Unit 3 Forms and Hooks in ReactJS

By: Prof.Khushbu Surati

React Forms:

- Forms are an integral part of any modern web application.
- It allows the users to interact with the application as well as gather information from the users.
- Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc.
- A form can contain text fields, buttons, checkbox, radio button, etc.
- **Creating Form**
- React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

React Forms:

- Controlled Component
- In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with setState() method.
- Controlled components have functions that govern the data passing into them on every onChange event, rather than grabbing the data only once, e.g., when you click a submit button. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.
- Controlled form components are defined with a value property. The value of controlled inputs is managed by React, user inputs will not have any direct influence on the rendered input. Instead, a change to the value property needs to reflect this change.

React Forms:

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

class Form extends Component {

  constructor(props) {
    super(props)

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

```
handleUsernameChange = (event) => {
  this.setState({
      username : event.target.value
  })
}

handleCommentChange = (event) => {
  this.setState({
      comment:event.target.value
  })
}
```

```
handleTopicChange = event => {
    this.setState({
        topic:event.target.value
    })
}

handleSubmit = (event) =>{
    alert(`${this.state.username} ${this.state.comment} ${this.state.topic}`)
    event.preventDefault()
}
```

React Forms

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
        <div>
            <label>User Name</label>
            <input
              type='text'
              value={this.state.username}
              onChange={this.handleUsernameChange}
            />
        </div> <br></br>

        <div>
            <label>Comments</label>
            <textarea
              value={this.state.comment}
              onChange={this.handleCommentChange}
            ></textarea>
        </div> <br></br>
        <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>
  )
 }
}
```

React Forms:

```
import './App.css';
import Greet from './components/Form'

function App() {
  return (
    <div className="App">
       <Form/>
    </div>
  );
}
export default App;
```

React Hooks

- Hooks are the new feature introduced in the React 16.8 version.
- It allows you to use state and other React features without writing a class.
- Hooks are the functions which "hook into" React state and lifecycle features from function components.
- It does not work inside classes.
- Hooks are backward-compatible, which means it does not contain any breaking changes.
- Hooks are simply function so we just have to call them.
- **When to use a Hooks**
- If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

React Hooks

- **Rules of Hooks**
- Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code.

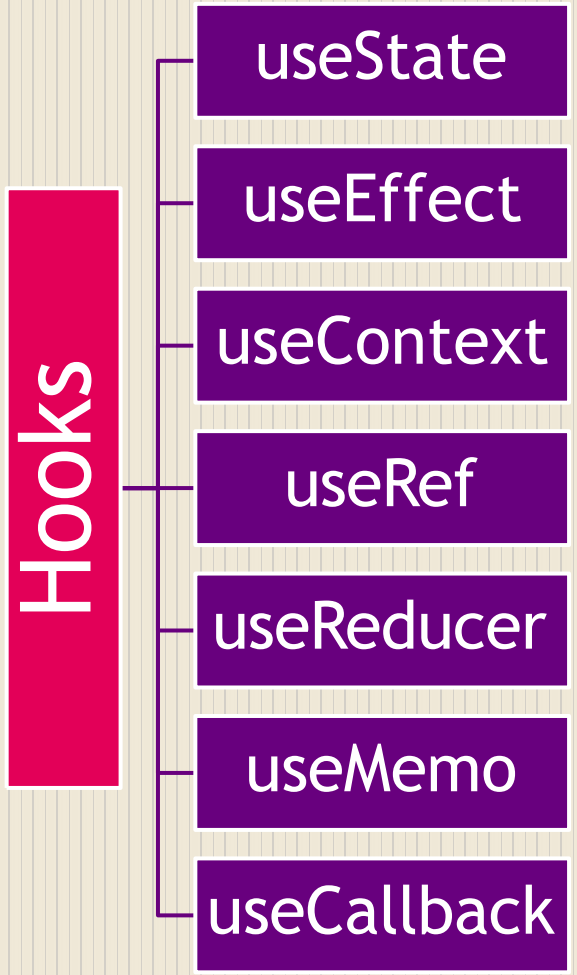
1. Only call Hooks at the top level

- Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions.

2. Only call Hooks from React functions

- You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

React Hooks


Hooks
useState
useEffect
useContext
useRef
useReducer
useMemo
useCallback

useState Hook

- The **useState()** is a Hook that allows you to have state variables in functional components.
- The useState is the ability to encapsulate local state in a functional component.
- The useState hook is a special function that takes the initial state as an argument and returns an array of two entries.
- UseState encapsulate only singular value from the state, for multiple state you need to have useState calls.
- Syntax:

  const [state, setState] = useState(initialstate)

- To use useState you need to import useState from react as shown below:
- import React, { useState } from "react"

useState Hook

HookCounter.js

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

function HookCounter() {
    const [count, setCount] =useState(0)

  return (
    <div>
        <button onClick={()=>setCount(count + 1)}>Count {count}</button>
    </div>
  )
}
export default HookCounter
```

