React useContext hook

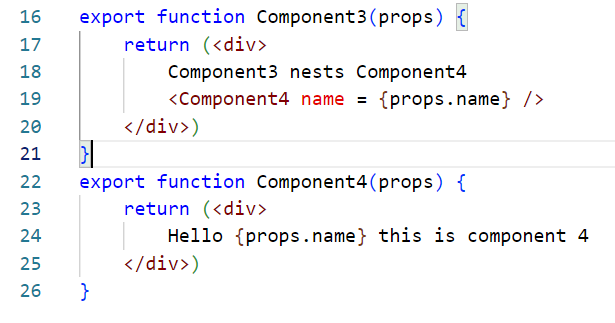
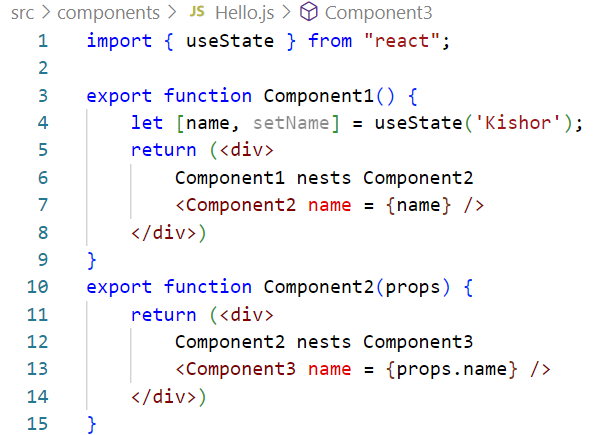
React Context is a way to manage the state globally, it is used when you want to share state between deeply nested child components

Without useContext we must do props drilling

When we want to share the state between the deeply nested components we need to pass the props through each nested components this is called props drilling and it will be not feasible in the complex application

ex:

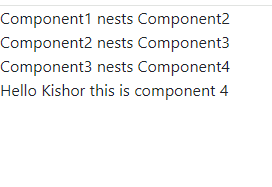
src/components/Hello.js



src/App.js



Output:



Even though Component2 & Component3 didn’t need the state, they had to pass the state along to reach the Component4, this is called props drilling, however it is not easy if there are more number of nested components

What is the solution for props drilling

Solution is to create Context

import {createContext} from ‘react’;  
const UserContext = createContext();

Now use the Context Provider to wrap the child components that need the states

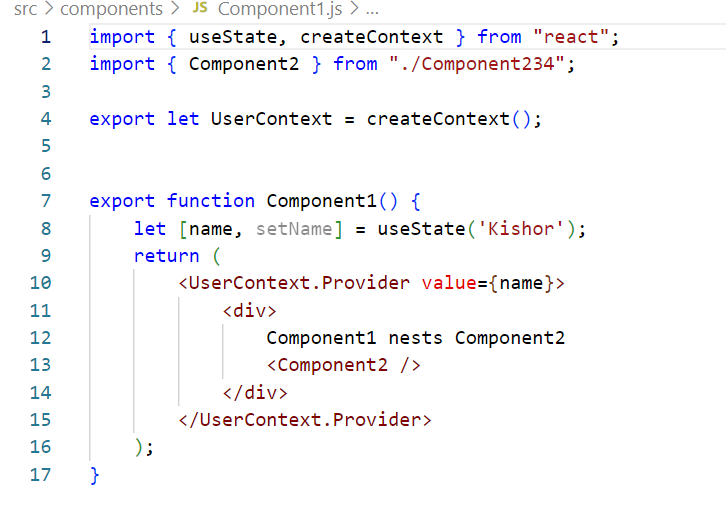
<UserContext.Provider value = {user}>  
 <Component3 />  
</UserContext>

Now the Component that need the state must use the context as below

import {useContext} from ‘react’;

function Component4() {  
 const user = useContext(UserContext);  
 return (<> Hello {user} </>);  
}

src/Component1.js



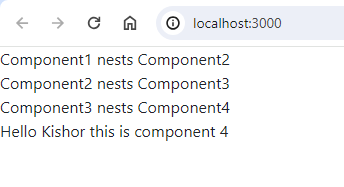
src/Components234.js



src/App.js



Output:



Passing multiple data is possible

Context Provider takes anything in the value so you can pass an object or array or simple value

ex:   
Below is the code you will use to pass multiple values  
<UserContext value = { [value1, value2,….]}>

