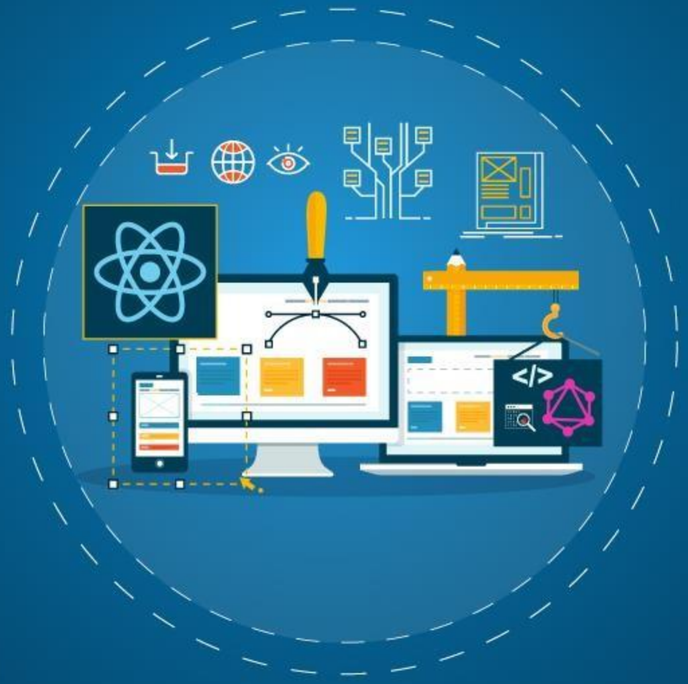
React With Redux Certification Training

COURSE OUTLINE
MODULE 06

1. Introduction to Web Development and React
2. Components and Styling the Application Layout
3. Handling Navigation with Routes
4. React State Management using Redux
5. Asynchronous Programming with Saga Middleware



6. **React Hooks**
7. Fetching Data using GraphQL
8. React Application Testing and Deployment
9. Introduction to React Native
10. Building React Native Applications with APIs

Topics

Following are the topics covered in this module:

- Caveat of JavaScript class
- Functional components and React hooks
- What are React hooks?
- Basic hooks
- useState() hook
- How to write useState() hook when state variable is an array of objects
- useEffect() hook
- Fetch API data using useEffect() hook
- useContext() hook
- Rules to write React hooks
- Additional hooks
- Custom hooks
- Build a weather application using React hooks

Objectives

After completion of this module you should be able to:

- Build stateful components using without JavaScript class
- Build applications using basic React hooks
- Follow React hook rules while working with hooks
- Implement other additional React hooks
- Write your own custom hooks
- Build a weather application using React hooks

Before React Hooks

Drawbacks Of JavaScript Classes

- Understand the functionality of "***this***" keyword to work with class components
- Remember to ***bind event handlers*** in class components
- While creating components for complex scenarios, such as ***data fetching*** and ***subscribing*** the events we need to make use of different ***component lifecycle methods***
- Data fetching is done in ***componentDidMount()*** and sometimes in ***componentDidUpdate()***
- For ***event listeners*** you set events in ***componentDidMount()*** and unsubscribe in ***componentWillUnmount()***
- This leads to ***splitting*** of code as per component lifecycle methods and not as per the components functional use
- Though use of multiple components may not matter in terms of view structure but can cause ***wrapper hell***

Example: Counter Application Using Class Component

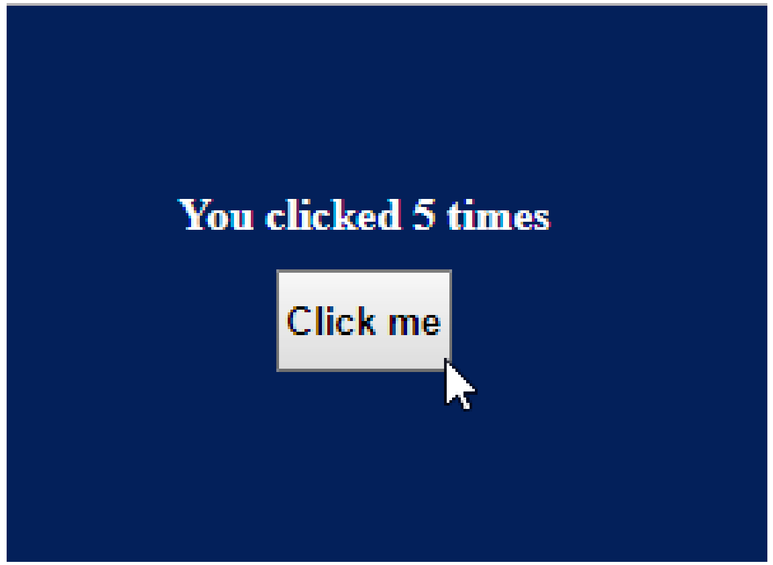
```
import React  from  'react';
import ReactDOM from 'react-dom';
import './App.css';

class Counter extends React.Component {

  constructor(props) {
    super(props);

    this.state = {
      count: 0,
    };
  }
  render() {
    return (
      <div>
        <h1>You clicked {this.state.count} times</h1>
        <button
          onClick={() =>

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

          }>
          <h1>Click me</h1>
        </button>
      </div>
    );}}
ReactDOM.render(<Counter/>, document.getElementById('root'));
```

Creates a class based component

Creates a state component and initializes it to zero

Method that sets this.state value

Output: Counter Application Using Class Component




Is it possible to work with states, without JavaScript classes to simplify the application code?
Yes, you can write functional components with React hooks.

