**What Is react js**

React.js, often simply referred to as React, is an open-source JavaScript library used for building user interfaces or UI components, particularly for single-page applications where the user interface needs to be highly dynamic and responsive. It was developed and is maintained by Facebook.

Key features and concepts of React include:

1. **Component-Based Architecture:** React allows developers to break down the UI into reusable and modular components. Each component manages its own state and can be composed to build complex user interfaces.
2. **Virtual DOM:** React uses a virtual DOM (Document Object Model) to optimize the rendering process. Instead of updating the entire DOM when there is a change, React compares the virtual DOM with the actual DOM and updates only the parts that have changed. This helps improve performance and ensures a smoother user experience.
3. **Declarative Syntax:** React uses a declarative syntax, making it easier to understand and debug. Developers describe the desired UI state, and React takes care of updating the DOM to match that state.
4. **JSX (JavaScript XML):** React uses JSX, a syntax extension that allows mixing HTML-like code within JavaScript. JSX makes it more readable and concise to describe the structure of UI components.
5. **Unidirectional Data Flow:** Data in a React application flows in a unidirectional manner, from parent components to child components. This makes it easier to understand how data changes and updates are propagated through the application.
6. **React Router:** React Router is a popular library for implementing navigation and routing in React applications. It enables the creation of single-page applications with multiple views.
7. **State and Props:** React components can manage their internal state, and data can be passed from parent to child components through props (short for properties). This enables the creation of dynamic and interactive user interfaces.

**Jsx**JSX, or JavaScript XML, is a syntax extension for JavaScript that looks similar to XML or HTML and is used with React to describe the structure of UI components. JSX provides a more concise and readable way to define the structure of user interfaces in React applications.

Here's a basic example of JSX:

const element = <h1>Hello, React!</h1>;

In this example, the <h1> tag is not a string, but a JSX element representing a heading element in the React component. JSX allows you to mix HTML-like syntax within your JavaScript code.

**Key features of JSX:**

**HTML-like Syntax**: JSX looks similar to HTML, making it more familiar and readable for developers. However, it's important to note that JSX is not HTML; it's a syntactic sugar for creating React elements.

**Embedding Expressions:** You can embed JavaScript expressions within curly braces {} inside JSX. This allows dynamic content and the use of variables or expressions within the JSX code.

const name = "John";

const element = <p>Hello, {name}!</p>;

**Attributes:** JSX allows you to use HTML-like attributes to define properties for React elements. These attributes are written in camelCase to match JavaScript conventions.

const element = <input type="text" className="my-input" />;

**JSX Elements:** JSX elements can represent HTML tags, React components, or fragments. They can also be used in expressions, assigned to variables, and passed as arguments to functions.

const MyComponent = () => <p>This is a custom component.</p>;

const element = (

<div>

<h2>Main Title</h2>

<MyComponent />

</div>

);

**No Browser Dependencies**: JSX is not interpreted by browsers. Instead, it needs to be transpiled into standard JavaScript using tools like Babel before being served to browsers.

Here's an example of how JSX might be transformed into JavaScript:

const element = <h1>Hello, React!</h1>;

**Transpiled to:**

const element = React.createElement('h1', null, 'Hello, React!');

The React.createElement function creates a virtual DOM representation of the element, which is used by React to efficiently update the actual DOM when changes occur.

**functional component lifecycle**In React, functional components are a type of component that are defined as JavaScript functions. Until the introduction of React Hooks, functional components were stateless and didn't have lifecycle methods. However, with the advent of React Hooks, functional components can now use state and lifecycle features through hooks.

Here's a brief overview of the lifecycle aspects for functional components with the use of hooks:

**Mounting Phase:**

**useState:** The useState hook allows functional components to have state. It returns an array with two elements: the current state and a function to update that state.

import React, { useState } from 'react';

function MyComponent() {

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

// ...

}

**useEffect (componentDidMount):** The useEffect hook can be used to perform side effects in functional components. When passed an empty dependency array, it behaves similarly to componentDidMount in class components.

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

// This code will run after the component is mounted

return () => {

// Cleanup code (equivalent to componentWillUnmount)

};

}, []); // Empty dependency array means it only runs once, like componentDidMount

}

Updating Phase:

**useEffect (componentDidUpdate):** If useEffect is used without a dependency array, it runs on every render (including updates). This is similar to componentDidUpdate in class components.

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

// This code will run after every render (including updates)

return () => {

// Cleanup code (equivalent to componentWillUnmount)

};

});

}

Unmounting Phase:

**useEffect (componentWillUnmount):** The cleanup function inside useEffect is called when the component is unmounted, providing similar functionality to componentWillUnmount in class components.

