**Which are the differences between real and virtual DOM in React?**

React is a popular JavaScript library for building user interfaces. One of its key features is the use of a virtual DOM, which is a lightweight copy of the actual DOM. Here are some key differences between the real and virtual DOM in React:

1. Representation: The real DOM is the actual representation of the web page and its structure, whereas the virtual DOM is a lightweight copy of the real DOM that is maintained by React.
2. Updating: When changes are made to the real DOM, the browser must re-render the entire page, which can be slow and resource-intensive. In contrast, when changes are made to the virtual DOM, React updates only the specific components that have changed, and then re-renders the virtual DOM. This makes updates faster and more efficient.
3. Manipulation: Manipulating the real DOM directly can be complex and error-prone, while the virtual DOM is simpler to manipulate, since it's just a JavaScript object.
4. Performance: Because the virtual DOM is faster to update and manipulate than the real DOM, React is generally faster and more performant than other JavaScript libraries that don't use a virtual DOM.
5. Flexibility: The virtual DOM allows React to support a wide range of platforms and environments, including web browsers, mobile devices, and server-side rendering.

In summary, the virtual DOM in React is a powerful tool that makes updates faster and more efficient, while also simplifying the process of manipulating the DOM.

**Which are the differences between callbacks, promises and async/await functions in React?**

In React, callbacks, promises, and async/await functions are all ways of managing asynchronous operations. Here are some key differences between these three approaches:

1. Callbacks: Callbacks are functions that are passed as arguments to other functions and are executed when an asynchronous operation completes. They are the oldest of the three approaches and can be used in any version of JavaScript. However, they can lead to callback hell when many asynchronous operations are nested.
2. Promises: Promises provide a cleaner way of managing asynchronous operations by representing a value that may not yet be available. They can be chained and allow for error handling through a single catch block. Promises were introduced in ES6 and have been widely adopted. However, they still require some boilerplate code.
3. Async/await functions: Async/await is a newer approach to managing asynchronous operations that allows for even cleaner code than promises. Async functions return a promise, and the await keyword can be used inside the function to wait for the resolution of a promise. Async/await functions can be used in any React application that supports ES6.

In summary, callbacks, promises, and async/await functions all provide ways of managing asynchronous operations in React, with each approach having its own advantages and disadvantages. Callbacks can be used in any version of JavaScript but can lead to callback hell, while promises and async/await provide cleaner and more concise code, with async/await being the most modern approach.

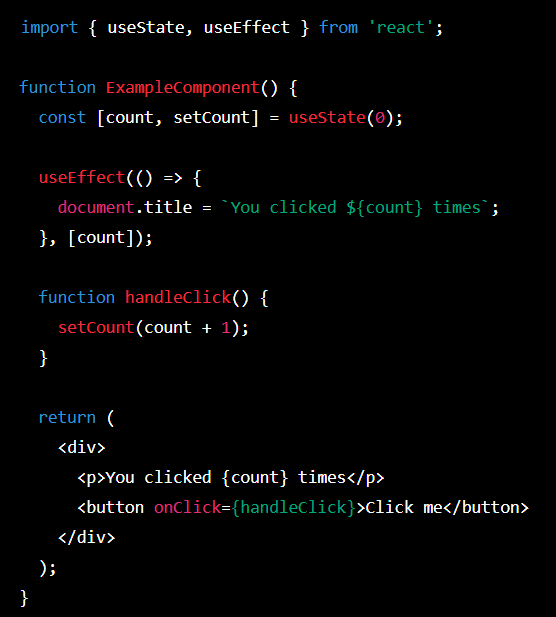
Use, definition and example of the following React:

useState, useEffect, useContext, useReducer, useCallback, useMemo, useRef

1. **useState** - This hook is used to add state to a functional component. It takes an initial state value and returns an array with two items: the current state value and a function to update the state. You can call the update function to change the state value, which will trigger a re-render of the component.