useState Hook

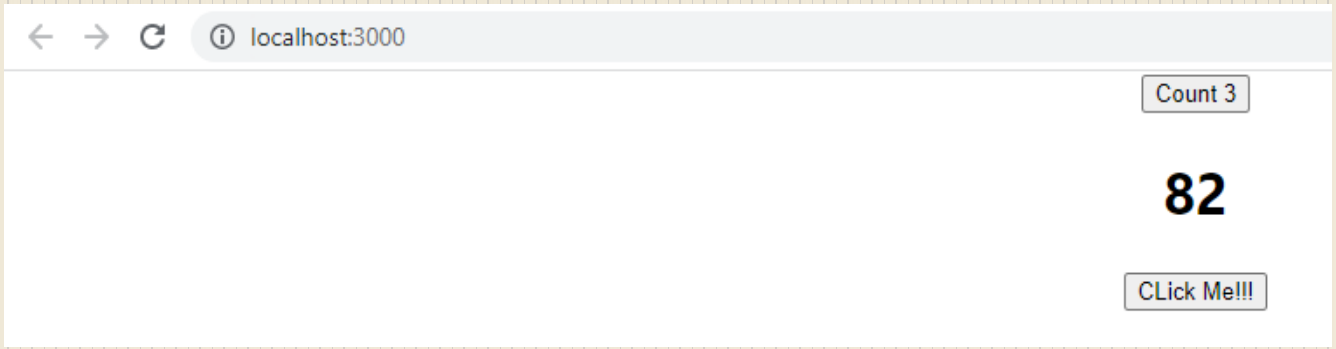
RandomNumber.js

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

function RandomNumber() {
    const [count, setRandomCount] =
        useState(function generateRandomInteger() {
            return Math.floor(Math.random() * 100);
        });