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

// This code will run after the component is mounted

return () => {

// Cleanup code (equivalent to componentWillUnmount)

};

}, []);

}

Note that these hooks have made it easier to manage the lifecycle of functional components, and they offer a more straightforward and consistent approach compared to class components. Each effect can be seen as a "mini-lifecycle" for the specific aspect it covers.

**State**Definition: In React, "state" refers to the data that a component maintains and can change over time. It is a way for a component to keep track of information and trigger re-renders when that information changes.

**Usage:** State is typically managed using the useState hook in functional components or through the state property in class components.

// Functional Component with state

import React, { useState } from 'react';

function ExampleComponent() {

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

return (

<div>

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

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

</div>

);

}

**Stateless:**

**Stateless components, also known as functional components, are components defined as JavaScript functions. They don't have state or lifecycle methods. They receive data through props and render UI based on those props.**

**Example of a stateless (functional) component:**

**import React from 'react';**

**const StatelessComponent = (props) => {**

**return (**

**<div>**

**<p>{props.message}</p>**

**</div>**

**);**

**};**

**export default StatelessComponent;**

**Stateful:**

Definition: A "stateful" component, often a class component, is one that manages its own internal state using the state property. It can handle and respond to changes in its state.

Example:

// Stateful (Class) Component

import React, { Component } from 'react';

class StatefulComponent extends Component {

constructor(props) {

super(props);

this.state = {

count: 0

};

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.setState({ count: this.state.count + 1 })}>

Increment

</button>

</div>

);

}

}

**State Management:**

State management in React involves handling and updating the state of your application in a way that is efficient and maintainable. React provides various tools for state management, and the choice often depends on the complexity of the application.

For more complex state management, you might use tools like React Context API, Redux, or Recoil.

Example using React Context API for simple state management:

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

const StateContext = createContext();

const StateProvider = ({ children }) => {

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

const incrementCount = () => {

setCount(count + 1);

};

return (

<StateContext.Provider value={{ count, incrementCount }}>

{children}

</StateContext.Provider>

);

};

const useGlobalState = () => {

return useContext(StateContext);

};  
export { StateProvider, useGlobalState };  
  
  
  
  
  
**Props**In React, "props" is short for properties, and it is a mechanism for passing data from one component to another. Props are a way to make components more dynamic and customizable, allowing parent components to pass information down to their child components.

Here's a basic overview of how props work in React:

**Passing Props:**

Parent components can pass data to child components by using attributes on the child component's JSX tag. These attributes are known as props.

// ParentComponent.js

import React from 'react';

import ChildComponent from './ChildComponent';

const ParentComponent = () => {

const message = "Hello from ParentComponent!";

return <ChildComponent message={message} />;

};

// ChildComponent.js

import React from 'react';

const ChildComponent = (props) => {

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

};

export default ChildComponent;

In this example, the message prop is passed from the ParentComponent to the ChildComponent.

**Accessing Props:**

In a functional component, you can access props by defining them as parameters in the function.

const ChildComponent = (props) => {

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

};

In a class component, you can access props using this.props.

class ChildComponent extends React.Component {

render() {

return <p>{this.props.message}</p>;

}

}

**Default Props:**

You can provide default values for props in case they are not passed by the parent component.

const ChildComponent = (props) => {

return <p>{props.message || "Default Message"}</p>;

};

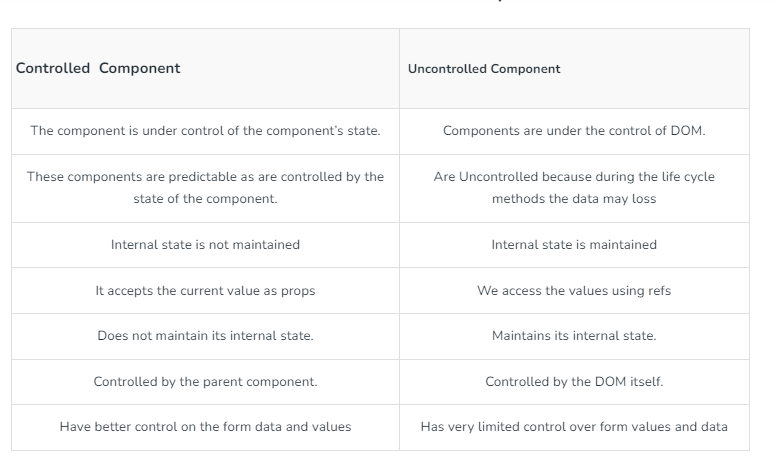
**Destructuring Props:**

To simplify the code, you can use destructuring to extract specific props.

