Do you have experience or do you know what CI/CD is?

I've got practical experience with Continuous Integration (CI) and Continuous Deployment (CD), two essential practices in modern software development. CI means frequently blending code changes from different developers into one central place. During this, unit tests are like a checkpoint. They're run automatically to make sure new code is up to snuff and won't mess up what's already there. It's like a safety net that keeps the code solid and everyone on the same page.

For Continuous Deployment, it's even cooler. If code passes the CI tests, it's sent off to production automatically. And guess what? Those unit tests are there again, making sure things work as they should. All this starts rolling when someone makes a pull request (PR), kicking off the CI process. That's why it's smart to catch issues early, even before they hit the big stage.

In a nutshell, my experience with CI/CD, paired with unit testing, shows that these methods make development smoother, deployments safer, and our software top-notch.

useEffect hook different use cases

1. componentDidMount
This lifecycle method is called once, after the component mounts. To mimic this with useEffect, you use an empty dependency array, which tells React to run the effect only after the first render.

```
useEffect(() => {
  // Code to run on component mount
}, []); // Empty dependency array
```

2. componentDidUpdate
This lifecycle method is invoked after every update (re-render) caused by changes to props or state. In useEffect, you can achieve similar behavior by specifying the state or props you want to watch in the dependency array.

```
useEffect(() => {
  // Code to run on component update
}, [dependency1, dependency2]); // Dependency array
```

If you want to mimic componentDidUpdate for all updates, you can omit the dependency array:

```
useEffect(() => {
  // Code to run on every update
});
```

3. componentWillUnmount

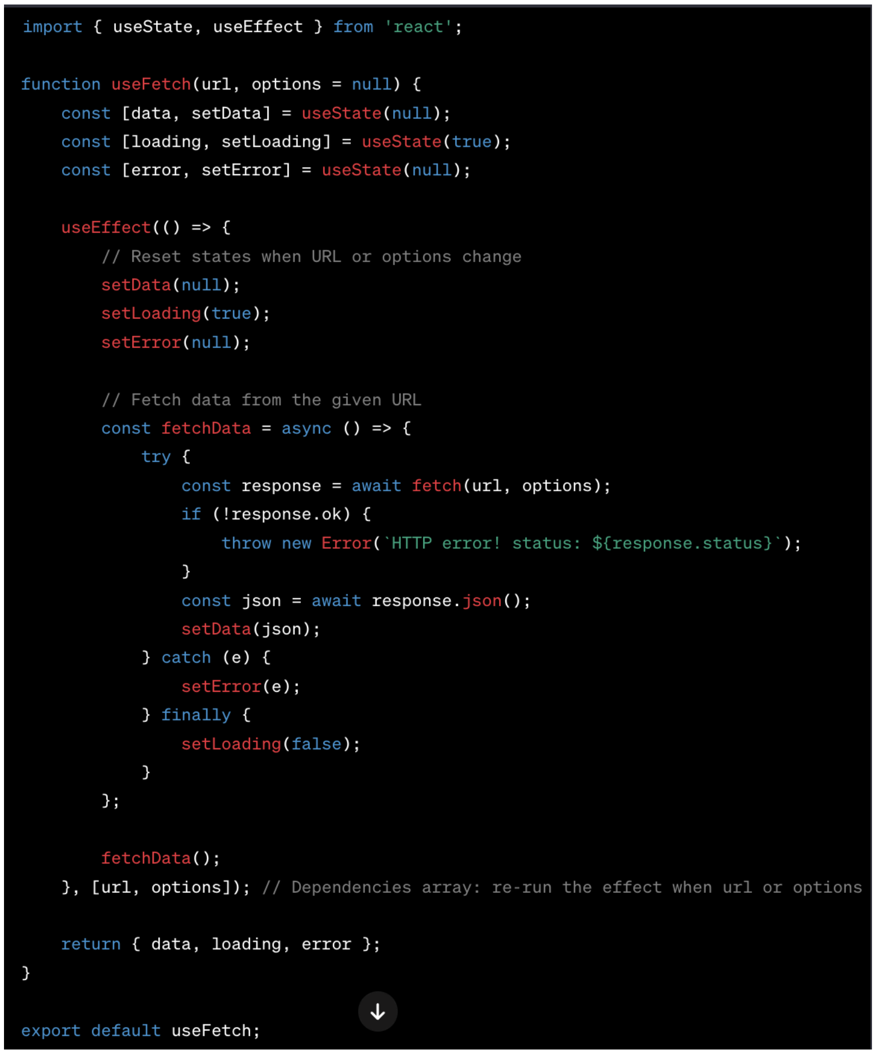
This lifecycle method is called right before a component is unmounted and destroyed. To mimic this in useEffect, you return a function from the effect, which will be called when the component is about to unmount.

```
useEffect(() => {
  // Code to run on component mount

  return () => {
    // Code to run on component unmount
  };
}, []); // Could have dependencies or be empty
```