        function clickHandler(e) {
            setRandomCount(Math.floor(Math.random() * 100));
          }
    return (
        <div>
            <h1>{count}</h1>
            <button onClick={clickHandler}> CLick Me!!!</button>
        </div>
    )
}
export default RandomNumber
```

useState Hook

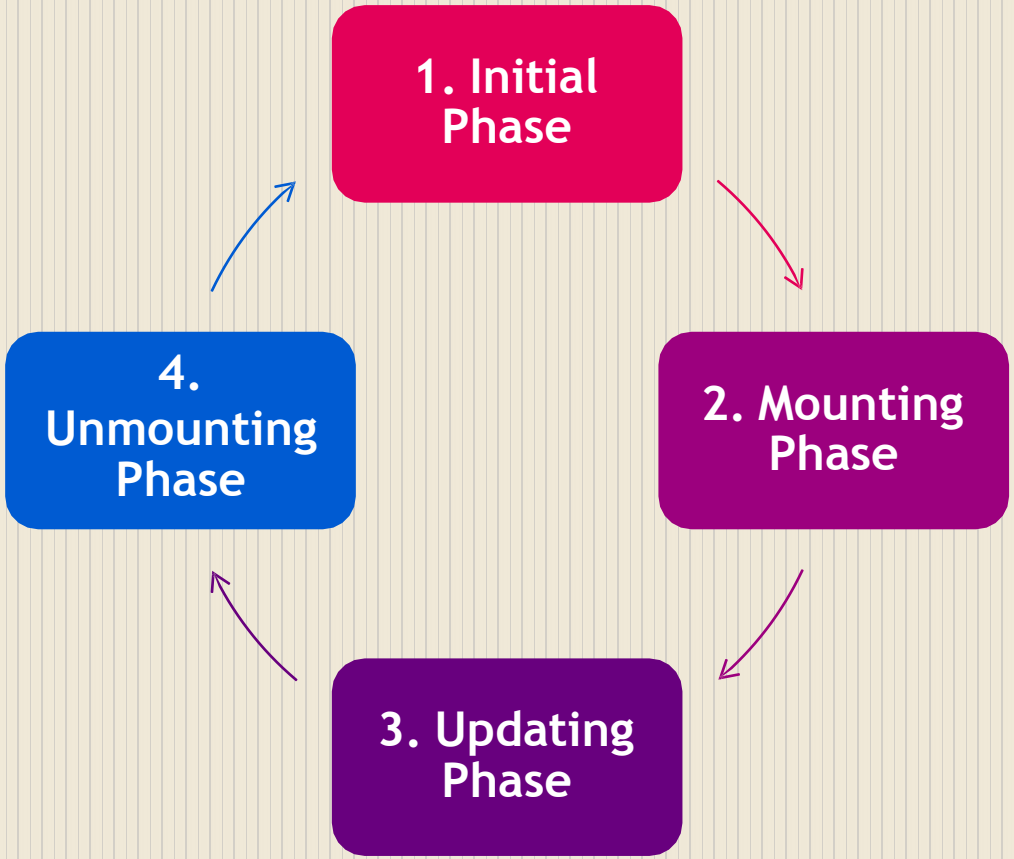
App.js

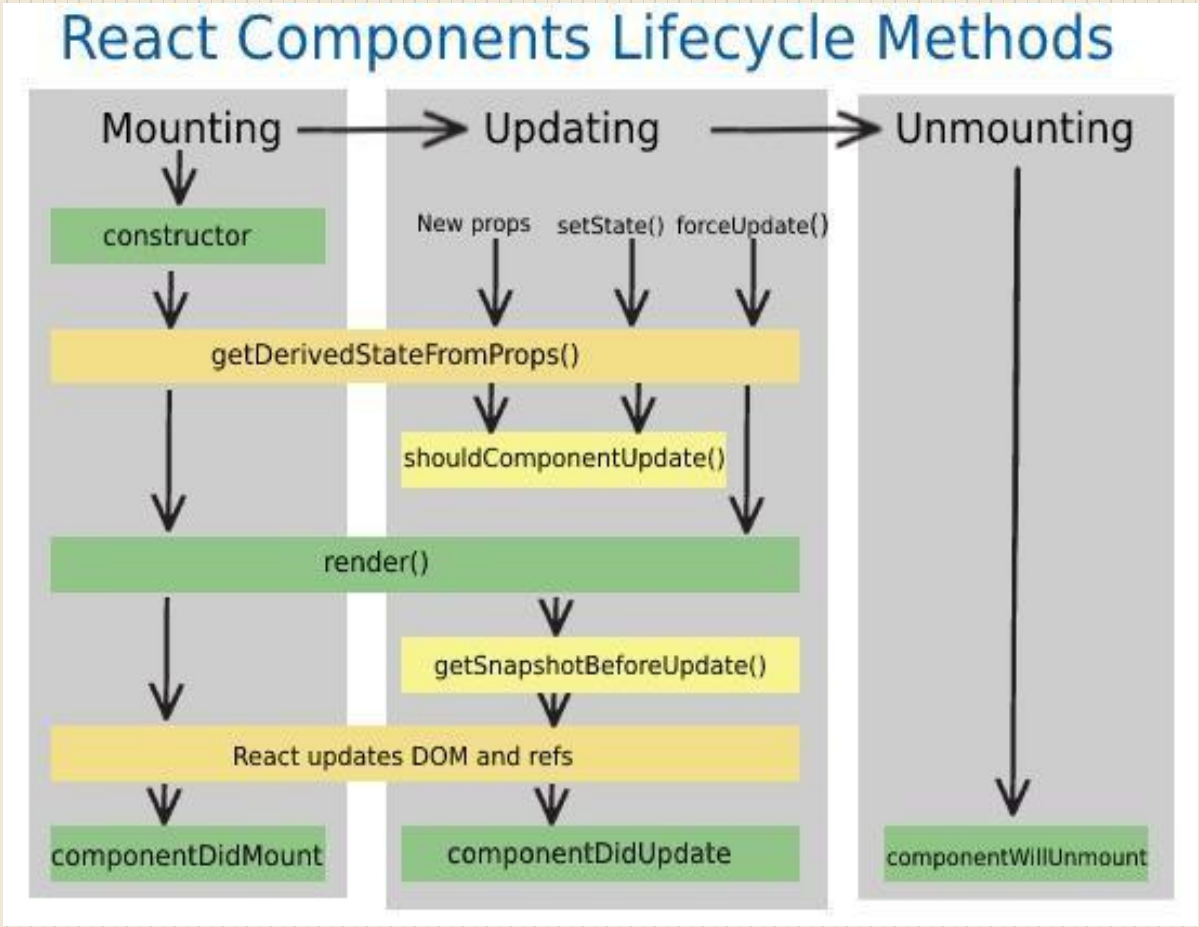
```
import './App.css';
import HookCounter from './components/HookCounter';
import RandomNumber from './components/RandomNumber';
function App() {
  return (
    <div className="App">
      <HookCounter />
      <RandomNumber/>
    </div>
  );
}
export default App;
```



Form Handling using - useState Hook

React Component Life-Cycle





useEffect Hook

- The motivation behind the introduction of useEffect Hook is to eliminate the side-effects of using class-based components.