const ChildComponent = ({ message }) => {

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

};

This is equivalent to const message = props.message;.  
 **Controlled and Uncontrolled**

**Controlled**  
import { useState } from "react";

import "./App.css";

**function** App() {

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

**function** handleSubmit() {

        alert(`Name: ${name}`);

    }

**return** (

        <div className="App">

            <h1 className="geeks">GeeksForGeeks</h1>

            <h3>Controlled Component</h3>

            <form onSubmit={handleSubmit}>

                <label>Name:</label>

                <input

                    name="name"

                    value={name}

                    onChange={(e) =>

                        setName(e.target.value)

                    }

                />

                <button type="submit">Submit</button>

            </form>

        </div>

    );

}

export **default** App;

**Uncontrolled**import React, { useRef } from "react";

import "./App.css";

**function** App() {

    const inputRef = useRef(**null**);

**function** handleSubmit() {

        alert(`Name: ${inputRef.current.value}`);

    }

**return** (

        <div className="App">

            <h1 className="geeks">GeeksForGeeks</h1>

            <h3>Uncontrolled Component</h3>

            <form onSubmit={handleSubmit}>

                <label>Name :</label>

                <input

                    type="text"

                    name="name"

                    ref={inputRef}

                />

                <button type="submit">Submit</button>

            </form>

        </div>

    );

}export **default** App;

**Virtual Dom**The Virtual DOM (Document Object Model) is a concept in React that involves an in-memory representation of the actual DOM elements. React uses the Virtual DOM to optimize and speed up the process of updating the user interface.

Here's how the Virtual DOM works in React:

1. **Rendering Components:**
   * When a React component's state or props change, the component re-renders.
2. **Virtual DOM Representation:**
   * Instead of updating the actual DOM directly, React creates a virtual representation of the updated UI in memory.
3. **Diffing Algorithm:**
   * React employs a reconciliation process that involves a "diffing" algorithm. This algorithm compares the new Virtual DOM with the previous one to identify the minimal set of changes needed to update the actual DOM.
4. **Reconciliation:**
   * React calculates the difference (diff) between the new Virtual DOM and the previous one. It determines which parts of the UI need to be updated, added, or removed.
5. **Batch Update:**
   * React batches these changes and updates the actual DOM in an efficient manner. Rather than updating the DOM immediately for each change, React batches the updates for better performance.

The key advantages of using the Virtual DOM in React are:

* **Efficiency:** Manipulating the Virtual DOM is faster than directly manipulating the actual DOM. React can optimize updates and minimize the number of changes needed.
* **Consistency:** The Virtual DOM ensures that the application's UI stays consistent with the underlying data. React components declare how the UI should look based on their current state, and React takes care of updating the DOM accordingly.
* **Abstraction:** Developers can work with the Virtual DOM and declarative programming in React, abstracting away the manual and error-prone manipulation of the actual DOM.

By leveraging the Virtual DOM, React aims to provide a more efficient and performant way of updating the UI, especially in applications with dynamic and frequently changing data. The use of the Virtual DOM contributes to React's reputation for building fast and responsive user interfaces.

**Prop Drilling**Prop drilling, also known as "threading props" or "passing props down the component tree," is a situation in React where data is passed from a higher-level component to a lower-level component through intermediary components in between. This can happen when a component needs to pass data to its descendant components that are not directly nested within it.

Here's an example to illustrate prop drilling:

// GrandparentComponent.js

import React, { useState } from 'react';

import ParentComponent from './ParentComponent';

const GrandparentComponent = () => {

const [data, setData] = useState("Hello from Grandparent!");

return (

<div>

<ParentComponent data={data} />

</div>

);

};

export default GrandparentComponent;

jsx

// ParentComponent.js

import React from 'react';

import ChildComponent from './ChildComponent';

const ParentComponent = ({ data }) => {

return (

<div>

<ChildComponent data={data} />

</div>

);

};

export default ParentComponent;

jsx

Copy code

// ChildComponent.js

import React from 'react';

const ChildComponent = ({ data }) => {

return (

<div>

<p>{data}</p>

</div>

);

};

export default ChildComponent;

In this example:

The GrandparentComponent holds the state data, and it passes that data down to the ParentComponent as a prop.

The ParentComponent then passes the same data down to the ChildComponent.