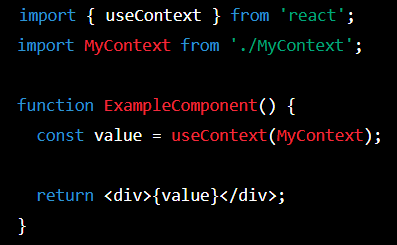
Here's an example:



1. **useEffect** - This hook is used to perform side effects in a functional component, such as fetching data or manipulating the DOM. It takes a function as its first parameter, which will be called after every render. You can also pass a second parameter to control when the effect should be re-run. Here's an example:

In this example, the **useEffect** hook is used to update the document title every time the **count** state value changes.

1. **useContext** - This hook is used to access a context object that was created with the **React.createContext** function. It takes the context object as its parameter and returns the current value of the context. This is useful for sharing state between components without having to pass it down through props. Here's an example:



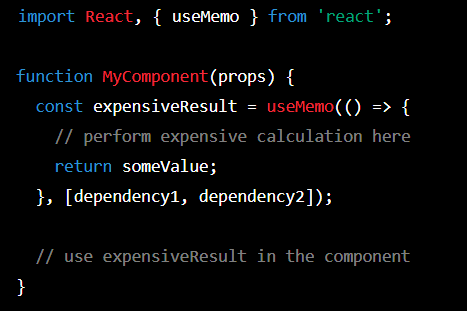
In this example, the **useContext** hook is used to access the current value of the **MyContext** object.

1. Text

   Description automatically generated**useReducer** - This hook is used to manage complex state in a functional component using a reducer function. It takes a reducer function and an initial state value, and returns an array with two items: the current state value and a dispatch function. You can call the dispatch function to update the state, which will trigger a re-render of the component. Here's an example:

In this example, the **useReducer** hook is used to manage the **count** state value with an increment and decrement action.

1. **useMemo** - This hook is used to memoize the result of a function, so that it can be cached and returned instead of recomputed every time the component re-renders. It takes two arguments: the first is the function to be memoized, and the second is an array of dependencies. The memoized value is only recomputed if one of the dependencies changes. Example:



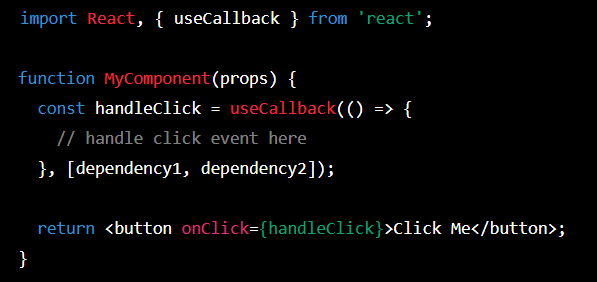
In this example, the **useMemo** hook is used to memoize the result of an expensive calculation, which is only recomputed if **dependency1** or **dependency2** changes.

1. **useRef** - This hook is used to create a mutable value that persists across component renders. It returns an object with a **current** property, which can be used to store and access the value. Example:



In this example, the useRef hook is used to create a reference to an input element, which can be accessed later in the handleClick function to give it focus.

1. **useCallback**- This hook is used to memoize a function, similar to **useMemo**, but for functions instead of values. It takes two arguments: the first is the function to be memoized, and the second is an array of dependencies. The memoized function is only recomputed if one of the dependencies changes. Example:

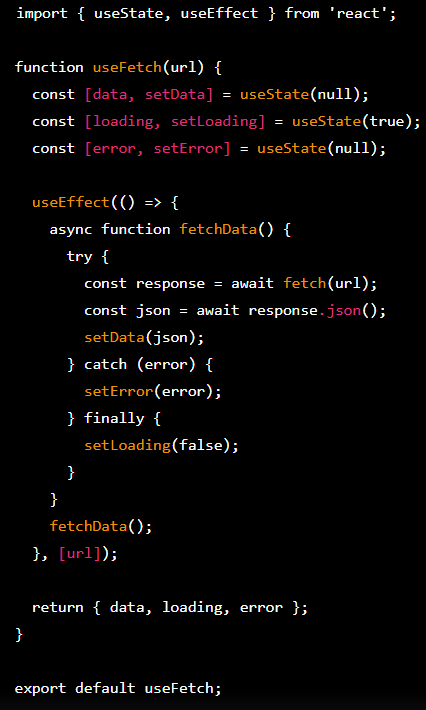


In this example, the useCallback hook is used to memoize a click handler function, which is only recomputed if dependency1 or dependency2 changes. The memoized function is then passed as a prop to a button component.