```
componentDidMount(){
   document.title = `you clicked ${this.state.count} times`;
}

componentDidUpdate(){
   document.title = `you clicked ${this.state.count} times`;
}
```

- Another side-effect by setting up a timer.

```
componentDidMount(){
   this.interval = setInterval(this.tick, 1000)
}

componentWillUnmount(){
   clearInterval(this.interval)
}
```

useEffect Hook

- The useEffect() is used for causing side effects in functional components and it is also capable for handling componentDidMount(), componentDidUpdate() and componentWillUnmount() life-cycle methods of class based components into functional component.

- Just like useState, useEffect is also a function. We simple have to call it. The useEffect( ) accepts a parameter of type function and this function will be executed after every render of the component.
- useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

useEffect Hook

ClassCounterOne.js

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

class ClassCounterOne extends Component {
  constructor(props) {
    super(props)
    this.state = {
      count: 0
    }
  }

  componentDidMount() {
    document.title = `Clicked ${this.state.count} times`
  }

  componentDidUpdate(prevProps, prevState) {
    document.title = `Clicked ${this.state.count} times`
  }

  render() {
    return (
      <div>
        <button onClick={() => this.setState(
          { count: this.state.count + 1 })}>
          Click {this.state.count} times
        </button>
      </div>
    )
  }
}
export default ClassCounterOne
```

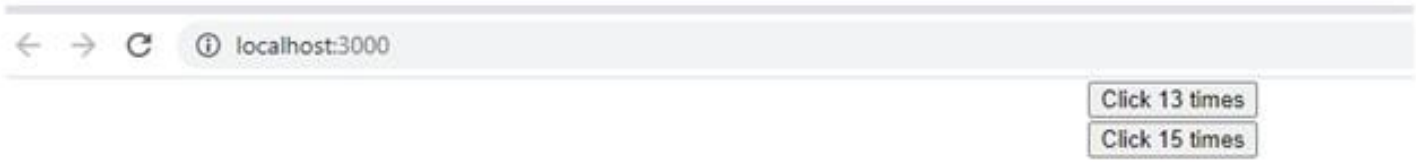
HookCounterOne.js

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

function HookCounterOne() {
  const [count, setCount] = useState(0)

  useEffect(() => {
    document.title = `You clicked ${count} times`
  })

