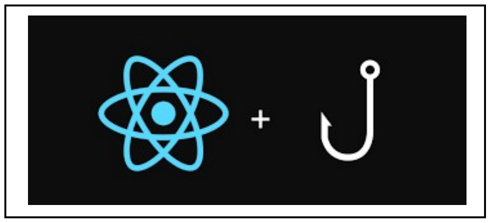
Hooks



Hooks are functions that enable you to use React functionalities like state and lifecycle features without using classes. Hooks have made it easier to share logic between components, reduce the need for higher-order components (HOCs) and render props, and create more concise and readable functional components. They are a significant improvement for managing complex state and side effects in React applications.

Note:

Before hooks were introduced, state management and side effects (lifecycle events) in React components were primarily handled by class components. Hooks were introduced in React 16.8 to enable state management, side effects, and other React features to be used in function components.

*React provides several built-in hooks,*and you can also create custom hooks to reuse stateful logic across multiple components. Some of the most commonly used React hooks are:

1. **useState:** useState allows function components to manage local state. You can use it to add state to your components and trigger re-renders when the state changes.

2. **useEffect:** useEffect allows you to perform side effects in your components. You can use it to fetch data, set up subscriptions, update the DOM, and more.

3. **useContext:**useContext enables you to access the context of a parent component in your child components, making it easier to share data or settings across your application.

4. **useReducer:**useReducer is an alternative to useState for managing more complex state logic. It is often used when you need to manage state transitions based on previous state.

5. **useRef:**useRef allows you to create mutable references to elements or values. It is often used to access or interact with DOM elements directly or to store values that don't trigger re-renders.

6. **useMemo and useCallback:** useMemo and useCallback allow you to optimize performance by memoizing the results of calculations and event handlers. They help prevent unnecessary calculations and function re-creations.

7. **useLayoutEffect:**useLayoutEffect is similar to useEffect but runs synchronously after the DOM is updated. It can be useful for performing DOM measurements or updates that need to be executed before the browser paints.

8. **useEffect with cleanup:** You can return a cleanup function from useEffect to handle resource cleanup (e.g., unsubscribing from a subscription) when the component unmounts.

RULES TO KEEP IN MIND WHEN USING HOOKS IN REACT

1. ***Only use hooks at the top level of your functional component:*** Hooks should always be used at the top level of a functional component, not inside nested functions, conditions, or loops.

```
function MyComponent () {
const [count, setCount] = useState (0);
   // ...
   return <div>{count}</div>;
  }
 // Avoid: Hooks inside if statement
  function MyComponent () {
   if (someCondition) {
     const [count, setCount] = useState (0); // Avoid this
   }
   // ...
   return <div>{count}</div>;
  }
```

2. **Always use hooks in the same order:** Ensure that you always call hooks in the same order in every render of your component. This order should be consistent, as React relies on it to associate state with hooks.
   e.g.

```
 function MyComponent () {
   const [count, setCount] = useState (0);
   const [name, setName] = useState('John');
   // ...
   return <div>{count}</div>;
 }

// Avoid: Changing the order of hooks
  function MyComponent () {
   const [name, setName] = useState('John');
   const [count, setCount] = useState (0); // Avoid changing the order
   // ...
   return <div>{count}</div>;
  }
```

Note:
If you change the order of hooks within a component, the code may appear to work without any immediate errors, especially if you don't rely on the order of hooks for your logic. React is designed to be flexible, and it will try to handle changes in the order of hooks gracefully by matching state updates to the corresponding components.

3. **Only call hooks from functional components:** ***You can only use hooks in functional components or custom hooks***. You should not call hooks from regular JavaScript functions or class components.

```
  // Using hooks in a functional component
  function MyComponent() {
   const [count, setCount] = useState(0);
   // ...
   return <div>{count}</div>;
  }
  // Avoid: Using hooks in a non-functional component
  class MyClassComponent extends React.Component {
render() {
     const [count, setCount] = useState(0); // Avoid this
     return <div>{count}</div>;
   }  }
```

USESTATE

The useState hook in React is a way to remember and change information in a functional component. It's like a container for holding a value, and you can use it to display and modify that value in your app. It's a fundamental tool for building dynamic and interactive user interfaces in React.

Syntax and how to use it

1. **Import React and the useState hook**: Make sure to import React and the `useState` hook at the beginning of your file:
   import React, { useState } from 'react';

2. **Declare and initialize state variables**

   To use the useState hook to declare and initialize one or more state variables within your functional component. The useState function takes an initial value as its argument and returns an array containing the current state value and a function to update that value. The typical pattern is to destructure this array into two variables.

   const [stateVariable, setStateVariable] = useState(initialValue);

   e.g.
   const [count, setCount] = useState(0); // Initializes 'count' with 0

3. **Use the state variable:** You can now use the stateVariable in your component to display or manipulate its value. In the example above, `count` can be used to display the current count in your component.
   <p> **state**: **{ stateVariable }**</p>
   e.g.
   <p>Count: **{count}**</p>

4. **Update the state variable:** To change the value of the state variable, you can use the corresponding setStateVariable function returned by useState. This function is responsible for updating the state and causing the component to re-render with the new value.
   <button onClick={() =>setStateVariable (prameter)}>caption of button</button>
   e.g.
   <button onClick={() => setCount(count + 1)}>Increment</button>

e.g. -1

e.g.

```
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  const increment = () => {
setCount(count + 1);
  };

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

Fuctional component without hook