Functional Components With React Hooks

01 Functional components with hooks are simpler, as there is no need to ***define constructors, this keyword, lifecycle methods, destructuring*** the same values multiple times

02 Hooks don’t split the components as per ***lifecycle methods***, rather they split a component into ***smaller functions*** based on related pieces

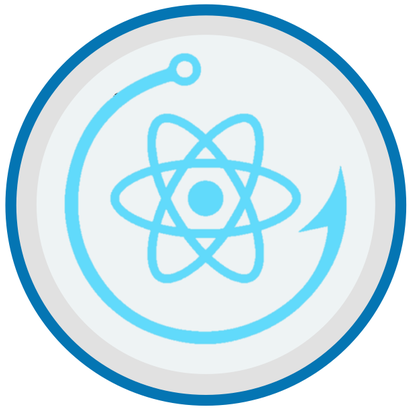
03 They ***organize logic*** inside a component into ***reusable isolated units***

04 They are easier to ***refactor and test***, allows developers to write clear and concise code

React Hooks

What Are React Hooks?

React hooks are basically functions that let us include react state and lifecycle features without JavaScript classes.



They allows us to use features of Stateful Components in ***functional components***

They mainly deal with ***state, effects*** and ***context*** in a React application

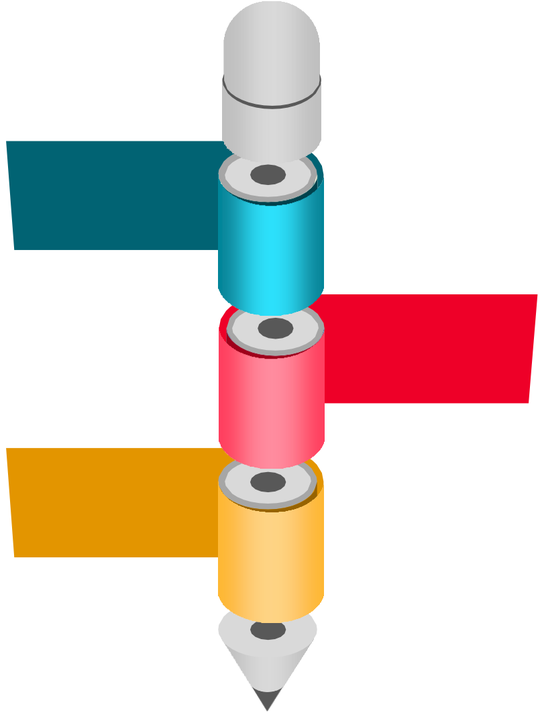
We need to ensure that the ***order of hooks*** always stays the same, we cannot use hooks in loops or conditionals

Basic Hooks

The ***basic hooks*** used to implement the features of stateful components in functional components are:

useState()

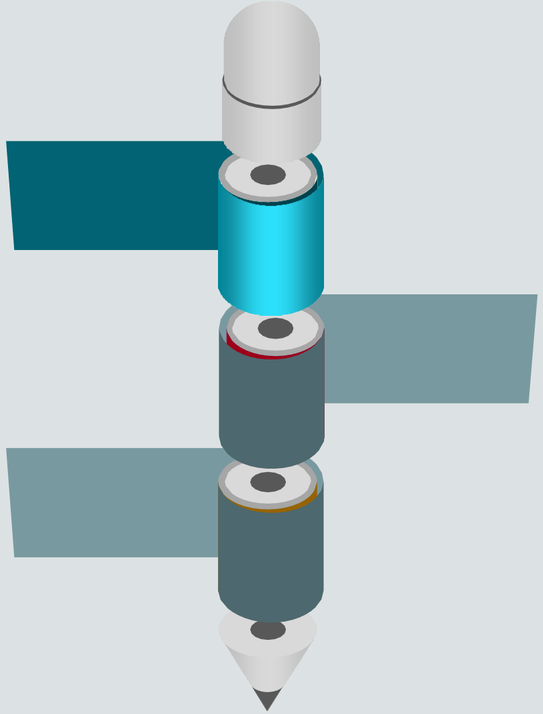
useContext()



useEffect()

useState()

useContext()



useEffect()

useState() Hook (State Hook)

useState hook is used to manage ***local states*** in functional component.

The ***useState() hook*** accepts an ***initial state*** as an argument and returns two variables, the first variable is the ***actual state*** and second variable is a ***setter function*** to update the state value

Syntax: useSate()

```
import { useState } from 'react'

const [ state, setState ] = useState(initialState)
```

New state

Setter function

Initial state

The useState() hook ***replaces this.state()*** and ***this.setState()*** methods used in class based components

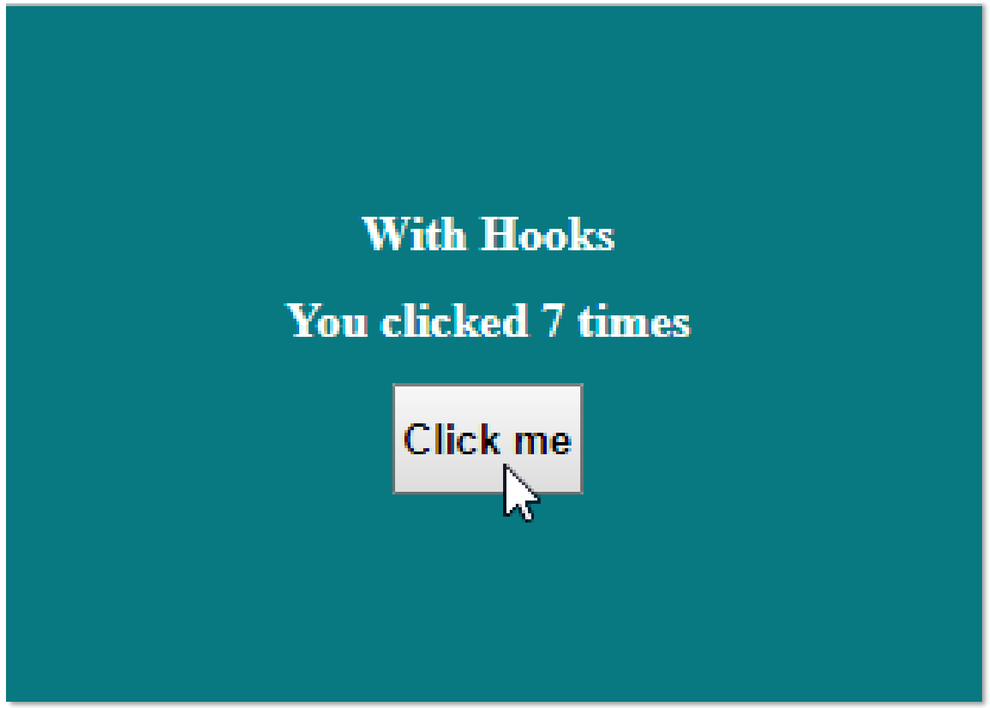
Example: Counter Application Using State Hook

