**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. Also, it does not replace your knowledge of React concepts.

React Hooks are in-built functions that allow React developers to use state and lifecycle methods inside functional components

**Basic Hooks**

* useState
* useEffect
* useContext

**Additional Hooks**

* useReducer
* useCallback
* useMemo
* useRef
* useImperativeHandle
* useLayoutEffect
* useDebugValue

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.

* Allows the removal of a lot of class components that we only used because we couldn’t have, e.g., a local state or ref, so the code for a component looks easier to read.
* Enables you to use less code for the same effect.
* Makes functions way easier to think about and test, e.g., by using the [react-testing-library](https://github.com/kentcdodds/react-testing-library).
* Can also take parameters, and the result of one can be easily be used by another hook (e.g., setState from useState in useEffect).
* Minifies way better than classes, which tend to be a bit more problematic for minifiers.
* Might remove HOCs and render props patterns in your app which introduced new problems despite having been designed to solve others.

**“state management and Lifecycle management”**

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. These rules are:

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. This rule ensures that Hooks are called in the same order each time a components renders.

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.

Pre-requisites for React Hooks

1. Node version 6 or above
2. NPM version 5.2 or above
3. Create-react-app tool for running the React App

React Hooks Installation

To use React Hooks, we need to run the following commands:

1. $ npm install react@16.8.0-alpha.1 --save
2. $ npm install react-dom@16.8.0-alpha.1 --save

The above command will install the latest React and React-DOM alpha versions which support React Hooks. Make sure the **package.json** file lists the React and React-DOM dependencies as given below.

1. "react": "^16.8.0-alpha.1",
2. "react-dom": "^16.8.0-alpha.1",

Hooks State

Hook state is the new way of declaring a state in React app. Hook uses useState() functional component for setting and retrieving state. Let us understand Hook state with the following example.

**Using Hooks**

**App.js**

import React, { useState } from 'react';

function CountApp() {

// Declare a new state variable, which we'll call "count"

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

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

export default CountApp;

In the above example, useState is the Hook which needs to call inside a function component to add some local state to it. The useState returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it. After that, we will call this function from an event handler or somewhere else. The useState is similar to this.setState in class. The equivalent code without Hooks looks like as below.

**Before hooks**

**App.js**

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

**class** CountApp **extends** React.Component {

  constructor(props) {

**super**(props);

**this**.state = {

      count: 0

    };

  }

  render() {

**return** (

      <div>

        <p><b>You clicked {**this**.state.count} times</b></p>

        <button onClick={() => **this**.setState({ count: **this**.state.count + 1 })}>

          Click me

        </button>

      </div>

    );

  }

}

export **default** CountApp;

Ex2:

import React, { useState } from 'react'

export default function App() {

const [age, setAge] = useState(19);

const handleClick = () => setAge(age + 1)

return (

I am {age} Years Old

<div>

<button onClick={handleClick}>Increase my age! </button>

</div>

</div>

);

Ex3:

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

export default function App() {

//Define State

const [name, setName] = useState({firstName: 'name', surname: 'surname'});

const [title, setTitle] = useState('BIO');

//Call the use effect hook

useEffect(() => {

setName({firstName: 'Ayushi', surname: 'M'})

}, [])//pass in an empty array as a second argument

return(

<div>

<h1>Title: {title}</h1>

<h3>Name: {name.firstName}</h3>

<h3>Surname: {name.surname}</h3>

</div>

);

};

Hooks Effect

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.

Side effects have common features which the most web applications need to perform, such as:

* Updating the DOM,
* Fetching and consuming data from a server API,
* Setting up a subscription, etc.

Ex2:

Before Hooks:

import React, { Component } from "react";

export default class Button extends Component {

constructor() {

super();

this.state = { buttonText: "Click me, please" };

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState(() => {

return { buttonText: "Thanks, been clicked!" };

});

}

render() {

const { buttonText } = this.state;

return <button onClick={this.handleClick}>{buttonText}</button>;

}

}

After Hooks:

import React, { useState } from "react";

export default function Button() {

const [buttonText, setButtonText] = useState("Click me, please");

return (

<button onClick={() => setButtonText("Thanks, been clicked!")}>

{buttonText}

</button>

);

}

useEffect:

Ex:3

export default class DataLoader extends Component {

state = { data: [] };

componentDidMount() {

fetch("http://localhost:8085/api/v1/employees")

.then(response => response.json())

.then(data =>

this.setState(() => {

return { data };

})

);

}

render() {

return (

<div>

<ul>

{this.state.data.map(el => (

<li key={el.firstName}>{el.lastName} {el.emailId}</li>

))}

</ul>

</div>

);

}

}

Using hooks

export default function DataLoader() {

const [data, setData] = useState([]);

useEffect(() => {

fetch("http://localhost:8085/api/v1/employees")

.then(response => response.json())

.then(data => setData(data));

});

return (

<div>

<ul>

{data.map(el => (

<li key={el.id}>{el.firstName} {el.lastName} {el.emailId}</li>

))}

</ul>

</div>

);

}

In React component, there are two types of side effects:

1. Effects Without Cleanup
2. Effects With Cleanup

Effects without Cleanup

It is used in useEffect which does not block the browser from updating the screen. It makes the app more responsive. The most common example of effects which don't require a cleanup are manual DOM mutations, Network requests, Logging, etc.

Effects with Cleanup

Some effects require cleanup after DOM updation. For example, if we want to set up a subscription to some external data source, it is important to clean up memory so that we don't introduce a memory leak. React performs the cleanup of memory when the component unmounts. However, as we know that, effects run for every render method and not just once. Therefore, React also cleans up effects from the previous render before running the effects next time.

**customHooks:**

useFetch.js

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

useEffect(() => {

fetch(url)

.then(response => response.json())

.then(data => setData(data));

}, []);

return data;

}

DataLoader.js

import React from "react";

import useFetch from "./useFetch";

export default function DataLoader(props) {

const data = useFetch("http://localhost:8085/api/v1/employees");

return (

<div>

<h1> using custom hooks</h1>

<ul> {data.map(el => (

<li key={el.id}>{el.firstName} {el.lastName} {el.emailId}</li>

))}

</ul>

</div> );

}

**Aync / Await :**

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

async function getData() {

const response = await fetch(url);

const data = await response.json();

setData(data);

}

useEffect(() => {

getData();

}, []);

return data;

}