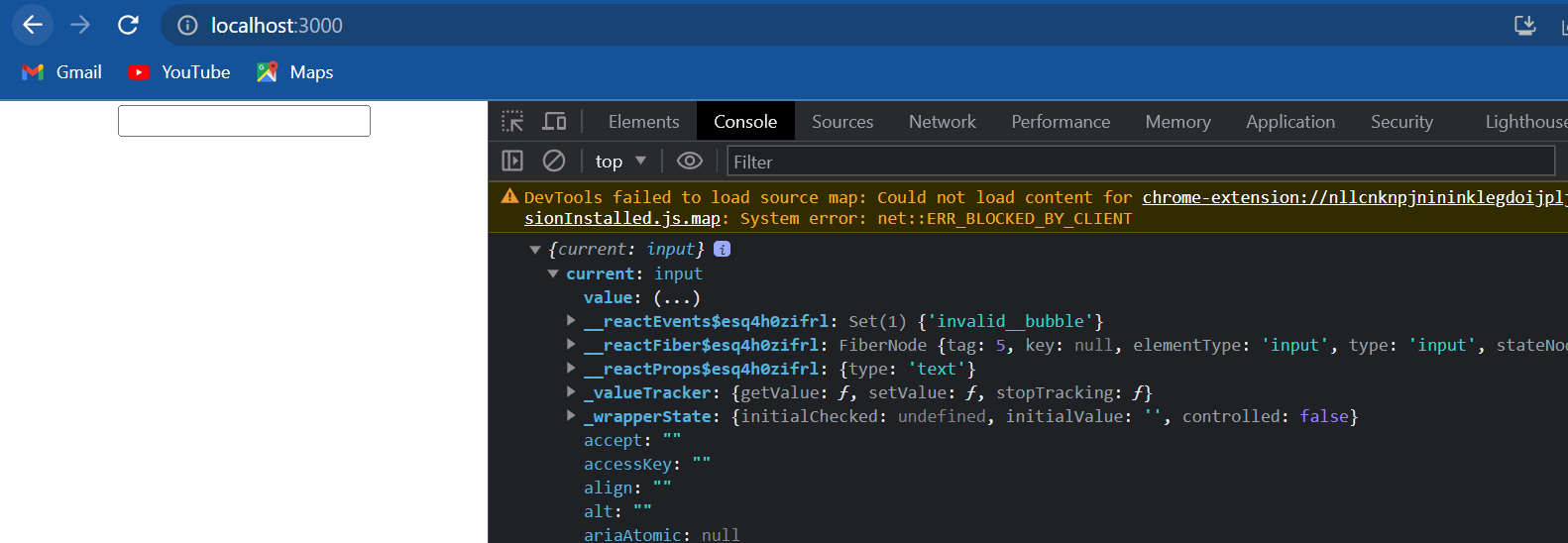
            </div>

        )

    }

}

*export* default RefsDemo



if we now go back to the browser open the console, we can see that we have an object logged in the console. if I expand it we can see a property called **current** of type **input** and this current property points to the actual DOM node. So to focus on the input element in **componentDidMount** we simply call the **focus()** method on the current property. So back in componentDidMount this dot inputRef dot current dot focus.

import React, { Component } from 'react'

class RefsDemo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef = React.createRef()

    }

    componentDidMount()

    {

        this.inputRef.current.focus()

        console.log(this.inputRef)

    }

    render()

    {

        return (

            <div>

                <input *type*="text" *ref*={this.inputRef} />

            </div>

        )

    }

}

*export* default RefsDemo



if we now save the file and take a look at the browser I click on refresh and we can see that on page load the input element has the focus. if the field was part of a login form the focus would be on the username field and the user can start typing in their name straightaway.

A **second possible use case for using refs would be to fetch the input value**. so back in code right after the input element I'm going to create a button with the text “click” and a click event handler **onClick** it's going to be equal to this dot clickHandler. now let's define the handler after component amount click handler which is going to be equal to an arrow function and we will simply alert this dot inputRef dot current dot value. basically we are accessing the **value** property of the input DOM node which is available as the current property.

import React, { Component } from 'react'

class RefsDemo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef = React.createRef()

    }

    componentDidMount()

    {

        this.inputRef.current.focus()

        console.log(this.inputRef)

    }

    clickHandler= ()=>

    {

        alert(this.inputRef.current.value)

    }

    render()

    {

        return (

            <div>

                <input *type*="text" *ref*={this.inputRef} />

                <button *onClick*={this.clickHandler}>Click</button>

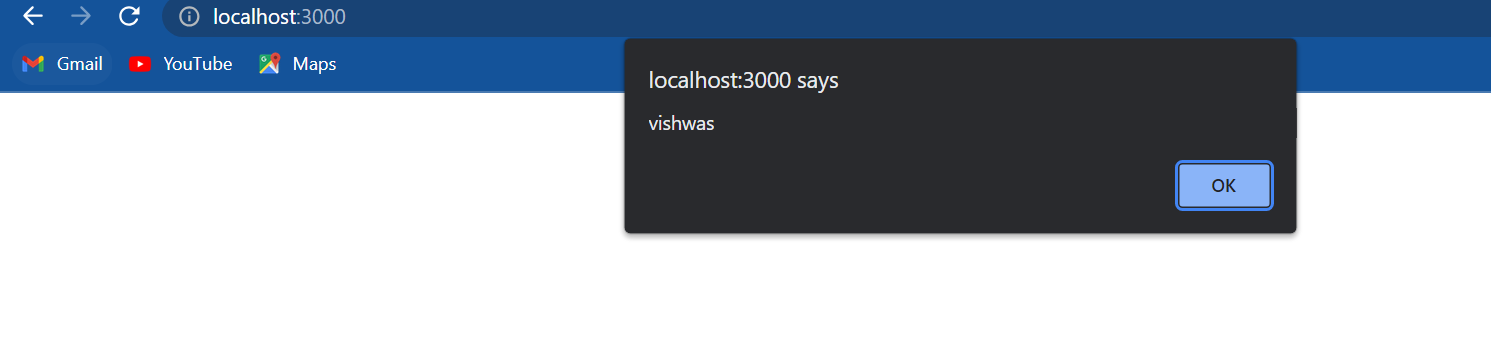
            </div>

        )

    }

}

*export* default RefsDemo



if we now go back to the browser enter the text as Vishwas and click on the button we should be able to see vishwas in the other dialog. And this is one way to create an access refs that is using the **createRef** method.

React also supports a second way to set refs which is called as callback refs. now this second approach is a slightly older approach to creating refs, but I still want to go through it in case we come across the usage of this approach in a project at work or when reading some articles or blog posts. Now to understand the callback refs approach, let's take a look at the same example of focusing an input element. This approach has four simple steps:

1. The first step is to create the ref. In the first approach we used the **createRef()** method. in this approach we create a property and assign a value of null. this dot **cbRef** for callback ref is going to be equal to null
2. In the second step though we create a method that will assign a DOM element to the ref we have just created in step 1. now I'm going to call this method as set callback ref so this dot **setCbRef** and it is going to accept an **element** a DOM element as its parameter and assign it to the callback ref, so this dot **cbRef** is equal to element.
3. The third step is to attach this ref to the input element. I am going to duplicate the input we already have and change the ref attribute to this dot **setCbRef**. if we remember this is the method that sets our ref. the input element is implicitly passed as a parameter which is assigned to the callback ref (cbRef) property.
4. Now for the fourth and final step in componentDidMount I will comment out the old code and replace it with code for the new callback ref. What we have to know about callback ref is that react will call the ref callback with the DOM element when the component mounts and call it with null when it unmount that is the reason it is really important to check if a value exists on the reference property and that it is not null. so in componentDidMount if this dot callback ref(cbRef) so we are checking if it is not null only then we are going to access this dot callback ref(cbRef) and then call the focus method. Now unlike the previous approach the DOM node is directly accessed using the ref property, we don't need current.

import React, { Component } from 'react'

class RefsDemo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef = React.createRef()

        this.cbRef=null

        this.setCbRef=element=>

        {

            this.cbRef=element

        }

    }

    componentDidMount()

    {

*// this.inputRef.current.focus()*

*// console.log(this.inputRef)*

        if(this.cbRef)

        {

            this.cbRef.focus()

        }

    }

    clickHandler= ()=>

    {

        alert(this.inputRef.current.value)

    }

    render()

    {

        return (

            <div>

                <input *type*="text" *ref*={this.inputRef} />

                <input *type*="text" *ref*={this.setCbRef} />

                <button *onClick*={this.clickHandler}>Click</button>

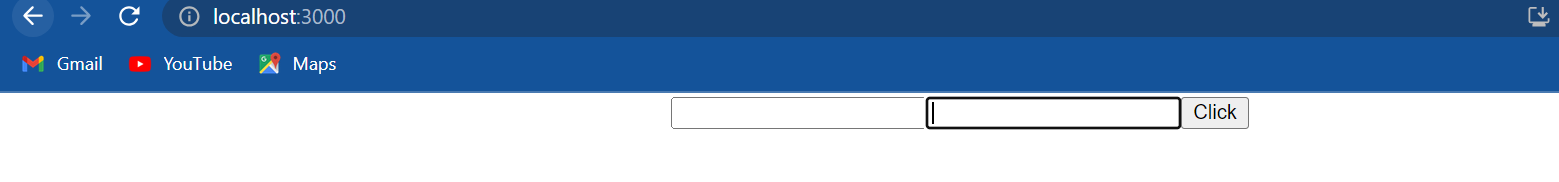
            </div>

        )

    }

}

*export* default RefsDemo



if we now save the file and take a look at the browser refresh and we can see that the second input has the focus on page load. so using callback refs we have focused the input element.

Summary:

* With create ref approach we create a reference using react dot createRef.
* With callback ref approach we first create a property and then create a method that assigns the property with a DOM element passed as a parameter.
* In create ref approach we attach the reference to an element using the ref prop and assigning the property.
* In callback of approach, we attach the ref to an element using the method that in turn assigns the element to the property.
* In create ref approach the element can be accessed using this dot reference variable dot current and
* In the callback approach it is directly accessed using this dot reference variable

**Refs with class component**

Previously we have seen that how to add refs to a normal HTML element like the input element.

It is also possible to add a ref to a class component. let's take a look at an example:

I'm going to create a new file called *Input.js* and within the file I'm going to create a class component. Now this component is basically going to be an implementation of what we learnt previously. So in the JSX I am going to add an input element input type is equal to text and in the constructor I am going to create a ref using the **createRef** method, so this dot inputRef is equal to react dot createRef. next I will attach this ref to the input element using the ref attribute, so ref is equal to this dot inputRef and we can focus on the input element in componentDidMount but for this example we don't want to do that instead we want to create a method that will in turn focus the input element. And that method will be called by the parent component. So in *Input.js* right after the constructor I'm going to have a method **focusInput()** within the method this dot inputRef dot current dot focus and that is it for the input component. we basically have an input element and a method focus input which when called will focus the input element.

import React, { Component } from 'react'

class Input *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef = React.createRef()

    }

    focusInput()

    {

        this.inputRef.current.focus()

    }

    render()

    {

        return (

        <div>

            <input *type*='text' *ref*={this.inputRef}/>

        </div>

        )

    }

}

*export* default Input

Now let's create the parent component for this input component. I'm going to create a new file and call it *FocusInput.js* and within the file I'm going to create a class component and in the JSX I will add the input component(<Input/>) that we have just created and with that I will also add a button with the text “focus input”. now what we want to achieve here is when we click on the focus input button in the parent component the input element in the child component should receive the focus. And we achieve that using refs on this input component. Just like previous we have three simple steps. in the parent component which is focus input component we are going to create a ref using the createRef method.

1. Step one, in the parent component which is the focus input component we are going to create ref using the createRef method. so add a constructor and then create a property this dot **componentRef** is equal to react dot createRef().
2. Second step re-attach the ref to the component. so on our input component(<Input/>) we are going to add the ref attribute and then attach this dot componentRef.
3. The final step add a click handler to the button and within the click handler called the child component method using the ref. so onClick is equal to this dot clickHandler which can be defined right after the constructor. clickHandler is going to be equal to an arrow function and within the body this dot component ref dot current which gives us access to the input component and then we can call the focusInput() method. Now this focusInput() method is of course defined an *Input.js* . let's go back to App component and include the focus input component(<FocusInput/>) so instead of <Refsdemo/>.

import React, { Component } from 'react'

import Input from './Input'

class FocusInput *extends* Component

{

    constructor(props)

    {

        super(props)

        this.componentRef=React.createRef()

    }

    clickHandler=()=>

    {

        this.componentRef.current.focusInput() *// defined in Input.js*

    }

    render()

    {

        return (

        <div>

            <Input *ref*={this.componentRef} />

            <button *onClick*={this.clickHandler}>Focus input</button>

        </div>

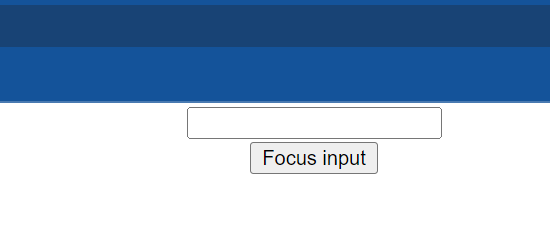
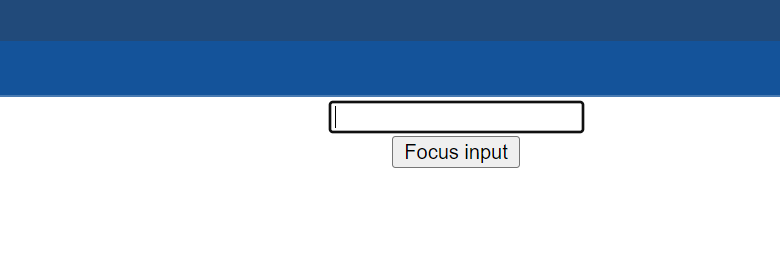
        )

    }

}

*export* default FocusInput

if we now save all the files and take a look at the browser we can see that we have an input element and a button right below it. If I click on the button the input element receives the focus.

So in special circumstances if at all we need a ref to a child component from a parent component know that it is definitely possible. The component however has to be a class component. **refs cannot be attached to functional components please do keep that in mind.**

**Forwarding refs**

refs forwarding is a technique for automatically passing a ref through a component to one of its children. this concept is best understood with an example so let's dive right into the code and then later on go over how it works.

I'm going to start off by creating a new file *FRInput.js* which stands for forward ref input, within the file I'm going to create a functional component and within the JSX I will add an input element. Next I am going to create another file *FRParentInput.js* (forward ref parent input) and within the file I will create a class component and in the JSX i will simply add the **<FRInput/>** component I will also add a button that says “focus input”. Back in App component I will include the **<FRParentINput/>** .

import React from 'react'

function FRInput()

{

    return (

        <div>

            <input *type*='text' />

        </div>

    )

}

*export* default FRInput

import React, { Component } from 'react'

import FRInput from './FRInput'

class FRParentInput *extends* Component

{

    render()

    {

        return (

        <div>

            <FRInput />

            <button>focus input</button>

        </div>

        )

    }

}

*export* default FRParentInput



if we save all the files and take a look at the browser we should be able to see the input and the button again. Similar to previous, what we want to achieve is when we click on the button and the parent component the input in the child component should receive focus. however unlike the previous where the ref was pointing to the class component, in this (this time) we will use the forwarding ref technique to allow the parent component to directly reference the native input element. now we have four simple steps:

1. First step create a ref and the parent component. so in *FRParentInput.js* I'm going to add constructor and I will add this dot **inputRef** is equal to react dot createRef.
2. Second step attach the ref to the child component using the ref attribute. so on <FRInput/> ref is equal to this dot inputRef.

import React, { Component } from 'react'

import FRInput from './FRInput'

class FRParentInput *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef=React.createRef()

    }

    render()

    {

        return (

        <div>

            <FRInput *ref*={this.inputRef}/>

            <button>focus input</button>

        </div>

        )

    }

}

*export* default FRParentInput

1. Third step we need to forward this ref to the input element in the child component. And ref forwarding can be achieved using the **forwardRef()** method from the react library. so in *FRInput.js* which is our child component we are going to modify how we create the functional component. I will comment out the existing code and show we the new way to define it.

First i am going to simply replace the traditional function with an arrow function. const FRInput is going to be equal to an arrow function we can see that we have not changed any functionality. we have simply rewritten the functional component using arrow functions. To forward ref we will use the react dot forwardRef method. the method is going to be assigned to the constant **React.forwardRef()**. The **forwardRef()** method takes in a component as its parameter, since we already have the functional component I will simply cut the arrow function and paste it inside the parentheses. so the arrow function is passed as a parameter to the **forwardRef()** method. We know that every functional component receives props as its parameter but let me now tell we when a component is passed as a parameter to the createRef method it receives the ref attribute as its second parameter. we can use this ref parameter and pass it as a value to the ref attribute on the native input element. so on the input element **ref** attribute and assign the value of the ref parameter. This ref parameter will point to the value of the **ref** attribute from the parent component or in other words the ref is being forwarded from the parent component to the native input element.

import React from 'react'

*// function FRInput()*

*// {*

*//     return (*

*//         <div>*

*//             <input type='text' />*

*//         </div>*

*//     )*

*// }*

const FRInput = React.forwardRef((props,ref)=>

{

    return (

        <div>

            <input *type*='text' *ref*={ref}/>

        </div>

    )

})

*export* default FRInput

1. So for our final step back in *FRParentInput.js* , we can define the clickHandler. button on click is equal to this dot clickHandler which we can define after the constructor. So clickHandler is going to be equal to an arrow function and within the body this dot inputRef dot current dot focus because we are using forward ref technique; ref dot current points to the native input element and not the FRInput component instance.

import React, { Component } from 'react'

import FRInput from './FRInput'

class FRParentInput *extends* Component

{

    constructor(props)

    {

        super(props)

        this.inputRef=React.createRef()

    }

    clickHandler=()=>

    {

        this.inputRef.current.focus()

    }

    render()

    {

        return (

        <div>

            <FRInput *ref*={this.inputRef}/>

            <button *onClick*={this.clickHandler}>focus input</button>

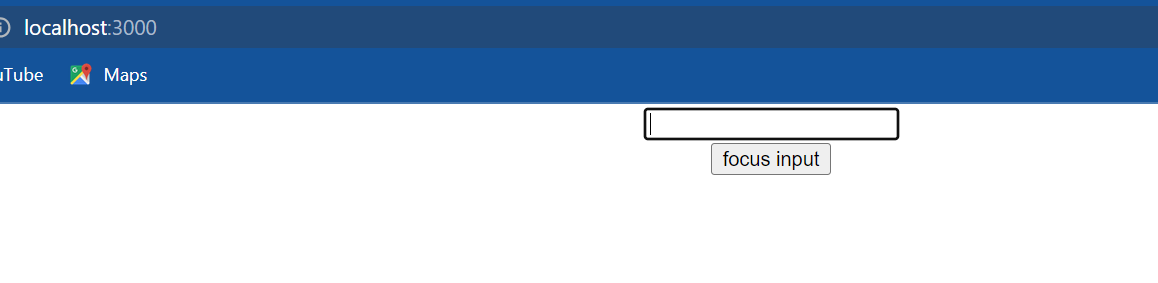
        </div>

        )

    }

}

*export* default FRParentInput



If we save all the files and go back to the browser click on the button we can see that the input element receives the focus.