Application Code

```
import React, { useState } from 'react';
import ReactDOM  from  'react-dom';
import './App.css';


function Counter() {

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

  return (
    <div>
        <h1>With Hooks</h1>
      <h1>You clicked {count} times</h1>
      <button onClick={() => setCount(count + 1)}>

<h1>Click me</h1>
      </button>
    </div>
  );
}

ReactDOM.render(<Counter/>, document.getElementById('root'));
```

- Creates a functional component
- Method that updates state variable
- Initial state property
- Current value of state variable
- Event handler

Output: Counter Application Using State Hook



Demo1: useState() With Previous State

Demo: useState() With Previous State

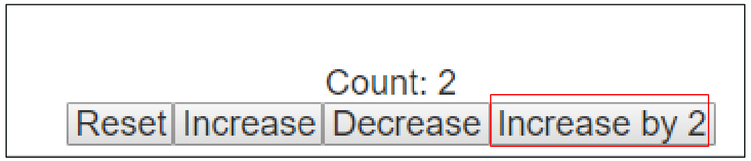
If you want to update the state value based on previous state value, then it's a good practise to pass a function in order to set new state value.

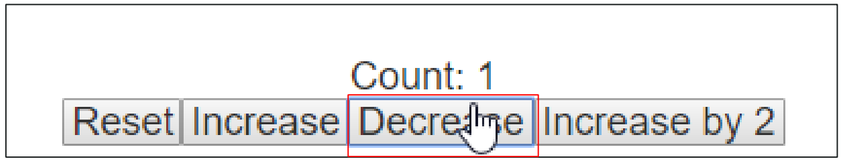
```
import React, {useState} from 'react';
function Counter() {

    const initialCount = 0
    const [count, setCount] = useState(initialCount)

    const increaseByTwo = () => {
        for(let i=0; i< 2; i++){

        setCount(prevCount => prevCount + 1)
}}
return(
    <div>
      <h1>
        Count: {count}
        <p>
        <button onClick= {() => setCount(initialCount)              } >Reset    </button>
        <button onClick= {() => setCount(prevCount => prevCount + 1)} >Increase</button>
        <button onClick= {() => setCount(prevCount => prevCount - 1)} >Decrease</button>
        <button onClick= {increaseByTwo}>                           Increase by 2</button>
       </p></h1>
    </div>
)}
ReactDOM.render(<Counter />, document.getElementById('root'));
```

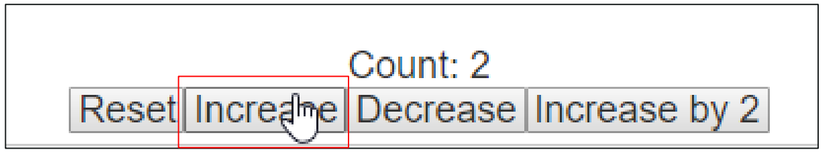
Previous state

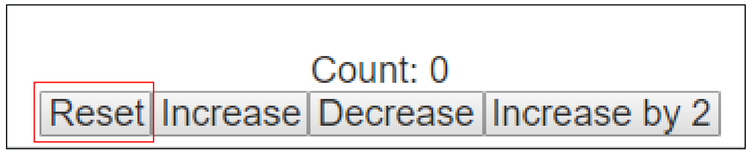
Passing function that has access to old value and later increments/decrements value by 1

Demo: Output


Count: 2
Reset
Increase
Decrease
Increase by 2


Count: 1
Reset
Increase
Decrease
Increase by 2


Count: 2
Reset
Increase
Decrease
Increase by 2


Count: 0
Reset
Increase
Decrease
Increase by 2

Demo 2: useState() Hook With Array

Demo: useState() Hook With Array

Below example will help us to understand how to use state hook when state variable is an array of objects.

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

function Counter() {
    const [numbers, setNumbers] = useState([])

    const addNumber =() => {
        setNumbers([...numbers, {
        id: numbers.length,
        value: Math.floor(Math.random() *10) +1
    }])}