**What's a custom hook in react, which is it common use and provide an example**

In React, a **custom hook** is a function that allows you to encapsulate reusable logic and stateful behavior so that it can be easily shared between different components. Custom hooks can be used to extract complex logic from components, making them more reusable and easier to understand.

A common use case for custom hooks is to handle fetching data from an API. For example, you might create a custom hook called useFetch that handles making an API request and updating the component state with the fetched data.

Here's an example implementation of a **useFetch** custom hook:

In this example, the useFetch hook takes a URL as an argument and returns an object with data, loading, and error properties. It uses the useState hook to initialize the state for these properties and the useEffect hook to fetch the data from the URL when the component mounts. Once the data is fetched, it updates the state accordingly.

You can then use this custom hook in any component that needs to fetch data from an API, like this:



In this example, MyComponent uses the useFetch custom hook to fetch data from an API and displays the data when it's loaded. The loading and error states are also handled by the hook, making the component more concise and easier to read.

**What's Redux, and the Redux flow? The Redux's actors: reducer, actions, store, dispatch and an example in React with each one**

Redux is a predictable state container for JavaScript applications, commonly used in combination with React to manage the state of an application in a centralized location. It provides a unidirectional data flow, allowing for a more controlled and maintainable architecture.

The Redux flow follows these steps:

**Actions** are dispatched by components when something happens, such as a user interaction.

**The store**, which holds the application state, receives the action.

**The reducer**, which is a pure function, takes the current state and the action as input and returns a new state.

**The store** updates its state to the new state returned by the reducer.

Any connected components are notified of the state change and can update accordingly.

**The key actors in Redux are:**

**Reducers:** These are pure functions that take the current state and an action as input and return a new state. They are responsible for updating the store's state.

**Actions:** These are plain JavaScript objects that represent an event in the application. They contain a type property and any necessary data.

**Store:** This is a centralized location that holds the application's state. It dispatches actions to reducers and notifies connected components of state changes.

**Dispatch:** This is a method on the store that takes an action as input and sends it to the reducers.

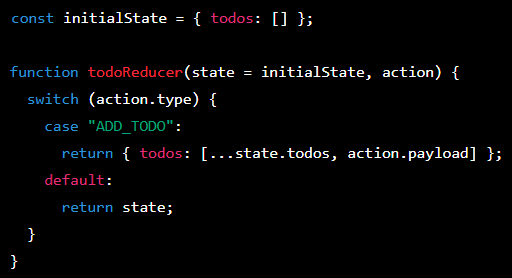
**Here's an example of where you could use Redux:**

Imagine you're building an e-commerce website that has a shopping cart. When a user adds an item to their cart, you want to update the cart's state and display the updated cart total. Without Redux, you might have to pass the cart state and update functions down through several layers of components, making the code more complex and harder to maintain.

With Redux, you can store the cart state in the store and use actions and reducers to update it. When a user adds an item to their cart, a dispatch is triggered, which sends an action to the reducer responsible for updating the cart state. The store updates its state to the new cart state, and any connected components are notified of the change and can update accordingly, displaying the updated cart total. This approach simplifies the code and makes it easier to manage the application's state.

**Here's an example of how you can use Redux with React, including each of the actors:**

**Reducer:** Suppose you have an application that manages a list of todos. The reducer function takes the current state and an action as input and returns a new state. For example, the reducer might look like this:

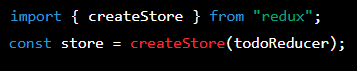


This reducer updates the state by adding a new todo when the **ADD\_TODO** action is dispatched.

Text

Description automatically generated**Actions:** You can create action objects using action creators, which are functions that return plain JavaScript objects. For example, an action creator might look like this:

This action creator returns an action object with a **type** property of **"ADD\_TODO"** and a **payload** property that contains the new todo.