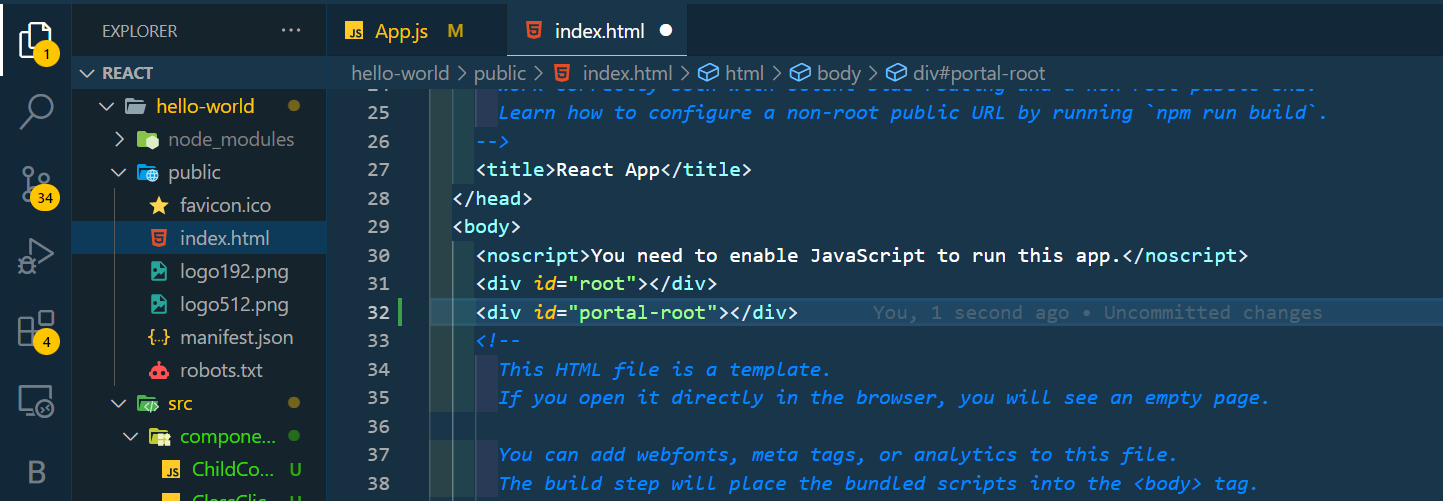
So let me quickly go over the code one last time: the parent component creates a ref and then attaches it to the child component instance. the child component sees that ref and tells the parent hey I am NOT the guy we are looking for, we want the native input element right let me directly introduce him to we. So the child component receives the ref from the parent component and attaches it to the native input element. now the parent component can directly access this input element using this dot reference variable dot current. the child element is basically forwarding this reference.

now ref forwarding is something we are rarely going to use in our application. this might come in handy when we are working with some libraries or when we're dealing with higher-order components but it's definitely a good idea to be aware of the different concepts in react.

**Portals**

React portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. now what does that mean? well so far we have had one DOM element in our HTML that we were mounting the react application into. if I go into the public folder and **index.html** we can see that element is the element with **id** is equal to root. In our **index.js** file we use react DOM render and mount our app component on to the root element. so if we take a look at the browser in the DOM tree every single react component in our application falls under the **root** element that is the div element with ID is equal to root( <div id=”root”> ). what react portals provide is the ability to break out of this DOM tree. so we can render a component onto a DOM node that is not under this root element. let me show we how to use portals with a simple example

the first step is to add a DOM node that falls outside the root element. so in **index.html** right below the root element I am going to add another div tag with ID is equal to “portal-root”.



for the second step I am going to create a new component. I will create a new file called *PortalDemo.js* and within the file I am going to create a functional component, in the JSX I will simply add a heading that says “portals demo”. after that I will include the component in App component.

import React from 'react'

function PortalDemo()

{

    return (

        <h1>

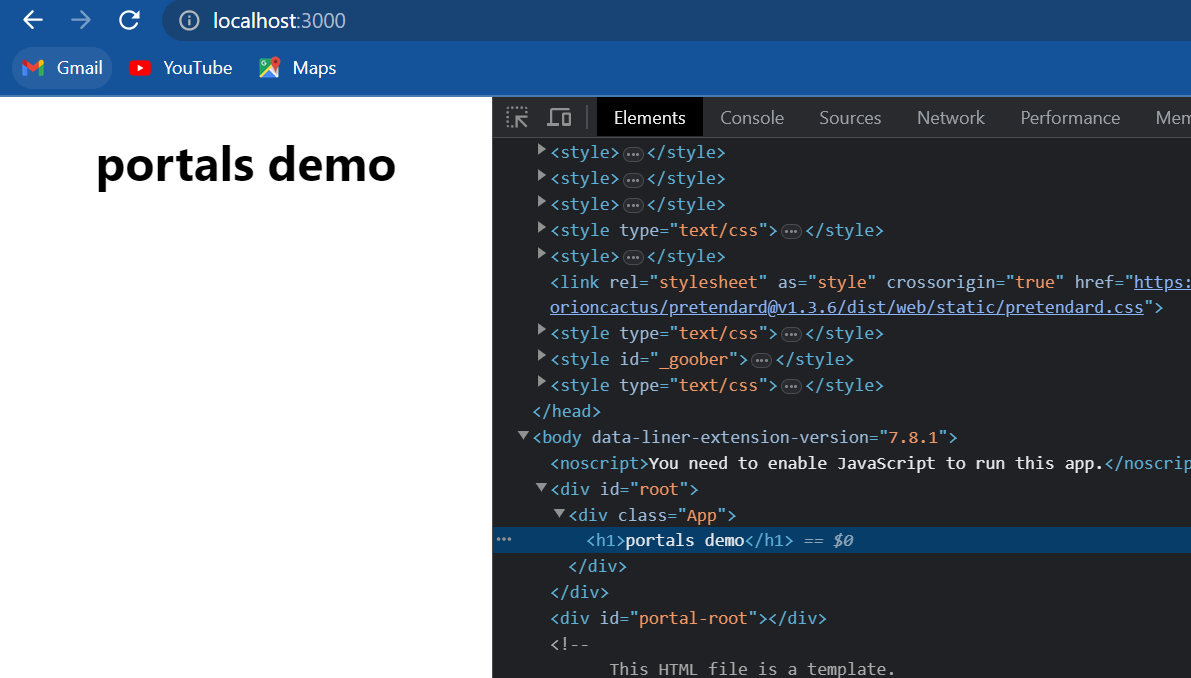
            portals demo

        </h1>

    )

}

*export* default PortalDemo



if we now save all the files and take a look at the browser we should be able to see the text “portals demo”. if I inspect the element we can see that the element falls under the root element and not under the portal-root element, Let's change that.

For step three we will use **ReactDOM.createPortal** method to insert this component under the portal-root node. so in PortalDemo component and the top we need to import the react DOM package. then in the render method instead of simply returning the JSX we are going to return **ReactDOM.createPortal()** and the method takes two parameters; first parameter is the JSX we want to render which is the heading “portals demo” and the second parameter is the DOM node to mount the element on to. So document dot get element by ID portal-root which is the ID of our second div element in **index.html**.

import React from 'react'

import ReactDOM from 'react-DOM'

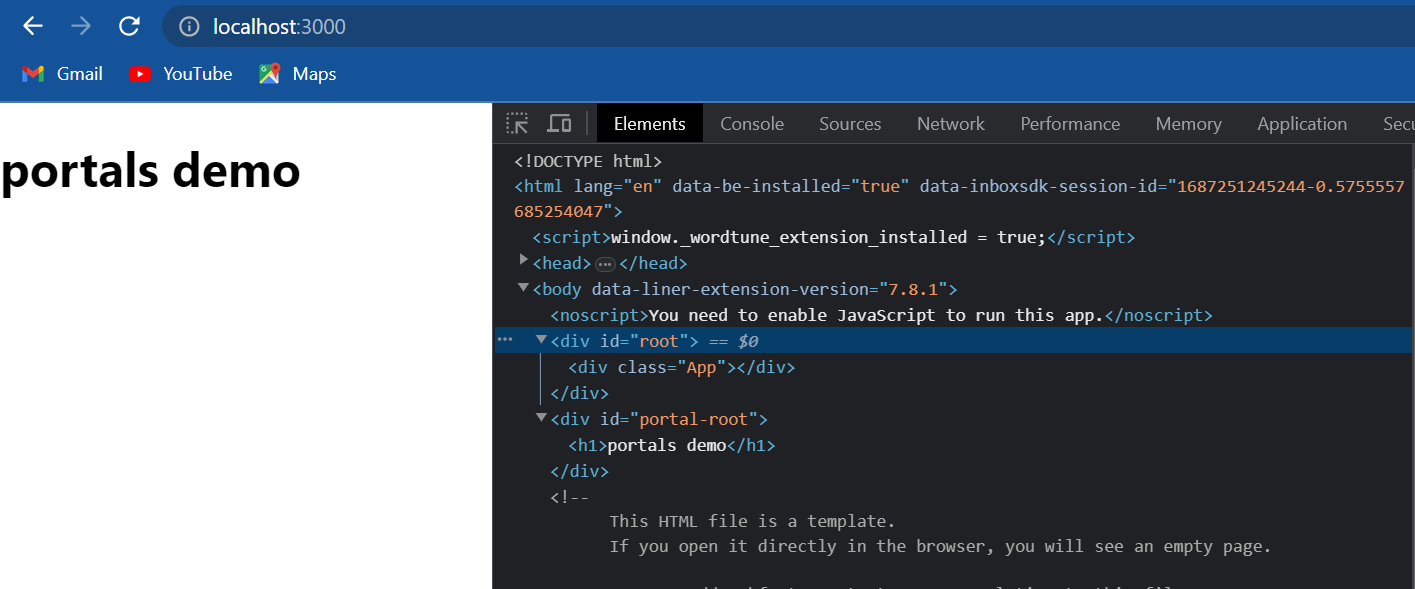
function PortalDemo()

{

    return ReactDOM.createPortal(<h1>portals demo</h1>,document.getElementById('portal-root'))

}

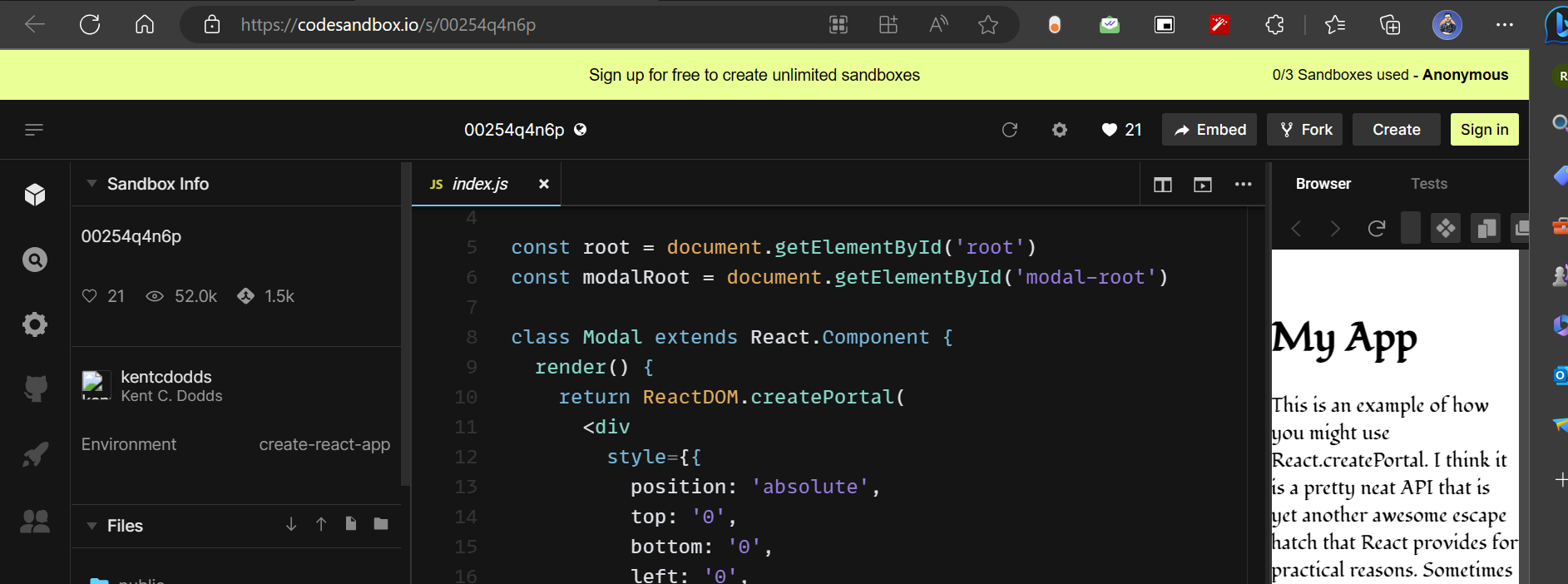
*export* default PortalDemo

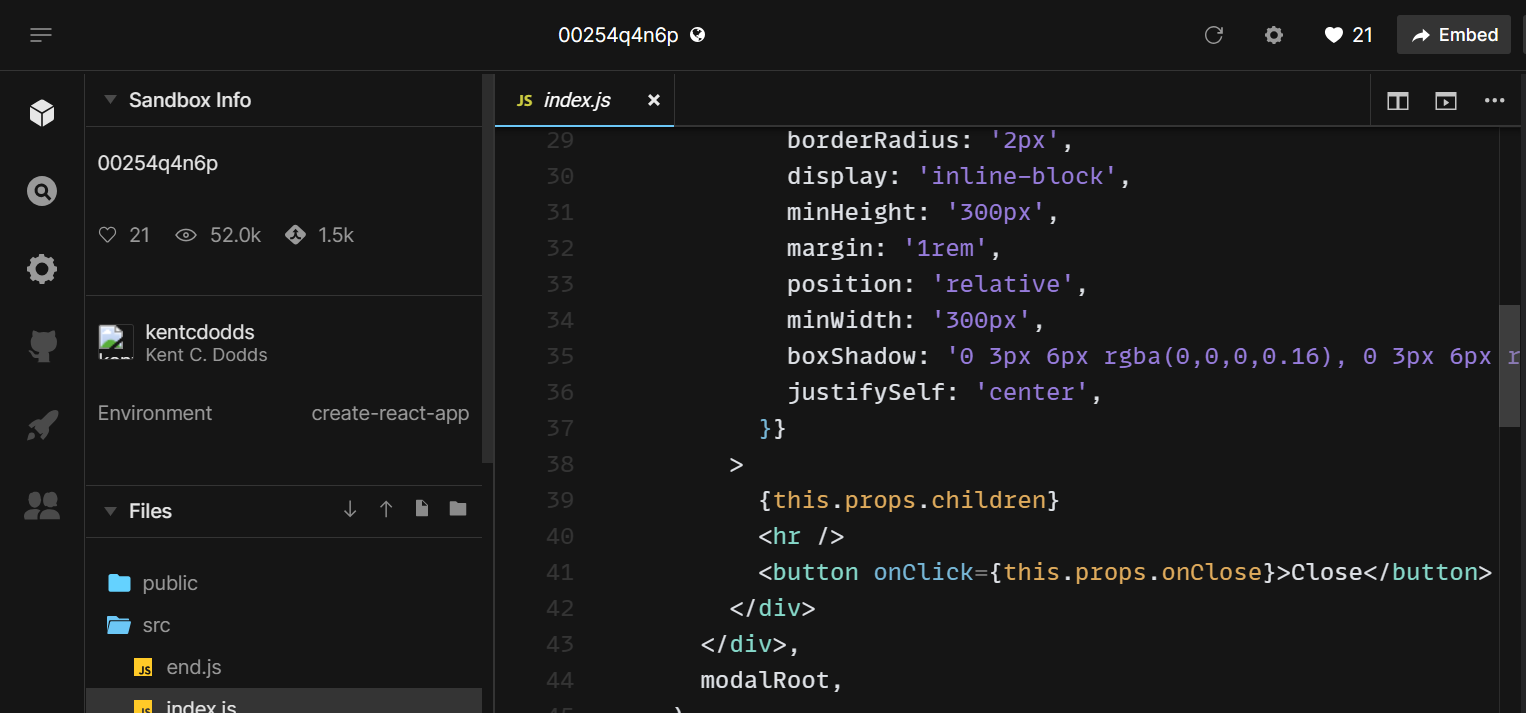


alright if we format the code save the files and take a look at the browser we should still be able to see the text portals demo but this time when I inspect the element we can see that the h1 tag is under the portal-root DOM node and not the root DOM node. so in our react application even though all the components are children to the App component and the App component is mounted onto the root DOM node it is possible to break away from that and mount on any DOM node that we wish to using react portals. let me tell we that the first parameter to create portal can be any element that react can render, it can be numbers strings JSX or even components.

