### **1.What are Pure Components?**

Pure components are the components which render the same output for the same state and props. In function components, you can achieve these pure components through memoized React.memo() API wrapping around the component. This API prevents unnecessary re-renders by comparing the previous props and new props using shallow comparison. So it will be helpful for performance optimizations.

But at the same time, it won't compare the previous state with the current state because function component itself prevents the unnecessary rendering by default when you set the same state again.

The syntactic representation of memoized components looks like below,

const MemoizedComponent = memo(SomeComponent, arePropsEqual?);

Below is the example of how child component(i.e., EmployeeProfile) prevents re-renders for the same props passed by parent component(i.e.,EmployeeRegForm).

import { memo, useState } from 'react';

const EmployeeProfile = memo(function EmployeeProfile({ name, email }) {

return (<>

<p>Name:{name}</p>

<p>Email: {email}</p>

</>);

});

export default function EmployeeRegForm() {

const [name, setName] = useState('');

const [email, setEmail] = useState('');

return (

<>

<label>

Name: <input value={name} onChange={e => setName(e.target.value)} />

</label>

<label>

Email: <input value={email} onChange={e => setEmail(e.target.value)} />

</label>

<hr/>

<EmployeeProfile name={name}/>

</>

);

}

In the above code, the email prop has not been passed to child component. So there won't be any re-renders for email prop change.

In class components, the components extending *React.PureComponent* instead of *React.Component* become the pure components. When props or state changes, *PureComponent* will do a shallow comparison on both props and state by invoking shouldComponentUpdate() lifecycle method.

Note: React.memo() is a higher-order component.

**2) In React, there are several ways to pass values from one component to another,**

depending on the relationship between the components and the nature of the data being passed. Here are some common approaches:

**Props**: The most common way to pass data from a parent component to a child component is through props. You can pass any data type as a prop, including functions, objects, or primitives.  
jsx  
Copy code  
// ParentComponent.jsx

const ParentComponent = () => {

const data = 'Hello from parent';

return <ChildComponent message={data} />;

};

// ChildComponent.jsx

const ChildComponent = ({ message }) => {

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

};

**Context**: Context provides a way to pass data through the component tree without having to pass props down manually at every level. It's useful for passing data that needs to be accessible by many components at different nesting levels.  
jsx  
Copy code  
// Context.js

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

const DataContext = createContext();

export const useData = () => useContext(DataContext);

export const DataProvider = ({ children }) => {

const data = 'Hello from context';

return <DataContext.Provider value={data}>{children}</DataContext.Provider>;

};

// ParentComponent.jsx

const ParentComponent = () => {

return <ChildComponent />;

};

// ChildComponent.jsx

const ChildComponent = () => {

const data = useData();

return <p>{data}</p>;

};

1. **State Management Libraries (e.g., Redux, Recoil)**: For complex applications, you might consider using state management libraries to manage the state of your application and share data between components.
2. **Callback Functions**: You can pass callback functions as props to child components, allowing child components to update the parent's state or trigger actions in the parent component.

**React Router Params**: If you're using React Router, you can pass values through route parameters.  
jsx  
Copy code  
// App.jsx

import { BrowserRouter as Router, Route } from 'react-router-dom';

const App = () => {

return (

<Router>

<Route path="/:id" component={ChildComponent} />

</Router>

);

};

// ChildComponent.jsx

import { useParams } from 'react-router-dom';

const ChildComponent = () => {

const { id } = useParams();

return <p>Route Param: {id}</p>;

};

These are some common ways to pass values between React components. The choice of method depends on the specific requirements and complexity of your application.

**3)What is simplified in react?**

**4) When to use class components and functional components in React.js?**

### **Functional Components**

* **Modern React Development**: With the introduction of React Hooks in version 16.8, functional components have become the preferred way to write components. Hooks allow functional components to manage state and side effects, which was previously only possible with class components.
* **Simpler Components**: If your component is stateless or only requires minimal state management, functional components are a straightforward and concise choice.
* **Performance**: Functional components are generally lighter and faster since they don't have the overhead of managing the component lifecycle methods that class components do.
* **Readability and Reusability**: Functional components are often easier to read and test. They encourage the use of hooks, which promotes code reuse and better organization.

### **Class Components**

* **Legacy Codebases**: If you’re working on a legacy React project that was built before Hooks were introduced, you’ll likely encounter and possibly need to maintain class components.
* **Complex Lifecycle Management**: Before Hooks, class components were necessary for managing more complex lifecycle events (componentDidMount, componentDidUpdate, componentWillUnmount). However, most of these use cases can now be handled using the useEffect hook in functional components.
* **Inheritance**: While React doesn’t encourage deep inheritance hierarchies, if you have a very specific use case that benefits from class inheritance, class components might be a consideration. However, this is rare.

### **When to Use Each**

* **Use Functional Components**:
  + When you want to leverage modern React features and best practices.
  + When you are building new components or starting a new project.
  + When you need to manage state or lifecycle with Hooks.
  + For smaller, presentational components.
* **Use Class Components**:
  + When you’re maintaining or extending an existing codebase that uses class components.
  + When working with libraries or tools that expect class components (though this is increasingly rare).