return(
    <div>
        <h1>
        <button onClick={addNumber}>Add a number</button>
        <ul>
            {numbers.map(number => (
            <li key={number.id}>{number.value}</li>
            ))}
        </ul>
        </h1>
    </div>
)
}
export default Counter
```

State variable in form of an array

Function defined to add numbers to array

Setter function

Sets array of numbers

Logic to push new objects to the array

Function defined to generate random values between 1 to 10

Button which pushes random numbers in array

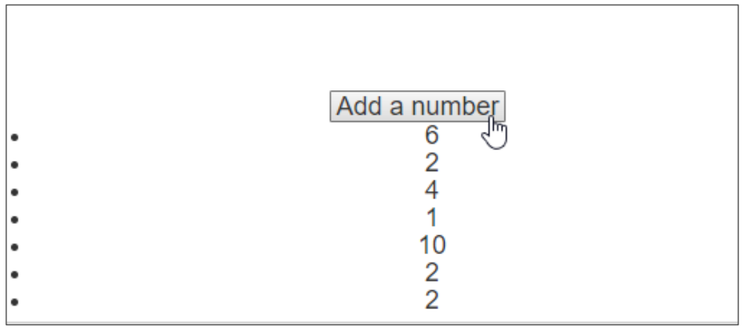
Data binding to display a number on corresponding click of button

Demo: Working Of Code

Whenever ***addNumber()*** is called we make a copy of all the numbers in the array using ***spread operator***

To the copied list we ***append*** another object (using id and value)

Output: Array of random numbers



For first iteration, array will ***not*** have any number (here id will be zero and value will be any random number between 1 to 10)

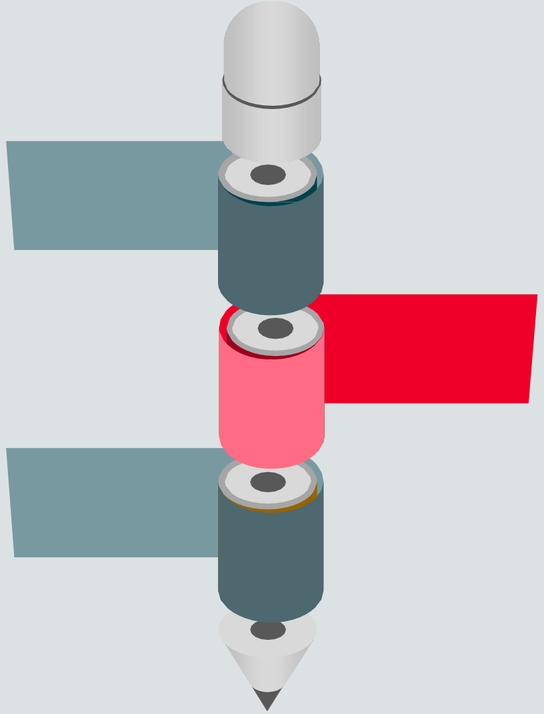
Next iteration we will have ***one number*** in numbers array, so we make copy of that and append an object to it (now id will be 1 and value will be again any random number between 1 to 10)

This process continues till the button is clicked

A ***setter function*** cannot merge and automatically update the numbers to array, we need to do this manually using ***spread operators.***

useState()

useContext()



useEffect()

useEffect() Hook (Effect Hook)

Effect Hook is used to manage the ***state and side-effects*** such as interactions with the browser/DOM API and external API like data fetching, timers, subscriptions, requests and more.

The useEffect hook replaces the ***componentDidMount, componentDidUpdate***, and c***omponentWillUnmount*** methods

componentDidUpdate can be implemented by using an ***effect hook*** with an empty array passed as the second argument.
For example, ***useEffect(() => console.log('did update'), [])***

componentDidMount can also be implemented by using an ***effect hook*** with an empty array passed as the second argument.
For example, ***useEffect(() => console.log('did mount'), [])***

componentWillUnmount can be implemented by returning a function from an ***effect hook*** with an empty array passed as the second argument.
For example***, useEffect(() => { return () => console.log('will unmount') }, [])***

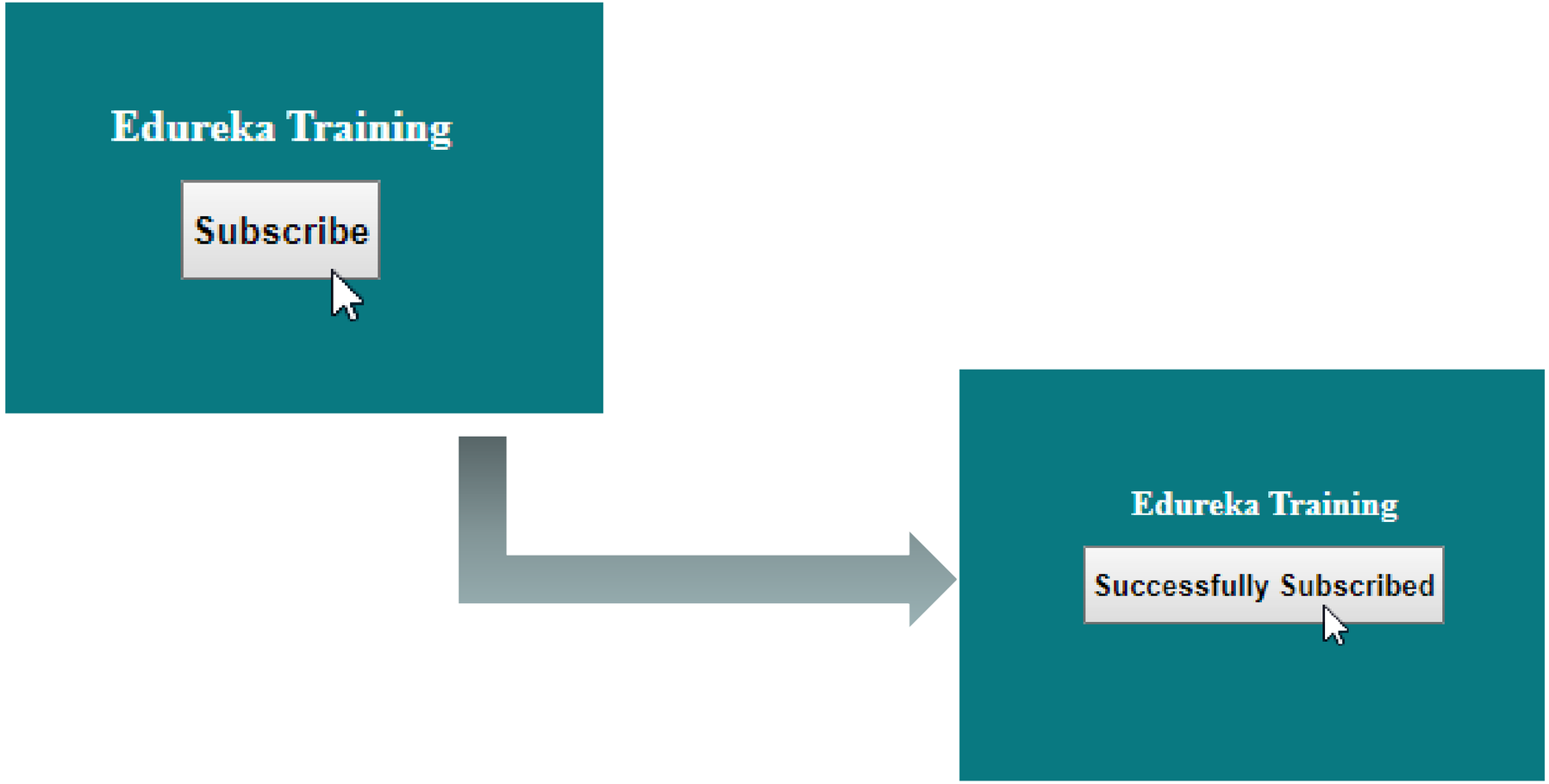
Syntax: useEffect()

