React Hooks

Kavya Srinivasa

React Hooks

- React Hooks are functions that provide a way to use state and other React features in functional components.
- Before hooks, stateful logic was limited to class components, which could lead to complex and nested code structures.
- With hooks, functional components can now manage state and lifecycle methods just like class components.

Hooks 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

useState Hook

- The 'useState' hook is the most basic and commonly used hook in React.
- It allows you to add state to your functional components.
- The hook returns a stateful value and a function to update that value.

Syntax

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

State: This is the current state value that we want to track.

setState: This is a function used to update the state.

initialState :The initial value set to the state

Lets code

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

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

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

  return (
    <div>
      <h1>Count: {count}</h1>
      <button
onClick={increment}>Increment</button>
    </div>
  );
};
export default Counter;
```

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

const useStateExample= () => {
  const [inputValue, setInputValue] = useState(“Galaxe solutions");

  let handleChange= (event) => {
    const newValue = event.target.value;
    setInputValue(newValue);
  };

  return (
    <div>
      <input placeholder="enter something..." onChange={handleChange} />
      {inputValue}
    </div>
  );
};

export default useStateExample;
```

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

const FormData = () => {
  const [formData, setFormData] = useState({
    username: '',
    email: '',
    password: '',
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  return (
    <form>
      <input type="text" name="username" value={formData.username} onChange={handleChange} />
      <input type="email" name="email" value={formData.email} onChange={handleChange} />
      <input type="password" name="password" value={formData.password} onChange={handleChange} />
    </form>
  );
};
export default FormData;
```

Rules of Using useState

- The useState hook must be called at the top level of the functional component.
- The order of hooks must be the same in each render call.

Benefits of useState

- Simplifies state management in functional components.
- Reduces the amount of code compared to using class components.
- Improves code readability and maintainability.
- No need to worry about the "this" keyword, as in class components.

useReducer Hook

- The 'useReducer' hook provides an alternative way to handle more complex state and logic in functional components
- The useReducer hook is a powerful tool in React that allows us to manage state in a more organized and structured way.
- It is an alternative to useState and is particularly useful when the state has complex transitions that involve multiple sub-values.
- useReducer follows the same principles as the Redux library, where state transitions are determined by a function called a "reducer."

Syntax

```
const [state, dispatch] = useReducer(reducer, initialState);
```

State: The current state value, similar to the state returned by useState

setState: A function that allows you to dispatch actions to trigger state transitions.

initialState : The initial value set to the state

Reducer : It takes two arguments: the current state and an action object that describes the state change. The reducer's responsibility is to return the new state based on the action type.

```
const reducer = (state, action) => {
  switch (action.type) {
    case ‘type1':
      return { corresponding action to type 1};
    case ‘type2':
      return {corresponding action to type 1};
    default:
      return default action;
  }
};
```

Lets code

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

const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1, showText:
state.showText };
    case "toggleShowText":
      return { count: state.count, showText:
!state.showText };
    default:
      return state;
  }
};

const ReducerTutorial = () => {
  const [state, dispatch] = useReducer(reducer, {
count: 0, showText: true });
```

```
return (
    <div>
      <h1>{state.count}</h1>
      <button
        onClick={() => {
          dispatch({ type:
"INCREMENT" });
          dispatch({ type:
"toggleShowText" });
        }}
      >
        Click Here
      </button>

      {state.showText && <p>This
is a text</p>}
    </div>
  );
};

export default ReducerTutorial;
```

Benefits of useReducer

- Helps manage complex state transitions and logic more effectively.
- Provides a predictable way to update state based on actions.

When to use UseReducer over useState

- If the state logic is simple, stick to: useState.
- If the state transitions are complex and involve multiple sub-values, consider, useReducer.
- If you find yourself writing multiple useState calls to handle related state, it might be a good candidate for useReducer.