The ChildComponent receives the data as a prop and renders it.

While this example is simple, in larger component trees, prop drilling can become a challenge:

Complexity: As the component tree grows, it can become challenging to manage and pass data through multiple levels of components.

Maintenance: Changes to the data structure or the need to add new props may require modifications to many components in the hierarchy.

**Advantage in react**React, an open-source JavaScript library for building user interfaces, offers several advantages that contribute to its popularity among developers. Here are some key advantages of using React:

1. **Declarative Syntax:**
   * React uses a declarative syntax, making it easier to understand and reason about your code. Developers can describe the desired state of the UI, and React takes care of updating the DOM to match that state.
2. **Component-Based Architecture:**
   * React promotes a component-based architecture, where UIs are broken down into small, reusable components. This modularity makes code more maintainable, scalable, and encourages a cleaner separation of concerns.
3. **Virtual DOM for Efficiency:**
   * React uses a Virtual DOM to optimize the rendering process. Instead of updating the entire DOM when there is a change, React compares the virtual DOM with the actual DOM and updates only the parts that have changed. This leads to improved performance and a smoother user experience.
4. **Reusability and Composition:**
   * React components can be easily reused throughout an application or shared across different projects. This reusability promotes a consistent design and reduces redundancy in code.
5. **Unidirectional Data Flow:**
   * React follows a unidirectional data flow, making it easier to understand how data changes and updates are propagated through the application. This helps prevent unexpected side effects and makes debugging more manageable.
6. **React Native for Cross-Platform Development:**
   * With React Native, developers can use React to build mobile applications for iOS and Android. The ability to use a single codebase for both platforms, along with the familiarity of React, simplifies cross-platform development.
7. **Rich Ecosystem and Community:**
   * React has a large and active community, resulting in a rich ecosystem of libraries, tools, and resources. This community support ensures that developers have access to a wealth of solutions and best practices.
8. **SEO-Friendly:**
   * React applications can be more SEO-friendly compared to traditional single-page applications. Server-side rendering (SSR) with React allows search engines to index content more effectively.
9. **Strong Developer Tools:**
   * React comes with a set of powerful developer tools (React DevTools) that facilitate debugging, inspecting component hierarchies, and profiling performance.
10. **Gradual Adoption:**
    * React can be gradually adopted into existing projects. Developers can introduce React components incrementally, allowing for a smooth transition and integration with legacy codebases.
11. **React Hooks:**
    * The introduction of React Hooks provides a more elegant way to manage state and side effects in functional components, eliminating the need for class components in many cases.
12. **Context API:**
    * React's Context API allows for efficient management and sharing of state between components without the need for prop drilling, simplifying state management in larger applications.

**Higher order components in react**

Higher-Order Components (HOCs) are a pattern in React that involves the use of functions to enhance or modify the behavior of a component. HOCs are not components themselves but functions that take a component and return a new, enhanced component. They are a way to reuse component logic, add functionality, and create more modular and composable code.

Here's a basic example of a Higher-Order Component:

// HigherOrderComponent.js

import React, { Component } from 'react';

const withLogging = (WrappedComponent) => {

class WithLogging extends Component {

componentDidMount() {

console.log(`Component ${WrappedComponent.name} is mounted.`);

}

componentWillUnmount() {

console.log(`Component ${WrappedComponent.name} is unmounted.`);

}

render() {

return <WrappedComponent {...this.props} />;

}

}

return WithLogging;

};

export default withLogging;

In this example:

withLogging is a higher-order component that takes a component (WrappedComponent) and returns a new component (WithLogging).

WithLogging logs messages when the component is mounted and unmounted.

The enhanced component is then exported.

You can use the HOC to enhance other components:

jsx

Copy code

// MyComponent.js

import React from 'react';

import withLogging from './HigherOrderComponent';

const MyComponent = () => {

return <div>My Component</div>;

};

export default withLogging(MyComponent);

Now, MyComponent has the logging behavior provided by the withLogging HOC.

Common use cases for HOCs include:

Code Reusability: HOCs allow you to encapsulate and reuse logic across multiple components.

Cross-Cutting Concerns: HOCs can encapsulate cross-cutting concerns like logging, authentication, or error handling.

Prop Manipulation: HOCs can modify or add props to the wrapped component.

Conditional Rendering: HOCs can conditionally render components based on certain conditions.

State Abstraction: HOCs can manage state and pass it down to the wrapped component.

While HOCs are a powerful pattern, it's worth noting that React Hooks (introduced in React 16.8) provide an alternative and more concise way to achieve similar functionality, especially with the useEffect and useState hooks. Consider using hooks when possible, but HOCs remain a relevant and valid pattern in React development.