To get the values in other components you will use the context as below

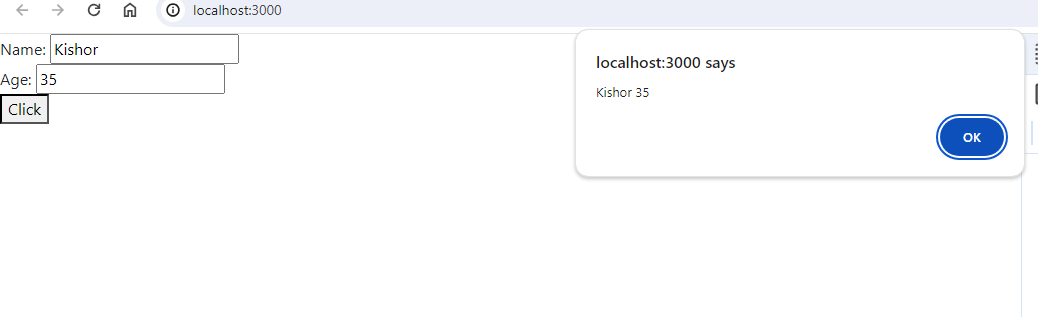
let values = userContext(UserContext);  
let name = values[0];  
let age = values[1];

useRef

You can access DOM elements through useRef, it returns an object current



Output:



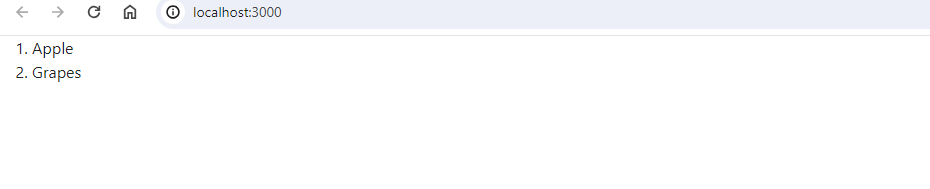
Fragments: <Fragment> or <>..</>

Fragment is used when you want to group the elements without a wrapper node, in React a component must always have a wrapper node like <div>, however when a component is reused in other components and you don’t want a wrapper node which might cause issue in creating a DOM tree then you can use the Fragments

return (<Fragment>  
 <td>Kishor</td><td>35</td>  
</Fragment>)



Output:



React Life cycle methods & hooks

life cycle methods are called automatically from the moment components are created to the point they are removed from the user interface, we can use these methods to control the component behaviour & perform some task at different stages of its life cycle, and also its not mandatory to use them in every components, you must know when they are called later on while working on the applications you will get an idea which one to use when.

In React, components have a life cycle that consists of different phases, each phase has a set of life cycle methods which are called at different phases, below are the different phases

1. Component mounting phase
2. Component updating phase
3. Component unmounting phase

Component mounting phase: This phase refers to the period when a component is being created & inserted into the DOM, this phase had 3 main life cycle methods which are called in the below order

1. constructor: Called when the component is first initialized
2. render: Called when the component is re-rendered, either because its props or state have changed, or because parent component has been re-rendered
3. componentDidMount: Called when component is mounted to the DOM, if you want to access browser related API’s or setting timers you can do it here

Component updating phase: Called when the props or state changes

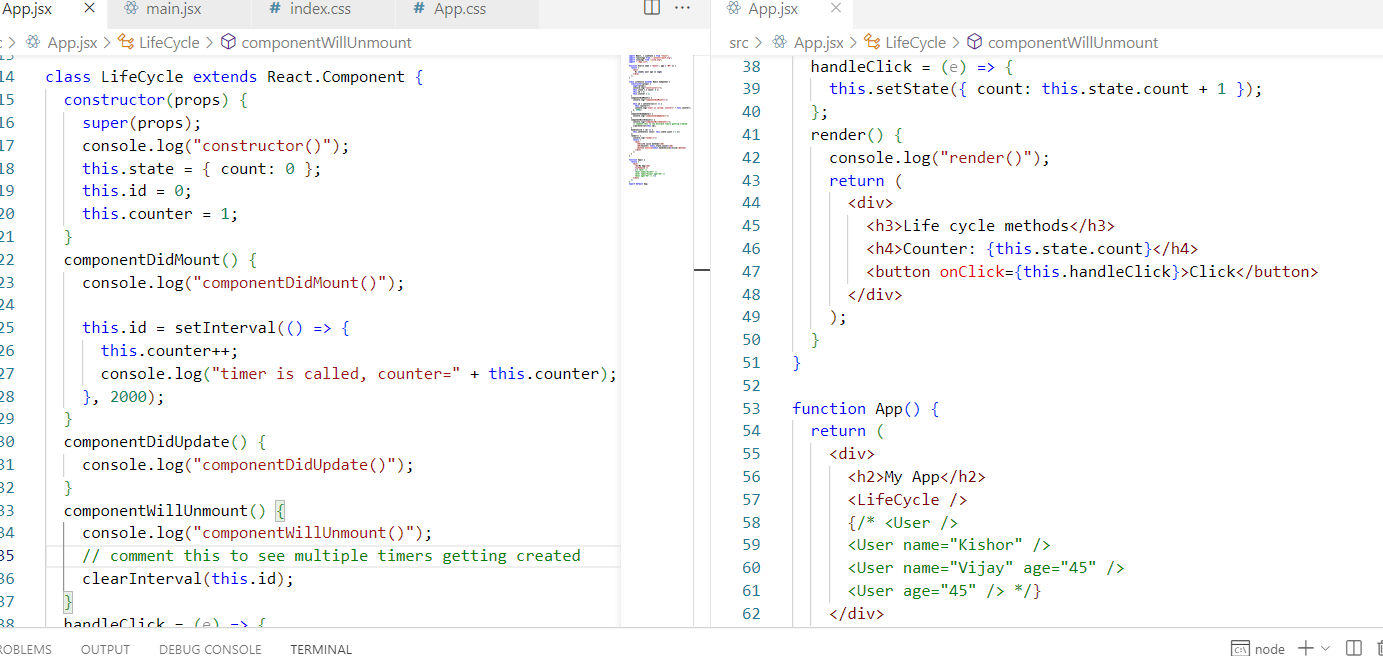
1. componentDidUpdate: Called when the component state or props changes