  return (
    <div>
      <button onClick = {() => setCount(count + 1)}>
          Click {count} times </button>
    </div>
  )
}
export default HookCounterOne
```

useEffect Hook

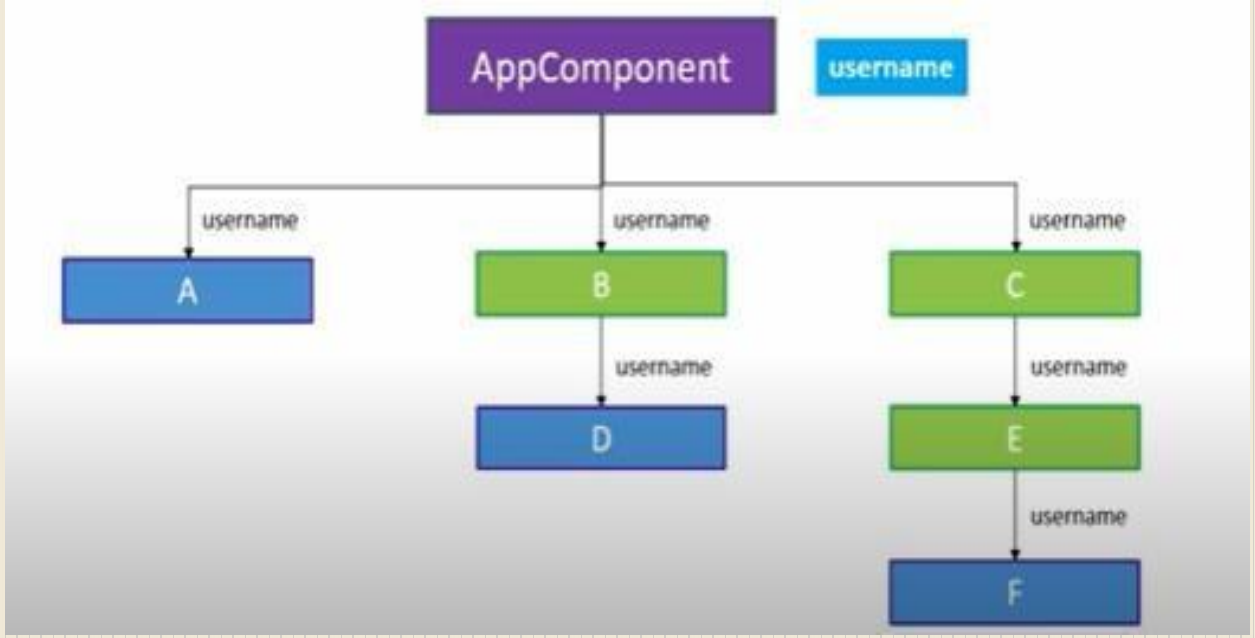
App.js

```
import './App.css';
import ClassCounterOne from './components/ClassCounterOne';
import HookCounterOne from './components/HookCounterOne';

function App() {
  return (
    <div className="App">
      <ClassCounterOne/>
      <HookCounterOne/>
    </div>
  );
}