```
import { useEffect } from 'react'

useEffect()
```

Example: Effect Hooks

```
import React, { useState, useEffect } from 'react';
import { render } from 'react-dom';

function App() {
  const [isOn, setIsOn] = useState(false);





return (
    <div>
      <h1>Edureka Training</h1>

      {!isOn && (
        <button type="button" onClick={() => setIsOn(true)}>
         <h1>Subscribe</h1>
        </button>
      )}

      {isOn && (
        <button type="button" onClick={() => setIsOn(false)}>
          <h1>Successfully Subscribed</h1>
        </button>
      )}
    </div>
  );
}
render(<App />, document.getElementById('root'));
```

Example: Output



Demo 3: Fetching Data Using useEffect()

Demo: Fetching Data Using useEffect()

Here we will be making use of ***axios*** to fetch data, so install it using: ***npm install axios***

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

function DataFetching(){
    const [albums, setalbums] = useState([])

    useEffect(() => {
        axios.get ('https://jsonplaceholder.typicode.com/albums')
        .then(res => {
            console.log (res)
            setalbums(res.data)
        })
        .catch(err => {
            console.log(err)
        })
    })

    return(
        <div>
            <ul>
                {albums.map(album => (
                <li key = {album.id}>{album.title}}</li>))
                }
            </ul>
        </div> )}
export default DataFetching
```

State variable

Sends request to metioned url

Updates albums state variable with API data

Promise to return data from API

Data binding

Demo: Output

List of albums (data) displayed via an API:

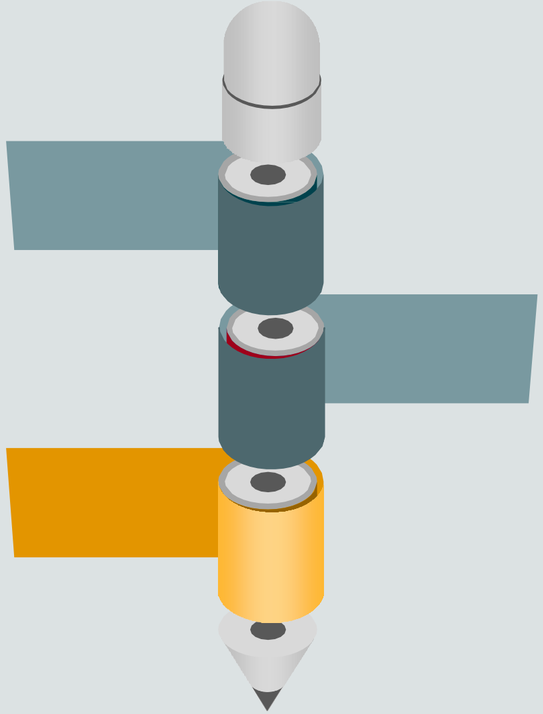
localhost:3000

- quidem molestiae enim}
- sunt qui excepturi placeat culpa}
- omnis laborum odio}
- non esse culpa molestiae omnis sed optio}
- eaque aut omnis a}
- natus impedit quibusdam illo est}
- quibusdam autem aliquid et et quia}
- qui fuga est a eum}
- saepe unde necessitatibus rem}
- distinctio laborum qui}
- quam nostrum impedit mollitia quod et dolor}
- consequatur autem doloribus natus consectetur}
- ab rerum non rerum consequatur ut ea unde}
- ducimus molestias eos animi atque nihil}
- ut pariatur rerum ipsum natus repellendus praesentium}
- voluptatem aut maxime inventore autem magnam atque repellat}
- aut minima voluptatem ut velit}
- nesciunt quia et doloremque}
- velit pariatur quaerat similique libero omnis quia}
- voluptas rerum iure ut enim}
- repudiandae voluptatem optio est consequatur rem in temporibus et}
- et rem non provident vel ut}
- incidunt quisquam hic adipisci sequi}
- dolores ut et facere placeat}
- vero maxime id possimus sunt neque et consequatur}
- quibusdam saepe ipsa vel harum}
- id non nostrum expedita}
- omnis neque exercitationem sed dolor atque maxime aut cum}
- inventore ut quasi magnam itaque est fugit}
- tempora assumenda et similique odit distinctio error}
- adipisci laborum fuga laboriosam}
- reiciendis dolores a ut qui debitis non quo labore}
- iste eos nostrum}
- cumque voluptatibus rerum architecto blanditiis}
- et impedit nisi quae magni necessitatibus sed aut pariatur}
- nihil cupiditate voluptate neque}
- est placeat dicta ut nisi rerum iste}
- unde a sequi id}
- ratione porro illum labore eum aperiam sed}

useState()

useEffect()

useContext()



Why Should We Use Context Hook?

- React applications can have ***multiple nested components*** as shown in the figure
- In case of ***passing the props***, you can pass a props only from a ***parent*** component to the ***child*** component
- Props from an ***App*** component can be directly passed to ***component U***
- In order to pass the prop to ***component W,*** we have to pass the prop to ***component V***
- ***Component Z*** will receive the prop only if it is passed to ***component X*** and **Component *Y*** consecutively
- This leads to ***wrapper hell*** if we have more than ***10-15 components***
- To avoid this ***React 16*** introduces ***context hook***