**5)The Document Object Model (DOM) -**

* The DOM is fundamental to web development as it bridges HTML and scripting languages like JavaScript.

**6)react virtual dom?**

The Virtual DOM is a lightweight, in-memory representation of the actual DOM elements. It’s essentially a copy or abstraction of the real DOM that React uses to efficiently update the UI.

### How It Works

1. Initial Rendering:
   * When a React component is first rendered, React creates a Virtual DOM tree that mirrors the actual DOM tree of the web page.
2. Re-rendering and Updates:
   * When the state or props of a component change, **React updates the Virtual DOM first, instead of directly manipulating the actual DOM.** This is because direct DOM manipulation can be slow and inefficient, especially with large or complex applications.
3. Diffing Algorithm:
   * **React then compares the updated Virtual DOM with the previous version** using a "diffing" algorithm. This comparison process identifies what has changed in the Virtual DOM.
   * The diffing algorithm is very efficient and quickly determines the minimal number of changes (i.e., what parts of the DOM need to be updated) by comparing the new Virtual DOM with the previous one.
4. Batch Updates:
   * After identifying the changes, **React efficiently updates the real DOM with only the necessary changes, rather than re-rendering the entire UI.** This reduces the number of direct manipulations of the real DOM, leading to better performance.
5. Reconciliation:
   * The process of updating the real DOM based on changes detected in the Virtual DOM is called "reconciliation." React ensures that the actual DOM is updated in a way that is both efficient and keeps the UI consistent with the component's state.

React's diffing algorithm is a key part of its virtual DOM system. The primary goal of the diffing algorithm is to efficiently update the DOM by minimizing the number of changes and re-renderings. Here’s a detailed overview of how it works and how often it compares:

### **How Often React Compares**

* **On State or Props Change**: React compares the virtual DOM trees whenever there’s a change in state or props of a component. This triggers a re-render of the component and a comparison of the new virtual DOM tree with the previous one.
* **On Force Update**: When you explicitly call forceUpdate() on a component, React will re-render that component and compare the new virtual DOM tree with the previous one.

### Benefits of the Virtual DOM

* Performance Optimization: By minimizing direct interactions with the real DOM, which is slow, React optimizes rendering and improves performance.
* Efficient Updates: The Virtual DOM allows React to batch updates and only apply the minimal changes needed to the actual DOM, avoiding unnecessary re-renders.
* Declarative UI: Developers can write their UIs declaratively in React, focusing on how the UI should look in a given state. React handles the efficient updating of the UI, thanks to the Virtual DOM.\

**React** is the most notable framework that uses the Virtual DOM, but other frameworks like **Vue.js**, **Preact**, **Inferno**, and earlier versions of **Svelte** also

**7)Best Practices**

When working with React.js, following best practices can help you build maintainable, efficient, and scalable applications. Here are some key best practices for React development:

### 1. Component Design

* Single Responsibility Principle: Each component should have a **single responsibility or focus. This makes components easier to understand, test, and maintain.**
* Reusable Components: Create r**eusable components to** avoid code duplication. Components should be generic and flexible enough to be reused in different contexts.
* Separation of Concerns: Separate presentational components (UI) from container components (logic). Use functional components for UI and class components or hooks for state and logic.

### 2. State Management

* **Local State: Use local state (useState hook) for simple and isolated component state management.**
* Context API: Use the Context API for managing global state that needs to be accessed by multiple components.
* External State Management Libraries: For complex state management, consider libraries like Redux, MobX, or Zustand.

### 3. Hooks Usage

* Custom Hooks: Create custom hooks for reusable logic across multiple components.
* Avoiding Side Effects in Render: Ensure that side effects (e.g., data fetching, subscriptions) are handled within useEffect or other appropriate lifecycle methods/hooks.

### 4. Performance Optimization

* Memoization: Use React.memo for component memoization and useMemo/useCallback for memoizing values and functions to prevent unnecessary re-renders.
* Code Splitting: Use React's lazy loading and React.lazy to split code and improve load times by loading components only when needed.
* Virtualization: Use libraries like react-window or react-virtualized to efficiently render large lists or tables.

### 5. Component Communication

* Props: Use props to pass data and event handlers between components. Ensure that props are well-documented and validate them using PropTypes or TypeScript.
* Callback Functions: Pass callback functions as props for handling events and interactions from child components.

### 6. Error Handling

* Error Boundaries: Implement error boundaries using componentDidCatch or ErrorBoundary components to catch JavaScript errors anywhere in the component tree.
* Graceful Degradation: Ensure your application can handle errors gracefully and provide user-friendly error messages.

### 7. Styling

* CSS Modules or Styled Components: Use CSS Modules or CSS-in-JS libraries like styled-components for scoped and maintainable styling.
* Avoid Inline Styles: Prefer using CSS classes or styled components over inline styles for better performance and maintainability.