export default App;
```

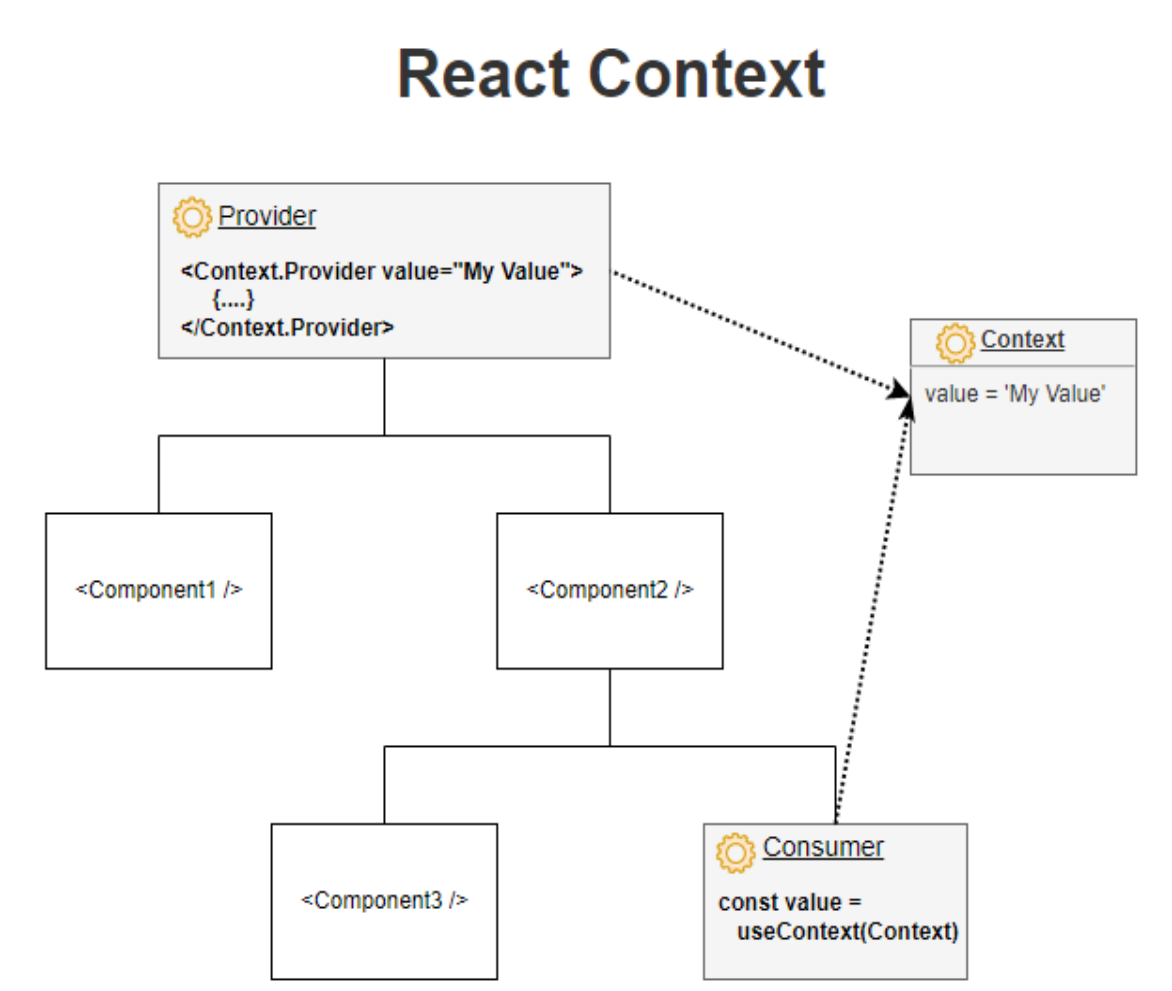


useContext Hook

The React Context provides a way to pass data or state through the component tree without having to pass props down manually through each nested component. It is designed to share data that can be considered as global data for a tree of React components, such as the current authenticated user or theme(e.g. color, paddings, margins, font-sizes).



useContext Hook (How to use the context)



useContext Hook (How to use the context)

A. Creating the context

- import { createContext } from 'react';
- const Context = createContext('Default Value');

B. Providing the context

- **Context.Provider** component available on the context instance is used to provide the context to its child components, no matter how deep they are.
- To set the value of context use the value prop available on the <Context.Provider value={value} />:

C. Consuming the context

useContext Hook (How to use the context)

```
import { createContext } from 'react';
const Context =
createContext('Default Value');
```

```
function Main() {
  const value = 'My Context Value';
return (
    <Context.Provider value={value}>
        <MyComponent />
    </Context.Provider>
);
}
```

```
import { useContext } from 'react';
function MyComponent() {
const value = useContext(Context);
return <span>{value}</span>;
}
```

```
function MyComponent() {
return (
<Context.Consumer>
{value => <span>{value}</span>}
</Context.Consumer>
);
}
```

useReducer Hook

- This hook is used for state management. It is an alternative to useState. useState is built using useReducer hook.
- **What is reducer?**
- The **reduce()** method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.
- reduce(callbackFn, initialValue)
- The callbackFn is a reducer function that accepts two parameeers and return single value.

useReducer Hook

- Perhaps the easiest-to-understand case for reduce() is to return the sum of all the elements in an array:

```
const array1 = [1, 2, 3, 4];
// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce(
  (previousValue, currentValue) => previousValue + currentValue,
  initialValue
);
console.log(sumWithInitial);
// expected output: 10
```

useReducer Hook

reduce in JS	useReducer in React
Array.reduce(reducer,initialValue)	useReducer(reducer,initialValue)
singleValue = reducer(accumulator, itemValue)	newState = reducer(currentState, action)
reduce method returns a single value	useReducer returns a pair of values. [newState, dispatch]

useReducer Hook

- **How to implement useReducer:**

1. Import useReducer from react as shown in the below snippet:

- import React, { useReducer } from 'react';

2. Call the useReducer( ) by following its syntax.

- const [state, dispatch] = useReducer(reducer, initialState);
- This hook function returns an array with 2 values. The first one is the state value, and the second value is the dispatch function which is further used to trigger an action with the help of array de-structuring.

useReducer Hook

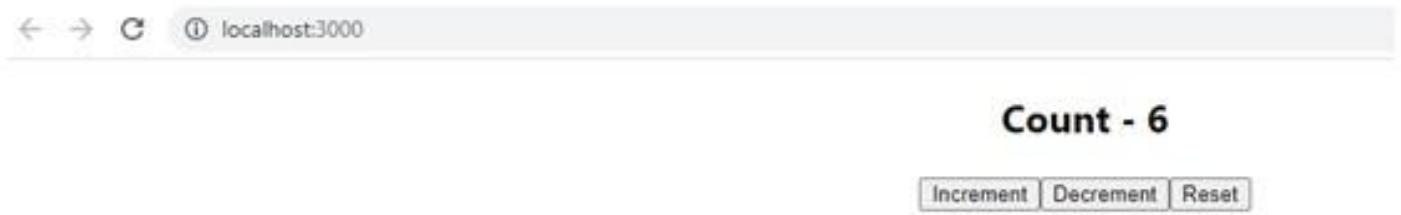
CounterOne.js

```
import React, {useReducer} from 'react'
const initialState = 0

const reducer = (state, action) =>{
    switch(action){
        case 'increment':
            return state + 1
        case 'decrement':
            return state - 1
        case 'reset':
            return initialState
        default:
            return state
    }

}

function CounterOne() {
    const [count, dispatch] = useReducer(reducer, initialState)

  return (
    <div>
        <h2>Count - {count}</h2>
        <button onClick={()=>dispatch('increment')}>Increment</button>
        <button onClick={()=>dispatch('decrement')}>Decrement</button>
        <button onClick={()=>dispatch('reset')}>Reset</button>
    </div>
  )
}

export default CounterOne
```

useReducer Hook

App.js

```
import './App.css';
import CounterOne from './Components/CounterOne';

function App() {
  return (
    <div className="App">
      <CounterOne />
    </div>
  );
}
export default App;
```



useCallback Hook

- The useCallback hook is used when you have a component in which the child is re-rendering again and again without need.
- Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders. So, basically it is used for performance optimization.

Syntax:

```
const memoizedCallback = useCallback(
 () => {
   doSomething(a, b);
 },
 [a, b],
);
```

- WithoutCallback.js & WithCallback.js

useMemo Hook

- This hook is also used for performance optimization like useCallback hook.
- The useMemo is a hook used in the functional component of react that returns a memoized value.
- The memoization is a concept used in general when we don't need to re- compute the function with a given argument for the next time as it returns the cached result.
- A memoized function remembers the results of output for a given set of inputs.
- For example, if there is a function to add two numbers, and we give the parameter as 1 and 2 for the first time the function will add these two numbers and return 3, but if the same inputs come again then we will return the cached value i.e 3 and not compute with the add function again.
- **Syntax:**