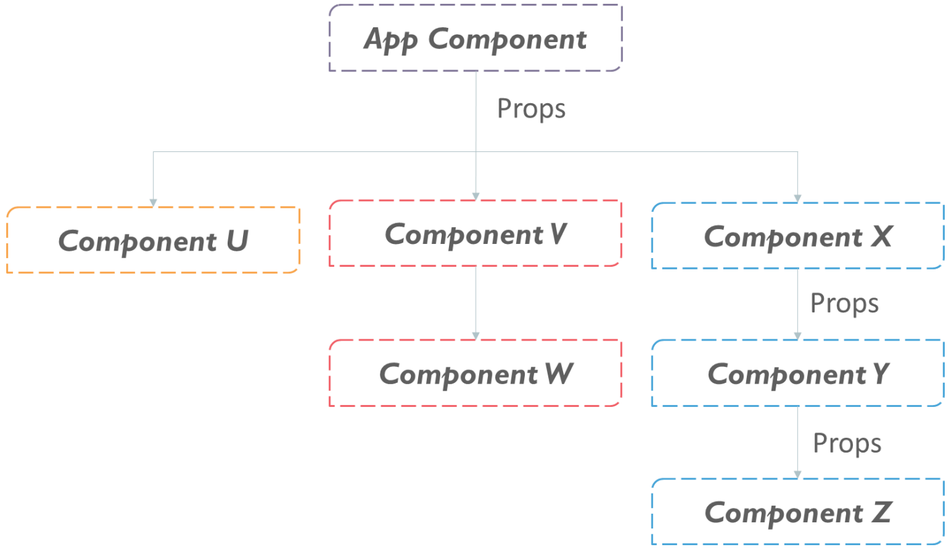


Fig: Passing the props within the component tree

useContext() (Context Hook)

Context provides a way to pass data through the component tree without having to pass props down manually at multiple levels.

- React contexts consist of a ***provider*** and a ***consumer***, provider is defined to pass the context value and consumer is defined to accept the context value

Syntax: useContext()

```
import { useContext } from 'react'

const value = useContext(MyContext)
```

- The ***useContext()*** hook is used to deal with context in React, it accepts a ***context object*** and returns the ***current context value***

How To Write Context Hook?

There are ***3 steps*** to implement ***context hook***:

1) Create a context using:

```
const<contextName>= React.createContext()
```

3) Consume the context value using:

```
<contextName.Consumer>
{
   Data => {
   Return()
}}
</ contextName.Consumer>
```

2) Provide a context value using:

```
<contextName.Provider value = { 'data'}
<componentName />
<contextName.Provider>
```

Demo 4: Multiple Context Using useContext() Hook

Demo: Nested Components

Create three components as shown below:

```
import React from 'react'
import ComponentY from './Component_Y.js'

function ComponentX(){
    return(
        <div>
            <ComponentY />
        </div>
    )
}

export default ComponentX
```

Component_X.js

```
import React from 'react'
import ComponentY from './Component_Z.js’

function ComponentY(){
    return(
        <div>
            <ComponentZ />
        </div>
    )
}

export default ComponentX
```

Component_Y.js

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

function ComponentZ(){

    return()

}
export default ComponentZ
```

Component_Z.js

Demo: App.js

Create the context, provide the created context a value, and children components who should receive this value must be wrapped in a provider.

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

import  ComponentX from './Component_X';
import './App.css';

export const OrganizationContext = React.createContext()
export const CourseContext = React.createContext()

function App() {
    return (
    <div className="App">

    <OrganizationContext.Provider value={'edureka!'}>

    <CourseContext.Provider value={'React with Redux Certification Training’}>

    <ComponentX />

    </CourseContext.Provider>

    </OrganizationContext.Provider>

    </div>
    )
}
export default App
```

Import the path of nested component

Creates Multiple contexts

Value to be passed, to the nested component

Component wrapped within provider

Demo: Component_Z.js

Define the ***consumer*** to accept the value passed by App.js.

```
import React, {useContext} from 'react';
import {OrganizationContext, CourseContext} from './App.js'

function ComponentZ(){

    const organization= useContext(OrganizationContext)
    const course= useContext(CourseContext)

     return(
         <div>
             <h1>
             {organization}
             <p>{course}</p>
             </h1>
         </div>
     )
 }

export default ComponentZ
```

Imports necessary context

Assigns the context values to the variables

Renders the data to be displayed on screen

Demo: Output

edureka!
React With Redux Certification Training

Rules To Write React Hooks

Do’s

Hooks can only be ***called*** at the beginning of React functional components or custom hooks

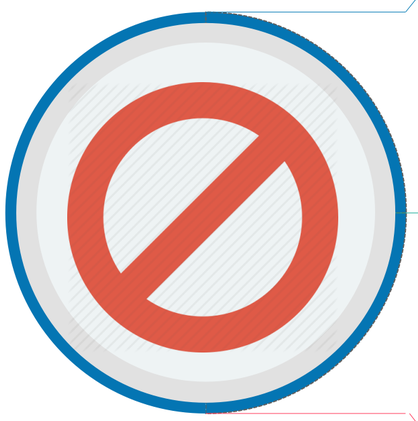
Hook function names should always start with a ***use prefix*** and then a name in ***CamelCase***. For example: useSomeHookName

We can use ***eslint*** with ***eslint-plugin-react-hooks*** to enforce the rules of hooks

Make use of ***exhaustive dependencies*** to ensure that in an ***effect hook*** all variables being used are listed as dependencies via the ***second argument***

To ***automatically fix linter*** warnings execute command: ***npm run lint -- --fix***, running this command will automatically enter all variables being used in ***effect hook*** as dependencies