Component unmount phase: Called when the component is removed from the DOM

1. componentWillUnmount: Called when the component is removed from the DOM and can perform resource releasing tasks like clearing the timers

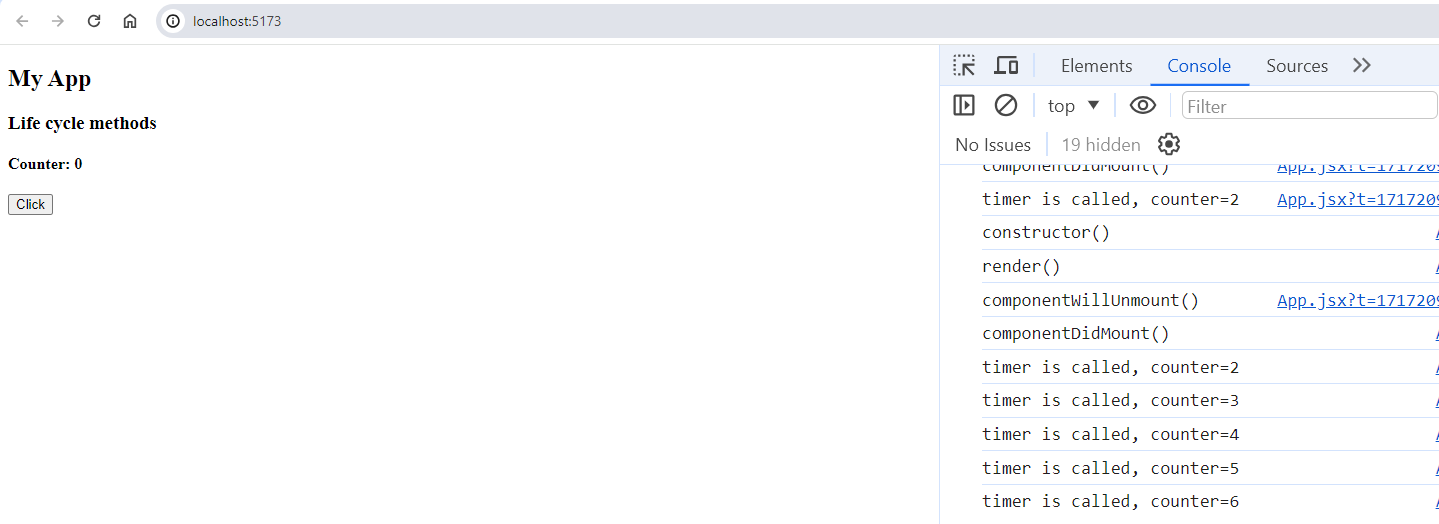
Example to understand componentDidMount & componentWillUnmount, we just need to make changes in the code and see the browser console to see the componentWillUnmount which is called when the code changes as the React removes the old DOM & adds the new DOM to the browser

App.js



Note: comment clearInterval() & modify the code a few times where you can see so many timers getting created and also componentWillUnmount() is getting called, however timer doesn’t stop, hence we can keep clearInterval() in the componentWillUnmount() which will clear these timers before the component is removed from the DOM

Output:



React Hooks

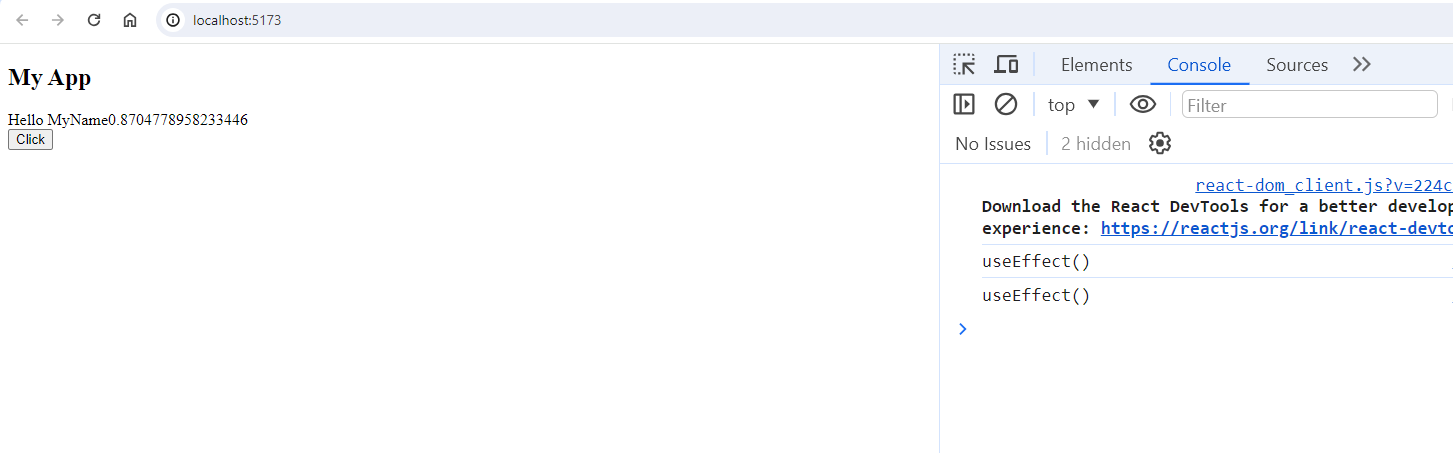
These are introduced in React V16, to add react features like life cycle methods, states in the functional components

1. useState(): to create states & methods
2. useEffect(): to add effects when the component is re-rendered or when props or state changes
3. useContext(): to share the state across the components without props drilling
4. useRef(): Used to manipulate the DOM

Using useEffect(callback, []) which is called only when the age changes



Output:



useRef example



Output:



Higher Order Components (HOC)

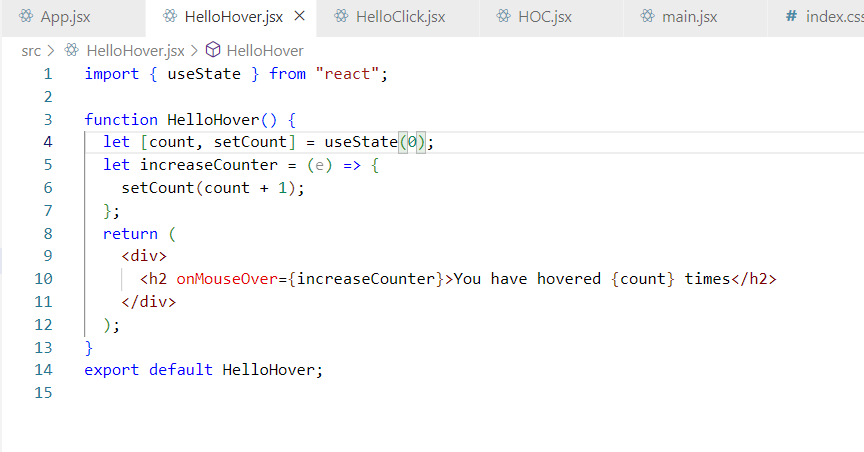
It is used to reuse the components logic when the same logic is required in multiple components you don’t have to write them in each component rather you can create an HOC which is a function that takes a component and returns a new component.

ex:

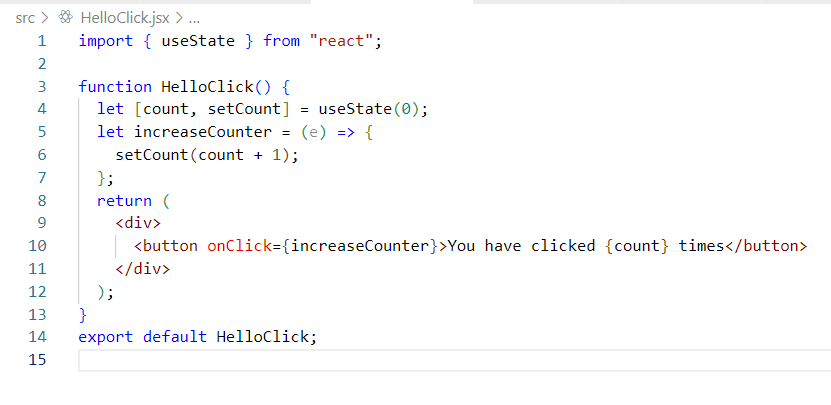
const EnhancedComponent = higherOrder(WrappedComponent)

