Unit-3

Unit-3:	***Forms and Hooks in React.JS***
3.1	***Forms: (Adding forms, Handling forms, Submitting forms)***
3.1.1	***event.target.name and event. Target.event, React Memo***
3.1.2	***Components (TextArea, Drop down list (SELECT))***
3.2	***Hooks: Concepts and Advantages***
3.2.1	***useState, useEffect, useContext***
3.2.2	***useRef, useReducer, useCallback, useMemo***
3.2.3	***Hook: Building custom hook, advantages and use***

Component Constructor

If there is a constructor() function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called state.

The constructor function is also where you have the inheritance of the parent component by including the super() statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (React.Component).

Example

Create a constructor function in the Car component, and add a color property:

Use the color property in the render() function:

```
class Furniture extends React.Component {
  constructor() {
    super();
    this.state = {material: "wood"};
  }
  render() {
    return <h2>the furniture is made of {this.state.material } material!</h2>;
  }
}
export default Furniture
```

Changing the state Object

To change a value in the state object, use the this.setState() method.

When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an onClick event that will change the color property:

```
class Car extends React.Component {
```

```
 constructor(props) {
  super(props);
  this.state = {
    brand: "Ford", model: "Mustang", color: "red", year: 1964
  };
 }
 changeColor = () => {
  this.setState({color: "blue"});
 }
 render() {
  return (
    <div>
      <h1>My {this.state.brand}</h1>
      <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}.
      </p>
      <button
        type="button"
        onClick={this.changeColor}
      >Change color</button>
    </div>
  );
 }
}export default  Car;
```

Difference state and props

Props	state
Props get passed to the component	State is managed within the component
Function parameters	Variables declared in the function body
Props are immutable	State can be changed
Props-functional Components this.props – Class Components	useState Hook-Functional Components this.state-Class Components

3.2 React Hooks

Hooks were added to React in version 16.8.
Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.
What is a Hook?
Hooks allow us to "hook" into React features such as state and lifecycle methods.

- You must import Hooks from react.
- Here we are using the useState Hook to keep track of the application state.
- State generally refers to application data or properties that need to be tracked.

Why the need for Hooks?
There are multiple reasons responsible for the introduction of the Hooks which may vary depending upon the experience of developers in developing React product. Some of them are as follows:

- **Use of *'this'* keyword:** The first reason has to do more with javascript than with React itself. To work with classes one needs to understand how 'this' keyword works in javascript which is very different from how it works in other languages. It is easier to understand the concept of props, state, and uni-directional data flow but using 'this' keyword might lead to struggle while implementing class components. One also needs to bind event handlers to the class components. It is also observed by the React developers team also observed that classes don't concise efficiently which leads to hot reloading(To keep the app running and to inject new versions of the files that you edited at runtime.) being unreliable which can be solved using Hooks.

- **Reusable stateful logics:** This reason touches advance topics in React such as Higher-order components(HOC A higher-order component (HOC) is **an advanced technique in React for reusing component logic**) and the render props pattern. There is no particular way to reuse stateful component logic (Stateful logic is any code that uses the state)to React. Though this problem can be solved by the use of HOC and render props patterns it results in making the code base inefficient which becomes hard to follow as one ends up wrapping components in several other components to share the functionality. Hooks let us share stateful logic in a much better and cleaner way without changing the component hierarchy.

  Note:
  Lifecycle of component
  Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.
  The three phases are: **Mounting**, **Updating**, and **Unmounting**.

- **Simplifying complex scenarios:** While creating components for complex scenarios such as data fetching and subscribing to events it is likely that all related code is not organized in one place are scattered among different life cycle methods.
  For example, actions like data, fetching are usually done in componentDidMount(The componentDidMount() method is called after the component is rendered.This is where you run statements that requires that the component is already placed in the DOM) or componentDidUpdate(The componentDidUpdate method is called after the component is updated in the DOM.), similarly, in case of event listeners, it is done in componentDidMount or componentWillUnmount (The componentWillUnmount method is called when the component is about to be removed from the DOM). These develop a scenario where completely different codes like data fetching and event listeners end up in the same code-block. This also makes impossible to brake components to smaller components because of stateful logic. Hooks solve these problems by rather than forcing a split based on life-cycle method Hooks to let you split one component into smaller functions based on what pieces are related.

Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

3.2.1useState Hook

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

Import useState

To use the useState Hook, we first need to import it into our component.

```
import { useState } from "react";
```

Initialize useState

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

```
import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}
```

Notice that again, we are destructuring the returned values from useState.

The first value, color, is our current state.

The second value, setColor, is the function that is used to update our state.

Commented [Ma1]: Destructuring Assignment is a JavaScript expression that allows to **unpack values** from arrays, or properties from objects, into distinct variables data can be extracted from *arrays, objects, nested objects* and **assigning to variables**. Eg:

```
<script>
var names = ["alpha", "beta", "gamma", "delta"];

var firstName = names[0];
var secondName = names[1];

console.log(firstName);//"alpha"
console.log(secondName);//"beta"
</script>/
```

Read State

We can now include our state anywhere in our component.

```
import { useState } from "react";
function FavoriteColor() {
  const [color, setColor] = useState("red");
  return <h1>My favorite color is {color}!</h1>
}export default FavoriteColor;
```

Update State

To update our state, we use our state updater function.

```
import { useState } from "react";
function FavoriteColor() {
  const [color, setColor] = useState("red");
 return (
   <>
     <h1>My favorite color is {color}!</h1>
     <button
       type="button"
       onClick={() => setColor("blue")}
     >Blue</button>
   </>
 )
}
export default FavoriteColor;
```

Updating Objects and Arrays in State

When state is updated, the entire state gets overwritten.

What if we only want to update the color of our car?

If we only called setCar({color: "blue"}), this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to overcome the above example.

Spread Operator

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Example

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
```

Example:

Use the JavaScript spread operator to update only the color of the car:

```
import { useState } from "react";
function Car() {
  const [car, setCar] = useState({
    brand: "Ford", model: "Mustang", year: "1964", color: "red"
  });
  const updateColor = () => {
    setCar(previousState => {
      return { ...previousState, color: "blue" }
    });
  }
  return (
    <>
      <h1>My {car.brand}</h1>
```

```
    <p>
      It is a {car.color} {car.model} from {car.year}.
    </p>
    <button
      type="button"
      onClick={updateColor}
    >Blue</button>
   </>
 )
}
export default Car;
```

React useEffect Hooks

The useEffect Hook allows you to perform side effects in your components.

Commented [Ma2]: Anything that affects something outside of the scope of the current function that's being executed.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

Let's use a timer as an example.

Example:

Use setTimeout() to count 1 second after initial render:

```
import { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);
```

```
 useEffect(() => {
  setTimeout(() => {
   setCount((count) => count + 1);
  }, 1000);
 });

 return <h1>I've rendered {count} times!</h1>;
}
export default Timer;
```

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

Here is an example of a useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again:

```
import React,{ useState, useEffect } from "react";
function Counter() {
 const [count, setCount] = useState(0);
 const [calculation, setCalculation] = useState(0);

 useEffect(() => {
  setCalculation(() => count * 2);
 }, [count]); // <- add the count variable here

 return (
  <>
   <p>Count: {count}</p>
   <button onClick={() => setCount((c) => c + 1)}>+</button>
```

```
    <p>Calculation: {calculation}</p>
  </>
 )export default Counter;
```

Example of useEffect to update the title with increment of count variable

```
import React,{useState,useEffect}from 'react'
const UseEffects1 = () => {
   const [count,setCount]=useState(0);
   useEffect(()=>{document.title =`Counts (${count})`
   //useEffect(()=>{console.log("Inside useEffects");
});
console.log("UseEffects Outside");
 return (
   <div>
   <h1>{count}</h1>
   <button class="btn" onClick={()=>setCount(count+1)}>Click </button>
   </div>
 )
}export default UseEffects1;
```

React Context hook

React Context is a way to manage state globally.

It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

The Problem is when State is held by the highest parent component in the stack that requires access to the state at the lowest child component.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

Example:

Passing "props" through nested components:

```
import { useState } from "react";
function Component1() {
  const [user, setUser] = useState("Learning React Hooks");
  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
```

```
      <h1>Component 3</h1>
      <Component4 user={user} />
    </>
  );
}

function Component4({ user }) {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 user={user} />
    </>
  );
}

function Component5({ user }) {
  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
export default Component1 ;
```

Solution:

The Solution

The solution is to create context.

Create Context

To create context, you must Import createContext and initialize it:

```
import { useState, createContext } from "react";

const UserContext = createContext()
```

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");


  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}
```

Now, all components in this tree will have access to the user Context.