Now we know what portals can do for us, the next question is why do we need them? one of the use cases which is brought up is having to deal with parent components CSS when that child component is a modal, a pop-up or a tooltip. There is already an example created by Kentcdodds that we can make use of. I'm at this codesandbox.io [link](https://codesandbox.io/s/00254q4n6p)

first we will take a look at the demo with portals and then I will remove the portal and show we what happens. let me start with ***index.html,*** we can see that we have two div tags one with ID equals root and the other with ID modal-root. In ***index.html*** we have a reference to both the elements. next we have a Model component there is a bit of styling and within the model content.





We display whatever is passed as children. And we have a button to close the model. but the most important part is that we are using a portal so we have **ReactDOM.createPortal** the first parameter is the JSX and the second parameter is the model-root DOM node. In the app component there is a bit of text which we can see on the right hand side and there is a button that works with the state to open and close the model. we also have the model component nested inside app component. if I now go to the UI, show model and it opens close and it closes. so it works perfectly fine. now let me remove the portal, so the model is rendered under the app component which falls under the root DOM node so I'm going to remove ReactDOM.createPortal and the second parameter. if we now head back to the UI and click on the model we can see that the model breaks. Why?, well if we can see the code in the app component the model component falls under a div tag which has a max width of 400 as a result the model also has a width restriction now and that messes up the UI. so sometimes it's useful to insert a child into a different location in the DOM and portals help we do that.

The last point to discuss about portals is event bubbling. what we should know about portals is that even though a portal can be anywhere in the DOM tree it behaves like a normal react child in every other way, this includes event bubbling. so an event fired from inside a portal will propagate to ancestors in the containing react tree even if those elements are not ancestors in the DOM tree.

I have another codepen.io example [link](https://codepen.io/gaearon/pen/jGBWpE) to help we understand. In the HTML we can see that we have two DOM nodes one for app component and the other for a modal component. if we take a look at the JSX we can see that we have the parent component this component has **clicks** state property which tracks the number of times we have clicked the button in the UI right here. The parent component includes the modal component which in turn includes a child component and it is this child component which has the click button and of course the modal component is rendered using ReactDOM.createPortal, so it is a portal. So the parent component is mounted on the app root DOM node whereas the modal is mounted on the modal-root DOM node but if we click on the button in the UI we can see that the app still works; number of clicks is now one, I click again it is now two, click again this three and so on. if we were to consider just the DOM tree this event bubbling would have never happened. since an event fired from inside a portal will propagate to ancestors in the containing react tree even bubbling still works. so a portal behaves like a normal react in every other way.

**Error boundary**

Let's learn yet another feature in react which is error boundaries. if we can recollect from the lifecycle hooks I briefly mentioned about the error handling phase which includes two life cycle methods **static getDerivedStateFromError(error)** and **componentDidCatch(error,info).** We have seen so far we see that runtime errors during rendering could put our application in a broken state. React basically unmount the whole react component tree. what would be great is if we could catch the errors anywhere in the component tree and display a fallback UI. this is where error boundaries take the spotlight.Now what exactly is an error boundary.

* It is a class component that implements either one or both of the lifecycle methods ***getDerivedStateFromError*** or ***componentDidCatch*** becomes an error boundary.
* The static method ***getDerivedStateFromError*** is used to render a fallback we ID after an error is thrown and the ***componentDidCatch*** method is used to log their information.

let's understand this better with an example. I'm going to go back to components folder create a new file *Hero.js,* within the file I'm going to create a functional component and this component will accept **heroName** as a prop and will render the same. For our understanding of error boundaries though we need to throw an error. so what I will add is if the heroName is Joker throw an error that says not a hero. now back in app component I will include two heroes the first one heroName is equal to “Batman” and the second one heroName is equal to “Superman”.

**Hero.js**

import React from 'react'

function Hero({heroName})

{

    if(heroName=='joker')

    {

        throw new Error('not a hero')

    }

    return (

        <div>

            {heroName}

        </div>

    )

}

*export* default Hero

**App.js**

import logo from './logo.svg';

import './App.css';

import Hero from './components/Hero'

function App() {

  return (

    <div *className*="App">

      <Hero *heroName*="batman"/>

      <Hero *heroName*="superman"/>

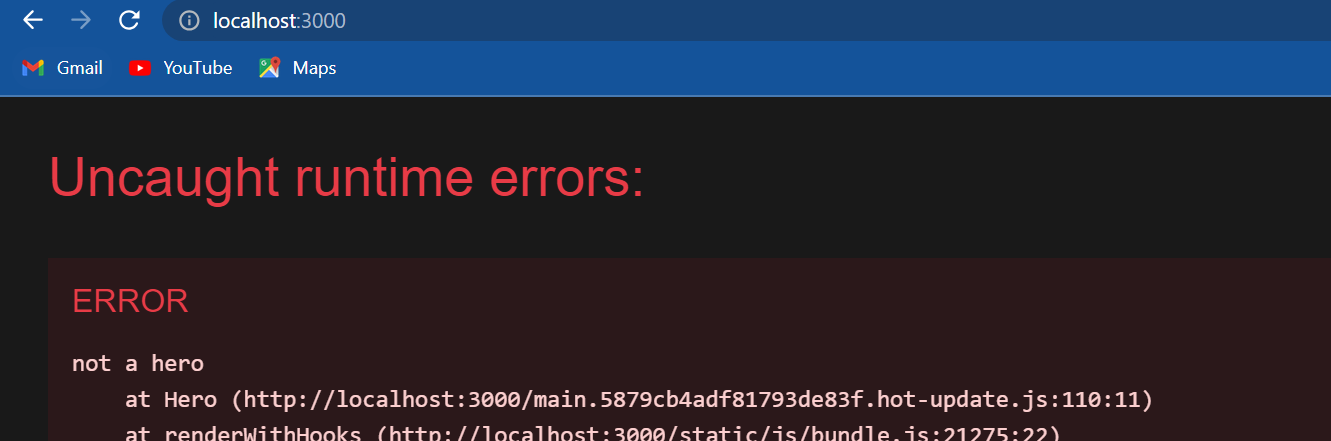
    </div>

  );

}

*export* default App;

if we take a look at the browser we can see both the hero's Batman and Superman. Now I am going to add another hero but this time passed “Joker” as the heroName.



if we now take a look at the browser we can see that our entire application crashes. this obviously is not good. what we want is if a particular component throws an error, only that component should fall back into a UI and the remaining components should be unaffected. let's see how to achieve that with an error boundary.

I'm going to create a new file called *ErrorBoundary.js* , within the file I'm going to create a new class component. for this class component to become an error boundary we need to define either the **getDerivedStateFromError** or **componentDidCatch** lifecycle methods.

let's start with the first one **static getDerivedStateFromError()** and it receives the error as a parameter within the body we are simply going to return the new state object, what we will do is set a property called **hasError** to true. Of course we don't have state right now so let's add it in our constructor Const initialize a state property called hasError to false. so what we are effectively doing is if at all there is an error when rendering any of the components we are setting the state has error property to true. This state property can now be used to create a fallback UI, so in the render method if this dot state dot has error then we return an <h1> tag that says “something went wrong”, if there is no error we return this dot props dot children. this dot props dot children refers to the component we are actually rendering, this will make more sense in just a minute but let me tell we that our error boundary is now complete.

**ErrorBoundary.js**

import React, { Component } from 'react'

class ErrorBoundary *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {hasError:false}

    }

*static* getDerivedStateFromError(error)

    {

        return {hasError:true}

    }

    render()

    {

        if(this.state.hasError)

        {

            return <h1>something went wrong</h1>

        }

        return this.props.children

    }

}

*export* default ErrorBoundary

The final step is to wrap the components with this error boundary so it App component wrap all the hero components with the error boundary component**( <ErrorBoundary> <ErrorBoundary/>** )and make sure to import the component at the top.

**App.js**

import logo from './logo.svg';

import './App.css';

import Hero from './components/Hero'

import ErrorBoundary from './components/ErrorBoundary'

function App() {

  return (

    <div *className*="App">

      <ErrorBoundary>

        <Hero *heroName*="batman"/>

        <Hero *heroName*="superman"/>

        <Hero *heroName*="joker" />

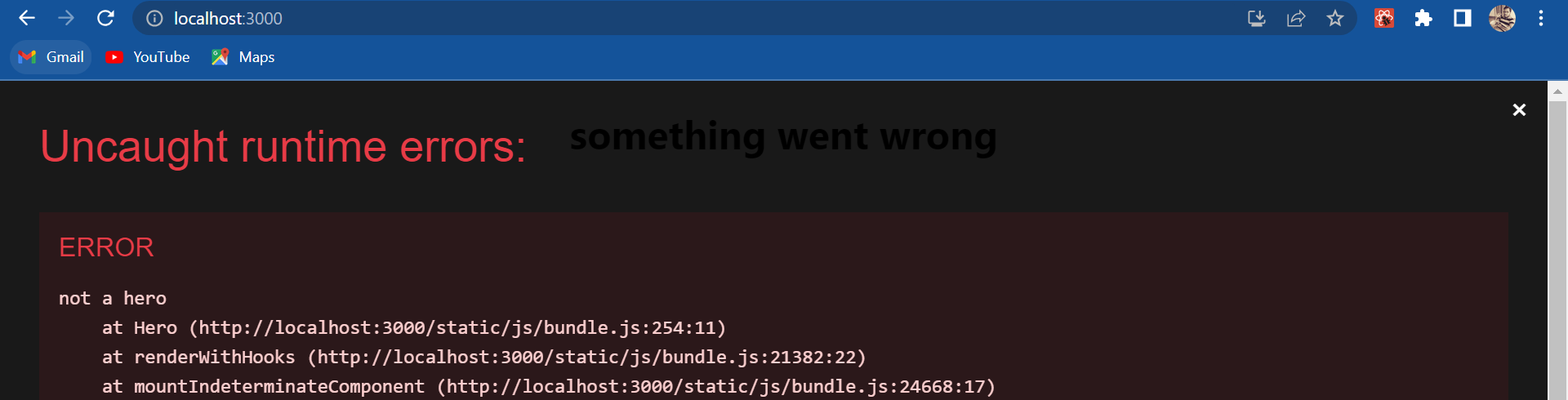
      </ErrorBoundary>

    </div>

  );

}

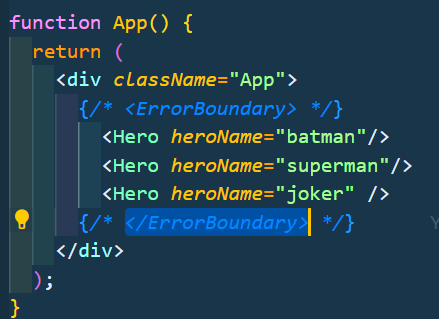
*export* default App;



now if I save all the files and take a look at the browser, we still see the error this might be confusing but let me tell we this is the intended behavior. the react team have mentioned that error boundaries are primarily useful for production but in development they want to make errors as highly visible as possible so we will always see this error during development but what we can do is on the top right click on the close button and we now have our application back and we can see that we have the text “something went wrong” being displayed.



if I go back to code and quickly comment out there <ErrorBoundary> head back to the browser click on the close button we can see that nothing is displayed.





So this is the difference. Error boundaryies will catch the error and display a fallback UI which in our case is the text “something went wrong”.

Now a very important point to discuss about error boundaries is where to place them. Right now our error boundary encloses all the hero components, so if there is an error everything inside the error boundary is hidden and the fallback UI is rendered. this however might not be ideal for example consider an e-commerce site where we display a thousand products. just because there is an error in one of the products it would not be a great idea to hide the other 999 products. similarly in our example it's not a good idea to hide the other two heroes when the third hero is throwing an error. so in App component what I will do is wrap each hero component(<hero>) with the error boundary(<ErrorBoundary>). Ideally we would want this as a reusable component but this will do for now.

import logo from './logo.svg';

import './App.css';

import Hero from './components/Hero'

import ErrorBoundary from './components/ErrorBoundary';

function App() {

  return (

    <div *className*="App">

      <ErrorBoundary>

        <Hero *heroName*="batman"/>

      </ErrorBoundary>

      <ErrorBoundary>

        <Hero *heroName*="superman"/>

      </ErrorBoundary>

      <ErrorBoundary>

        <Hero *heroName*="joker" />

      </ErrorBoundary>

    </div>

  );

}

*export* default App;



if I go back to the browser we can see that we still have the error overlay. I close it and we can now see that the first two heroes are displayed and only the third hero we have the fallback UI; “Batman”, “Superman”, and “something went wrong”. the placement of the error boundary is completely up to us. We can just wrap the top level component or wrap any nested individual components so that only that component has a fallback UI leaving the rest of our user interface working as expected.

Now that I did mention to lifecycle methods so let's get to the second one **componentDidCatch.** So right after getDerivedStateFromError we have componentDidCatch which takes two parameters: error and info which is the information related to the error and this method is pretty much used to log the errors. So if we have a logging service we can call it passing in the error and the info parameters. for now I will simply log them to the console. this will seem redundant though as we will see in just a second.

**ErrorBoundary.js**

import React, { Component } from 'react'

class ErrorBoundary *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {hasError:false}

    }

*static* getDerivedStateFromError(error)

    {

        return {hasError:true}

    }

    componentDidCatch(error,info)

    {

        console.log(error)

        console.log(info)

    }

    render()

    {

        if(this.state.hasError)

        {

            return <h1>something went wrong</h1>

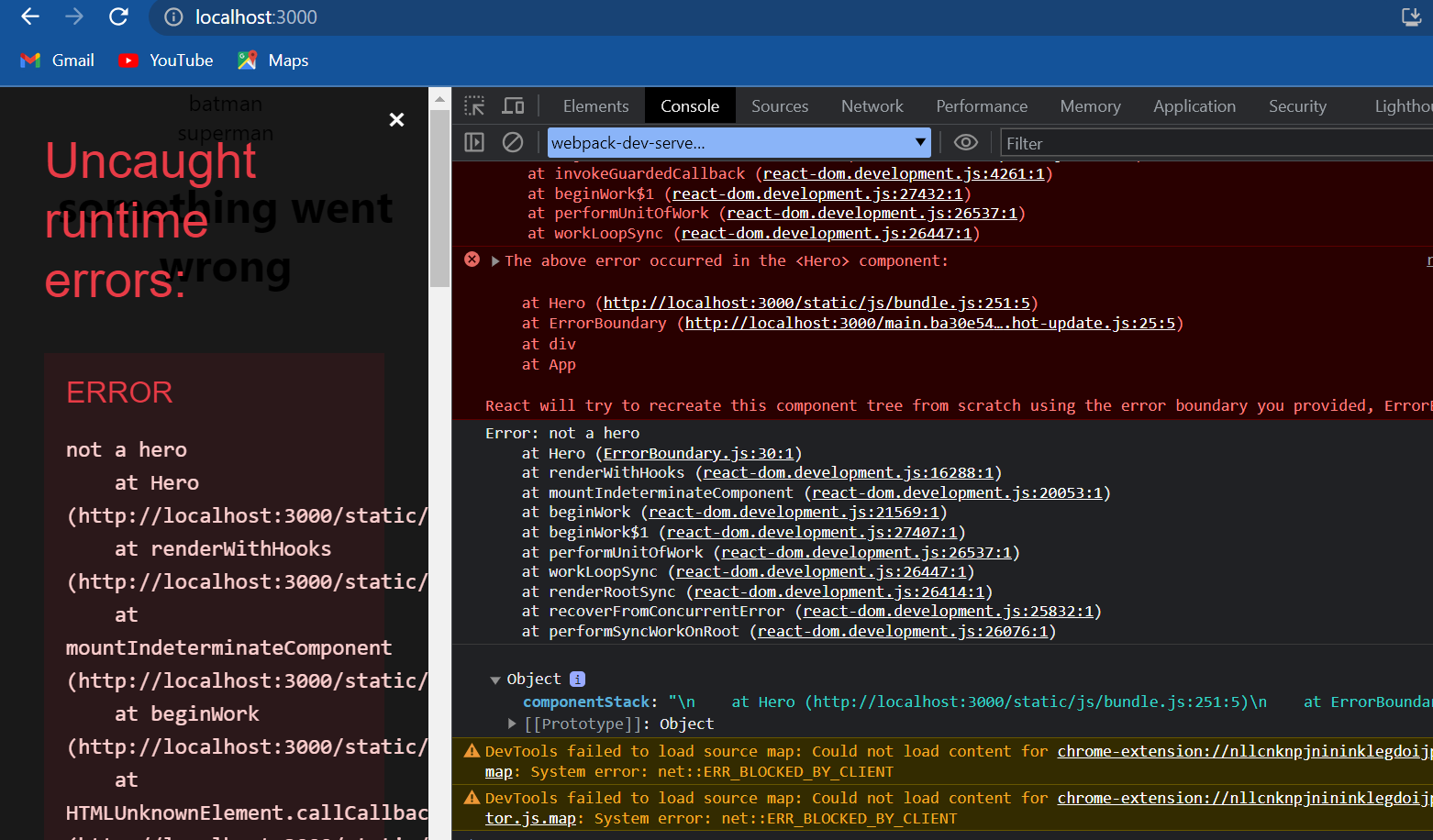
        }

        return this.props.children

    }

}

*export* default ErrorBoundary



we need to save the file and head to the browser and when I open the developer tools and take a look at the console we can see that we have our console logs we have the error object and the information related to that error but as it turns out during development react automatically logs every error to the console. so the same is already logged as we can see here but this is what the error and info object will contain in componentDidCatch.

Now the final point is that we should know that our boundaries catch errors during rendering in lifecycle methods and in the constructors of the whole tree below them. however they do not catch errors inside event handlers. if we have an **onClick** handler and want to catch an error we just need to use the regular try catch statements and not error boundaries.