**Routing**

In web development, routing refers to the mechanism that enables navigation between different pages or views in a web application. In React, several libraries can be used for routing, with React Router being the most commonly used one.

Here, I'll provide a basic example using React Router, which is a declarative routing library for React applications.

Installation:

To use React Router, you need to install it first:

npm install react-router-dom

Basic Example with React Router:

jsx

Copy code

// App.js

import React from 'react';

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

import Home from './Home';

import About from './About';

import Contact from './Contact';

const App = () => {

return (

<Router>

<div>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/about">About</Link>

</li>

<li>

<Link to="/contact">Contact</Link>

</li>

</ul>

</nav>

<hr />

<Route exact path="/" component={Home} />

<Route path="/about" component={About} />

<Route path="/contact" component={Contact} />

</div>

</Router>

);

};

export default App;

// Home.js

import React from 'react';

const Home = () => {

return <h2>Home Page</h2>;

};

export default Home;

// About.js

import React from 'react';

const About = () => {

return <h2>About Page</h2>;

};

export default About;

// Contact.js

import React from 'react';

const Contact = () => {

return <h2>Contact Page</h2>;

};

export default Contact;

In this example:

The BrowserRouter component is used to wrap the entire application and provide the routing context.

The Link component is used to create navigation links.

The Route component is used to define the mapping between a URL path and a React component.

Three routes (Home, About, and Contact) are defined with corresponding components.

When a user clicks on a link, React Router updates the UI to render the corresponding component based on the URL path.

This is a basic example, and React Router provides more advanced features like nested routes, route parameters, and programmatic navigation. Depending on your application's complexity, you may need to explore these features for a more sophisticated routing setup.

**Router**In React, a router is a library or tool that enables navigation and handling of different views or pages in a web application. It helps manage the application's URL and allows you to render different components based on the current URL. There are several routing libraries available for React, and one of the most commonly used ones is React Router.

**React Router:**

React Router is a declarative routing library for React applications. It provides a set of components that you can use to define the navigation structure of your application. The two most commonly used components are:

**BrowserRouter:**

BrowserRouter is a component that wraps your application and provides the context for routing. It uses the HTML5 history API to keep the UI in sync with the URL.

jsx

Copy code

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

const App = () => {

return (

<Router>

{/\* Your application components with Route, Link, etc. \*/}

</Router>

);

};

Route:

Route is a component that renders some UI when the URL matches its path.

jsx

Copy code

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

const Home = () => <div>Home Page</div>;

const App = () => {

return (

<Router>

<Route path="/" component={Home} />

</Router>

);

};

Link:

Link is a component that provides navigation links. It renders an anchor (<a>) tag but prevents the default behavior to perform a full-page reload.

jsx

Copy code

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

const Navigation = () => {

return (

<nav>

<Link to="/">Home</Link>

<Link to="/about">About</Link>

</nav>

);

};

Switch:

Switch is a component that renders the first Route or Redirect that matches the current location. It's useful for grouping routes.

jsx

Copy code

import { Switch, Route } from 'react-router-dom';

const App = () => {

return (

<Router>

<Switch>

<Route path="/" exact component={Home} />

<Route path="/about" component={About} />

<Route path="/contact" component={Contact} />

</Switch>

</Router>

);

};

These are just some of the key components provided by React Router. React Router also supports nested routes, route parameters, and other advanced features.

Installation:

To use React Router, you need to install it first:

bash

Copy code

npm install react-router-dom

React Router is widely used in React applications to handle client-side routing, and its documentation is comprehensive and well-maintained, providing detailed information on various features and use cases.

**React router**

React Router is a declarative routing library for React applications that helps manage navigation and view rendering based on the application's URL. It allows you to define a mapping between different URL paths and React components, enabling a single-page application to have multiple views.

Here's a basic overview of React Router:

Installation:

To use React Router, you need to install it using npm or yarn:

bash

Copy code

npm install react-router-dom

or

bash

Copy code

yarn add react-router-dom

Basic Usage:

jsx

Copy code

// App.js

import React from 'react';

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

import Home from './Home';

import About from './About';

import Contact from './Contact';

const App = () => {

return (

<Router>

<div>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/about">About</Link>

</li>

<li>

<Link to="/contact">Contact</Link>

</li>

</ul>

</nav>

<hr />

<Route exact path="/" component={Home} />

<Route path="/about" component={About} />

<Route path="/contact" component={Contact} />

</div>

</Router>

);

};

export default App;

In this example:

The Router component from react-router-dom is used to wrap the application.