Use the useContext Hook

In order to use the Context in a child component, we need to access it using the useContext Hook.

First, include the useContext in the import statement:

```
import { useState, createContext, useContext } from "react";
```

Then you can access the user Context in all components:

```
function Component5() {
  const user = useContext(UserContext);
  return (
    <>  <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

Full Example:

Here is the full example using React Context:

```
import { useState, createContext, useContext } from "react";
const UserContext = createContext();
function Component1() {
  const [user, setUser] = useState("Jesse Hall");
          return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
```

```
    </>
  );
}

function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}

function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
export default Component1;
```

useCallback Hook

The useCallback hook is used when you have a component in which the child is rerendering 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.

Syntax:

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

Without useCallback Hook: The problem is that once the counter is updated, all three functions are recreated again. The alert increases by three at a time but if we update some states all the functions related to that states should only re-instantiated. If another state value is unchanged, it should not be touched. Here, the filename is App.js**With useCallback hook:** To solve this problem we can use the useCallback hook. Here, the filename is App.js.

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

var funccount = new Set();
const App = () => {
  const [count, setCount] = useState(0)
  const [number, setNumber] = useState(0)
const incrementCounter = useCallback(() => {setCount(count + 1)}, [count])
const decrementCounter = useCallback(() => {setCount(count - 1)}, [count])
const incrementNumber = useCallback(() => {setNumber(number + 1)},
[number])
funccount.add(incrementCounter);
```

Commented [Ma3]: **Set() constructor**
The **`Set` constructor** lets you create `Set` objects that store unique values of any type, whether primitive values or object references.
Eg: `let mySet = new Set()`

```
mySet.add(1)            // Set [ 1 ]
mySet.add(5)            // Set [ 1, 5 ]
mySet.add(5)            // Set [ 1, 5 ]
mySet.add('some text') // Set [ 1, 5,
'some text' ]
let o = {a: 1, b: 2}
mySet.add(o)
```

```
funccount.add(decrementCounter);
funccount.add(incrementNumber);
alert(funccount.size);
 return (
   <div>
     Count: {count}
     <button onClick={incrementCounter}> Increase counter </button>
     <button onClick={decrementCounter}>Decrease Counter </button>
     <button onClick={incrementNumber}>Increase number  </button>
   </div>
 )
}
export default App;
```

Output: As we can see from the below output when we change the state 'count' then two functions will re-instantiated so the set size will increase by 2 and when we update the state 'number' then only one function will re-instantiated and the size of the set will increase by only one.

React useMemo Hook

The useMemo is a hook used in the functional component of react that returns a memoized value. In Computer Science, memoization is a concept used in general when we don't need to recompute 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. In react also, we use this concept, whenever in the React component, the state and props do not change the component and the component does not re-render, it shows the same output. The useMemo hook is used to improve performance in our React application.

Syntax:

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

Example:

In an example, we can see that even if we changed the input number once, but clicked on-increment counter multiple times our function squareNum got executed whenever we clicked the increment counter button multiple times. This is happening because the App component re-renders whenever we change the state of the counter.

Now let's solve this problem using the useMemo hook.