Don’ts



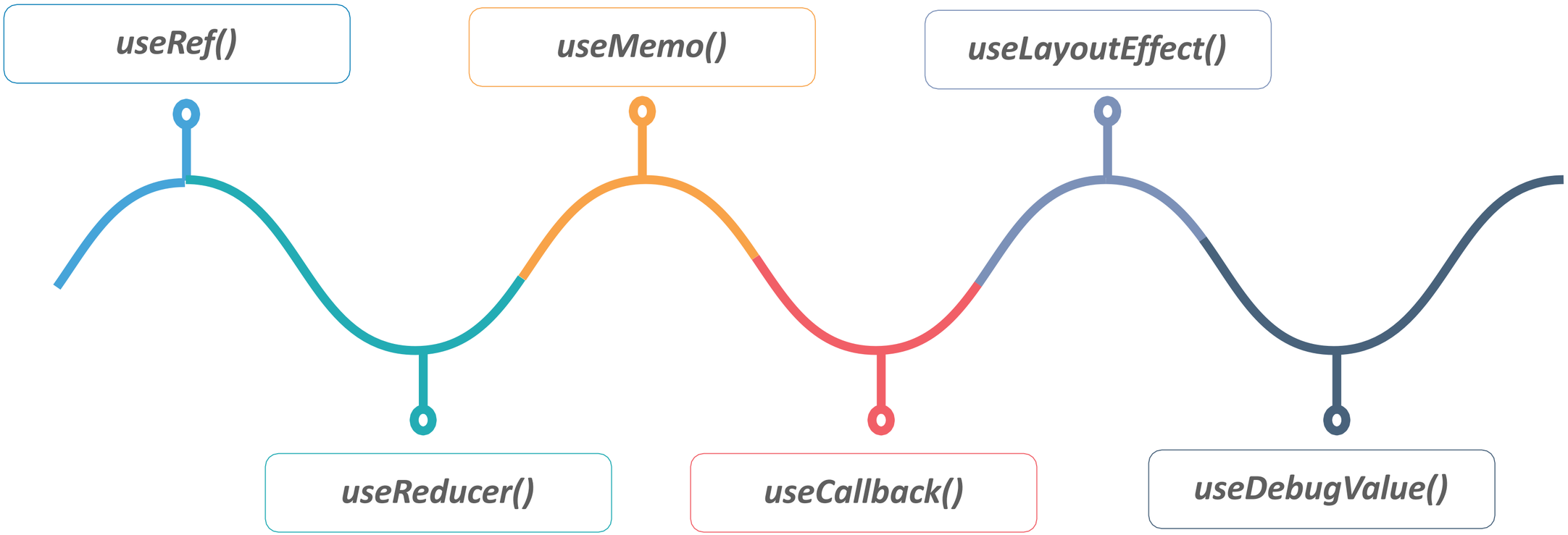
It is ***not*** possible to use hooks in React ***class components***

The order of hooks should never change, as it is used to track the values of various hooks

Hooks ***cannot be called*** inside conditionals, loops, or nested functions, because that would change the order of hooks

Additional Hooks

Additional Hooks



useRef() Hook

- ***useRef***
- *useReducer*
- *useMemo*
- *useCallback*
- *useLayoutEffect*
- *useDebugValue*

This hook returns a mutable ***ref object***, where the ***current*** property is initialized to the passed argument (initialValue).

Syntax

```
import { useRef } from 'react'

const refContainer = useRef(initialValue)
```

Use case: It is used to deal with references to elements and components in React. We can set a reference by passing the ***ref prop*** to an element or a component, as follows: ***<ComponentName ref={refContainer} />***

useReducer() Hook

useRef

useReducer

useMemo

useCallback

useLayoutEffect

useDebugValue

This hook is an ***alternative*** to ***useState() hook*** and works similarly to the Redux library.

Syntax

```
import { useReducer } from 'react'

const [ state, dispatch ] = useReducer(reducer,
initialArg, init)
```

Use case: The useReducer() hook is used to deal with ***complex state logic.***

useMemo() Hook

useRef

useReducer

useMemo

useCallback

useLayoutEffect

useDebugValue

Memoization is an optimization technique where the result of a function call is ***cached***, then returned when the same input occurs again. The ***useMemo()*** hook allows us to compute a value and memoize it.

Syntax

```
import { useMemo } from 'react'

const memoizedValue = useMemo(() =>
computeExpensiveValue(a, b), [a, b])
```

Use case: The ***useMemo*** hook is useful for optimization when we want to avoid re-executing expensive operations.

useCallback() Hook

useRef

useReducer

useMemo

useCallback

useLayoutEffect

useDebugValue

This hook allows us to pass an ***inline callback function***, an ***array of dependencies*** and will return a ***memoized*** version of the callback function.

Syntax

```
import { useCallback } from 'react'
const memoizedCallback = useCallback(
  () => {
    statement(a, b)
  },
  [a, b]
)
```

Use case: It is useful when passing callbacks to optimized child components. It works similar to the ***useMemo()*** hook, but for callback functions.

useLayoutEffect() Hook

useRef

useReducer

useMemo

useCallback

useLayoutEffect

useDebugValue

The ***useLayoutEffect()*** hook can be used to read information from the DOM. It is identical to ***useEffect(),*** but it only fires after all Document Object Model (DOM) mutation.

Syntax

```
import { useLayoutEffect } from
'react'useLayoutEffect(didUpdate)
```

Use case: Use ***the useEffect()*** hook when possible, because ***useLayoutEffect()*** will block visual updates and slow down your application.

useDebugValue() Hook

useRef

useReducer

useMemo

useCallback

useLayoutEffect

useDebugValue

This hook can be used to display a ***label*** in React DevTools, while creating ***custom hooks***.

Syntax

```
import { useDebugValue } from 'react'

useDebugValue(value)
```

Make sure to use this hook in custom hooks to display the current state of your hooks, as it will make it easier to debug them.


Till now we learnt, how to make use of hooks provided by React 16, Now its time to learn how to create our own hooks.

Custom Hooks

Custom Hooks

A ***custom hook*** is a JavaScript function whose name starts with “use”.

- A custom hook can ***call*** other hooks as per requirement
- You can even take advantage of ***built-in hooks*** and build your ***own hook***
- They are mainly used to ***share logic*** between two or higher components

Demo 5: Custom Hooks

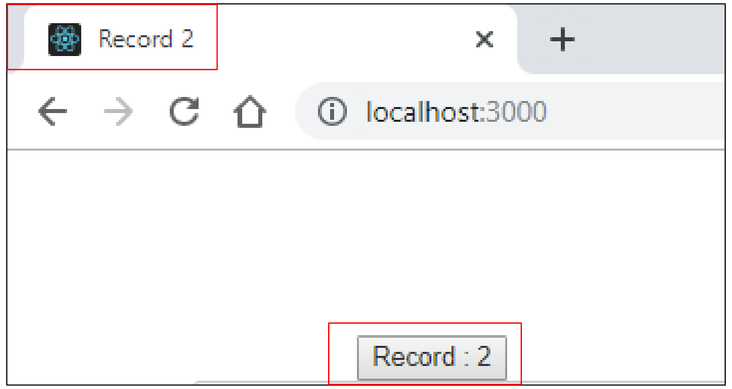
Demo: Custom Hooks

In this demo we will be creating a custom hook to update the document title. For this we will make use of a counter, as the count value is updated the document name should be updated.