The Link component is used to create navigation links.

The Route component is used to define the mapping between a URL path and a React component.

Three routes (Home, About, and Contact) are defined with corresponding components.

When a user clicks on a link, React Router updates the UI to render the corresponding component based on the URL path.

More Advanced Features:

React Router provides more advanced features, including:

Nested Routes: You can nest routes to create a hierarchical structure.

Route Parameters: Extract dynamic values from the URL path.

Programmatic Navigation: Navigate to different routes programmatically.

Switch: Render the first route that matches the current location.

React Router's documentation is comprehensive and provides in-depth information on these features and more. For more complex applications, you might also explore features like server-side rendering (SSR) and code splitting for optimized performance.

**Fragment**

In React, a Fragment is a way to group multiple children elements without adding an extra node to the DOM. It's a lightweight and syntactic sugar for wrapping multiple elements without introducing an additional parent element when rendering.

In React versions prior to 16.2, developers commonly used an empty tag (<> ... </>) to achieve a similar effect. However, with the introduction of the Fragment component in React 16.2, there is now an official and more readable way to accomplish this.

Here's an example of using Fragment:

jsx

Copy code

import React, { Fragment } from 'react';

const MyComponent = () => {

return (

<Fragment>

<h1>Hello</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</Fragment>

);

};

In this example, the Fragment component is used to wrap multiple elements without introducing an additional parent <div> or any other HTML element. When this component is rendered, only the <h1> and <p> elements will be present in the DOM, and there won't be an extra wrapper element.

Alternatively, you can use the shorthand syntax <> ... </> instead of Fragment:

jsx

Copy code

import React from 'react';

const MyComponent = () => {

return (

<>

<h1>Hello</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</>

);

};

Both approaches are functionally equivalent, but using Fragment may be more explicit, especially if you need to pass keys or other props to the fragment.

The use of Fragment is particularly helpful in scenarios where you want to group elements logically without affecting the structure of the rendered HTML. It helps in keeping the DOM clean and avoids unnecessary wrapper elements.

**Hooks**

React Hooks are functions that enable functional components to use state and other React features that were previously only available in class components. They were introduced in React version 16.8 to provide a more concise and expressive way to work with state and side effects in functional components.

Here are some of the most commonly used React Hooks:

**1. useState:**

useState is used to add state to functional components.

import React, { useState } from 'react';

const Counter = () => {

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

return (

<div>

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

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

</div>

);

};

**2. useEffect:**

useEffect is used to perform side effects in functional components, such as data fetching, subscriptions, or manually changing the DOM.

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

const DataFetcher = () => {

const [data, setData] = useState(null);

useEffect(() => {

// Fetch data from an API

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

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

.then(data => setData(data))

.catch(error => console.error('Error fetching data:', error));

}, []); // Empty dependency array means it runs once after the initial render

return (

<div>

<p>Data: {data}</p>

</div>

);

};

**3. useContext:**

useContext allows functional components to consume values from the React context.

jsx

Copy code

import React, { useContext } from 'react';

import MyContext from './MyContext';

const MyComponent = () => {

const contextValue = useContext(MyContext);

return (

<div>

<p>Context Value: {contextValue}</p>

</div>

);

};

**4. useReducer:**

useReducer is a hook for managing more complex state logic in functional components. It's similar to Redux in terms of handling state updates.

import React, { useReducer } from 'react';

const initialState = { count: 0 };

const reducer = (state, action) => {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

return state;

}

};

const CounterWithReducer = () => {

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

return (

<div>

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

<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

<button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

</div>

);

};

**5. useCallback and useMemo:**

useCallback and useMemo are used for performance optimization.

import React, { useState, useCallback, useMemo } from 'react';

const ExpensiveComponent = ({ data }) => {

// Memoized function using useCallback

const handleClick = useCallback(() => {

console.log('Button clicked!');

}, []);

// Memoized result using useMemo

const processedData = useMemo(() => {

console.log('Data processed!');

return data.map(item => item \* 2);

}, [data]);

return (

<div>

<button onClick={handleClick}>Click me</button>

<p>Processed Data: {processedData.join(', ')}</p>

</div>

);

};

React Hooks have become an integral part of React development, offering a more concise and readable way to manage state and side effects in functional components. They have simplified code and reduced the need for class components in many scenarios.