```
const memoizedValue = useMemo(functionThatReturnsValue,
                              arrayDepencies)
```

useRef Hook

- The useRef is a hook that allows to directly creating a reference to the DOM element in the functional component.
- **Syntax:**

  const refContainer = useRef(initialValue);

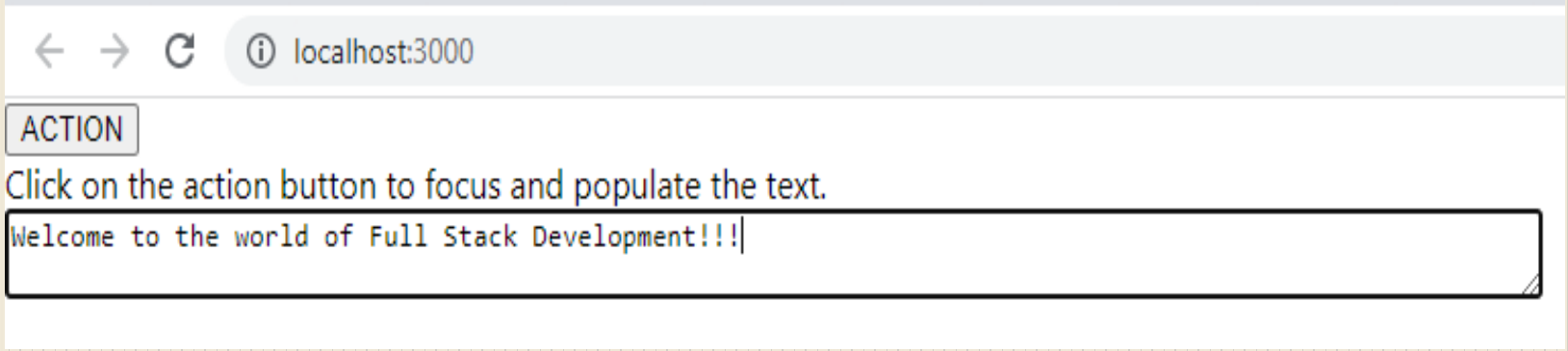
- The useRef returns a mutable ref object. This object has a property called .current. The value is persisted in the refContainer.current property. These values are accessed from the current property of the returned object. The .current property could be initialised to the passed argument initialValue e.g. useRef(initialValue). The object can persist a value for a full lifetime of the component.

useRef Hook

App.js

```
import React, {Fragment, useRef} from 'react';

function App() {

// Creating a ref object using useRef hook
const focusPoint = useRef(null);
const onClickHandler = () => {
  focusPoint.current.value =
  "Welcome to the world of Full Stack Development!!!";
  focusPoint.current.focus();
};
return (
  <Fragment>
  <div>
    <button onClick={onClickHandler}>
    ACTION
    </button>
  </div>
  <label>
  Click on the action button to
  focus and populate the text.
  </label><br/>
  <textarea ref={focusPoint} cols='100'/>

  </Fragment>
);
};

export default App;
```

useRef Hook


localhost:3000
ACTION
Click on the action button to focus and populate the text.
Welcome to the world of Full Stack Development!!!

Custom Hooks

- React.Js provides lots of built-in hooks that you can use in your React apps.
- But besides them, you can make your own custom hooks and use it in your apps resulting in better readability and a reduced amount of code.
- The main reason for which you should be using Custom hooks is to maintain the concept of DRY(Don't Repeat Yourself) in your React apps.
- For example, suppose you have some logic that makes use of some built-in hooks and you need to use the logic in multiple functional components.
- Creating a custom hook is the same as creating a JavaScript function whose name starts with "use". It can use other hooks inside it, return anything you want it to return, take anything as parameters.

Custom Hooks

useDocumentTitle.js

```
import { useEffect } from 'react'
function useDocumentTitle(count) {
  useEffect(() => {
    document.title = `Count ${count}`
  }, [count])
}
export default useDocumentTitle
```

DocTitleOne.js

```
import React,{useState,useEffect} from 'react'
import useDocumentTitle from './useDocumentTitle'

function DocTitleOne() {
    const [count, setCount] = useState(0)

    // useEffect(()=>{
    //     document.title=`Count ${count}`
    // },[count])
  useDocumentTitle(count)
  return (
    <div>
        <button onClick={()=> setCount(count+1)}>Count - {count}</button>
    </div>
  )
}
export default DocTitleOne
```

DocTitleTwo.js

```
import React,{useState,useEffect} from 'react'
import useDocumentTitle from './useDocumentTitle'

function DocTitleTwo() {
    const [count, setCount] = useState(0)

    // useEffect(()=>{
    //     document.title=`Count ${count}`
    // },[count])
    useDocumentTitle(count)
  return (
    <div>
        <button onClick={()=> setCount(count+1)}>Count - {count}</button>
    </div>
  )
}
export default DocTitleTwo
```

Custom Hooks

App.js

```
import logo from './logo.svg';
import './App.css';
import DocTitleOne from './Components/DocTitleOne';
import DocTitleTwo from './Components/DocTitleTwo';

function App() {
  return (
    <div className="App">
      <DocTitleOne/>
      <DocTitleTwo/>
    </div>
  );
}

export default App;
```