Summary:

* Error boundaries are react components that catch JavaScript error in their child component tree, log those errors, and display a fallback UI.
* A class component becomes an error boundary by defining either or both of ***getDerivedStateFromError*** and ***componentDidCatch*** lifecycle methods.

In our example, in the lifecycle method we set the hasError state to true which causes the fallback UI to re-rendered.

* The placement of the error boundary also matters as it controls if the entire application should have the fallback UI or just the component causing the problem.
* Error boundaries basically provide a way to gracefully handle error in application code.

**High order component**

Let’s understand why there is a need for something like a higher-order component. I'm going to create a new file called *ClickCounter.js* and within the file I'm going to create a class component and for the JSX I will add a button with the text “clicked x times” after formatting the code I will add this component in app component.

import React, { Component } from 'react'

class ClickCounter *extends* Component

{

    render()

    {

        return (

        <div>

            <button>clicked x times</button>

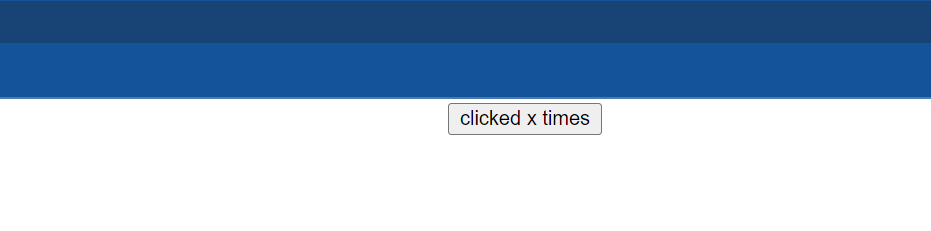
        </div>

        )

    }

}

*export* default ClickCounter



If we now save the files and take a look at the browser we should be able to see a button that says “click x times”. now let's go back and implement the actual click counter. I'm going to start off by adding the constructor and creating a state property called **count** initialized to zero. I will also add a click handler on the button, so button onClick is going to be equal to this dot incrementCount. now we can define the handler **incrementCount()** is going to be equal to an arrow function within the body we call setState() to increment the count value by one. Since we need the previous state to increment the value the argument to setState() will not be a simple object. instead it takes in a function which gets previous state as a parameter and returns the new state. so **prevState** and return the new state count set the prevState dot count plus one. finally in the render method I will extract count from the state and include it in the JSX const count this equal to this dot state and instead of rendering X I'm going to render count.

import React, { Component } from 'react'

class ClickCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <button *onClick*={this.incrementCount}>clicked {count} times</button>

    }

}

*export* default ClickCounter



If we now save the files and take a look at the browser we can see that when I click on the button the count value increments clicked 5 times, so we have successfully created a click counter.

we show this to the client and the client is really happy about it but the client now comes up with a new requirement. hey I want another piece of UI which works very similar to this click counter instead of button clicks though I want a heading which indicates the number of times we hover over it. so basically a click counter with click functionality replaced by our functionality. let's implement that.

I will create a new file called *HoverCounter.js* and within the file I will create a class component and for the JSX I will add a heading that says “hovered x times”. On the heading we will handle the **onMouseOver** event. The handler is going to be this dot **incrementCount.** now we need a counter functionality and that was already implemented in the *ClickCounter.js* , so let me simply copy the constructor and the handler and paste it. and I'm going to format the code now in the render method I will extract the count value and display it as part of the heading const count is equal to this dot state and “hovered {count} times”. finally back in app component we can include the HoverCounter.

import React, { Component } from 'react'

class HoverCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <h2 *onMouseOver*={this.incrementCount}>hovered {count} times</h2>

    }

}

*export* default HoverCounter



if we now save the files and take a look at the browser, I hover on the heading and the count increases. so we have successfully created the hover counter as well.

Now the client comes back with another requirement they want an input element that counts the number of key presses. for example on key up in an input element they want to increment a counter value and display it. we can of course implement it just like the click counter or the hover counter but we start to realize that we are duplicating code and not really reusing the functionality. In our click counter and how our counter we have the counter functionality which could have been reused but instead has been duplicated. so if 10 different components needed a counter functionality we would be writing the exact same code over and over again. So the question now is how can we reuse this code. The immediate thought is to **lift this state to the parent component and pass down the handler as a prop**. So define the counter functionality in app component and provide the state and the handler as props to click counter and hover counter. This would definitely work in our scenario where we have the counter components as children of the same parent. But imagine a scenario where the counter components are scattered in the react component tree, lifting the state would definitely not be the correct solution. So there is a need to share common functionality between components without repeating code and that is where the concept of higher-order components comes into picture.

in the next video we will see what exactly is a higher-order component and how can we create one to reuse code in a counter example

**what is a higher order component (HOC)?**

Higher order component which will be referred to as HOC from now on is a pattern where a function takes a component as an argument and returns a new component. In simple code it will look something like this:

***const NewComponent = higherOrderComponent(originalComponent)***

Typically an HOC adds additional data or functionality to the original component. so the new component can also be referred to as enhanced component:

***const EnhancedComponent = higherOrderComponent(originalComponent)***

if we were to understand this from a non technical point of view we have Iron Man is equal to with suit passing in Tony Stark as a parameter.

***const IronMan = withSuit(TonyStark)***

Here TonyStark is the original component withSuit is the function that will enhance TonyStark and return Iron Man which of course is the enhanced component. From react point of view we have a function which accepts the original component adds functionality and returns an enhanced component or in other words we have the HOC pattern. Alright now that we know what the HOC pattern is we can head back to code and learn how to implement a basic HOC and then implement it for the counter example.

let's start with the bare minimum code for an HOC. I'm going to create a new file called *withCounter.js*. Not going to worry much about the folder structure, I'm going to create it in the components folder *withCounter.js* , filename is an HOC convention which we will discuss in a bit. Now let us create our HOC. First let's import react; **import React from ‘react’**. next let us create a function, the function will of course be an arrow function const **updatedComponent** is equal to an arrow function, the HOC accepts the original component as its parameter. the HOC also returns a new component so within the function body let's create a new component. **class NewComponent extends React.Component**. In a class component the render method is a required method but what do we render we simply return the original component, so render and return the original component (<OriginalComponent/>). now that the **NewComponent** has been defined we return new component in the arrow function returned **NewComponent** and there we go we have a very first higher-order component. it is a function that accepts the original component and returns a new component. this HSE though doesn't really add anything to the original component, let's change that. On the original component in the JSX I'm going to add a prop **name** is equal to “vishwas”. now to be able to use this HOC function we need to export it; **export default UpdatedComponent.**

**withCounter.js**

import React from 'react'

const UpdatedComponent=OriginalComponent=>

{

    class NewComponent *extends* React.Component

    {

        render()

        {

            return <OriginalComponent *name*='vishwas'/>

        }

    }

    return NewComponent

}

*export* default UpdatedComponent

Now let's apply this HOC pattern to our click counter and hover counter. In click counter I will import updated component from with counter and while exporting I will call the updated component function passing in the click counter component. let's do the same for hover counter as well I'm going to copy this import statement paste it in our counter and export the updated component with hover counter.

**ClickCounter.js**

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class ClickCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <button *onClick*={this.incrementCount}>clicked {count} times</button>

    }

}

*export* default UpdatedComponent(ClickCounter)

**HoverCounter.js**

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class HoverCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <h2 *onMouseOver*={this.incrementCount}>hovered {count} times</h2>

    }

}

*export* default UpdatedComponent(HoverCounter)

So what we are doing here is instead of exporting the click counter or the hover counter component we export the higher-order component. the HOC in addition to be the click counter or the hover counter. now has a new prop called name. so in both the components we can actually render this prop. so in the JSX this dot props dot name and the same in hover counter as well.

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class ClickCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <button *onClick*={this.incrementCount}>{this.props.name} clicked {count} times</button>

    }

}

*export* default UpdatedComponent(ClickCounter)

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class HoverCounter *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount =()=>

    {

        this.setState(prevState =>

            {

                return {count:prevState.count+1}

            })

    }

    render()

    {

        const {count}=this.state

        return <h2 *onMouseOver*={this.incrementCount}> {this.props.name} hovered {count} times</h2>

    }

}

*export* default UpdatedComponent(HoverCounter)

**App.js**

import logo from './logo.svg';

import './App.css';

import HoverCounter from './components/HoverCounter'

import ClickCounter from './components/ClickCounter';

function App() {

  return (

    <div *className*="App">

      <ClickCounter/>

      <HoverCounter/>

    </div>

  );

}

*export* default App;

**withCounter.js**

import React from 'react'

const UpdatedComponent=OriginalComponent=>

{

    class NewComponent *extends* React.Component

    {

        render()

        {

            return <OriginalComponent *name*='vishwas'/>

        }

    }

    return NewComponent

}

*export* default UpdatedComponent

if we now save the files and take a look at the browser we should be able to see the prop being rendered “vishwas” in the button and “vishwas” in the heading:



so our HOC injects a name prop to any component required. for our counter example though that is not really what we want, we want the counter functionality to be shared amongst the components. So let's modify our HOC. I will copy the constructor and the incrementCount() method from the click counter component into the HOC. so copy the code which has to be shared and paste it in the HOC. The **count** state and the **incrementCount()** method is the common functionality we want to share. so the new component now maintains a count state and a method to increment that state this is the common functionality we want to share. I will remove the same piece of code from both click counter as well as hover counter. This will avoid duplication of code.

Now in the HOC we need to pass down the state and the increment count method as props so that the original component can make use of that functionality. I will remove the **name** prop and instead add count is equal to this dot state dot count and incrementCount is equal to this dot incrementCount. we can now make use of these props in click counter and hover counter.

**withCounter.js**

import React from 'react'

const UpdatedComponent=OriginalComponent=>

{

    class NewComponent *extends* React.Component

    {

        constructor(props)

        {

            super(props)

            this.state = {count:0}

        }

        incrementCount =()=>

        {

            this.setState(prevState =>

                {

                    return {count:prevState.count+1}

                })

        }

        render()

        {

            return <OriginalComponent *count*={this.state.count} *incrementCount*={this.incrementCount}/>

        }

    }

    return NewComponent

}

*export* default UpdatedComponent

In click counter we are going to de-structure **count** and **incrementCount** from this dot props and I will get rid of the state (const {count}=this.state). In the onClick handler it is now going to be just increment count. I will also get rid of the name that was being rendered. Now let's repeat the same for our counter as well const **count** comma **incrementCount** is going to destructure from this dot props. I will remove this keyword from on mouse over event handler and also the name prop.

**ClickCounter.js**

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class ClickCounter *extends* Component

{

    render()

    {

*//const {count}=this.state*

        const {count,incrementCount}=this.props

        return <button *onClick*={incrementCount}>clicked {count} times</button>

    }

}

*export* default UpdatedComponent(ClickCounter)

**HoverCounter.js**

import React, { Component } from 'react'

import UpdatedComponent from './withCounter'

class HoverCounter *extends* Component

{

    render()

    {

*//const {count}=this.state*

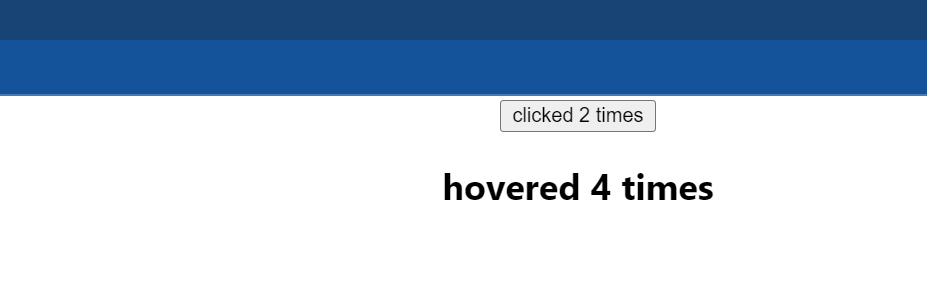
        const {count, incrementCount}=this.props

        return <h2 *onMouseOver*={incrementCount}>hovered {count} times</h2>

    }

}

*export* default UpdatedComponent(HoverCounter)



if we now save the files and take a look at the browser we should see that both the counter components are working as before but the difference now is that we are reusing code rather than duplicating it that is what HOC s bring to the table.

let me go over the code one more time so that we get a proper understanding of how HOC s work:

I will begin from *App.js* here we have included the click counter component the <ClickCounter/> is defined in *ClickCounter.js* file now over here we can notice that we are exporting the updated component HOC passing in the ClickCounter. the UpdatedComponent is defined in *withCounter.js*. let's take a look at that file the UpdatedComponent is a function which accepts a component or an OriginalComponent as its parameter and returns a NewVomponent. In our case the OriginalComponent refers to the ClickCounter. the NewComponent has functionality to maintain the state of count property and also a method to increment that count property, both of them are passed as props to the OriginalComponent. the OriginalOomponent enhanced with these props are then returned. The control goes back to click counter where the count and increment count props passed in from the HSE are destructured and used in the return statement. so the HOC is basically taking care of maintaining the state and incrementing it. when we click on the button the count is incremented and displayed in the UI. the same execution flow also happens for the hover counter component but what we should know is that both the components received separate state. Incrementing the count in click counter will not affect the hover counter and vice-versa.So that is how an HOC works, it is a function that accepts a component and returns an enhanced component.

Now let's talk about the naming convention. The component and functions we typically see are different from what I have used so let me change them. The function and the file name is usually the same it indicates the functionality that will be added to the components. So const **withCounter** and export the same. The OriginalComponent is usually referred to as **WrappedComponent** that is the component wrapped with the HOC, so let's change the parameter name and the return statement. the NewComponent is usually the same as the function name but in Pascal case the NewComponent will be **WithCounter** and we will return the same.

import React from 'react'

const withCounter = WrappedComponent=>

{

    class WithCounter *extends* React.Component

    {

        constructor(props)

        {

            super(props)

            this.state = {count:0}

        }

        incrementCount =()=>

        {

            this.setState(prevState =>

                {

                    return {count:prevState.count+1}

                })

        }

        render()

        {

            return <WrappedComponent *count*={this.state.count} *incrementCount*={this.incrementCount}/>

        }

    }

    return WithCounter

}

*export* default withCounter

In click counter and hover counter we are going to change UpdatedComponent to **withCounter** which is the function name. withCounter make sure to import it and repeat the same in hover counter. We can now read the export statement as “withCounter functionality export the HoverCounter component”. Now if there is no rule which tells we to name our components or HOC s this way but this is kind of the convention used in most of the videos and articles.

So, an HOC is a pattern where a function accepts a component and returns an enhanced component. The HOC pattern is used to share common functionality between components without having to repeat the code.

I want to discuss two things:

**The first one** is about passing down the props. In app component I'm going to pass a **name** prop on the <ClickCounter/> component name is equal to “vishwas”. Now in click counter I will render that name prop this dot props dot name.

import logo from './logo.svg';

import './App.css';

import HoverCounter from './components/HoverCounter'

import ClickCounter from './components/ClickCounter';

function App() {

  return (

    <div *className*="App">

      <ClickCounter *name*='vishwas'/>

      <HoverCounter/>

    </div>

  );

}

*export* default App;

import React, { Component } from 'react'

import withCounter from './withCounter'

class ClickCounter *extends* Component

{

    render()

    {

*//const {count}=this.state*

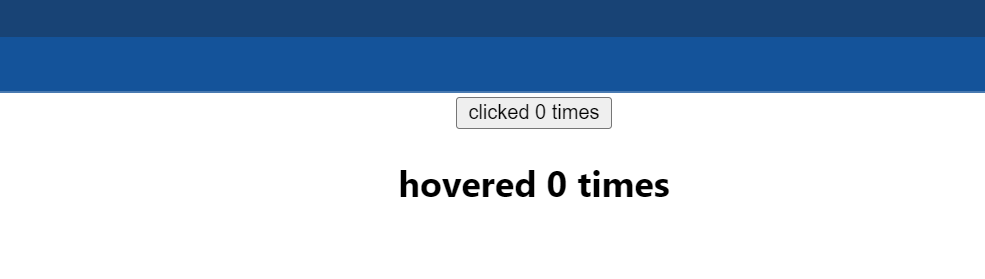
        const {count,incrementCount}=this.props

        return <button *onClick*={incrementCount}> {this.props.name} clicked {count} times </button>

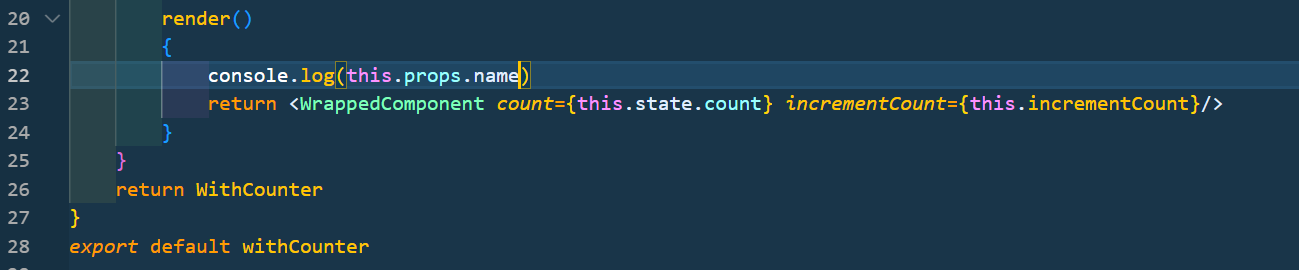
    }

}

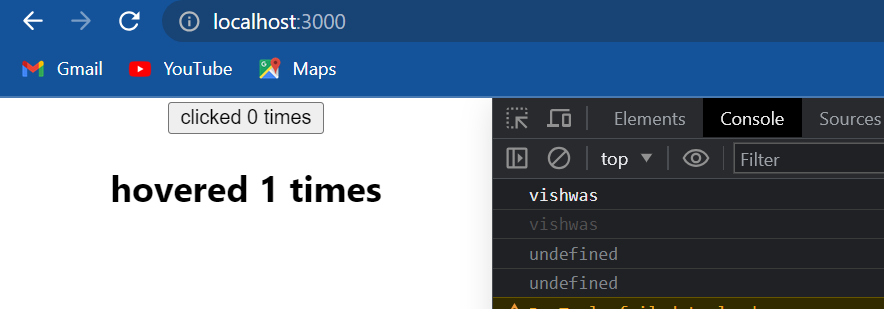
*export* default withCounter(ClickCounter)



if we go back to the browser we can see that the name “vishwas” is not displayed and this is a common mistake that happens when we start off with HOCs. The problem here is that when we specify props on the click counter component, the props are sent down to the HOC and not to click counter. In the HOC in the render method if I console dot log this dot props dot name that is **console.log(this.props.name)**.



take a look at the developer tools we can see that for click counter we do have the **name** prop set to “vishwas”, for the hover counter it is undefined since we are not passing in any props in app component.