**pages in frontnd**

It looks like there might be a typo in your question, and you might be referring to "pages in frontend." If you're asking about structuring pages in a frontend web application, particularly in the context of a Single Page Application (SPA) or a multi-page application, here's some information:

1. **Single Page Application (SPA):**
   * In SPAs, the application typically consists of a single HTML file, and the content is dynamically loaded as the user interacts with the application.
   * Pages are often represented as components or views that get rendered based on the user's actions, without requiring a full page reload.
   * Frameworks like React, Angular, and Vue are commonly used to build SPAs.
2. **Component-Based Architecture:**
   * Organize your frontend code into modular components. Each component should represent a specific piece of UI or functionality.
   * Components can be reused across different pages or views, improving code maintainability and scalability.
3. **Routing:**
   * Implement client-side routing to manage navigation within your SPA. This allows users to navigate between different "pages" without triggering a full server request.
   * Libraries like React Router, Vue Router, or Angular Router can assist in setting up client-side routing.
4. **State Management:**
   * Use state management libraries (e.g., Redux, Vuex) to manage the application's state globally. This ensures that data can be shared between components seamlessly.
5. **Lazy Loading:**
   * Implement lazy loading for components and resources to improve initial page load times. Only load the resources needed for the current view, and load additional resources as the user navigates.
6. **Responsive Design:**
   * Design pages with responsiveness in mind to ensure a seamless experience across various devices and screen sizes.
7. **CSS Preprocessors and Styling:**
   * Use CSS preprocessors like Sass or Less for better maintainability and organization of styles.
   * Adopt a consistent styling approach, such as BEM (Block Element Modifier) methodology.
8. **SEO Considerations:**
   * If your application requires good search engine optimization (SEO), consider server-side rendering (SSR) or pre-rendering to ensure that search engines can crawl and index your pages effectively.
9. **Testing:**
   * Implement testing strategies for your frontend code, including unit tests, integration tests, and end-to-end tests. This helps catch issues early and ensures code stability.
10. **Accessibility:**
    * Design your pages with accessibility in mind to ensure that the application is usable by people with disabilities.

Remember that the specific architecture and practices may vary based on the framework or library you are using and the requirements of your project. The principles mentioned above are general guidelines that can be applied to many frontend development scenarios.

## **React js limitaions**

**Limited Functionality:** React primarily focuses on the view layer, meaning it handles rendering UI components and managing state. It doesn't offer built-in features for routing, forms, state management, etc. This requires developers to use additional libraries or write custom code, adding complexity and potentially increasing development time.

**Steeper Learning Curve:** Compared to some frameworks, React has a steeper learning curve. Concepts like JSX, virtual DOM, and component lifecycles require a good understanding of JavaScript and can be challenging for beginners.

**Large Ecosystem Dependence:** With its focus on UI, React relies heavily on a vast ecosystem of third-party libraries and tools for functionalities like routing, state management, testing, and more. While this offers flexibility, it can also be overwhelming to choose and manage these dependencies, especially for newcomers.

**Potential Performance Issues:** While generally performant, React applications can encounter performance issues, especially with large amounts of data or complex component hierarchies. Techniques like code splitting and memoization can help, but optimization requires careful consideration.

**JSX Complexity:** While JSX makes writing UI code visually appealing, it can be a barrier for developers unfamiliar with XML-like syntax. It also raises concerns about accessibility and potential security vulnerabilities.

**Lack of Official Documentation:** While React has extensive community-driven documentation, the official documentation can be sparse and lag behind the rapid evolution of the library and surrounding ecosystem. This can lead to difficulties finding reliable information and keeping up with best practices.

**SEO Challenges:** Single-page applications (SPAs) built with React can pose challenges for search engine optimization (SEO). This is because the initial page content may not be readily accessible to search engine crawlers. Techniques like server-side rendering or static site generation can help mitigate this.

It's important to remember that these limitations don't negate React's strengths. Its component-based architecture, virtual DOM, and focus on reusability make it a powerful tool for building dynamic and flexible UIs. However, it's crucial to be aware of these limitations and weigh them against the benefits when deciding if React is the right choice for your project.

React JS is a powerful library for building user interfaces, but it also has some limitations. Here are some of the key ones to consider:

**Limited functionality:**

* Library, not a framework: React focuses solely on the UI layer, meaning it doesn't provide built-in features like routing, state management, or data fetching. You'll need to add external libraries or write custom code for these functionalities.
* Steeper learning curve: While React is praised for its flexibility, its unique concepts like JSX and virtual DOM can be challenging for beginners to grasp.