**Store:** You can create a store using the **createStore** function from the Redux library. For example, the store might look like this:

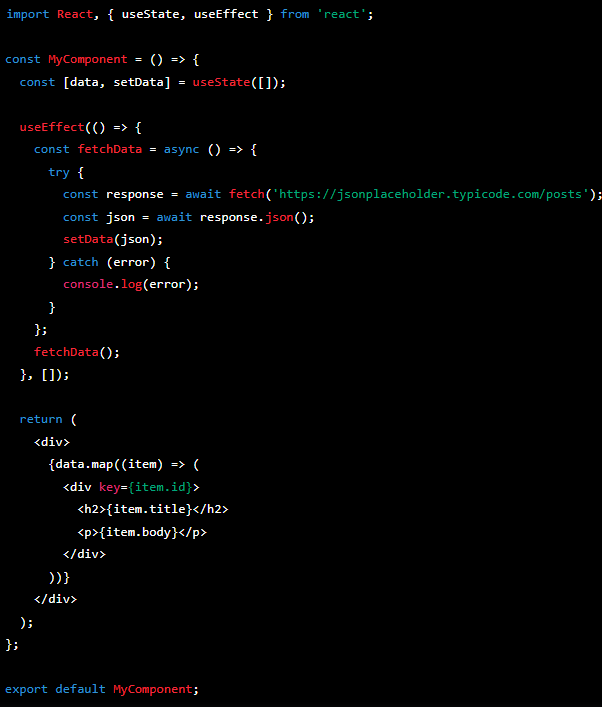
This creates a store that uses the **todoReducer** function to update its state.

**Dispatch:** You can dispatch actions using the **dispatch** method on the store. For example, to dispatch the **addTodoAction** action, you can do this:

This dispatches the **addTodoAction** action with a payload of **"Buy milk"**, which updates the state and triggers any connected components to update.

An example of where you might use Redux with React is in a large-scale e-commerce application that manages complex state, such as user authentication, cart contents, and product listings. Using Redux allows you to centralize this state and manage it more efficiently, reducing the complexity of passing state through multiple layers of components. For example, you might use Redux to manage the state of a user's shopping cart, so that when they add or remove items from their cart, the cart state is updated and displayed consistently across the application.

**Example of a javaScript fetch function, getting the data, parsing it, and render the data in a react component. Also use try catch and error handling.**



In this example, we're using **useState** to create a state variable called **data** which will hold our fetched and parsed data. We're also using the **useEffect** hook to fetch the data when the component mounts. Inside the **fetchData** function, we're using **try/catch** to handle any errors that may occur during the fetch or parsing process. If an error occurs, we log it to the console.

In the **return** statement, we're mapping over the **data** array and rendering a **div** for each item. We're using the **key** prop to assign a unique key to each **div** based on the **id** of the item. We're also displaying the **title** and **body** properties of each item in the **div**.

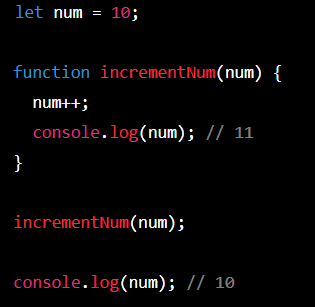
**Whats the difference in passing by value and by reference in JavaScript? Give examples**

In JavaScript, primitive types such as numbers and strings are passed by value, while objects and arrays are passed by reference.

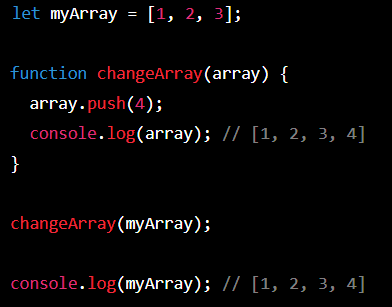
When passing by value, a copy of the value is created and passed to the function. Any changes made to the value inside the function do not affect the original value outside the function.

When passing by reference, a reference to the original object is passed to the function. Any changes made to the object inside the function will affect the original object outside the function.

Here are some examples**, passing by value**:



In this example, the **incrementNum** function takes a copy of the **num** value and increments it inside the function. However, the original value of **num** outside the function is not affected.

**Passing by reference:**

In this example, the **changeArray** function takes a reference to the **myArray** object and modifies it by adding the value **4** to the end of the array. This change is reflected in the original **myArray** object outside the function.

What’s the React state?

In React, the "state" is a JavaScript object that represents the current state of a component. The state object stores data that can change over time and affects how the component is rendered on the screen.

When a component's state changes, React will automatically re-render the component to reflect the new state. This is what allows React to create dynamic, interactive user interfaces.