So the prop is passed to the HOC but not to the component that is wrapped. To fix this issue we need to pass down the remaining props to the wrap component using the **spread operator**. so what is happening here is that the HOC adds two props to the WrappedComponent and then simply passes down whatever remaining props have been specified, in our case it passes down the name prop.

import React from 'react'

const withCounter=WrappedComponent=>

{

    class WithCounter *extends* React.Component

    {

        constructor(props)

        {

            super(props)

            this.state = {count:0}

        }

        incrementCount =()=>

        {

            this.setState(prevState =>

                {

                    return {count:prevState.count+1}

                })

        }

        render()

        {

            console.log(this.props.name)

            return <WrappedComponent *count*={this.state.count} *incrementCount*={this.incrementCount} {... this.props}/>

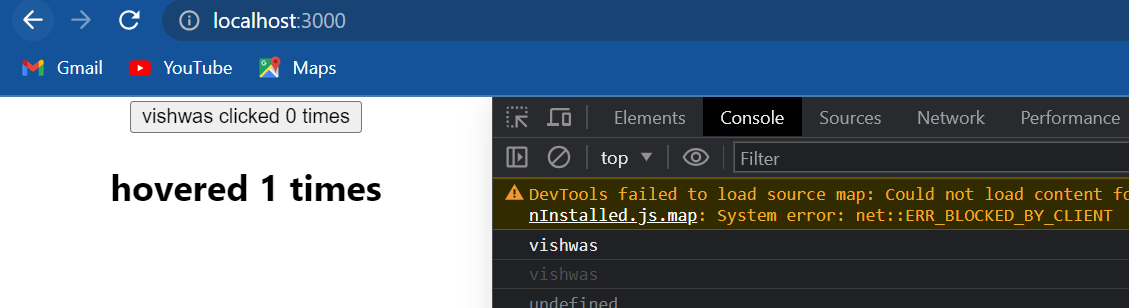
        }

    }

    return WithCounter

}

*export* default withCounter



if we now go back to the browser we should be able to see “vishwas” as part of the button text. so when we create HOCs please make sure to pass down the rest of the props.

**The second** thing to discuss is passing parameters to the HOC function. In our HOC let's say instead of incrementing the count value by one I want to increment it by different numbers for both the counter components. We can do that by passing a parameter to the HOC function. So the arrow function will now have two parameters: the first one is the **WrappedComponent** and the second one is the **incrementNumber.** Instead of incrementing the count by 1 we will now increment it by the incrementNumber parameter.

**withCounter.js**

import React from 'react'

const withCounter=(WrappedComponent,incrementNumber)=>

{

    class WithCounter *extends* React.Component

    {

        constructor(props)

        {

            super(props)

            this.state = {count:0}

        }

        incrementCount =()=>

        {

            this.setState(prevState =>

                {

                    return {count:prevState.count + incrementNumber}

                })

        }

        render()

        {

            console.log(this.props.name)

            return <WrappedComponent *count*={this.state.count} *incrementCount*={this.incrementCount} {... this.props}/>

        }

    }

    return WithCounter

}

*export* default withCounter

Back in click counter I will add a second argument which is 5 and then in hover component I will add the second argument as 10.

**ClickCounter.js**

import React, { Component } from 'react'

import withCounter from './withCounter'

class ClickCounter *extends* Component

{

    render()

    {

*//const {count}=this.state*

        const {count,incrementCount}=this.props

        return <button *onClick*={incrementCount}>{this.props.name} clicked {count} times</button>

    }

}

*export* default withCounter(ClickCounter,5)

**HoverCounter.js**

import React, { Component } from 'react'

import withCounter from './withCounter'

class HoverCounter *extends* Component

{

    render()

    {

*//const {count}=this.state*

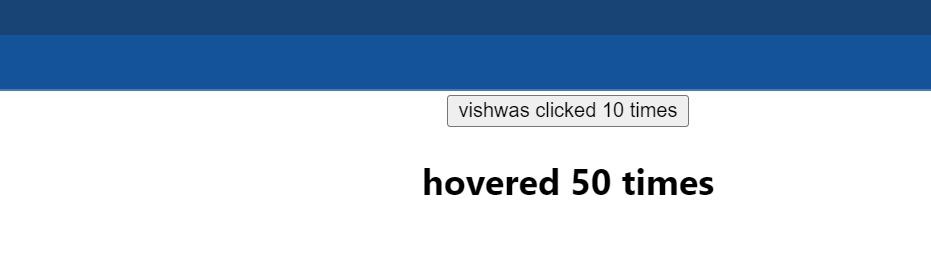
        const {count, incrementCount}=this.props

        return <h2 *onMouseOver*={incrementCount}>hovered {count} times</h2>

    }

}

*export* default withCounter(HoverCounter,10)



if we now go back to the browser I click on the button and we can see that increment is a multiple of 5, I hover and increment is in multiples of 10. so it's really easy to make some parameters into an HOC.

We might come across HOC s in some popular react libraries for example connect HOC in redux with router in react router with styles in material UI and so on. It's a nice little pattern that can be used to share common functionality between react components.

**Render props**

Now let's take a look at another pattern for sharing code between react components which is the render prop pattern. Let’s quickly revisit why there is a need for render props?

let's begin, I'm going to create a new file called *ClickCounterTwo.js* and within the file I'm going to create a class component and for the JSX I will add a button with the text “clicked x times”. Next I'm going to add a constructor and create a state property called **count** and initialize it to zero. I will also add a click handler to the button onClick is going to be equal to this dot incrementCount. Now we can define the handler increment count is going to be equal to an arrow function and within the body we call this dot set state to increment the count value by 1. the argument is going to be a function which accepts the previous state(**prevState**) as a parameter and returns count set to prevState.count plus one. Finally in the render method I will extract count from the state and include it in the JSX; const count is equal to this dot state and “clicked {count} times”. Now let me include this click counter(ClickCounterTwo) in app component.

import React, { Component } from 'react'

class ClickCounterTwo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount=()=>

    {

        this.setState(prevState =>{return {count:prevState.count+1}})

    }

    render()

    {

        const {count}=this.state

        return (

        <button *onClick*={this.incrementCount}>clicked {count} times</button>

        )

    }

}

*export* default ClickCounterTwo



if we now save the files and take a look at the browser we can see that when I click on the button the count value increments, so we have successfully created a click counter. Now we have a new requirement for a hover counter. so basically a click counter with click functionality replaced by hover functionality let's implement that. we'll go back to code and create a new file called *HoverCounterTwo.js* and within the file I will create a class component and for the JSX I will add a heading that says “hovered x times”. On the heading we will handle the **onMouseOver** event on onMouseOver is equal to this dot increment count. now we need the counter functionality which I will copy paste from ClickCounterTwo. Copy the constructor and **incrementCount()** method and paste it in HoverCounterTwo. Next in the render method the destructure count from this dot state and then display it as part of the heading. finally back in app component we can include the hover component <HoverCounterTwo/>.

import React, { Component } from 'react'

class HoverCounterTwo *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount=()=>

    {

        this.setState(prevState =>{return {count:prevState.count+1}})

    }

    render()

    {

        const {count}=this.state

        return (

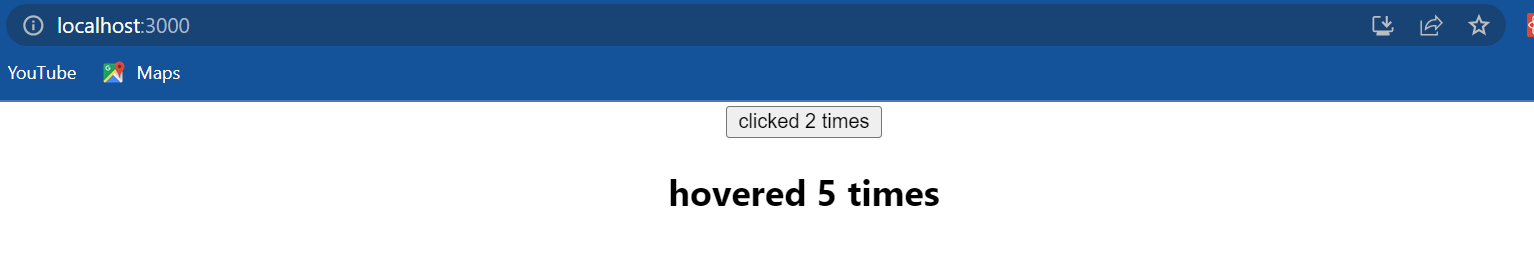
        <h2 *onMouseOver*={this.incrementCount}>hovered {count} times</h2>

        )

    }

}

*export* default HoverCounterTwo



if we now save the files and take a look at the browser, I hover on the heading and the count increases so we have successfully created the hover counter as well.

Now we have yet another requirement, the client wants an input element that counts the number of key presses. for example on key up in an input element they want to increment a counter value and display it. we can of course implement it just like the click counter or the hover counter but we're going to realize that we are duplicating code and not really reusing the functionality. so there is a need to share this common functionality between components. once it's pattern capable of achieving that is the higher-order component(HOC) pattern which we learnt previously. There is also another approach well-suited for sharing functionality between react components and that is the render props pattern. What is the render props pattern and how do we implement it let's take a look at that.

Let's understand how the render props pattern is implemented that will in turn help us to understand what is render props. I'm going to create a new file called *User.js* and within the file I'm going to create a class component and for the JSX I will add the text “vishwas”. Next I will include the component in app component.

**User.js**

import React, { Component } from 'react'

class User *extends* Component

{

    render()

    {

        return (

        <div>

            vishwas

        </div>

        )

    }

}

*export* default User

**App.js**

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

function App() {

  return (

    <div *className*="App">

      <ClickCounterTwo />

      <HoverCounterTwo />

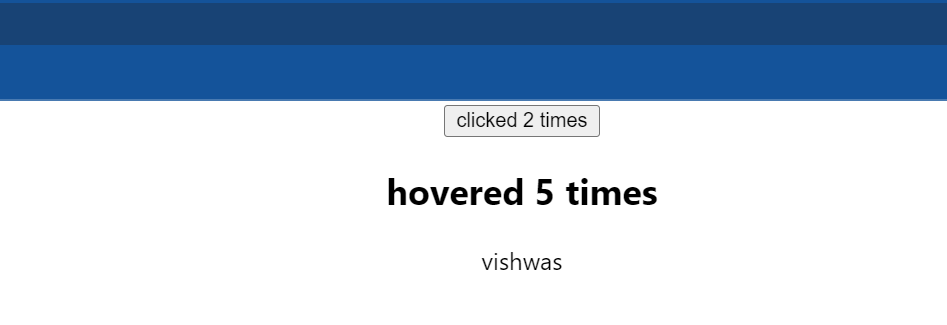
      <User />

    </div>

  );

}

*export* default App;



if we now save all the files and take a look at the browser we should be able to see the text “vishwas”. Now let's make changes to this user component, and I'll make sure we are taking baby steps:

The first step(**step1**) instead of hard-coding the name Vishwas let's pass it as a prop, so in app component name is equal to “vishwas” and in User component we will render this dot props dot name.

**App.js**

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

function App() {

  return (

    <div *className*="App">

      <ClickCounterTwo />

      <HoverCounterTwo />

      <User *name*="vishwas"/>

    </div>

  );

}

*export* default App;

import React, { Component } from 'react'

class User *extends* Component

{

    render()

    {

        return (

        <div>

            {this.props.name}

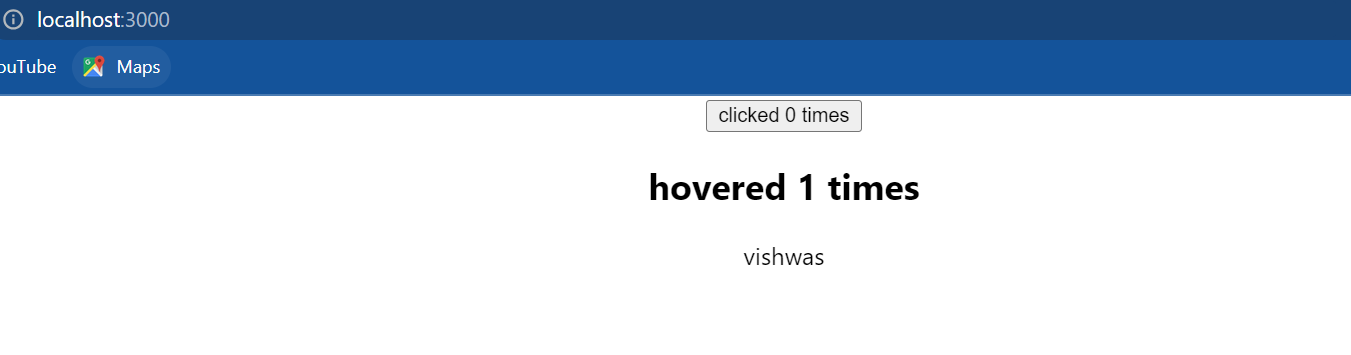
        </div>

        )

    }

}

*export* default User



save the files and take a look at the browser still works perfectly fine. For our next step(**step2**) I want to go a little crazy and instead of simply passing the string “vishwas” as name prop, I want to pass a function which will return the string “vishwas”, so the name prop is going to be equal to a function which will return the string “vishws”.

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

function App() {

  return (

    <div *className*="App">

      <ClickCounterTwo />

      <HoverCounterTwo />

      <User *name*={()=> "vishwas"}/>

    </div>

  );

}

*export* default App;

Now in our User component we will still have this dot props dot name but the only difference now is that it contains a reference to a function. To actually display the name “vishwas” we need to call the function so add parentheses.

import React, { Component } from 'react'

class User *extends* Component

{

    render()

    {

        return (

        <div>

            {this.props.name()}

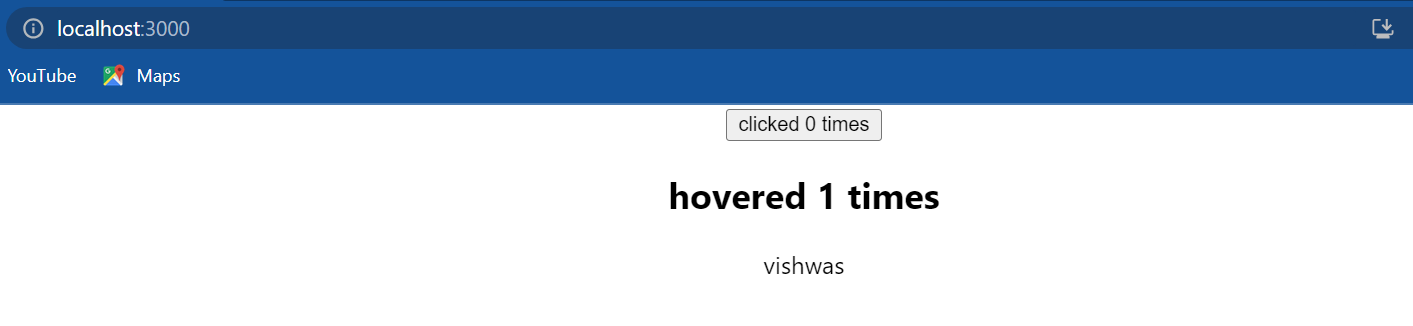
        </div>

        )

    }

}

*export* default User



if we now save the files and take a look at the browser everything still works perfectly fine. For our next step(**step3**) I want to have parameters to the function in thename prop based on the parameter I want to change what is rendered by the user component. So I will pass in **isLoggedIn** as a parameter and the function will now return the string “vishwas” or guest based on the value of isLoggedIn, so isLoggedIn and we are going to use the ternary operator if it is true return “Vishwas” if it is false return “guest”.

