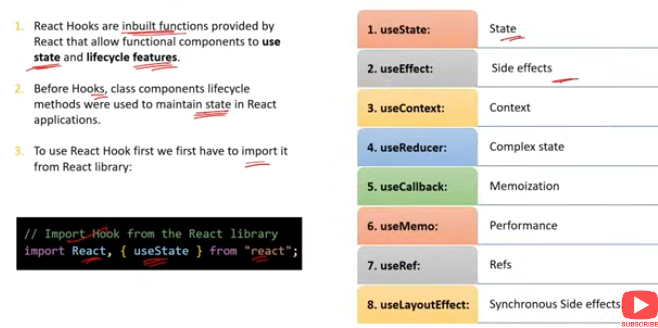
Q1)What are React Hooks ? What are the top React Hooks ?



### What are React Hooks?

**React Hooks** are functions that allow you to use state and other React features in functional components. Introduced in React 16.8, hooks enable functional components to manage state, perform side effects, and access the component lifecycle without the need for class components.

### Top React Hooks

useState:

* 1. Manages state in functional components.
  2. Returns a stateful value and a function to update it.

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

useEffect:

* 1. Performs side effects in functional components, such as fetching data, subscribing to services, or manually updating the DOM.
  2. Replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

useEffect(() => {

// Effect logic

return () => {

// Cleanup logic

};

}, [dependencies]);

useContext:

* 1. Accesses the value from a React Context without having to pass props manually through every level of the component tree.

const value = useContext(MyContext);

useReducer:

* 1. Manages complex state logic in functional components, similar to useState but more powerful.
  2. Useful for managing state transitions in a predictable manner using a reducer function.

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

useRef:

* 1. Creates a mutable reference that persists across renders.
  2. Commonly used to access DOM elements directly or store any mutable value.

const inputRef = useRef(null);

useMemo:

* 1. Memoizes a computed value to avoid expensive recalculations on every render.
  2. Recomputes the memoized value only when its dependencies change.

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

useCallback:

* 1. Memoizes a callback function to prevent unnecessary re-creations on each render.
  2. Useful when passing callbacks to optimized child components that rely on reference equality.

const memoizedCallback = useCallback(() => {

doSomething(a, b);

}, [a, b]);

useLayoutEffect:

* 1. Similar to useEffect, but it runs synchronously after all DOM mutations. Used for reading layout from the DOM and synchronously re-rendering.

useLayoutEffect(() => {

// Layout effect logic

}, [dependencies]);

useImperativeHandle:

* 1. Customizes the instance value that is exposed to parent components when using ref in conjunction with forwardRef.

useImperativeHandle(ref, () => ({

customMethod() {

// Logic

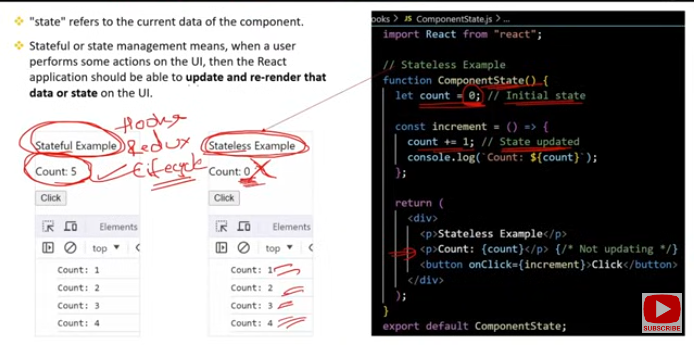
}

}));

### Summary:

* **React Hooks**: Enable state management, side effects, and more in functional components.
* **Top Hooks**: Include useState, useEffect, useContext, useReducer, useRef, useMemo, useCallback, useLayoutEffect, and useImperativeHandle.

Q2)What are state,stateless.stateful,and state Management term ?



### State, Stateless, Stateful, and State Management in React

#### 1. ****State****:

* **Definition**: State is an object that holds data that influences the rendering and behavior of a React component. State is mutable and can be updated, triggering a re-render of the component to reflect the changes.
* **Usage**: Managed within components using hooks like useState in functional components or this.state in class components.

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

this.state = { count: 0 }; // Class component

#### 2. ****Stateless**** (Presentational) Components:

* **Definition**: Components that do not manage their own state and rely on props to receive data and configuration. They focus purely on rendering UI based on the given props.
* **Usage**: Often used for simple UI elements or as child components that receive state data from parent components.

function Display({ message }) {

return <div>{message}</div>;

}

#### 3. ****Stateful**** (Container) Components:

* **Definition**: Components that manage and hold their own state. These components are responsible for data logic and state management, and often pass state data to child components as props.
* **Usage**: Used when a component needs to track user input, manage data fetching, or handle complex interactions.

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

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

</div>);

}

#### 4. ****State Management****:

* **Definition**: State management refers to the techniques and patterns used to manage and share state across components in a React application. It involves ensuring that state is updated consistently and efficiently across different parts of an application.
* **Techniques**:
  + **Local State**: Managed within individual components using hooks (useState, useReducer).
  + **Lifted State**: State is lifted up to a common ancestor component when multiple components need to share the same data.
  + **Context API**: Used for passing state globally to components without prop drilling.
  + **State Management Libraries**: External libraries like Redux, MobX, or Zustand manage state across larger applications.

import React, { useState, createContext, useContext } from 'react';

const MyContext = createContext();

function App() {

const [value, setValue] = useState('Hello');

return (

<MyContext.Provider value={value}>

<Child />

</MyContext.Provider>

);

}

function Child() {

const value = useContext(MyContext);

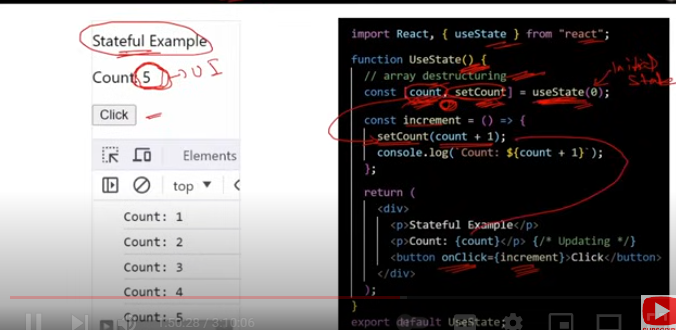
return <div>{value}</div>;

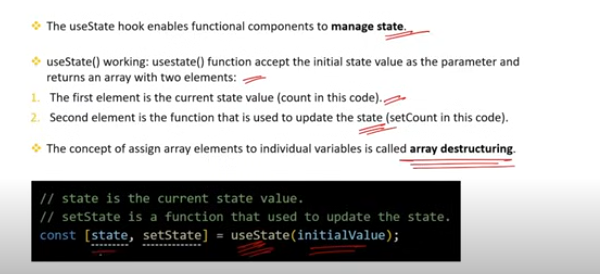
}

### Summary:

* **State**: Holds data that determines the component's behavior and rendering.
* **Stateless Components**: Do not manage state; rely on props.
* **Stateful Components**: Manage their own state internally.
* **State Management**: Techniques and patterns for managing state across components and the entire application.

Q3)What is the role of useState() hook and how it works ?





### Role of useState() Hook in React

The useState() hook is a fundamental React hook that allows functional components to have and manage state. It enables you to add stateful logic to functional components, which previously was only possible in class components.

### How useState() Works

**Initialization**:

* 1. useState() is called with an initial state value. This value can be any data type: string, number, object, array, etc.
  2. It returns an array with two elements:
     1. The **current state value**.
     2. A **function to update the state**.

const [state, setState] = useState(initialValue);

**State Management**:

* 1. The state value is preserved across renders. When the state is updated using the provided function, React re-renders the component with the new state value.
  2. The state-updating function can take a new state value or a function that receives the current state and returns the updated state.

const [count, setCount] = useState(0); // Initialize state with 0

const increment = () => {

setCount(count + 1); // Update state

};

**Reactivity**:

* 1. When you call the state-updating function (setState), React triggers a re-render of the component with the updated state.
  2. The updated state is then used in the next render.

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

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

</div>

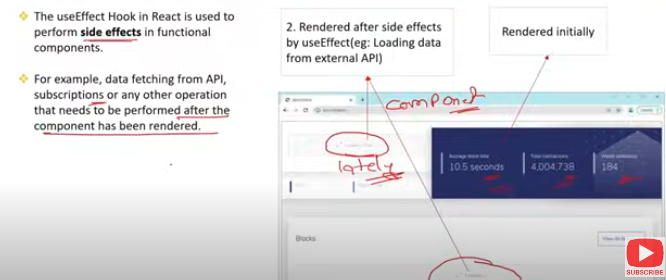
);

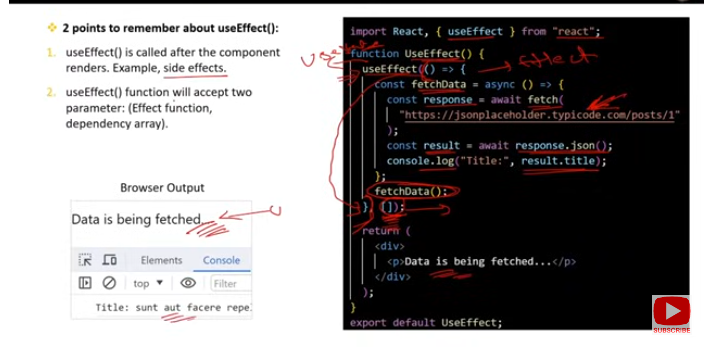
}

### Summary:

* **Role**: useState() allows functional components to hold and manage state.
* **Initialization**: Call useState() with an initial value; it returns the state and a function to update it.
* **State Updates**: The state-updating function triggers re-renders with the new state value.

Q4)What is the is the role of useEffect() .How it works and what is its use ?





### Role of useEffect() in React

The useEffect() hook allows you to perform side effects in functional components. Side effects include tasks like data fetching, subscriptions, manual DOM manipulations, and logging. useEffect() enables functional components to mimic lifecycle methods from class components, such as componentDidMount, componentDidUpdate, and componentWillUnmount.

### How useEffect() Works

**Basic Usage**:

* 1. useEffect() takes two arguments:
     1. **Effect function**: A function where you write the code for the side effect.
     2. **Dependencies array** (optional): An array of values that the effect depends on. The effect function is re-run when any of these values change.
  2. By default, useEffect() runs after every render.

useEffect(() => {

// Code to run after the component renders

});

**Running Once on Mount:**

* 1. If you pass an empty array [] as the second argument, the effect runs only once after the component mounts, similar to componentDidMount.

useEffect(() => {

// Code runs only once after the component mounts

}, []);

**Cleaning Up**:

* 1. useEffect() can return a cleanup function. This cleanup function is called when the component unmounts or before the effect is re-run due to changes in dependencies. This is useful for tasks like unsubscribing from a service or clearing timers.

useEffect(() => {

const subscription = someService.subscribe();

return () => {

// Cleanup: unsubscribe or clear timers

subscription.unsubscribe();

};

}, [dependencies]);

**Re-running Effects:**

* 1. The effect function re-runs whenever any of the dependencies in the array change. If no dependencies are provided, the effect runs after every render.

useEffect(() => {

// Code runs every time 'count' changes

console.log(`Count changed to: ${count}`);

}, [count]);

### Use Cases for useEffect()

**Data Fetching**:

* 1. Fetch data from an API when the component mounts and update the state with the result.

useEffect(() => {

fetch('https://api.example.com/data')

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

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

}, []);

**Subscribing to Events**:

* 1. Subscribe to events like window resizing or WebSocket updates and clean up the subscription when the component unmounts.

useEffect(() => {

const handleResize = () => console.log(window.innerWidth);

window.addEventListener('resize', handleResize);

return () => window.removeEventListener('resize', handleResize);

}, []);

**Synchronizing with External Systems**:

* 1. Update a document title based on the current state or synchronize component state with local storage.

useEffect(() => {

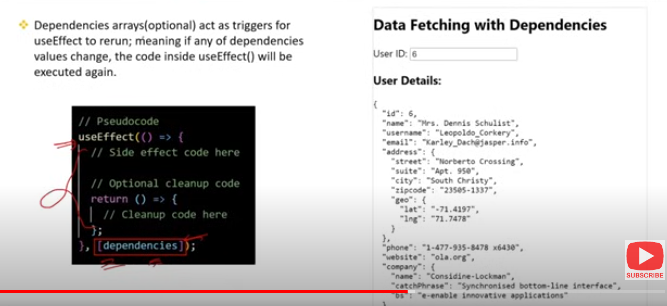
document.title = `Count: ${count}`;

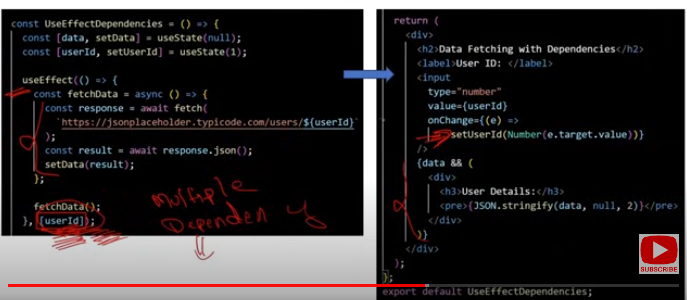
}, [count]);

### Summary:

* **Role**: useEffect() handles side effects in functional components.
* **How It Works**: Runs after render, optionally dependent on specified values. Supports cleanup.
* **Use Cases**: Data fetching, subscriptions, manual DOM updates, and more.

Q5)What is Dependency Array in useEffect() hook ?





### Dependency Array in useEffect() Hook

The **dependency array** in the useEffect() hook determines when the effect function should re-run. It is the second argument passed to useEffect() and is essential for optimizing performance and controlling when side effects occur.

### Key Points:

**Purpose**:

* 1. The dependency array lists all the variables that the effect depends on. React will re-run the effect function whenever any of these dependencies change.

**Usage**:

* 1. **Empty Array** []: The effect runs only once after the initial render (mimics componentDidMount).
  2. **No Array**: The effect runs after every render (mimics both componentDidMount and componentDidUpdate).
  3. **Array with Dependencies**: The effect runs only when one or more of the specified dependencies change.

### Examples:

**Empty Dependency Array** []:

* 1. The effect runs only once when the component mounts.

useEffect(() => {

console.log('Component mounted');

}, []); // Runs only once

**No Dependency Array**:

* 1. The effect runs after every render, which can lead to performance issues if not used carefully.

useEffect(() => {

console.log('Effect runs after every render');

}); // Runs on every render

**Array with Specific Dependencies**:

* 1. The effect runs only when the specified variables in the array change.

useEffect(() => {

console.log(`Count changed: ${count}`);

}, [count]); // Runs only when 'count' changes

**Multiple Dependencies**:

* 1. The effect runs when any of the variables in the dependency array change.

useEffect(() => {

console.log(`Count or user changed: ${count}, ${user}`);

}, [count, user]); // Runs when 'count' or 'user' changes

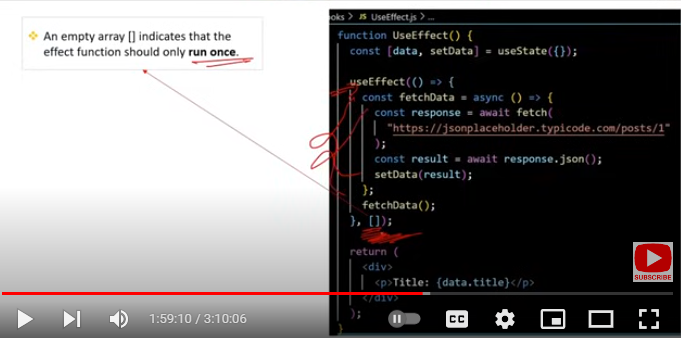
### Why It Matters:

* **Performance Optimization**: By controlling when the effect runs, you avoid unnecessary computations or operations, improving the performance of your application.
* **Correct Behavior**: Ensures that the effect reflects the current state of your application, as it will only re-run when necessary.

### Summary:

* **Dependency Array**: Controls when the useEffect() function re-runs, based on the specified variables.
* **Usage**:
  + []: Run once on mount.
  + No array: Run after every render.
  + [dependencies]: Run when specified dependencies change.

Q6)What is the meaning of empty array [] in the useEffect ?



### Meaning of an Empty Array [] in useEffect

When you pass an empty array [] as the second argument to the useEffect() hook, it tells React to run the effect function **only once** after the initial render of the component. This behavior mimics the componentDidMount lifecycle method in class components.

### Key Points:

**Runs Once After Initial Render**:

* 1. The effect function will execute only once, right after the component mounts. It will not re-run on subsequent renders.

useEffect(() => {

console.log('Component mounted');

// This effect runs only once

}, []); // Empty array as dependency

**No Dependencies**:

* 1. Since the array is empty, there are no dependencies to monitor. Thus, the effect does not re-run unless the component is unmounted and remounted.

**Common Use Cases**:

* 1. **Fetching Data**: Perform an API call to fetch data only when the component mounts.
  2. **Subscribing to Services**: Set up event listeners or subscriptions that should only be initialized once.
  3. **Initializing State**: Run any setup logic that needs to happen just once.

**Cleanup**:

* 1. You can return a cleanup function from the effect, which will be called when the component is unmounted.

useEffect(() => {

console.log('Component mounted');

return () => {

console.log('Component unmounted');

};

}, []); // Runs once and cleans up on unmount

### Summary:

* **Empty Array** []: Tells useEffect() to run the effect function only once after the component mounts, mimicking componentDidMount.
* **No Re-Runs**: The effect does not re-run on updates unless the component is unmounted and remounted.
* **Use Cases**: Ideal for actions like data fetching, subscriptions, and initial setup that should occur only once.