Do you know how to create a custom hook that do call api?

Creating a custom hook in React for fetching data:



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

function useFetch(url, options = null) {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        // Reset states when URL or options change
        setData(null);
        setLoading(true);
        setError(null);

        // Fetch data from the given URL
        const fetchData = async () => {
            try {
                const response = await fetch(url, options);
                if (!response.ok) {
                    throw new Error(`HTTP error! status: ${response.status}`);
                }
                const json = await response.json();
                setData(json);
            } catch (e) {
                setError(e);
            } finally {
                setLoading(false);
            }
        };

        fetchData();
    }, [url, options]); // Dependencies array: re-run the effect when url or options

    return { data, loading, error };
}

export default useFetch;
```

What is HOC, do we still need in react functional component?
In React, a Higher-Order Component (HOC) is an advanced technique for reusing component logic. It's a pattern derived from React's compositional nature. Essentially, a HOC is a function that takes a component and returns a new component, thereby enhancing the original component with additional properties or behaviors.

HOCs are a powerful pattern for abstracting and reusing component logic in React applications. However, with the introduction of Hooks in React 16.8, many scenarios where HOCs were used can now be handled more elegantly with hooks, especially in functional components. Nonetheless, understanding HOCs is still important for working with many existing React codebases and libraries.

What is Promise?

A Promise in JavaScript is an object representing the eventual completion or failure of an asynchronous operation. Essentially, it's a container for a future value or error. Promises are used to handle asynchronous operations like network requests, file system tasks, or any operations that take time to complete.

Key Concepts of Promises:
States: A Promise is in one of these states:

Pending: Initial state, neither fulfilled nor rejected.
Fulfilled: The operation completed successfully.
Rejected: The operation failed.

What is an event loop?

The Event Loop enables JavaScript, a single-threaded language, to handle asynchronous operations. In a JavaScript runtime like V8, functions are pushed to the Call Stack for execution. Asynchronous functions are moved to Web APIs, where they're processed separately. Upon completion, their callbacks enter the Task Queue. The Event Loop constantly monitors the Call Stack and, when it's empty, transfers callbacks from the Task Queue to the Call Stack. This system allows JavaScript to perform asynchronous tasks efficiently without blocking the main thread.

Do you know React LifeCycle?
The React component lifecycle encompasses three main phases: mounting, updating, and unmounting. During mounting, a component is rendered for the first time, primarily using the render() method for UI rendering and componentDidMount for initial tasks like AJAX calls. In the updating phase, triggered by subsequent renders, render() continues to be essential, while shouldComponentUpdate can optimize rendering, and componentDidUpdate allows tracking of state and prop changes. Finally, in the unmounting phase, componentWillUnmount is used for cleanup as the component is removed from the DOM. Each phase has specific methods enabling control and manipulation of the component's behavior and lifecycle.

What is hoisting in JS?

Hoisting in JavaScript is a behavior in which variable and function declarations are moved to the top of their containing scope during the compilation phase. This means that variables and functions are hoisted to the top of the scope, either the function in which they are defined, if they are locally scoped, or to the global scope, if they are outside any function.

Key Points About Hoisting:

Variable Hoisting: With var declarations, only the declaration is hoisted, not the initialization. If you try to access a variable before it is declared and initialized, it will result in undefined. However, with let and const, hoisting behaves differently. They are hoisted, but not initialized, leading to a ReferenceError if accessed before declaration.