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

function App() {

  return (

    <div *className*="App">

      <ClickCounterTwo />

      <HoverCounterTwo />

      <User *name*={(isLoggedIn) => isLoggedIn ? "vishwas" : "guest"}/>

    </div>

  );

}

*export* default App;

Now in User component the name prop will accept an argument which I will pass as true this implies isLoggedIn is set to true and the string “Vishwas” should be rendered in the browser.

import React, { Component } from 'react'

class User *extends* Component

{

    render()

    {

        return (

        <div>

            {this.props.name(true)}

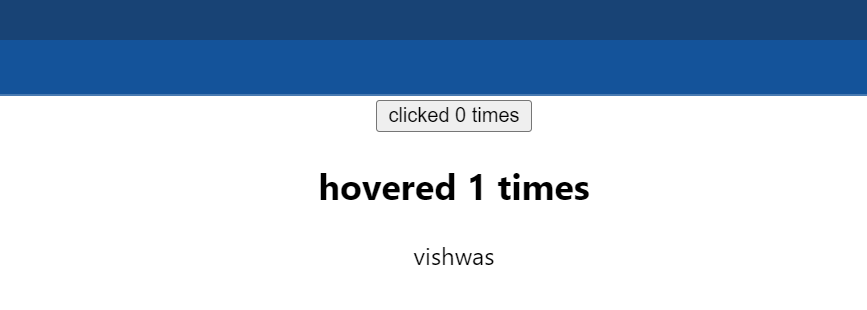
        </div>

        )

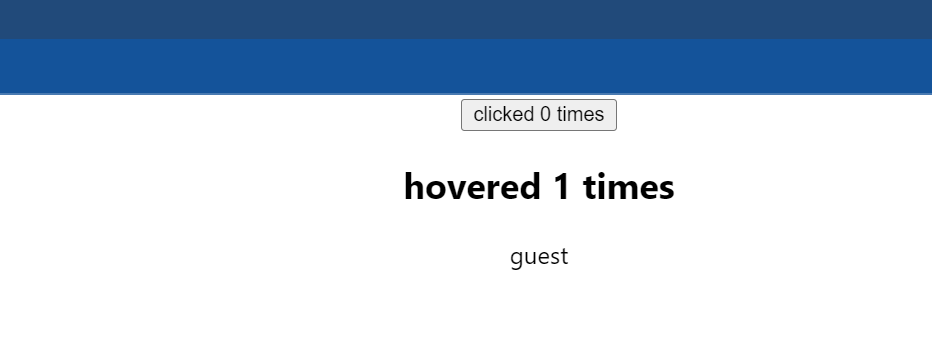
    }

}

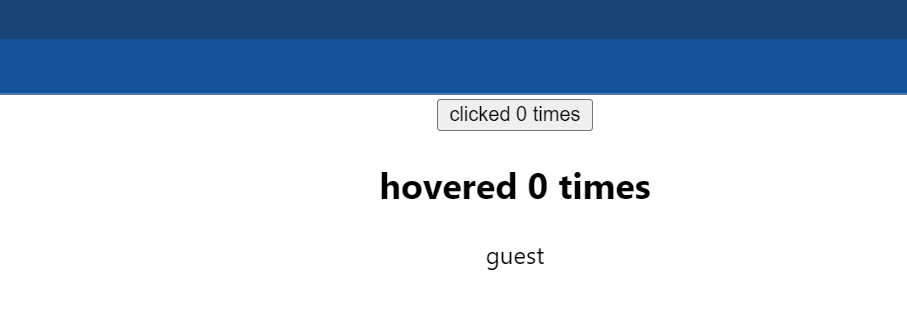
*export* default User



which is what we have here. I change it to false and the string “guest” is rendered in the browser:



Now for another baby step(**step4**), in app component I will simply rename the name prop to **render**  and in the User component I will change name to **render,** now this is perfectly valid and will not conflict with the render lifecycle method in any way. if we save the files and take a look at the browser our UI still works as expected:



From this example here is what I want we to keep in mind, **in react it is possible to use a prop whose value is a function to control what is actually rendered by a component** and why do I want us to remember this well it is pretty much what the render props pattern is based on. To answer the question what is render props; **the term render prop refers to a technique for sharing code between react components using a prop whose value is a function**. Two parts to this definition **sharing code** and **prop whose value is a function**. We have just learnt how to use a prop whose value is a function. let us now see how we can share functionality. To understand that we will go back to the problem at hand we have two counter components click counter(ClickCounterTwo) and hover counter(HoverCounterTwo) both the components contain the code that can be shared between them, that is the **count** state and the **incrementCount** method. let us now see how to share that code by using the render props pattern.

I will begin by creating a new file *Counterr.js* and within the file I'm going to create a class component and leave the JSX empty for now. This Counter component is going to be our container component where we implement the common functionality and that is the state along with the incrementCount method. so from *ClickCounterTwo.js*, I will pull out that code and paste it in *Counterr.js.* I will also remove the same piece of code from *HoverCounterTwo.js* as well.

**ClickCounterTwo.js**

import React, { Component } from 'react'

class ClickCounterTwo *extends* Component

{

    render()

    {

        const {count}=this.state

        return (

        <button *onClick*={this.incrementCount}>clicked {count} times</button>

        )

    }

}

*export* default ClickCounterTwo

**HoverCounterTwo.js**

import React, { Component } from 'react'

class HoverCounterTwo *extends* Component

{

    render()

    {

        const {count}=this.state

        return (

        <h2 *onMouseOver*={this.incrementCount}>hovered {count} times</h2>

        )

    }

}

*export* default HoverCounterTwo

So the common functionality has been removed from both the components and moved to the Counterr component. The JSX is empty because the rendered prop is going to control what will be rendered by this Counterr component, for our example we will render ClickCounterTwo and HoverCounterTwo and pass down the count state and incrementCount method. so in the JSX in the div tag this dot props dot render; we are going to pass two arguments this dot state dot count and the method this dot incrementCount. The Counterr component is basically telling hey take the count state and the increment count method and render whatever we want to I will handle everything regarding the counter logic, we just worry about what to render.

**Counterr.js**

import React, { Component } from 'react'

class Counterr *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount=()=>

    {

        this.setState(prevState =>{return {count:prevState.count+1}})

    }

    render()

    {

        return (

        <div>

            {this.props.render(this.state.count,this.incrementCount)}

        </div>

        )

    }

}

*export* default Counterr

So back an app component we can now comment out these existing components and include the <Counterr/> component. we are going to add the render prop whose value is a function, so render is a prop its value is a function. the function receives the count and incrementCount method and will return ClickCounterTwo component(<ClickCounterTwo/>) passing the same as props. so one prop is going to be count is equal to count the other prop incrementCount is equal to incrementCount. I will format this and we are going to repeat the same for HoverCounterTwo as well. I'm going to copy this paste it and change (<ClickCounterTwo/> (<HoverCounterTwo/>.

**App.js**

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

import Counterr from './components/Counterr'

function App() {

  return (

    <div *className*="App">

      <Counterr *render*={(count,incrementCount)=>(<ClickCounterTwo *count*={count} *incrementCount*={incrementCount}/>)}/>

      <Counterr *render*={(count,incrementCount)=>(<HoverCounterTwo *count*={count} *incrementCount*={incrementCount}/>)}/>

{*/\* <ClickCounterTwo />*

*<HoverCounterTwo />*

*<User render={(isLoggedIn) => isLoggedIn ? "vishwas" : "guest"}/> \*/*}

    </div>

  );

}

*export* default App;

Next let's make the changes in the actual components and ClickCounterTwo we are now going to destructure count and increment count from this dot props and the onClick handler I'm going to remove the this keyword. let's see the same in HoverCounterTwo as well, count comma incrementCount from this dot props and remove the this keyword.

**ClickCounterTwo.js**

import React, { Component } from 'react'

class ClickCounterTwo *extends* Component

{

    render()

    {

        const {count,incrementCount}=this.props

        return (

        <button *onClick*={incrementCount}>clicked {count} times</button>

        )

    }

}

*export* default ClickCounterTwo

**HoverCounterTwo.js**

import React, { Component } from 'react'

class HoverCounterTwo *extends* Component

{

    render()

    {

        const {count, incrementCount}=this.props

        return (

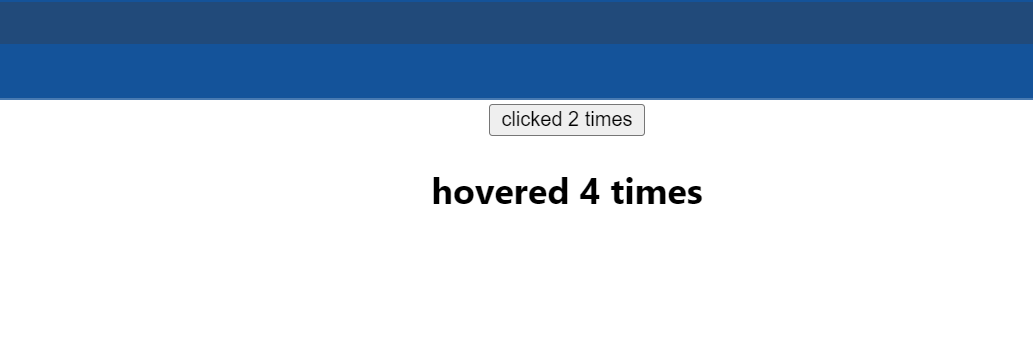
        <h2 *onMouseOver*={incrementCount}>hovered {count} times</h2>

        )

    }

}

*export* default HoverCounterTwo



If we now save all the files and take a look at the browser we should have the click counter and the hover counter working as expected but this time by sharing the functionality implemented using the render props pattern.

Let me go over the pattern one more time so as to get a better understanding of how it really works. In App component we come across the Counterr component, in the Counter component we have a count state and incrementCount method. the Counter component however does not render anything on its own, it is simply going to render whatever is passed as the render prop and while doing so it passes on the count state and the increment count method. Now what is a render prop, it is the ClickCounterTwo component. The count State and increment count method from the Counter component are passed as props to the ClickCounterTwo component. The ClickCounterTwo makes use of the passed in props to render the actual UI. When we click on the button and call the incrementCount method or try to display the count value it is pretty much what the Counter component has provided. It is also what happens with the HoverCounterTwo component. Now even though they share common code the Counter component instance will be different and hence there is no conflict between the count state values. What is also important here is to note that the prop need not be called as render. It could be called anything we wish to but render is kind of the convention. In fact there is a variation of the render props pattern which doesn't even make use of the prop, instead the children prop is used. We have to make two simple changes: instead of render prop we pass in the function in between the component opening and closing tags, and the same for HoverCounter as well. Next in the Counterr component we change this props dot render to this dot props dot children.

**App.js**

import logo from './logo.svg';

import './App.css';

import ClickCounterTwo from './components/ClickCounterTwo';

import HoverCounterTwo from './components/HoverCounterTwo';

import User from './components/User';

import Counterr from './components/Counterr'

function App() {

  return (

    <div *className*="App">

      <Counterr>

        {(count,incrementCount)=>(<ClickCounterTwo *count*={count} *incrementCount*={incrementCount}/>)}

      </Counterr>

      <Counterr>

        {(count,incrementCount)=>(<HoverCounterTwo *count*={count} *incrementCount*={incrementCount}/>)}

      </Counterr>

      {*/\* <ClickCounterTwo />*

*<HoverCounterTwo />*

*<User render={(isLoggedIn) => isLoggedIn ? "vishwas" : "guest"}/> \*/*}

    </div>

  );

}

*export* default App;

**Counterr.js**

import React, { Component } from 'react'

class Counterr *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {count:0}

    }

    incrementCount=()=>

    {

        this.setState(prevState =>{return {count:prevState.count+1}})

    }

    render()

    {

        return (

        <div>

            {this.props.children(this.state.count,this.incrementCount)}

        </div>

        )

    }

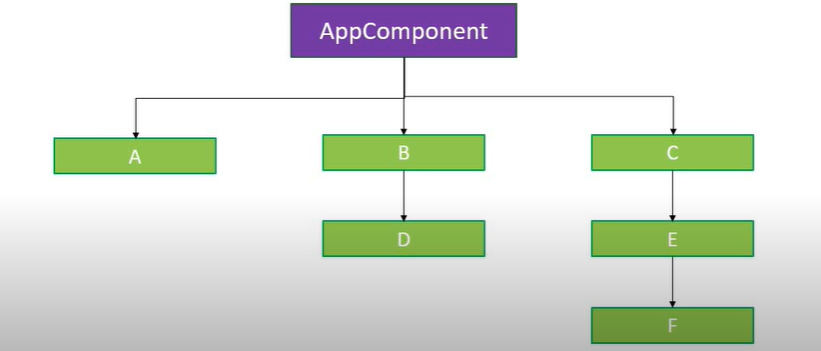
}

*export* default Counterr

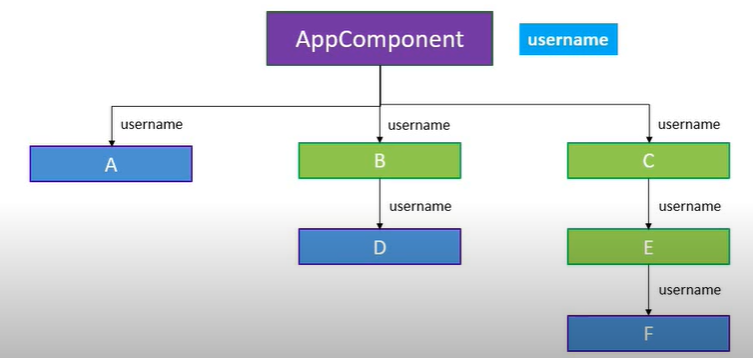
Remember anything within the components opening and closing tags will be passed as the children prop which is then accessed to render the UI. So we are still using a prop this value is a function to render UI and share functionality.

**Context**

Let’s understand what was the need for the context API. Consider a react application that has lots of components, we have app component which is the root component and nested within the app component at different levels are several other components.

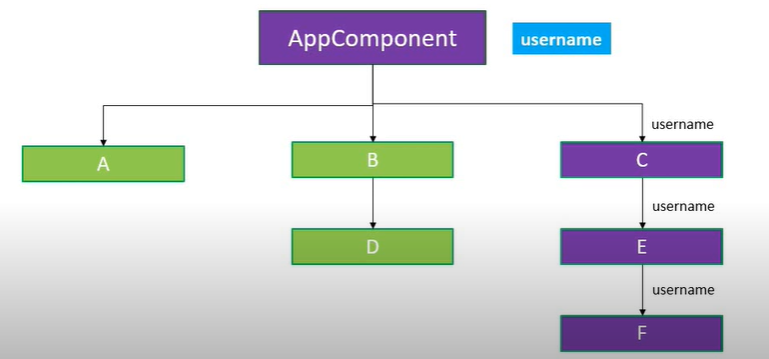


At the first level we have components A, B, and C. Nested within component B is component D and nested within component C is component E and as it turns out component F is nested inside component E. So we have three levels in total. The requirement in our application is that the components A, D and F are supposed to display the logged in user name that information is maintained as a property in the app component. so to be able to display the username in the nested components we need to pass down the username as a prop.



For component A it is pretty straightforward, directly pass it as props. For component D however we have the intermediate component B ,so we have to pass down the username as a prop to component B and that in turn has to pass down the prop to component D. Now the scenario is somewhat similar for component F as well, the prop has to be passed through component C and then component E and then finally to component F. Now even though components B, C, and E do not need the prop we still have to send the prop through them to be able to pass it to come opponents further down in the tree. Imagine if the component were to be nested five or ten levels deep all the components in between would have to forward the prop. this especially becomes a problem for certain types of props such as language preference, UI team, and authenticated user which are pretty much required by many components in our application. What would be nice is if we could directly send data to the required component without having to manually drill down the props through every level of the component tree. This is where context comes into picture. **Context provides a way to pass data through the component tree without having to pass props down manually at every level**. So we now know why context is needed.

We will implement a user context which will enable a component at any level to read a prop that is passed at the top level. For the demo I will only consider the rightmost part of our component tree that is components C, E, and F.



Our goal is to pass the user name value from app component and read that value in component F using context. To get us started I have created the components and nested them at the appropriate level. In app component we have componentC within componentC we have componentE and finally within componentE we have componentF and componentF simply renders the text “component F”.

**App.js**

import logo from './logo.svg';

import './App.css';

import React, {Component} from 'react'

import ComponentC from './components/ComponentC'

class  App *extends* Component {

  render()

  {

    return (

      <div *className*="App">

        <ComponentC/>

      </div>

    )

  }

}

*export* default App;

**ComponentC.js**

import React, { Component } from 'react'

import ComponentE from './ComponentE'

class ComponentC *extends* Component

{

    render()

    {

        return <ComponentE/>

    }

}

*export* default ComponentC

**ComponentE.js**

import React, { Component } from 'react'

import ComponentF from './ComponentF'

class ComponentE *extends* Component

{

    render()

    {

        return <ComponentF/>

    }

}

*export* default ComponentE

**ComponentF.js**

import React, { Component } from 'react'

class ComponentF *extends* Component

{

    render()

    {

        return (

            <div>

                Component F

            </div>

        )

    }

}

*export* default ComponentF

Now let us understand how to get data from the app component to componentF using context. There are three steps to implement when making use of context:

1. Create the context
2. Provide a context value
3. Consume the context value in the necessary components

let's begin with step 1 creating the context. I'm going to go back to code and create a new file *userContext.js* and within the file we will use the createContext method from react to create a user context. **import React from ‘react’** and then const **UserContext** is equal to React dot createContext(). now it so happens that every context object created using the createContext method comes with a provider and a consumer react component. we need those components for steps two and three which means we have to export them. while exporting though I like to assign them to a more readable name, so const UserProvider is equal to UserContext dot Provider and const UserConsumer is equal to UserContext dot Consumer. now we can export them. All right so that is our step 1 creating the user context.

**userContext.js**

import React from 'react'

const UserContext=React.createContext()

const UserProvider=UserContext.Provider

const UserConsumer=UserContext.Consumer

*export* {UserProvider, UserConsumer}

For step 2 we need to provide this userContext using the UserProvider component and the place we provide is important because only the descendant components can consume it. app component is usually a good place because pretty much all components fall under it. so in app component I will wrap componentC with UserProvider. Also make sure to include the import statement at the top. Now the provider component is responsible for providing a value for all the descendant components. the value we want to provide is the user name and the way we provided is using the value attribute on the provider component, so value is equal to “vishwas”. That completes step two, we are now providing a value in the app component that can be consumed by componentC and any component nested inside componentC.

**App.js**

import logo from './logo.svg';

import './App.css';

import React, {Component} from 'react'

import ComponentC from './components/ComponentC'

import { UserProvider } from './components/userContext'

class  App *extends* Component {

  render()

  {

    return (

      <div *className*="App">

        <UserProvider *value*="vishwas">

          <ComponentC/>

        </UserProvider>

      </div>

    )

  }

}

*export* default App;

The final step is to consume the username in the desired component. for our demo we need to consume the username value in componentF. To consume a context value we need to use the consumer component. so in componentF in the render method as part of the return statement include the UserConsumer component and import it at the top. In between the opening and closing tags of the consumer component we need to specify a function. so start with curly braces and include an arrow function. the function gets the user context value which is “vishwas” as its parameter which can then be used within the function body to return a react element. in our example the parameter is username which is what the context provides and within the body we will return hello {username}. so what is happening here is that we are passing a function as a child to the consumer component. the consumer component tells componentF hey we need the username right let me provide it to we what we want to do with the user name is up to we just make sure we return proper JSX. componentF will consume the username and simply render it as part of the JSX.