Without HOC you need to write same logics in multiple components

HelloHover.jsx

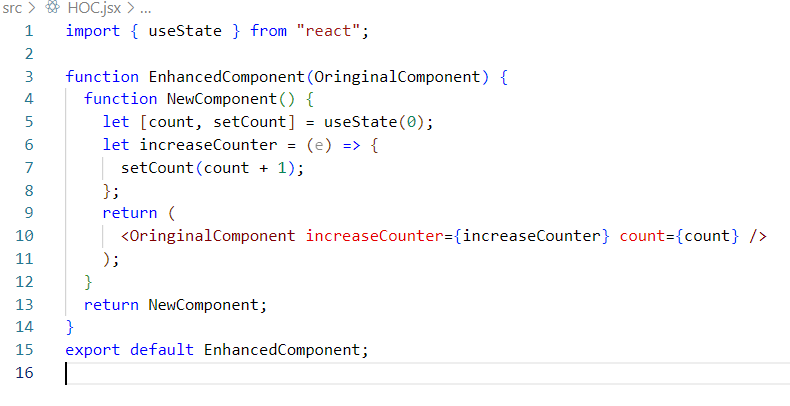


HelloClick.jsx

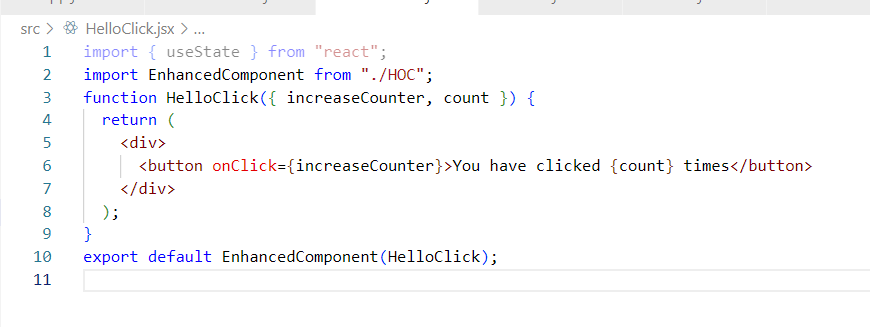


Creating HOC solves this problem

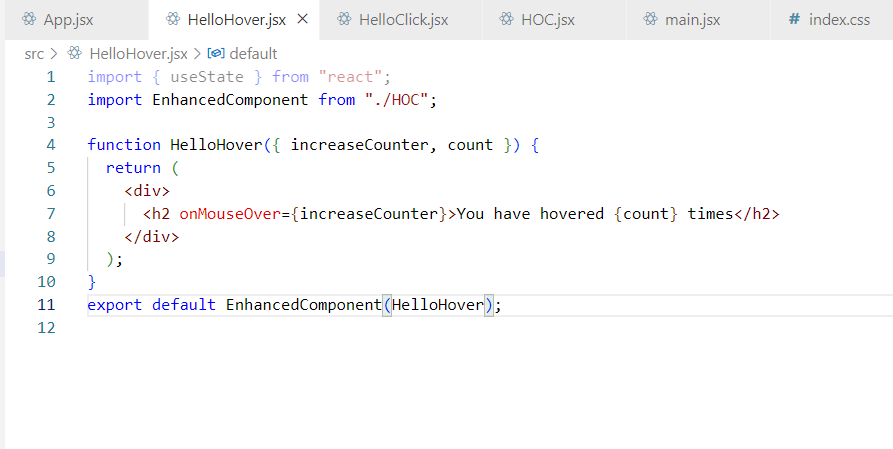
HOC.jsx



HelloClick.jsx



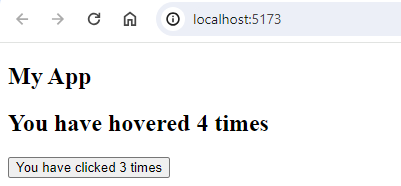
HelloHover.jsx



App.jsx

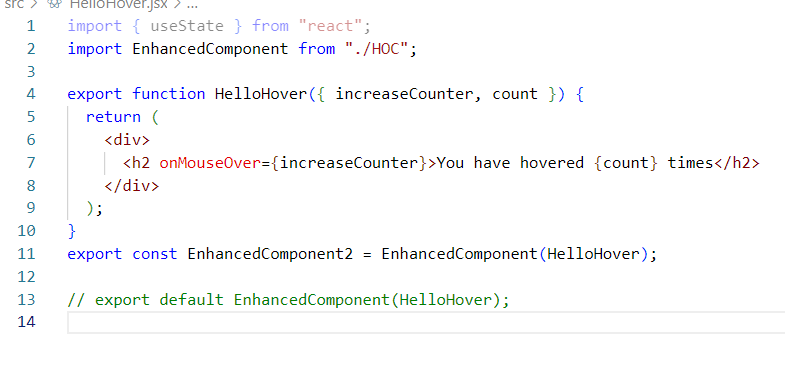


Output:

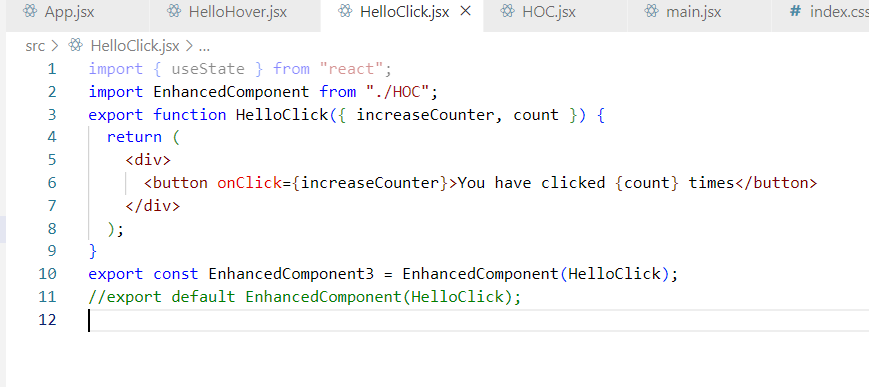


You can also use with named exports for HOC as below

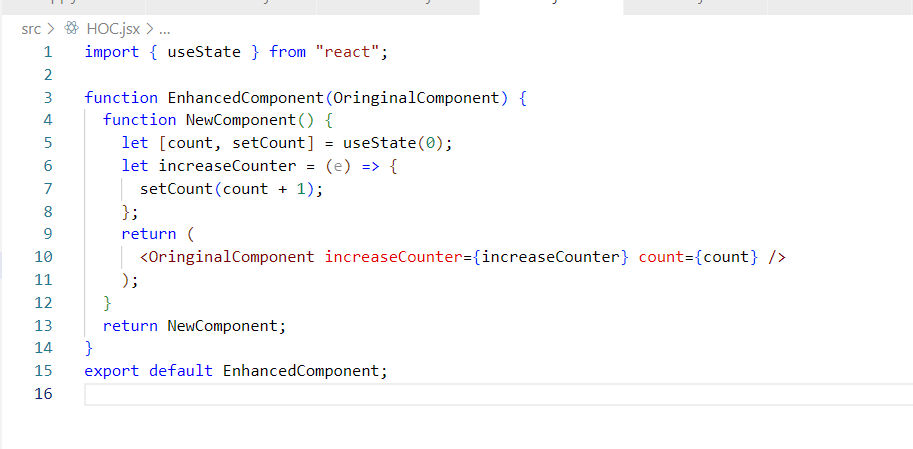
HelloHover.jsx



HelloClick.jsx



HOC.jsx



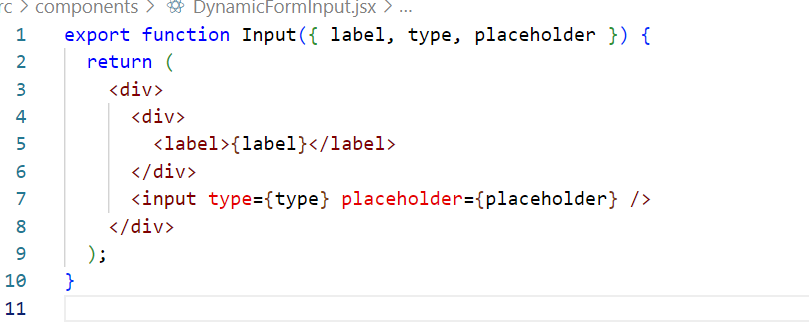
App.jsx



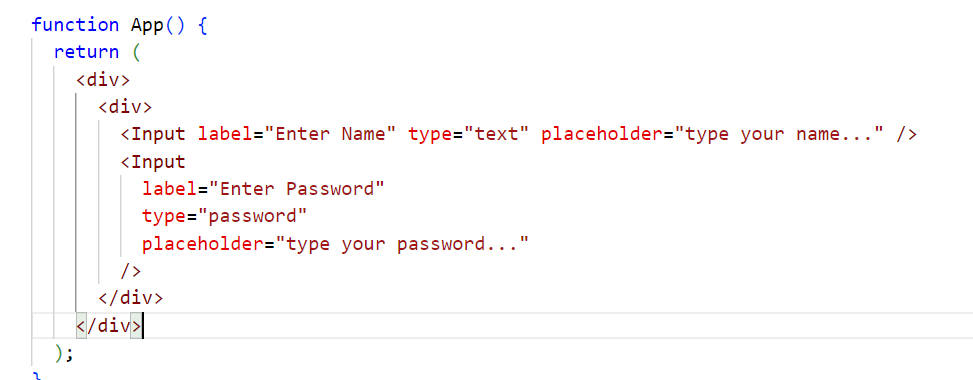
Custom Dynamic Input

We can create custom dynamic input and reuse in our forms

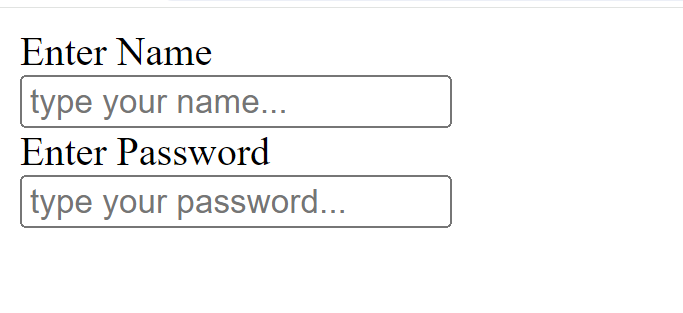
DynamicFormInput.jsx: This is a JS Config for the Form