```
console.log(x); // undefined
var x = 5;

console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 10;
```

Function Hoisting: Function declarations are hoisted, and the entire function is available at the top of the scope. This allows a function to be used before it is declared in the code.

```
hello(); // "Hello, world!"

function hello() {
   console.log("Hello, world!");
}
```

Function Expressions: Unlike function declarations, function expressions are not hoisted. If a function expression is defined using var, the variable name is hoisted but not the function definition.

```
foo(); // TypeError: foo is not a function

var foo = function() {
   console.log("bar");
};
```

What is the “This” keyword in JS

In JavaScript, the this keyword is a special identifier that's automatically defined in the scope of every function and refers to the object that is executing the current piece of code. Understanding how this behaves in different contexts is crucial for effective JavaScript programming.

Understanding this in Different Contexts:
Global Context: In the global execution context (outside of any function), this refers to the global object. In a browser, it's window, and in Node.js, it's global.

```
console.log(this === window); // true in a browser
```

Function Context:

Regular Functions: In regular function calls, this refers to the global object (in non-strict mode) or undefined (in strict mode).
Method Calls: When a function is called as a method of an object, this refers to the object the method is called on.

```
function regularFunction() {
  console.log(this);
}
regularFunction(); // `this` will be `window` in non-strict mode or `undefined` in strict mode

const obj = {
  method: function() {
    console.log(this);
  }
};
obj.method(); // `this` refers to `obj`
```

Constructor Functions: When a function is used as a constructor (using new keyword), this refers to the newly created object.

```
function Person(name) {
  this.name = name;
}
const person = new Person('Alice');
console.log(person.name); // Alice
```

Arrow Functions: Arrow functions do not have their own this context; they inherit this from the parent scope at the time they are defined.

```
const obj = {
  method: () => {
        console.log(this);
    }
};
obj.method(); // `this` is not `obj`, it's the `this` value from the outer scope
```

call, apply, and bind Methods:
These methods are used to control the value of this in a function.

call Method: It calls a function with a given this value and arguments provided individually.

```
function greet() {
    console.log(`Hello, ${this.name}`);
}
const user = { name: 'Alice' };
greet.call(user); // Hello, Alice
```

apply Method: Similar to call, but arguments are passed as an array.

```
function greet(greeting, punctuation) {
    console.log(`${greeting}, ${this.name}${punctuation}`);
}
const user = { name: 'Alice' };
greet.apply(user, ['Hello', '!']); // Hello, Alice!
```

bind Method: It creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
function greet() {
    console.log(`Hello, ${this.name}`);
}
const user = { name: 'Alice' };
const boundGreet = greet.bind(user);
boundGreet(); // Hello, Alice
```

Use Cases:
Event Handlers: In DOM event handlers, this refers to the element that received the event, unless an arrow function is used.
Object Methods: To access properties or other methods within the same object.
Callback Functions: Control the context in which a callback function is executed.
Functional Programming: To set the context for utility functions.
Understanding this and its manipulation methods (call, apply, and bind) is fundamental for writing effective and predictable JavaScript code, especially when dealing with object-oriented programming or manipulating execution context.

Different between Function class component

Hooks: Function components can use Hooks, which allow for more reusable stateful logic without the complexity of classes.
Lifecycle Methods vs. Effects: Class components use lifecycle methods, while function components use the useEffect hook.
this Keyword: Class components often require binding this in callbacks, whereas function components don’t use this.
Syntax and Boilerplate: Function components tend to be more concise and have less boilerplate compared to class components.

With the advent of Hooks, function components are now capable of almost everything class components can do, leading to a shift in the React community towards functional components for their simplicity and reduced boilerplate. However, understanding class components is still important, especially for working with older React codebases.