**ComponentF.js**

import React, { Component } from 'react'

import { UserConsumer } from './userContext'

class ComponentF *extends* Component

{

    render()

    {

        return (

            <UserConsumer>

                {

                    (username)=>

                    {

                        return <div>Hello {username}</div>

                    }

                }

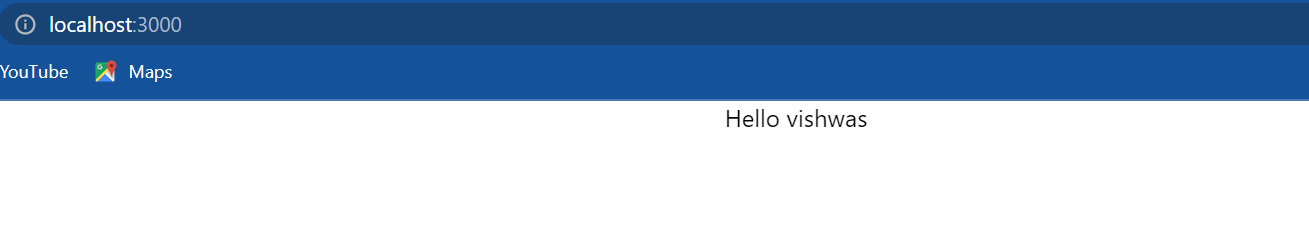
            </UserConsumer>

        )

    }

}

*export* default ComponentF



if we now save all the files and head to the browser we should be able to see the text “hello vishwas” displayed from component. we have successfully used context to provide a value to a deeply nested component without having to pass that value as a prop through every intermediate component.

let me go over the steps one more time:

* Step 1 create the user context using the create context method from react. Make sure to export the provider and consumer components as well.
* Step 2 at the top level include the provider component and provide a value using the value attribute. This value can now be consumed in any of the descendant components.
* Step 3 in the component where the user name is required use the consumer component and pass in a function as its child. the function receives the context value as its parameter which can then be used to return the desired JSX.

We can choose to just display it or even use it for some rendering logic but this is pretty much how we use react context.

Let us discuss two more points about the context API. The first point is that **we can set a default value to our context and the default value is set while creating the context.** it is passed as an argument to the create context method. Now in *userContext.js* I will add the string “codevolution” as the default value.

import React from 'react'

const UserContext=React.createContext("codevolution")

const UserProvider=UserContext.Provider

const UserConsumer=UserContext.Consumer

*export* {UserProvider, UserConsumer}

Back in app component I will comment out the provider component:

**App.js**

import logo from './logo.svg';

import './App.css';

import React, {Component} from 'react'

import ComponentC from './components/ComponentC'

import { UserProvider } from './components/userContext'

class  App *extends* Component {

  render()

  {

    return (

      <div *className*="App">

        {*/\* <UserProvider value="vishwas"> \*/*}

          <ComponentC/>

        {*/\* </UserProvider> \*/*}

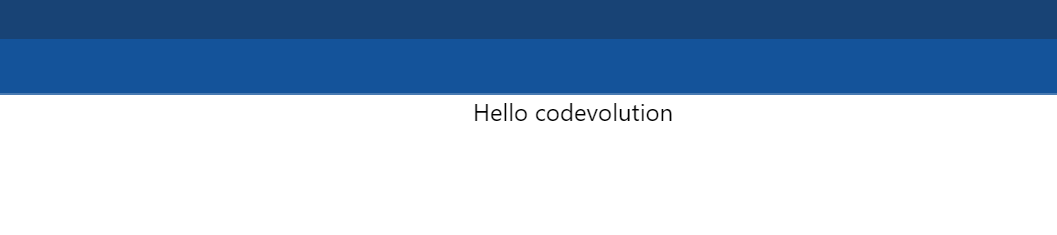
      </div>

    )

  }

}

*export* default App;



if we save all the files and take a look at the browser we should be able to see the text “hello codevolution”. if I add the component back and take a look at the browser we can see that “hello vishwas” is being displayed.

**App.js**

import logo from './logo.svg';

import './App.css';

import React, {Component} from 'react'

import ComponentC from './components/ComponentC'

import { UserProvider } from './components/userContext'

class  App *extends* Component {

  render()

  {

    return (

      <div *className*="App">

        <UserProvider *value*="vishwas">

          <ComponentC/>

        </UserProvider>

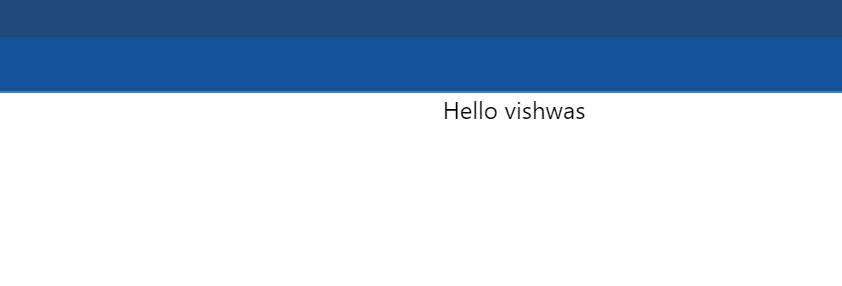
      </div>

    )

  }

}

*export* default App;



So the default value will only be used when a component does not have a matching provider above it in the component tree.

The second and final point to discuss about is the context type property. We have learnt previously how to use the consumer component to consume the context value, turns out there is another way to do that and that is using the context type property on a class.

let's see how that works by consuming the user context value in ComponentE.

The **first step** in *userContext.js* we need to export the user context itself so export default UserContext.

**userContext.js**

import React from 'react'

const UserContext=React.createContext("codevolution")

const UserProvider=UserContext.Provider

const UserConsumer=UserContext.Consumer

*export* {UserProvider, UserConsumer}

*export* default UserContext

**Second step** assign this user context to the context type property on the class. so in *ComponentE.js,* outside the code for class ComponentE we are going to have ComponentE dot contextType is equal to UserContext and make sure to import it at the top. now in the render method the UserContext value is available as this dot context. so in the render method in addition to rendering ComponentF I will also include “component E context” which is going to be this dot context and then include ComponentF.

**ComponentE.js**

import React, { Component } from 'react'

import ComponentF from './ComponentF'

import UserContext from './userContext'

class ComponentE *extends* Component

{

    render()

    {

        return (

            <div>

                Component E context {this.context}

                <ComponentF/>

            </div>

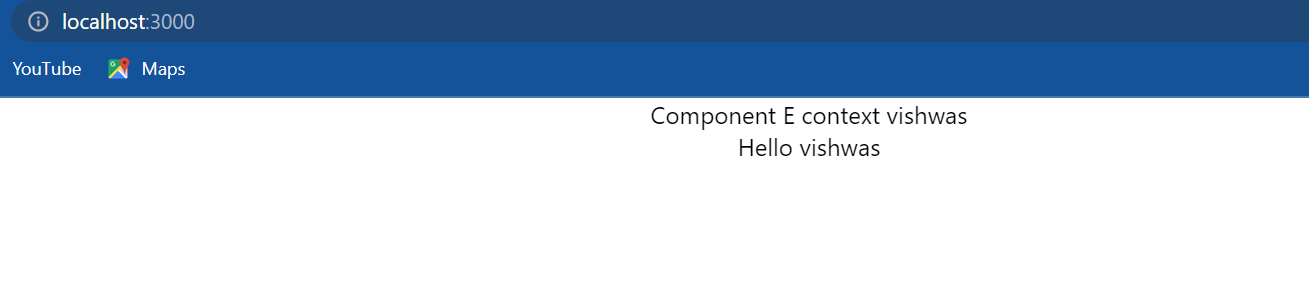
        )

    }

}

ComponentE.contextType=UserContext

*export* default ComponentE



if we now save the files and take a look at the browser we should see the expected output “component E context Vishwas” which is the value and then “hello vishwas” from ComponentF. so ComponentE also is now able to render the username. And if out application supports the public class view syntax we can replace ComponentE dot context type with static contextType is equal to UserContext and the application still works the same:

**ComponentE.js**

import React, { Component } from 'react'

import ComponentF from './ComponentF'

import UserContext from './userContext'

class ComponentE *extends* Component

{

*static* contextType=UserContext

    render()

    {

        return (

            <div>

                Component E context {this.context}

                <ComponentF/>

            </div>

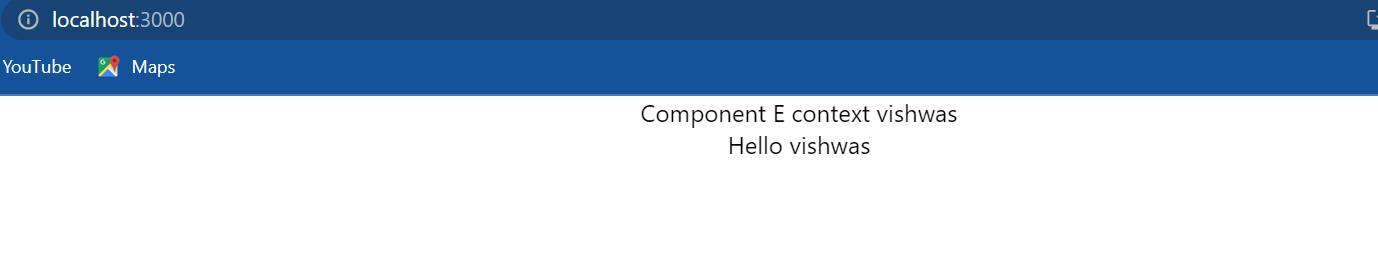
        )

    }

}

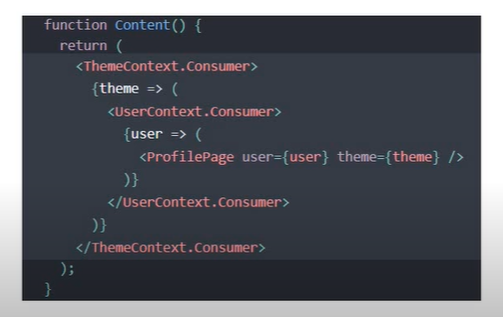
*//ComponentE.contextType=UserContext*

*export* default ComponentE



Now we might think this looks much simpler compared to the consumer components in tags why should we not just stick to context type.

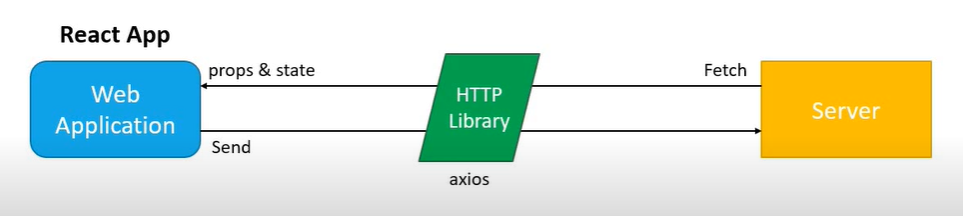
Well there are two limitations: the **first one is that it only works with class components**. The **second limitation is that we can only subscribe to a single context using context type**. Many a times in our application we need to read more than one context in which scenario the consumer component is the way to go. And the code to consume multiple contexts looks somewhat like this:



we can see that we have two contexts ThemeContext and UserContext. ThemeContext consumer accepts a function as a child passing in the theme value. Within the function body we have another function as a child which provides the UserContext value. both of them are then passed as props to a component. alright those are the two additional points I wanted to discuss about react context default values and context type.

**HTTP and React**

When we're building web applications we more often than not have to reach out to the server to fetch some data or send some data based on user interaction and this web application in our context is of course a react application. But if we can recollect from starting, I mentioned that react is a library for building user interfaces. It is in no way concerned about HTTP. This raises a very important question how do we make Ajax requests in react or how do we make API calls in react. To answer that question first let me tell we that react itself does not have a particular way to fetch or send data to the server, infact as far as react is concerned it doesn't even have to know that there is a server in the picture.



Remember react components simply read props and state and render the UI. Therefore to use some data from the server we just have to get the data into our components props or state. Now if react isn't going to handle the requests who will be doing that this is where we will have to make use of an HTTP library. There are a few popular ones out there but I prefer Axios. The fetch API is also a good consideration as it is more standardized but for our understanding I will stick to Axios. Our focus is not really on the library we use rather how we use it with react.

All right with that introduction let's get started by adding the Axios package to our application. I'm going to go back to vs code and as we can see I have created a new project using the create react app the command is **npx create-react-app react-HTTP** which is the name of our project. Once we run command we should have a project. Next step is to install the Axios package, so make sure we are inside the project folder and then run the command **npm install axios**. When the command completes Axios should be added to the list of dependencies so we can now start making HTTP requests from our application.

**HTTP GET request**

We will learn how to make a get request using Axios and render the fixed data in a react component. To fetch the data we need to have an API endpoint for that I will be making use of JSON placeholder. As we can see [here](https://jsonplaceholder.typicode.com/) , it is basically a fake online REST API for testing and prototyping. I want to focus on fetching data in our react application and not on creating an API. if we scroll down to the routes section we can see the possible HTTP requests we can make. For this topic I will be making a get request to slash posts which will fetch an array of posts to display in the UI. let's get started I'm going to go back to vs code and create a new component, so within the source folder I'm going to add a new folder called **components** and within the folder a new file *PostList.js* and within the file I'm going to create a class component. Now as of this, react version 16.8 is out which does give us access to hooks. however I want to first show we how to work with HTTP and class components and talk about hooks separately. So back to our PostList component and for the JSX simply add the text “list of posts”. Back in app component I will include PostList component.

**App.js**

import logo from './logo.svg';

import './App.css';

import React, { Component } from 'react'

import PostList from './components/PostList'

*export* class App *extends* Component

{

  render() {

    return (

      <div *className*="App">

        <PostList/>

      </div>

    )

  }

}

*export* default App

**PostList.js**

import React, { Component } from 'react'

class PostList *extends* Component

{

    render()

    {

        return (

        <div>

            list of posts

        </div>

        )

    }

}

*export* default PostList

if we now save all the files and take a look at the browser we should be able to see the text “list of posts”:



Now for data fetching, **first step** we import the Axios library in PostList component i.e. **import axios from axios**. **Second step** we need to create a state property which will store the list of posts. I'm going to add a constructor and then a state property called **posts** initialized to an empty array. **Third step** we are going to use Axios to make a get request to the JSON placeholder API. the question though is where do we place the code to make the get request. the answer is in the componentDidMount() lifecycle method, this method will be executed when the component mounts for the first time and this only executed once during components lifetime; a perfect place for our get request. Now to make a get request we invoke the get method on the Axios library. this method accepts the API endpoint as its argument. Go back to the browser ([here](https://jsonplaceholder.typicode.com/)) and click on the [/posts](https://jsonplaceholder.typicode.com/posts) route, copy the URL from the address bar and paste it as a string parameter to the get() method. alright we are now making a get request but how do we access the data that is returned. Well Axios is a promise based library so we can add **then** and **catch blocks**. **then** accepts an arrow function as its argument which gives us access to the response and I'm going to simply log to the console **response**. also if at all something went wrong we simply lock that error in the console, so again an arrow function which receives the error and log error.

**PostList.js**

import React, { Component } from 'react'

import axios from 'axios'

class PostList *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {posts:[]}

    }

    componentDidMount()

    {

        axios.get("https://jsonplaceholder.typicode.com/posts")

        .then(response =>

        {

            console.log(response)

        })

        .catch(error =>

            {

                console.log(error)

            })

    }

    render()

    {

        return (

        <div>

            list of posts

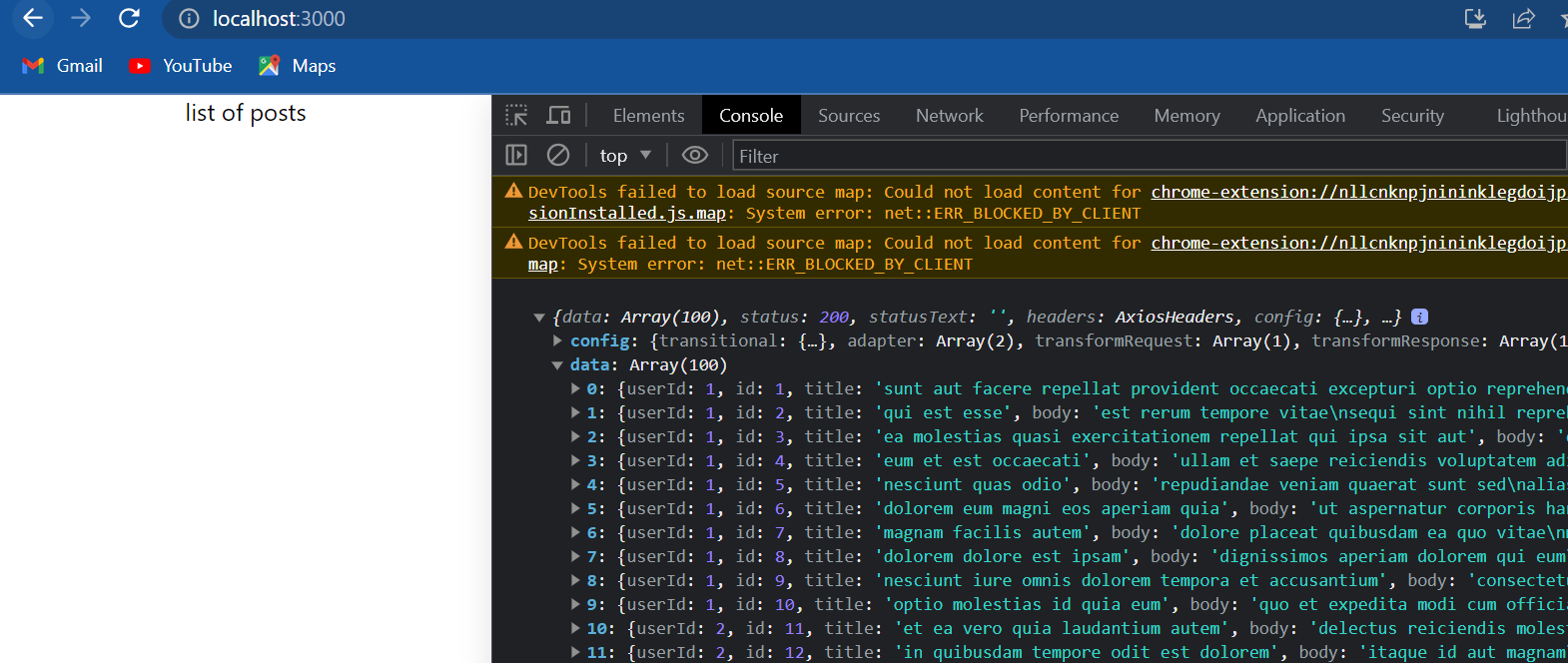
        </div>

        )