App.jsx



Output:



Try to validate the form and handle form input referencing to below website

<https://www.freecodecamp.org/news/how-to-validate-forms-in-react/>

Debugging in React

We can use

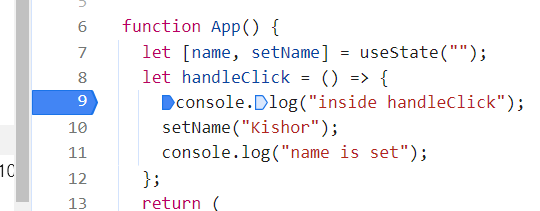
1. Google Developer tools
2. React Developer tools

Debugging in Google Dev tools

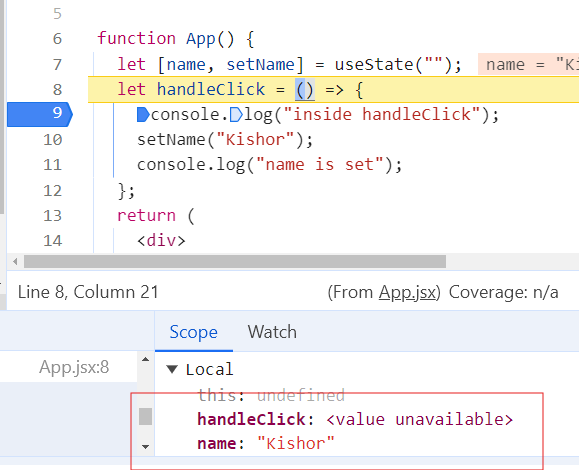
App.jsx: Create a button and an handler to update the state



Open Google chrome and create a break point inside handleClick



Now click on the button you will see the debugger pauses inside the handleClick, keep using the step over you will also observe the state getting updated when App is re-rendered