```
import React from 'react';
function Counter() {
  let count = 0;
  const increment = () => {
    count = count + 1;
    // This won't trigger a re-render
  };
  return (
<div>
<p>Count: {count}</p>
<button onClick={increment}>Increment</button>
</div>
  );
}
export default Counter;
```

e.g.-2

```
import React,{useState} from 'react';
function MessageDisplay() {
    const [message,changeMessage]= useState('Hello, World!')


  const setMesssage = () => {
changeMessage('Welcome to React!')
    console.log(message);
  };

  return (
<div>
<p>{message}</p>
<button onClick={setMesssage}>Change Message</button>
</div>
  );
}
export default MessageDisplay;
```

e.g.-3

```
import React, { useState } from 'react';
function ToggleButton() {
  const [isToggled, setToggled] = useState(false);
  const toggle = () => {
setToggled(!isToggled);
  };
  return (
<div>
<button onClick={toggle}>
        {isToggled ? 'ON' : 'OFF'}
</button>
</div>
  );
}
export default ToggleButton;
```

e.g.-4

```
import React, { useState } from 'react';
function TextInput() {
  const [text, setText] = useState('');
  const handleChange = (event) => {
setText(event.target.value);
  };
  return (
<div>
<input type="text" value={text} onChange={handleChange} />
<p>You entered: {text}</p>
</div>
  );
}
export default TextInput;
```

e.g.-5

Event Handling Systax in javascript
<input type="text" onchange="myFunction(event)">

```
import React, { useState } from 'react';
function Form() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [message, setMessage] = useState('');

  const handleSubmit = (event) => {
event.preventDefault();
    // Perform form submission logic using name, email, and message
console.log('Submitted:', { name, email, message });
    // Reset the form inputs
setName('');
setEmail('');
setMessage('');
  };
  return (
<form onSubmit={handleSubmit}>
<label>
        Name:
<input
          type="text"
          value={name}
onChange={(event) => setName(event.target.value)}
        />
</label>
<label>
        Email:
<input
          type="email"
          value={email}
onChange={(event) => setEmail(event.target.value)}
        />
</label>
<label>
        Message:
<textarea
          value={message}
onChange={(event) => setMessage(event.target.value)}
        />
</label>
<button type="submit">Submit</button>
</form>
  );
}
export default Form;
```

USEEFFECT

The useEffect hook in React allows you to ***perform side effects*** in functional components. It tells React that you want to do something after the component has rendered such as fetching data from an API, updating the browser's title, setting up event listeners, or making changes to the DOM.

Its runs after every render of your component. It allows you to specify what should happen when the component is rendered or when certain dependencies change.

Understand the working

1. When a component renders, React checks for any useEffect hooks inside the component.
2. React executes the code inside the useEffect hook after the component has rendered and the browser has painted the changes to the screen.
3. You can provide a callback function as the first argument to useEffect. This callback function contains the code you want to run after each render.
4. By default, the callback function runs after every render. However, you can provide a second argument to useEffect which is an array of dependencies. If any of the dependencies change between renders, the useEffect callback will run again.

Syntax and how to use it

useEffect (callbackFunction, dependencyArray);

- **callbackFunction:** This is a function that contains the code you want to run as a side effect. It's executed after the component has rendered.
- **dependencyArray (optional):** This is an array of dependencies that tells React when the effect should run. If this array is empty, the effect will run after every render. If you provide dependencies, the effect will only run when one of the dependencies has changed.

1. **Import React and the useEffect hook:** Make sure to import React and the useEffect hook at the beginning of your file:
   import React, { useEffect } from 'react';

2. **Use the useEffect hook:** Inside your functional component, you can use the useEffect hook to perform side effects. For example, you might want to fetch data from an API when the component mounts
```
useEffect(() => {
   // Code to run after the component has rendered
   // This is where you typically put side effects, like data fetching.
  }, []); // Empty dependency array means this effect runs once, when the component mounts.
```

3. **Adding Dependencies (optional):** If you want your effect to run only when certain values change, you can specify those values in the dependency array. For example, if you want to fetch data when a “userId” prop changes, you would include it in the dependency array:
```
useEffect(() => {
   // Code to run when userId changes
}, [userId]);
```

4. **Cleaning Up:** You can also return a cleanup function from the `useEffect`. This function will be executed when the component unmounts or when the dependencies change. It's used for cleaning up resources like subscriptions or event listeners.

```
useEffect(() => {
  // Code to run when the component mounts

  return () => {
   // Code to clean up when the component unmounts or when dependencies change
  };
}, [/* dependencies */]);
```

e.g.-1

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

function MyComponent() {
 const [count1, setCount1] = useState(0);
 const [count2, setCount2] = useState(0);
useEffect(() => {
  // This code will run after each render
console.log( `Counter-1: ${count1}`);
console.log( `Counter-2: ${count2}`);

},[count1]); // Run the effect only when `count1` changes
return (
<div>
<p>Count: {count1}</p>
<p>Count: {count2}</p>
<button onClick={() => setCount1(count1 + 1)}>Increment</button>
<button onClick={() => setCount2(count2 + 1)}>Increment</button>
</div>
 );
}
export default MyComponent;
```

e.g.-2

```
import React, { useEffect, useState } from 'react';
function DataFetchingComponent() {
 const [data, setData] = useState([]);
useEffect(() => {
  // Replace 'https://api.example.com/data' with your actual API endpoint.
  fetch('https://jsonplaceholder.typicode.com/users')
.then((response) =>response.json())
.then((data) => setData(data));
 }, []);
 return (
<div>
<ul>
    {data.map((item) => (
<li key={item.id}>{item.name}</li>
    ))}
</ul>
</div>
 );
}
export default DataFetchingComponent;
```