    }

}

*export* default PostList



if we now go back to the browser and open dev tools we should see an object logged in the console and if I expand the data property we can see the list of 100 posts that have been retrieved. All that is left now is to assign this data array to the state probability and then render it in the JSX. So back in vs code, when we get back the **response** we are going to set the response data to the posts array so this dot setState() posts is going to be response dot data i.e. {posts:response.data} and in the render method first destructure the state property then display this list of posts using the map method, I'm going to have curly braces if at all there is at least one post so **posts.length** then we are going to use the map method and display each post title so post key is going to be post dot ID and we are going to render just post dot title. if post is an empty array we are not going to render anything.

import React, { Component } from 'react'

import axios from 'axios'

class PostList *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {posts:[]}

    }

    componentDidMount()

    {

        axios.get("https://jsonplaceholder.typicode.com/posts")

        .then(response =>

        {

            console.log(response)

            this.setState({posts:response.data})

        })

        .catch(error =>

            {

                console.log(error)

            })

    }

    render()

    {

        const {posts}=this.state

        return (

        <div>

            list of posts

            {

                posts.length ?

                posts.map(post => <div *key*={post.id}> {post.title} </div>):

                null

            }

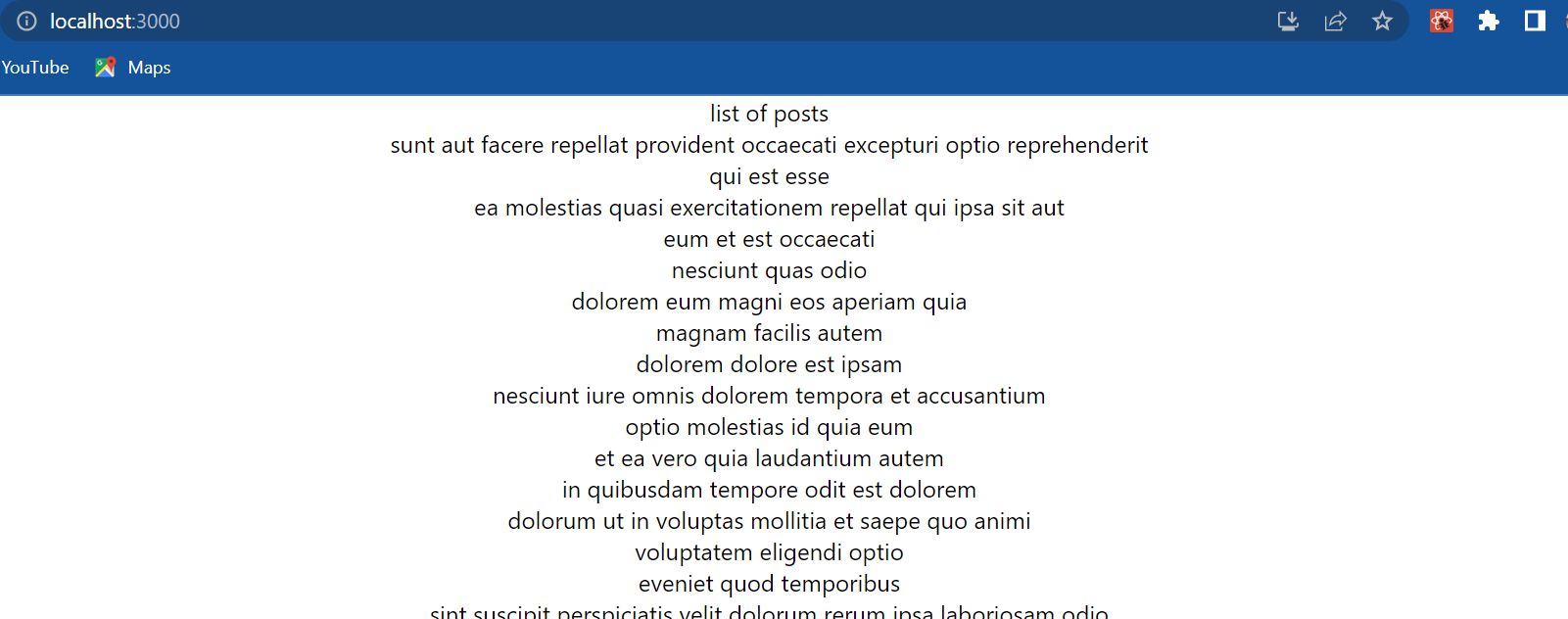
        </div>

        )

    }

}

*export* default PostList



save the file go back to the browser and we should be able to see the list of items fetched from the API being displayed in the browser. Now let me explain how this complete process happens as it is really important. We start off with the constructor, we have a state property called hosts which is an empty array then the control flows to the render method the text “list of posts” is displayed but the actual list is not rendered because the array is an empty array at the moment. then the control flows to componentDidMount over here we make our get request to the API endpoint. once the data is retrieved we then update the State posts property. when we change state the component will re-render and this time the array is not empty and hence the list of titles are rendered in the browser.

Now let me also quickly show we how to display an error message when the API fails. I will add another state property called error message initialized to an empty string in the catch block I will set it to the string “error retrieving data” this dot said state “error retrieving data”, in the JSX if there is an error message I will display it. so the structure and conditionally render it. finally I will alter the URL to an invalid URL.

import React, { Component } from 'react'

import axios from 'axios'

class PostList *extends* Component

{

    constructor(props)

    {

Wrong URL

        super(props)

        this.state = {posts:[], errorMsg:''}

    }

    componentDidMount()

    {

        axios.get("https://jsonplaceholder.typicode.com/postswef")

        .then(response =>

        {

            console.log(response)

            this.setState({posts:response.data})

        })

        .catch(error =>

            {

                console.log(error)

                this.setState({errorMsg: 'error retrieving data'})

            })

    }

    render()

    {

        const {posts, errorMsg}=this.state

        return (

        <div>

            list of posts

            {

                posts.length ?

                posts.map(post => <div *key*={post.id}> {post.title} </div>):

                null

            }

            {errorMsg ? <div> {errorMsg} </div>:null}

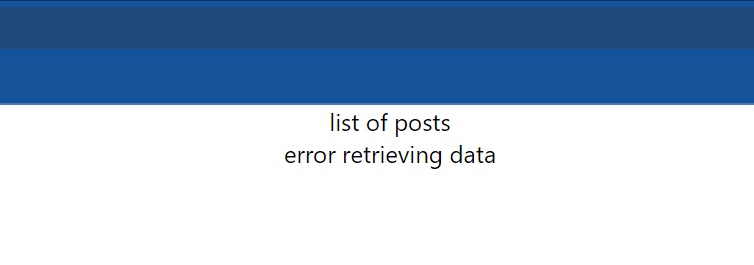
        </div>

        )

    }

}

*export* default PostList



if we now head back to the browser we should see the message “error retrieving data”, so that is how we make get requests with Axios and react.

**HTTP Post request**

let's see how to post data to an API from our react application. Now if we take a look [here](https://jsonplaceholder.typicode.com/posts) at the object properties for an individual post we have userId, title, and body the ID itself is self incrementing and doesn't have to be sent as part of the post request. so what we are going to do is create three input fields one each for userId, title, and body. we will store the data in the state object and make a post request on click of the submit button. let's get started we need to go back to vs code and create a new file called *PostForm.js* and within the file I'm going to create a class component and for the JSX I am going to create a form with three input elements one for userId, one for title, and one for body. Next let's create state properties for these three fields and link them back to the input elements. So in constructor userId, title, and body. Next let's be structure them in the render method and add their values to the value attribute of the input elements. destructure and assign to the value attribute. Next we add the onChange handler to track the change in input values and keep them in sync with the state object. so onChange is equal to this dot changeHandler. I am going to copy paste the same for the other two inputs as well and this changeHandler() is going to be an arrow function which is going to accept the event **e** as its argument and within the body we are going to call setState where the key is going to be **e** dot target dot name and the value is going to be **e** dot target dot value. By making use of the **name** attribute (in <input> tag) we don't have to have separate handlers for each input. The last thing we need is the submit handler so at the very end I'm going to add a button of type submit and then on the form tag we listen to the **onSubmit** event and assign a handler this dot submitHandler. Now let's define the handler so submitHandler is going to be an error function and within the body for now we will preventDefault to avoid page refresh and simply log to the console the state object. Back in app component I will include PostForm component i.e. <PostForm/>.

**App.js**

import logo from './logo.svg';

import './App.css';

import React, { Component } from 'react'

import PostForm from './components/PostForm'

*export* class App *extends* Component

{

  render() {

    return (

      <div *className*="App">

        <PostForm/>

      </div>

    )

  }

}

*export* default App

import React, { Component } from 'react'

class PostForm *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {userId:'', title:'', body:''}

    }

    changeHandler=(e)=>

    {

        this.setState({[e.target.name]:e.target.value})

    }

    submitHandler=(e)=>

    {

        e.preventDefault()

        console.log(this.state)

    }

    render()

    {

        const {userId, title, body}=this.state

        return (

        <div>

            <form *onSubmit*={this.submitHandler}>

                <div>

                    <input *type*="text" *name*="userId" *value*={userId} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <input *type*="text" *name*="title" *value*={title} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <input *type*="text" *name*="body" *value*={body} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <button *type*="submit">Submit</button>

                </div>

            </form>

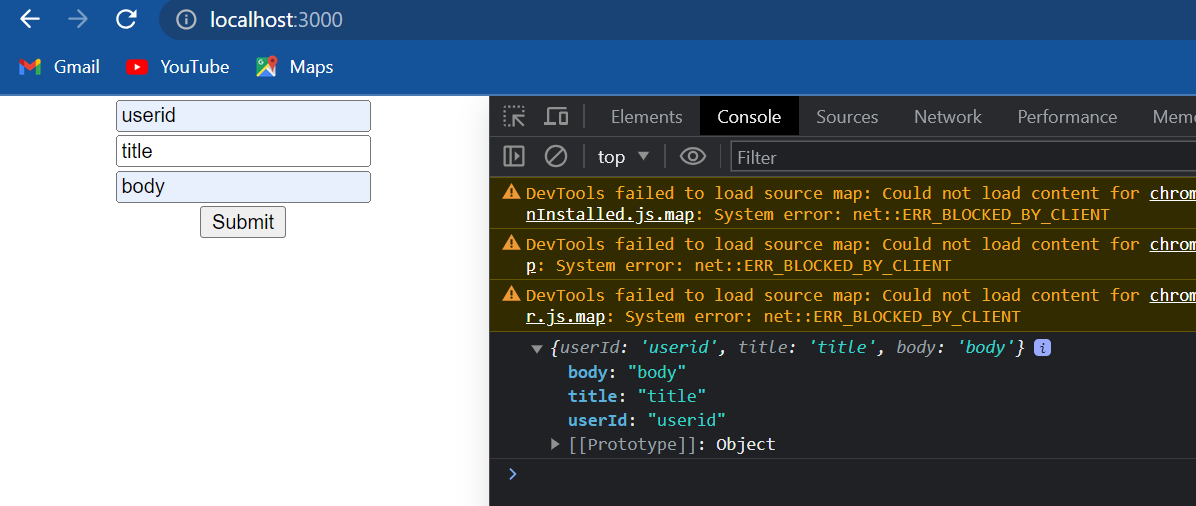
        </div>

        )

    }

}

*export* default PostForm



if we now go back to the browser fill in the fields and click on submit, we should have the state object with the values logged in the console. Now that we have the object we want to post let's do the same with Axios. this is pretty much like the get request with a small change. first at the top import Axios **import axios from ‘axios’**. Next in the submitHandler we make the post request axios dot post and the first argument is the URL which as it turns out is the same as the get request. so copy the URL and paste it. now the two endpoints are different though because of the get and post HTTP verbs. The post method takes in a second argument which is the data that has to be sent, in our case it is the state object this dot state. Once the request completes we either get a response or get an error. To keep it simple I will simply love them both to the console.

import React, { Component } from 'react'

import axios from 'axios'

class PostForm *extends* Component

{

    constructor(props)

    {

        super(props)

        this.state = {userId:'', title:'', body:''}

    }

    changeHandler=(e)=>

    {

        this.setState({[e.target.name]:e.target.value})

    }

    submitHandler=(e)=>

    {

        e.preventDefault()

        console.log(this.state)

        axios.post("https://jsonplaceholder.typicode.com/posts", this.state)

        .then(response =>

            {

                console.log(response)

            })

        .catch(error =>

            {

                console.log(error)

            })

    }

    render()

    {

        const {userId, title, body}=this.state

        return (

        <div>

            <form *onSubmit*={this.submitHandler}>

                <div>

                    <input *type*="text" *name*="userId" *value*={userId} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <input *type*="text" *name*="title" *value*={title} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <input *type*="text" *name*="body" *value*={body} *onChange*={this.changeHandler} />

                </div>

                <div>

                    <button *type*="submit">Submit</button>

                </div>

            </form>

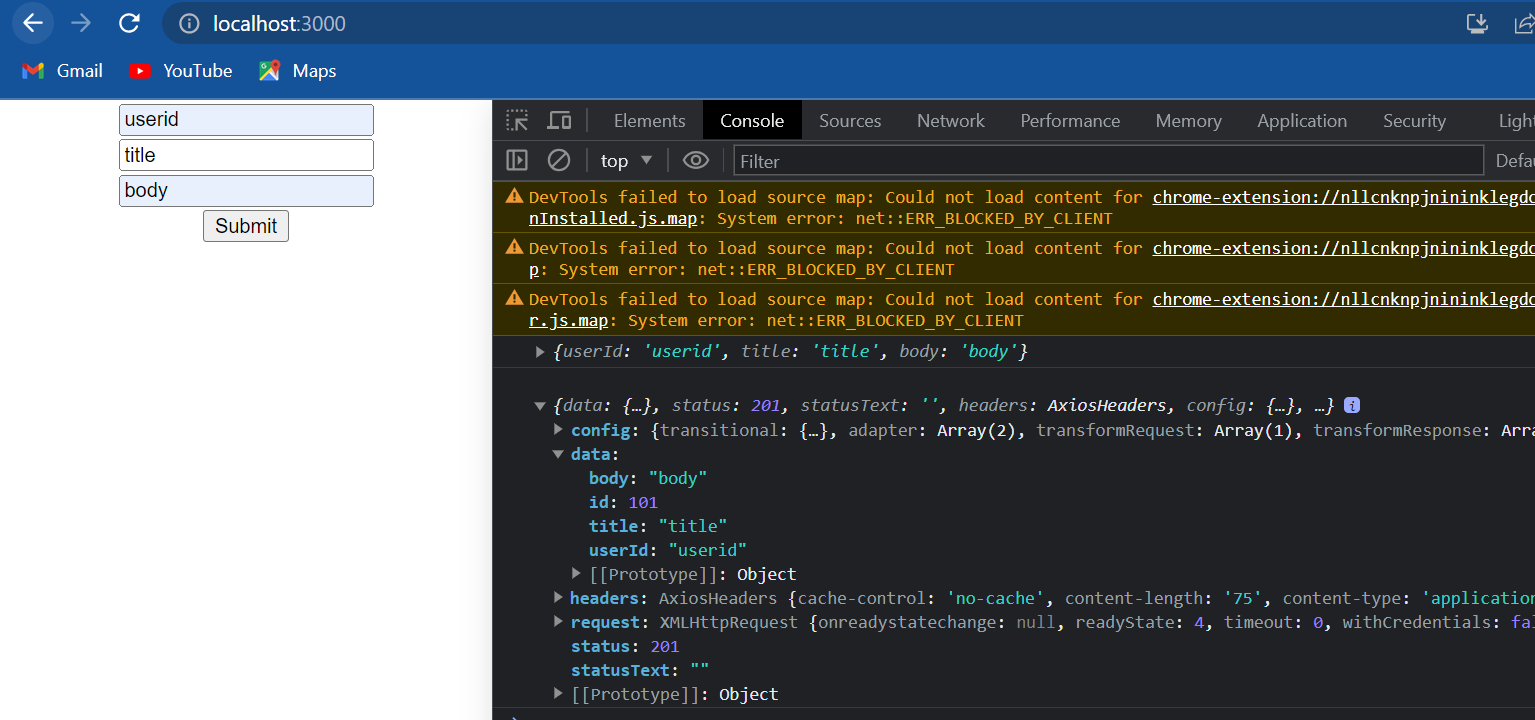
        </div>

        )

    }

}

*export* default PostForm



now let's go back to the browser and test it out I will fill in userId, title, and body and click on submit we should now be able to see the response in the console if we expand the data object we pretty much get back the same object we posted but this time with the unique ID property. This data object can then be used for any necessary changes in the UI. for example we can add this object to the end of the post list which might be displayed in the browser. so there we go a post request in our react application with Axios; create the input fields, tie the values to the state object,t and make a post request in the submit handler. Now posting data generally requires form validation and so on and we might want to take a look at a library like formic which helps we with forms in react. Also if we're using redux we might want to take a look at redux Axios middleware.

Now we've seen in this video is the simplest way to post data to a server. now once we get the hang of get and post requests making put or delete requests is pretty much the same so I will leave that for we guys to experiment with. now then I did mention earlier hooks are officially released as part of version 16.8.

----