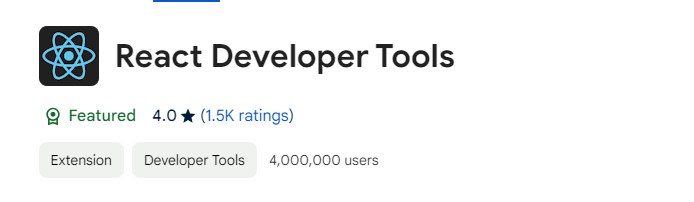


Note: If any errors are present you can see where debugger exits

Note: You need to step out when the debugger goes inside the inbuilt functions

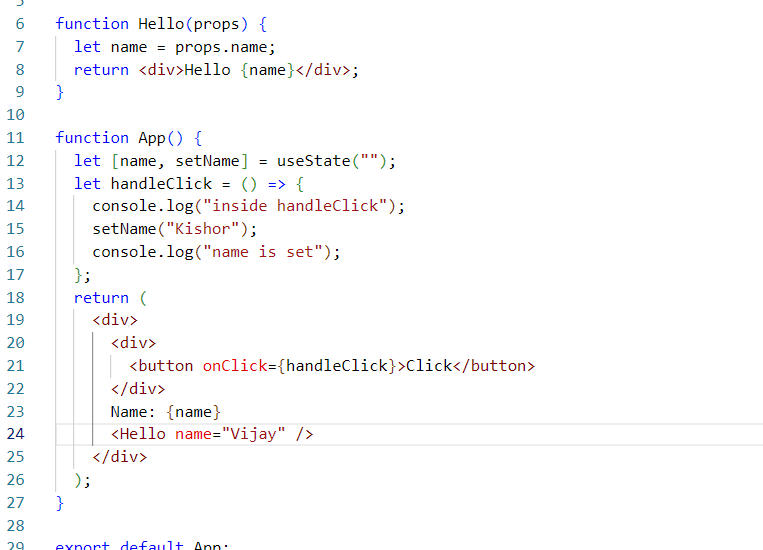
Using React Dev tools

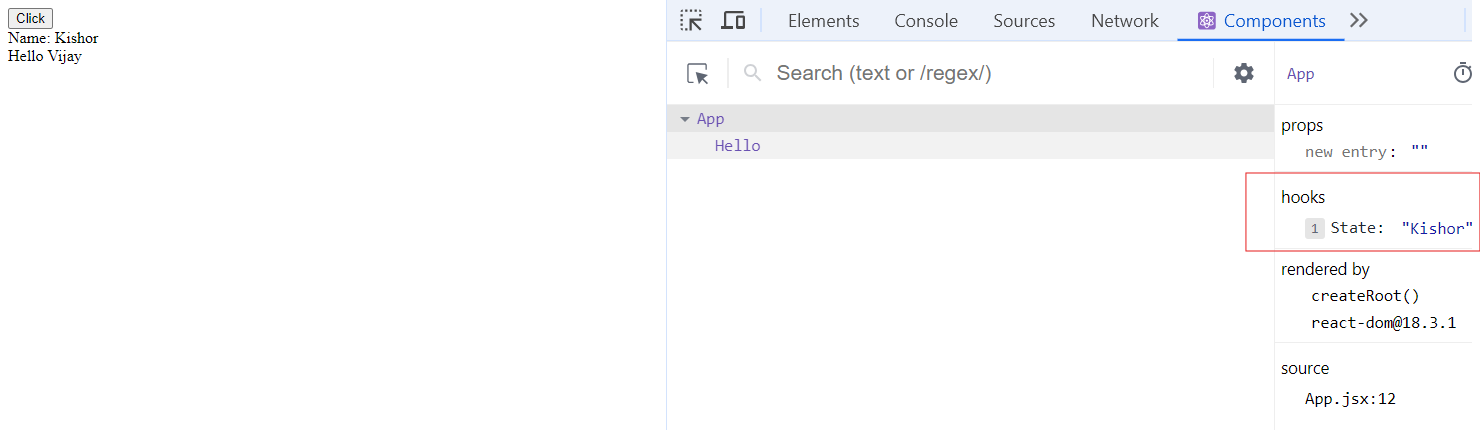
Add the below plugin



In this you can see the props, state of the components & also highlight the components to see how they appear

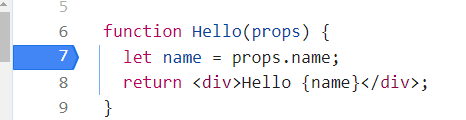
Create Hello Component and render in App



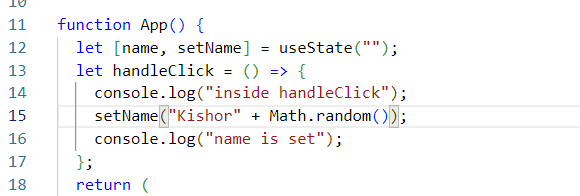


You can observe the component Hello getting re-rendered when the state is modified

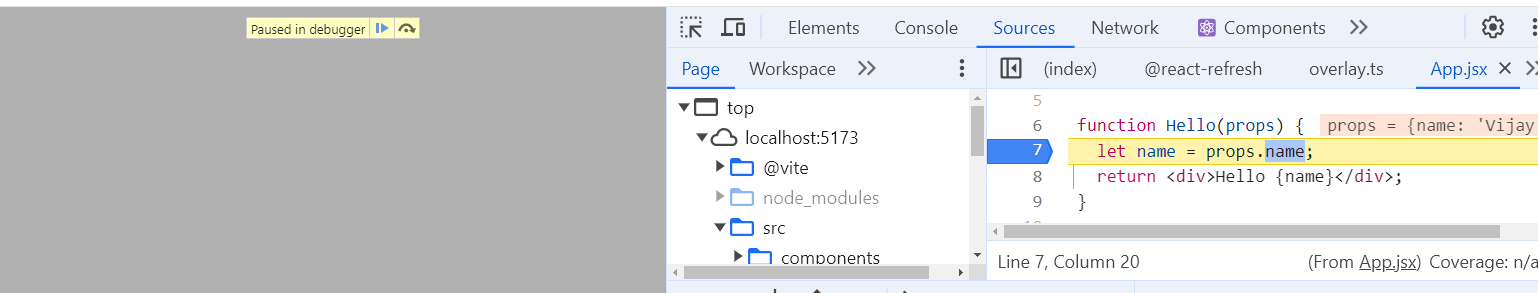
Keep the breakpoint in the Hello



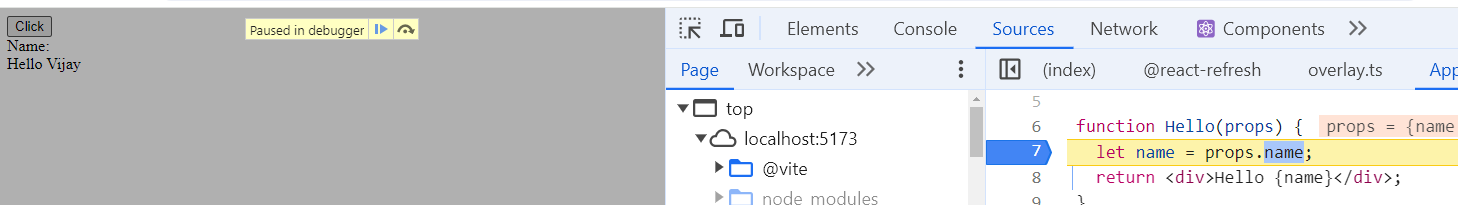
Use Math.random() in the setName() of App



Output:

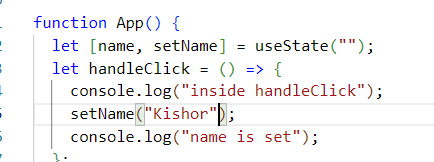


When you click on the button then you will see the Hello getting re-rendered in the debug mode, because the debugger again stops at the 7th line, this is because setName() of App is rendering random values on each click



When the setName() is same you can see Hello not getting re-rendered,

Change the setName() as below



Output:

