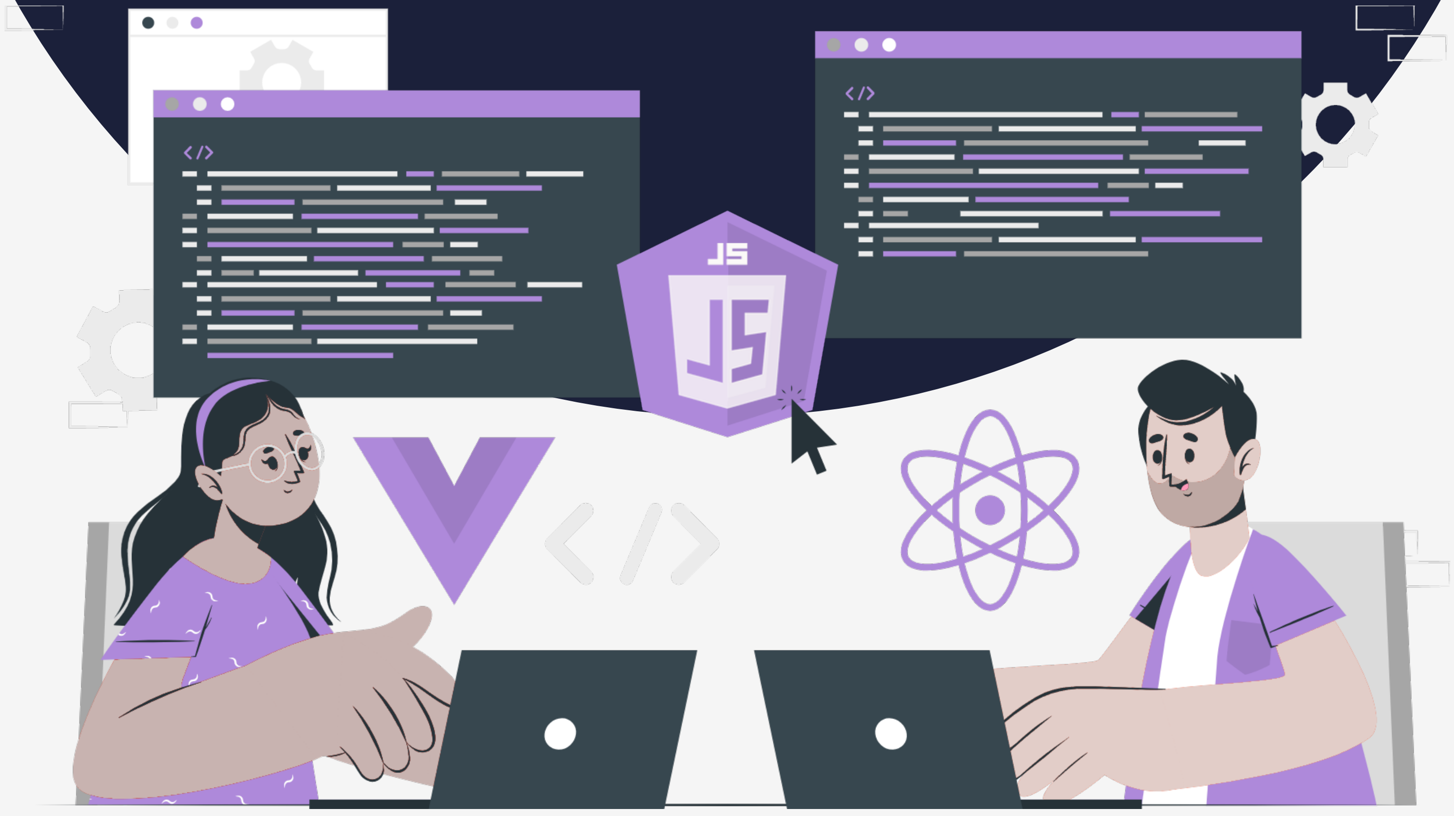
Lesson:

useEffect()



SKILLS

Topics Covered:

1. What are side effects ?
2. What is the useEffect() hook.
3. Uses of useEffect()

What are side effects?
In React, a "side effect" is any code that produces an effect outside of the current component or function, such as modifying the DOM, fetching data from an API, updating a global variable, or setting a timer.

React components are designed to be predictable and deterministic, meaning that given the same input, they should always produce the same output. Side effects can make components less predictable and harder to reason about, since they can cause unexpected changes or actions to happen outside of the component. To manage side effects We can use the `useEffect()` hook

The useEffect() hook allows us to perform side effects in our components. Basic examples are fetching data, directly updating the DOM etc.

The useEffect hook helps perform side effects by isolating it from the rendering logic.

```
Unset
useEffect(callback, [dependencies])
```

The useEffect hook takes two arguments:
- A callback function to define and clean up a side effect.
- An optional dependency array that ensures when to run a side effect defined inside the callback function.

When we pass the callback function to the useEffect hook, it runs the side effect.By default react runs it on every render of a component.However, side effects can be expensive and performance-intensive to run on every render. We can control it using the dependency array argument we pass to the `useEffect` hook.

Uses of useEffect:

1.After every render, side effect run

If we don't pass the dependency array to the useEffect hook, the callback function executes on every render.Thus React will run the side effect defined in it after every render.

```JavaScript
useEffect(()=>{
  //Side effect
});
```

2.After initial render, Side effect runs only once

If we want to run the side effect just once after the initial render, For example we want to fetch the data and make an API call and store the responses into a state variable after the initial render.You can pass an empty array as the second argument to the useEffect hook to tackle this use case.

```javascript
useEffect(() => {
  // Side Effect
}, []);
```

In this case, the side effect runs only once after the initial render of the component.

3.Side effects runs after props value change

We can also use props as a dependency to run the side effect.In this case, the side effect will run every time there is a change to the props passed as a dependency.

```javascript
useEffect(() => {
//Side effect

},[props]);
```

Let's take an example using useEffect hooks,

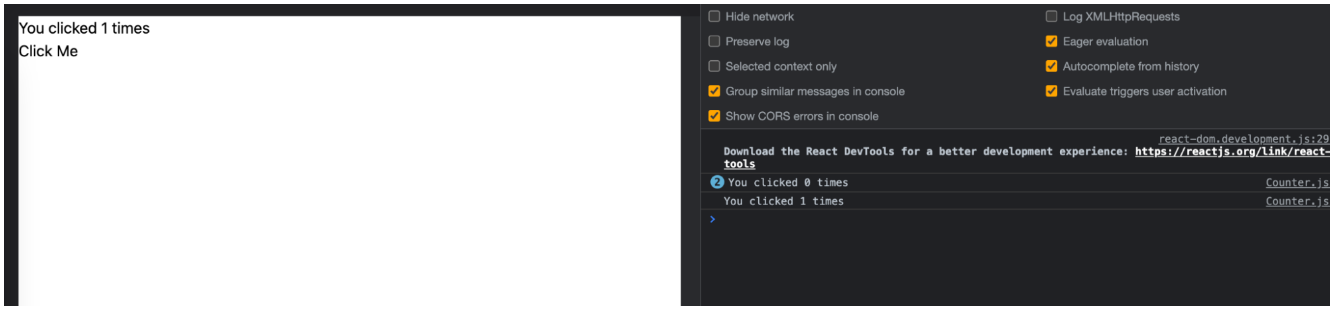
```javascript
import react, { useState, useEffect } from "react"
```

```javascript
function Counter() {
    const [count, setCount] = useState(0)
    useEffect(() => {
        setTimeout(() => {
            console.log(`You clicked ${count} times`);
        }, 3000);
    });

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>Click
Me</button>
        </div>
    )
}

export default Counter;
```

Output:



```
You clicked 1 times
Click Me
```

☐ Hide network ☐ Log XMLHttpRequests
☐ Preserve log ☑ Eager evaluation
☐ Selected context only ☑ Autocomplete from history
☑ Group similar messages in console ☑ Evaluate triggers user activation
☑ Show CORS errors in console

```
                                                         react-dom.development.js:29
Download the React DevTools for a better development experience: https://reactjs.org/link/react-
tools
2 You clicked 0 times                                                      Counter.js
  You clicked 1 times                                                      Counter.js
>
```

Here we are using the `useEffect` hook and setting the `setTimeout` function. Everytime we click the button, It displays on the console after every 3sec.

Note:
Whenever the user clicks on the click me button, the log is printed two times because of the strict mode but it will be working fine in the production environment by running only one time.

If we want to run it only once , then we can remove strict mode from index.js which is not recommended.

`useEffect` is usually the place where data fetching happens in React. Data fetching means using asynchronous functions, and using them in useEffect might not be as straightforward as we would think.

Wrong way to do data fetching using useEffect:

```javascript
. . .
// ❌ don't do this
useEffect(async () => {
  const data = await fetchData();

}, [fetchData])
. . .
```

We know that the first argument of `useEffect` is supposed to be a function that returns either nothing i.e undefined or a function to clean up side effects. But here an async function returns a Promise which can't be called as function!.It's just not what the useEffect hook expects as a first input.

Then how does a useEffect use asynchronous code?

Usually the solution is to write the data fetching code inside the useEffect itself.

JavaScript

```
. . .
useEffect(() => {
  // declare the data fetching function
  const fetchData = async () => {
    const data = await fetch('https://yourapi.com');
  }

  // call the function
  fetchData()
    // make sure to catch any error
    .catch(console.error);
```

```
}, [])
. . .
```