### 8. Testing

* Unit Testing: Write unit tests for components using libraries like Jest and React Testing Library to ensure they work as expected.
* Integration Testing: Test interactions between components and APIs to ensure they integrate correctly.
* End-to-End Testing: Use tools like Cypress or Puppeteer for end-to-end testing to simulate user interactions and validate application behavior.

### 9. Code Quality and Maintainability

* Linting and Formatting: Use ESLint and Prettier to enforce code style and quality rules. This helps in maintaining a consistent codebase.
* Type Checking: Use TypeScript or PropTypes to provide type safety and catch errors early in development.
* Documentation: Document your components, hooks, and utilities to help other developers (or future you) understand and use them effectively.

### 10. Security

* Sanitize User Input: Always sanitize user inputs to prevent cross-site scripting (XSS) attacks.
* Use HTTPS: Ensure your application is served over HTTPS to protect data in transit.

**8)Hooks of React Router-**

**React-Router-** is one of the React libraries that is used for client-side routing. It will let users access pages that are rendered within the router. Since it’s a variation of React Library, it performs with the virtual DOM and the navigation performs from inside of the component, therefor it makes user navigation fast.

React-Router has 4 Hooks APIs.

* useHistory
* useParams
* useLocation
* useRouteMatch

**9)React context API-**

The React Context API is a powerful feature for managing state and passing data through a React application without having to prop-drill (i.e., pass props through multiple layers of components). It allows you to share values between components without having to explicitly pass props through every level of your component tree.

Here's a basic overview of how to use the Context API:

### 1. Create a Context

First, you need to create a context using React.createContext(). This function returns a Context object which includes a Provider and a Consumer.

javascript

Copy code

import React from 'react';

const MyContext = React.createContext();

### 2. Provide the Context

Wrap your component tree with the Provider component of the context. This will make the context available to all components in the tree.

javascript

Copy code

const App = () => {

const contextValue = { user: 'John Doe', loggedIn: true };

return (

<MyContext.Provider value={contextValue}>

<ComponentA />

</MyContext.Provider>

);

};

### 3. Consume the Context

There are several ways to consume context in your components:

#### Using useContext Hook (Functional Components)

javascript

Copy code

import React, { useContext } from 'react';

const ComponentB = () => {

const context = useContext(MyContext);

return (

<div>

User: {context.user}

<br />

Logged In: {context.loggedIn ? 'Yes' : 'No'}

</div>

);

};

#### Using Context.Consumer (Class Components or Functional Components)

javascript

Copy code

import React from 'react';

const ComponentC = () => (

<MyContext.Consumer>

{context => (

<div>

User: {context.user}

<br />

Logged In: {context.loggedIn ? 'Yes' : 'No'}

</div>

)}

</MyContext.Consumer>

);

### 4. Update Context Value

If you need to update the context value (e.g., for managing state), you can include state management in the provider.

javascript

Copy code

const App = () => {

const [user, setUser] = React.useState('John Doe');

const [loggedIn, setLoggedIn] = React.useState(true);

const contextValue = { user, loggedIn, setUser, setLoggedIn };

return (

<MyContext.Provider value={contextValue}>

<ComponentA />

</MyContext.Provider>

);

};

In this example, ComponentA and its descendants can access and update the context value.

### Summary

The Context API is ideal for managing global states like user authentication, theme settings, or any other data that needs to be accessible across many components. It avoids the complexity of prop-drilling and is a great solution for many use cases in React applications.

**10:React class component and its life cycle:**

### Structure of a Class Component

A class component must include a render() method, which returns the JSX to be rendered. Here's a basic example:

import React, { Component } from 'react';

class MyComponent extends Component {

render() {

return (

<div>

<h1>Hello, Class Component!</h1>

</div>

);

}

}

export default MyComponent;

### **Class Component Lifecycle Methods**

These phases include:

* **Mounting** (when the component is first rendered in the DOM)-
* constructor(props)**-**Called before the component is mounted.
* Used to initialize state or bind event handlers.
* **componentDidMount()-**Called immediately after the component is mounted.
* Ideal for performing side effects such as fetching data from an API or setting up subscriptions.
* **Updating** (when the component re-renders due to changes in props or state)
* **Unmounting** (when the component is removed from the DOM)-**componentWillUnmount()**
  + Invoked just before the component is unmounted and destroyed.
  + Ideal for cleaning up (e.g., canceling network requests, removing event listeners).
* **Error Handling** (handling errors in the component tree)-componentDidCatch(error, info)
  + Called when an error occurs in the child components.
  + Useful for logging errors or displaying fallback UI.

**11)React's Re-render Mechanism**

* React components only re-render when there is a change in state or props.
* In your code, roundedNum was declared as a regular variable. Since regular variables do not trigger re-renders when they change, React wasn't aware that it needed to update the UI when the value of roundedNum was modified inside the axios callback.
* **Key Concept:** Only changes to state (using useState or other state management approaches) will trigger React to update the UI.