In React, the useState Hook is a function that allows functional components to have state. The useState Hook takes an initial value as an argument and returns an array with two elements:

1. The current state value
2. A function to update the state value

**What’s props drilling and why Redux helps with that?**

Prop drilling is a problem that occurs in React applications where props need to be passed down through multiple layers of components, even if some intermediate components don't need to use those props. This can make the code more difficult to read, harder to maintain, and can lead to performance issues if there are many levels of nesting.

Redux is a state management library that helps solve the prop drilling problem by providing a centralized store for application state. Instead of passing props down through multiple levels of components, components can simply subscribe to the store and retrieve the data they need directly from it.

With Redux, the state of the application is managed in a single location, and any component can access that state without having to pass props down through intermediate components that don't need them. This can make the code cleaner, easier to understand, and more performant.

In summary, Redux helps with prop drilling by providing a centralized store for application state, which allows components to access the data they need without having to pass props down through multiple levels of components.

**What is hoisting, currying, high order functions, high order components, pure components, and scope and how do you deal with the different types of variables?**

**Hoisting:**

Hoisting is a JavaScript behavior where variable and function declarations are moved to the beginning of their scope before the code is executed. That means, even if a variable or function is declared after its use, JavaScript will move it to the beginning of the scope, and the variable or function will be available throughout the scope.

**Currying:**

Currying is a programming technique where a function that takes multiple arguments is transformed into a series of functions that take a single argument. This allows creating specialized functions from a general function, which can improve code readability and make it more modular.

**High Order Functions:**

High Order Functions are functions that take one or more functions as arguments and/or return a function as a result. They allow abstracting common logic of a set of functions and reusing it in different contexts.

**High Order Components:**

High Order Components are React components that take a component as input and return a new component with certain properties and/or functionalities added. They allow reusing common logic in different React components.

**Pure Components:**

Pure Components are React components that implement the shouldComponentUpdate() method to avoid unnecessary component update when their props and state do not change. This can improve the performance of the application as the cost of re-rendering the component is avoided.

**Scope:**

Scope is the context in which a variable or function is accessible. In JavaScript, there are two types of scope: global scope and local scope. Local scope refers to the scope inside a function, while global scope refers to the scope outside any function.

**Variables:**

In JavaScript, there are three types of variables: var, let, and const. Variables declared with var have a global or local scope depending on where they are declared, while variables declared with let and const have a block scope (local scope) and are only available within the block where they are declared. The difference between let and const is that variables declared with let can be reassigned, while variables declared with const cannot be reassigned.

**What’s lifecycle in React?**

Lifecycle in React refers to the sequence of methods that are invoked at various stages of a component's existence. These methods allow the component to handle different events and perform certain tasks at different stages of its lifecycle.

The React component lifecycle is divided into three main phases: mounting, updating, and unmounting.

1. **Mounting Phase:** In this phase, a component is created and inserted into the DOM. The following methods are invoked in this phase:

* constructor(): This method is called first when a component is created. It is used to initialize the component's state and bind methods to the component.
* render(): This method is called next and it returns the JSX (JavaScript XML) that represents the component.
* componentDidMount(): This method is called after the component has been inserted into the DOM. It is used to perform tasks that require the component to be present in the DOM, such as fetching data from an API.

1. Updating Phase: In this phase, a component receives new props or state and re-renders. The following methods are invoked in this phase:

* static getDerivedStateFromProps(): This method is called when the component receives new props. It is used to update the component's state based on the new props.
* shouldComponentUpdate(): This method is called before the component re-renders. It is used to determine if the component needs to be updated or not. If it returns false, the component will not update.
* render(): This method is called again to re-render the component with the new props or state.
* componentDidUpdate(): This method is called after the component has been updated in the DOM. It is used to perform tasks that require the component to be updated, such as updating a third-party library.

1. **Unmounting Phase:** In this phase, a component is removed from the DOM. The following method is invoked in this phase:

* componentWillUnmount(): This method is called before the component is removed from the DOM. It is used to perform tasks such as cleaning up event listeners or timers that were set up in componentDidMount().

These lifecycle methods allow developers to handle different events and perform certain tasks at different stages of a component's lifecycle, making React a powerful and flexible library for building user interfaces.