When we use useMemo Hook

```
import React, {useState} from 'react';
  function App() {
  const [number, setNumber] = useState(0)
  // Using useMemo
  const squaredNum = useMemo(()=> {
    return squareNum(number);
  }, [number])
  const [counter, setCounter] = useState(0);

 // Change the state to the input
  const onChangeHandler = (e) => {
    setNumber(e.target.value);
  }

  // Increases the counter by 1
  const counterHander = () => {
    setCounter(counter + 1);
  }
  return (
    <div className="App">
      <h1>Learning Usememo hook</h1>
      <input type="number" placeholder="Enter a number"
          value={number} onChange={onChangeHandler}>
      </input>

      <div>OUTPUT: {squaredNum}</div>
      <button onClick= {counterHander}>Counter ++</button>
```

```
      <div>Counter : {counter}</div>
    </div>
  );
}

// function to square the value
function squareNum(number){
  console.log("Squaring will be done!");
  return Math.pow(number, 2);
}

export default App;
```

Output: Now in the above example, we have used the user memo hook, here the function that returns the value i.e squareNum is passed inside the useMemo and inside the array dependencies, we have used the number as the squareNum will run only when the number changes. If we increase the counter and the number remains the same in the input field the squareNum doesn't run again.

ReactJS useReducer Hook

The **useReducer** Hook is the better alternative to the **useState** hook and is generally more preferred over the **useState** hook when you have complex state-building logic or when the next state value depends upon its previous value or when the components are needed to be optimized.

The **useReducer** hook takes three arguments including reducer, initial state, and the function to load the initial state lazily(Sometimes instead of passing a primitive value, an object or an array as argument, you can also pass a function. The value returned by the function passed is used for initializing state.That is referred to as **lazy state initialization**. **Lazy state initialization** is necessary if you are performing a computationally expensive process for initializing state.).

Syntax:
const [state, dispatch] = useReducer(reducer, initialArgs, init);

Example: Here reducer is the user-defined function that pairs the current state with the dispatch method to handle the state, initialArgs refers to the initial arguments and init is the function to initialize the state lazily.
App.js: Program to demonstrate the use of useReducer Hook:

```
import React, { useReducer } from "react";

// Defining the initial state and the reducer
const initialState = 0;
const reducer = (state, action) => {
  switch (action) {
    case "add":
      return state + 1;
    case "subtract":
      return state - 1;
    case "reset":
      return 0;
    default:
      throw new Error("Unexpected action");
  }
};

const App = () => {
    // Initialising useReducer hook
  const [count, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={() => dispatch("add")}>
        add
      </button>
      <button onClick={() => dispatch("subtract")}>
        subtract
      </button>
```

```
      <button onClick={() => dispatch("reset")}>
        reset
      </button>
    </div>
  );
};
export default App;
```

React JS useRef Hook

The useRef hook is the new addition in React 16.8.
The useRef is a hook that allows to directly create 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.

Example: How to access the DOM using useRef hook.

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

function App() {

  // Creating a ref object using useRef hook
  const focusPoint = useRef(null);
```

```
  const onClickHandler = () => {
    focusPoint.current.value =
      "The quick brown fox jumps over the lazy dog";
      focusPoint.current.focus();
  };
  return (
    <>
      <div>
        <button onClick={onClickHandler}>
         ACTION
        </button>
      </div>
      <label>
       Click on the action button to
       focus and populate the text.
      </label><br/>
      <textarea ref={focusPoint} />
    </>
  );
};

export default App;
```

Output: In this example, we have a button called ACTION, whenever we click on the button the onClickHandler is getting triggered and it's focusing the textarea with help of useRef hook. The focusPoint is the useRef object which is initialised to null and the value is changing to onClick event.

Hook: Building custom hook, advantages and use

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use". Example **useTitleCount.jxs**

App.jsx

```
import Test from './Test'

import './App.css';

function App() {

  return (

    <div className="App">

      <Test/>

     </div>

  );

}export default App;
```

Text.jsx

```
import React,{useState}from 'react'
import useTitleCount from './useTitleCount'
const UseEffects1 = () => {
    const [count,setCount]=useState(0);
   useTitleCount(count);
console.log("UseEffects Outside");
  return (
<div>
    <h1>{count}</h1>
    <button class="btn" onClick={()=>setCount(count+1)}>Click </button>
    </div>
  )
}

export default UseEffects1
```

useTitleCount.jxs

```
import {useEffect} from "react";

const useTitleCount=(count) => { useEffect(()=>{

    if (count>=1){document.title =`Counts (${count})`

  //useEffect(()=>{console.log("Inside useEffects");

}

else {document.title =`Counts`}},[count]);}

export default useTitleCount;
```

3.1 Forms: (Adding forms, Handling forms, Submitting forms)

Adding Forms in React

You add a form with React like any other element:

Example:

Add a form that allows users to enter their name: MyForm.jsx

```
function MyForm() {
 return (
  <form>
   <label>Enter your name:
    <input type="text" />
   </label>
```

```
  </form>)
}
export default MyForm;
```

Handling Forms

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the onChange attribute.

We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

```
import { useState } from 'react';
function MyForm() {
 const [name, setName] = useState("");
 return (
  <form>
   <label>Enter your name:
    <input
     type="text"
     value={name}
     onChange={(e) => setName(e.target.value)}
    />
   </label>
  </form>
 )
```

}export default MyForm;

3.1.1 event.target.name and event. Target.event, React Memo

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

function Form() {
   const [formData, setFormData] = useState({
   name:"",
   email:""
});


     const changeHandler = (event) => {
       setFormData({
...formData,
[event.target.name]:event.target.value
     })
}
   const handleSubmit = (event) => {
      alert('Recieved Data');
     }
 return (
   <form onSubmit={handleSubmit}>
      <div>
      <label>Name</label>
      <input type="text" value={formData.name}
      onChange={changeHandler}
```

```
        placeholder="your name plz" name="name" />
        </div>
        <div>
        <label>Email</label>
        <input type="text" value={formData.email}
        onChange={changeHandler}
        placeholder="your email plz" name="email" />

        </div>

<div>
   <button type="submit" >submit</button>
</div>

   </form>
 )
}export default Form;
```

React Memo

Using memo will cause React to skip rendering a component if its props have not changed.

This can improve performance

Problem

In this example, the Todos component re-renders even when the todos have not changed.

Example:

App.js:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };

  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};
export default App;
Todos.js:
const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
```

```
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>
  );
};
export default Todos;
```

When you click the increment button, the Todos component re-renders.

If this component was complex, it could cause performance issues.

Solution

To fix this, we can use memo.

Use memoto keep the Todos component from needlessly re-rendering.

Wrap the Todos component export in memo:

Example:
index.js:

```
import { useState } from "react";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };
```

```
 return (
  <>
   <Todos todos={todos} />
   <hr />
   <div>
    Count: {count}
    <button onClick={increment}>+</button>
   </div>
  </>
 );
};
export default App;
```

Todos.js:

```
import { memo } from "react";

const Todos = ({ todos }) => {
 console.log("child render");
 return (
  <>
   <h2>My Todos</h2>
   {todos.map((todo, index) => {
    return <p key={index}>{todo}</p>;
   })}
  </>
 );
};

export default memo(Todos);
```

3.1.2 Components (TextArea, Drop down list (SELECT))

Forms using Components (TextArea, Drop down list (SELECT))

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

function Myform() {
   const [username, setUsername] = useState("");
   const [mySubject, setSubject] = useState("React");
   const [textarea, setTextarea] = useState(
      "Enter the description here"
    );
    const handleChange = (event) => {
      setTextarea(event.target.value)
    }

 const handleSubjectChange = (event) => {
  setSubject(event.target.value)
 }
   const handleUsernameChange = (event) => {
      setUsername(event.target.value);
    }
    const handleSubmit = (event) => {
      alert(`${username} ${textarea} ${mySubject}`);
    }
 return (
  <form onSubmit={handleSubmit}>
```

```
      <div>
      <label>Username</label>
      <input type="text" value={username}
      onChange={handleUsernameChange}/>
      </div>
      <div>
      <label>Description</label>
      <textarea value={textarea} onChange={handleChange} />
     </div>
     <div>
      <label>Subject</label>
      <select value={mySubject} onChange={handleSubjectChange}>
      <option value="React">React</option>
      <option value="AngularJS">AngularJS</option>
      <option value="Vue">Vue</option>
    </select>
</div>
<div>
   <button type="submit" >submit</button>
</div>

   </form>
 )
}
```

export default Myform;