Title1.js

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

function Title1() {

return (
<div>
<button onClick={() => setRecord(record + 1)}> Record : {record}</button>
</div>
)
}
export default Title1
```

Output



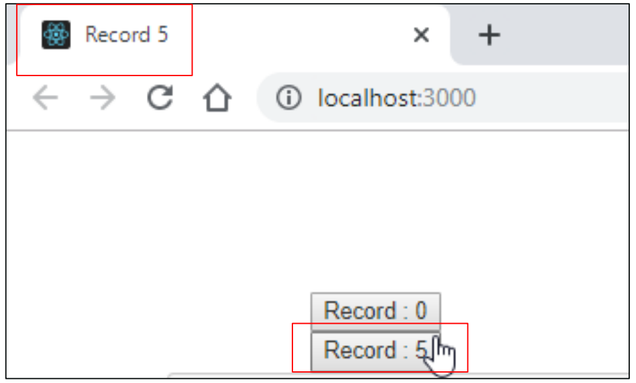
Demo: Custom Hooks (contd.)

Now we will create an another component and this time update the file name using second component: ***Title2.js***

Title2.js

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


function Title2() {

    const [record, setRecord] = useState(0)

    useEffect(() => {
        document.title = `Record ${record}`
    }, [record])


return (
    <div>
        <button onClick={() => setRecord(record + 1)}> Record : {record
}</button>
    </div>
)
}
export default Title2
```

Output




As you can see in both the components logic is repeating, this is where the ***custom hooks*** should be introduced.

Demo: Custom Hooks (contd.)

Create a custom hook and extract the logic in it, reuse the custom hook in different components

CustomHook.js

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

function useTitle(record) {


    useEffect(() => {

        document.title = `Record ${record}`
    }, [record])
}

export default useTitle
```

Custom hook

Logic to be passed to different components

Demo: Custom Hooks (contd.)

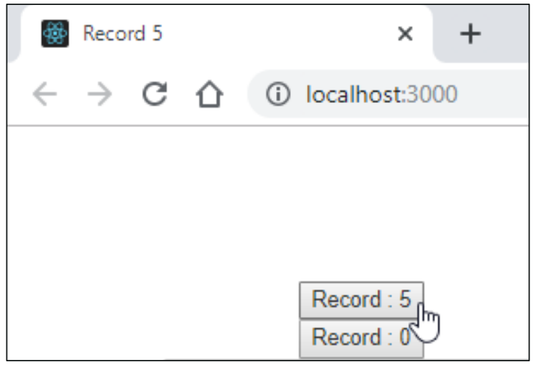
```
import React, { useState } from 'react';
import useTitle from './CustomHook.js'

function Title2() {
    const [record, setRecord] = useState(0)




return (
<div>
<button onClick={() => setRecord(record + 1)}>
 Record : {record}</button>
</div>
)}
export default Title2
```

Title2.js

Similarly update the components and check the Output.

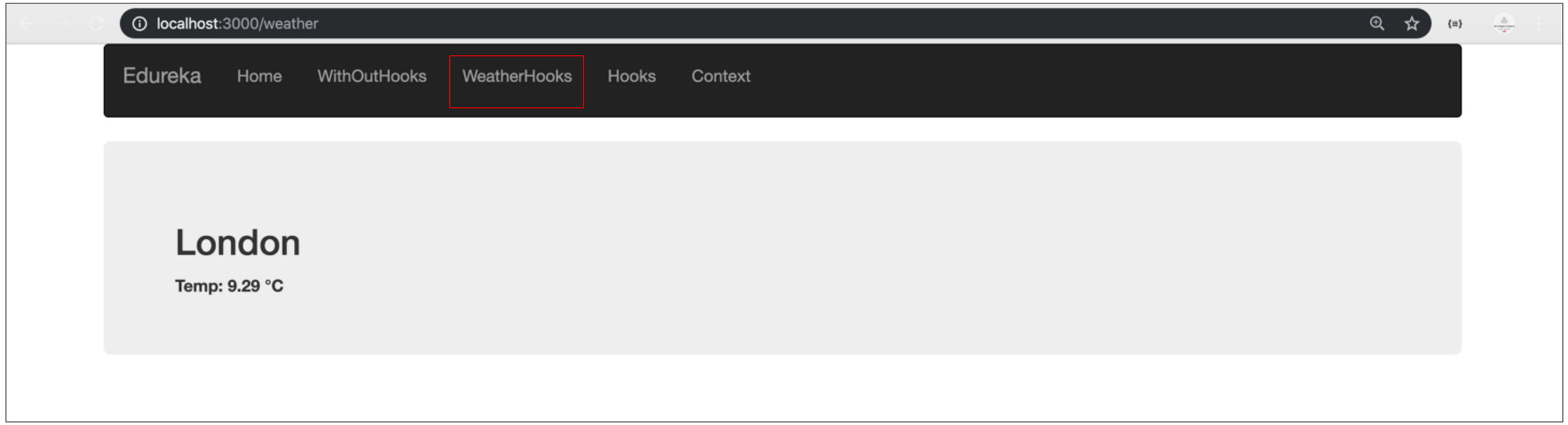
```
import React, { useState} from 'react';
import useTitle from './CustomHook.js'

function Title1() {
    const [record, setRecord] = useState(0)



return (
<div>
<button onClick={() => setRecord(record + 1)}>
 Record : {record}</button>
</div>
)
}
export default Title1
```

Title1.js



Demo 6 :Weather Application Using React Hooks

Demo Weather Application



Questions


Comments
Ratings
Suggestions
FEEDBACK
Survey
Likes
Ideas