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 Hooks: Buliding custom Hook, Advantages and use

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))

Adding Forms

- You add a form with React like any other element:

Example

```
function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>
  )
}
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

- This will work as normal, the form will submit and the page will refresh.
- But this is generally not what we want to happen in React.
- We want to prevent this default behavior and let React control the form.

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.

Example:

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

function Statedemo()
{
    const[count,setCount]=useState(0);
    const IncNum=()=>{
        setCount(count+1);
        console.log("Clicked")
    }

    return(
        <>
        <h1>{count}</h1>
        <button onClick={IncNum}>Click</button>
        </>
    )
}
export default Statedemo;
```

Submitting Forms

- You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

Example:

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

function FormSubmit() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}
export default FormSubmit;
```

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

Multiple Input Fields

- You can control the values of more than one input field by adding a name attribute to each element.
- We will initialize state with an empty object.
- To access the fields in the event handler use the event.target.name and event.target.value syntax.
- To update the state, use square brackets [bracket notation] around the property name.

Example:

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

export default function EventsTarget() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))
  }

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(inputs);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
      <input
        type="text"
        name="username"
        value={inputs.username || ""}
        onChange={handleChange}
      />
      </label>
      <label>Enter your age:
        <input
          type="number"
          name="age"
          value={inputs.age || ""}
          onChange={handleChange}
        />
        </label>
        <input type="submit" />
    </form>
  )
}
```

React Memo

- Using memo will cause React to skip rendering a component if its props have not changed.
- memo() is a great tool to memoize functional components. When applied correctly, it prevents useless re-renderings when the next props equal to previous ones. Take precautions when memoizing components that use props as callbacks.

MainDemo.js

```
import { useState } from "react";
import MemoDemo from './MemoDemo.js'
function MainDemo()
{
    const[count,setCount]=useState(0);
    const[data,setData]=useState(100);
    return(
        <div>Demo {count}
            <MemoDemo data={data}/>
            <button onClick={()=>setCount(count+1)}>Click Me</button>
        </div>
    );
}
export default MainDemo;
```

MemoDemo.js

```
import { memo } from "react";
function MemoDemo(props)
{
    console.log(props.data)
    return(
        <div>
            <h1>TYBCA</h1>
        </div>
    );
}
export default memo(MemoDemo);
```

3.1.2 Components (TextArea, Drop Down List (Select))

Textarea

- The textarea element in React is slightly different from ordinary HTML.

- In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>.

Example : HTML

```
<textarea>
        Content of the textarea.
</textarea>
```

Example: React

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

function TextArea() {
  const [textarea, setTextarea] = useState(
    "This is Just a Demo text"
  );

  const handleChange = (event) => {
    setTextarea(event.target.value)
  }

  return (
    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}
export default TextArea;
```

Example: HTML

```
<select>
  <option value="Oreo Silk">Oreo Silk</option>
  <option value="Kitkat" selected>Kitkat</option>
  <option value="5 Star">5 Star</option>
</select>
```

Example : React

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

export default function Cmb() {
  const [myChocolate, setMyChocolate] = useState("Oreo Silk");

  const handleChange = (event) => {
    setMyChocolate(event.target.value)
  }

  return (
    <form>
      <select value={myChocolate} onChange={handleChange}>
        <option value="Oreo Silk">Oreo Silk</option>
                <option value="Kitkat" selected>Kitkat</option>
                <option value="5 Star">5 Star</option>
      </select>
    </form>
  )
}
```

3.2 Hooks : Concepts and Advantages

3.2.1 useState, useEffect, useContext

3.2.2 useRef, useReducer, useCallback, useMemo

3.2.3 Hooks: Buliding custom Hook, Advantages and use

Hooks

- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- 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.

Hook Rules

- 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

- **Note:** Hooks will not work in React class components.

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

function Statedemo()
{
    const state=useState();
    const[count,setCount]=useState(0);
    const IncNum=()=>{
        setCount(count+1);
        console.log("Clicked")
    }

    return(
        <>
        <h1>{count}</h1>
        <button onClick={IncNum}>Click</button>
        </>
    )
}
export default Statedemo;
```

Example:

```
import React, { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
      <button
        type="button"
        onClick={() => setColor("red")}
      >Red</button>
      <button
        type="button"
        onClick={() => setColor("pink")}
      >Pink</button>
      <button
        type="button"
        onClick={() => setColor("green")}
      >Green</button>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

- 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.

3.2.1 useState, useEffect, useContext

useState()

- 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";
- useState accepts an initial state and returns two values:
  - The current state.
  - A function that updates the state.

Example:

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

function Statedemo()
{
    const state=useState();
    const[count,setCount]=useState(0);
    const IncNum=()=>{
        setCount(count+1);
        console.log("Clicked")
    }

    return(
        <>
        <h1>{count}</h1>
        <button onClick={IncNum}>Click</button>
        </>
    )
}
export default Statedemo;
```

useEffect()

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

Example:

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

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

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

  return <h1>Timer Started :  {count} times!</h1>;
}

export default UseEffect;
```

useContext()

- 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.

Example: (Problem)

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

function Component1() {
  const [user, setUser] = useState("20BCA");

  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;
```

Example : (Solution)

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

const UserContext = createContext();

function Components1() {
  const [user, setUser] = useState("20BCA");

  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 Components1;
```

3.2.2 useRef, useReducer, useCallback, useMemo

useRef()

- The useRef Hook allows you to persist values between renders.
- It can be used to store a mutable(value can be change) value that does not cause a re-render when updated.
- It can be used to access a DOM element directly.

Example:

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

function UseRef() {
  const inputElement = useRef();

  const focusInput = () => {
    inputElement.current.focus();
  };

  return (
    <>
      <input type="text" ref={inputElement} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
export default UseRef;
```

useReducer()

- The useReducer Hook is similar to the useState Hook.
- It allows for custom state logic.
- If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

Example:

- useReducer(<reducer>, <initialState>)

- const [val, setVal] = useReducer(reducer, initialTodos);

useCallback()

- The React useCallback Hook returns a memoized callback function.
- Think of memorization as caching a value so that it does not need to be recalculated.
- The useCallback Hook only runs when one of its dependencies update.
- This can improve performance.

useMemo()

- The React useMemo Hook returns a memoized value.
- The useMemo Hook only runs when one of its dependencies update.

Example:

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

const UseMemo = () => {
  const [count, setCount] =
useState(0);
  const [todos, setTodos] =
useState([]);
  const calculation = useMemo(()
=> expensiveCalculation(count),
[count]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New
Todo"]);
  };
  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => {
          return <p key={index}>{todo}</p>;
        })}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
        <h2>Expensive Calculation</h2>
        {calculation}
      </div>
    </div>
  );
};

const expensiveCalculation = (num) => {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {
    num += 1;
  }
  return num;
};
export default UseMemo;
```

3.2.3 Hooks: Buliding custom Hook, Advantages and use

Build a Hook

- 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: useFetch.

Advantages and Use

- Improving Readability of Component Tree.
- Encapsulating Side Effects.
- Composable and Reusable Logic.

Example: useFetch.js

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

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

Example : index.js

```
import ReactDOM from "react-dom/client";
import useFetch from "./useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};
```