**Complexity:**

* **JSX complexity:** JSX, an extension of JavaScript for writing HTML-like code, can add complexity to the codebase, especially for developers unfamiliar with it.
* **Choice overload:** The vast ecosystem of libraries and tools can be overwhelming for beginners and lead to decision fatigue. Choosing the right tools requires careful consideration and expertise.

**Performance:**

* **Potential performance issues:** While React can be performant, improper component design or overuse of virtual DOM updates can lead to performance bottlenecks.

Other limitations:

* SEO challenges: Single-page applications built with React can face SEO challenges due to the dynamic nature of the content.
* Documentation gaps: The rapid pace of development can sometimes leave the official documentation lagging behind, making it challenging to find comprehensive information.

Overall, React's limitations shouldn't deter you from using it. Its strengths, like component-based architecture, virtual DOM, and performance optimization potential, make it a valuable tool for building dynamic and user-friendly web applications. However, it's important to be aware of its limitations and plan accordingly.

Here are some additional points to consider:

* The limitations often come with benefits: For example, the lack of a pre-built framework gives you more flexibility to choose the tools and approach that best suit your project.
* The community is constantly improving: The React community is large and active, constantly working to address limitations and improve documentation.
* Many tools and libraries exist to address limitations: For example, libraries like Redux and MobX can help manage state, and React Router handles routing.

Ultimately, the decision of whether to use React JS depends on your specific needs and priorities. Weighing the limitations against its strengths will help you make an informed decision.

**React js features**  
React JS boasts a powerful set of features that make it a popular choice for building user interfaces. Here are some key features to highlight:

Component-based architecture: The core of React revolves around reusable components, each encapsulating a specific UI element and its logic. This promotes code modularity, maintainability, and scalability.

**JSX:** This extension of JavaScript allows you to write HTML-like code within JavaScript files, simplifying UI development and enhancing readability.

**Virtual DOM:** React keeps an internal representation of the UI called the virtual DOM. When data changes, only the necessary parts are updated in the virtual DOM, leading to efficient and performant UI rendering.

**State and props:** Components can have their own state (internal data) and props (external data passed from parent components). This allows for dynamic behavior and interaction within the UI.

One-way data binding: Data flows down from parent to child components, making your application more predictable and easier to debug.

Additional features:

* **React Hooks:** Introduced in React 16.8, Hooks allow you to reuse stateful logic without writing a full-blown class component.
* **Server-side rendering:** React can pre-render content on the server for better SEO and initial page load performance.
* Large ecosystem: A vast library of open-source tools and libraries exist to address various development needs, like routing, state management, and testing.

Benefits of using React JS:

* **Improved developer experience:** The component-based approach and features like JSX and Hooks make coding easier and more enjoyable.
* **Enhanced performance:** Virtual DOM and efficient updates ensure smooth and responsive UIs.
* **Reusable code:** Components save time and effort by allowing code reuse across different parts of your application.
* **Scalability and maintainability**: The modular architecture makes it easier to maintain and scale your application as it grows.

Overall, React JS offers a powerful and flexible set of features that can significantly improve your web development experience. While it has limitations, understanding its strengths and weaknesses can help you decide if it's the right tool for your project.

**how to pass child components to parent components in react js**

In React, data typically flows down the component tree from parent to child components. However, there are scenarios where you might need to pass data or functions from child components to their parent components. One common way to achieve this is by passing callback functions as props.

Here's a simple example:

ChildComponent.js:

import React, { useState } from 'react';

const ChildComponent = ({ onChildButtonClick }) => {

const handleClick = () => {

// Some logic in the child component

// ...

// Call the callback function passed from the parent

onChildButtonClick('Data from child');

};

return (

<button onClick={handleClick}>

Click me in the child component

</button>

);

};

export default ChildComponent;

ParentComponent.js:

import React from 'react';

import ChildComponent from './ChildComponent';

const ParentComponent = () => {

const handleChildButtonClick = (dataFromChild) => {

// Handle the data received from the child component

console.log('Data received from child:', dataFromChild);

// Perform some actions based on the data from the child component

};

return (

<div>

<h1>Parent Component</h1>

{/\* Pass the callback function as a prop to the child component \*/}

<ChildComponent onChildButtonClick={handleChildButtonClick} />

</div>

);

};

export default ParentComponent;

In this example, the ParentComponent has a callback function handleChildButtonClick. It passes this function as a prop called onChildButtonClick to the ChildComponent. When the button in ChildComponent is clicked, it calls the onChildButtonClick callback function, passing data from the child to the parent.

**Lazy loading in React**

Lazy loading in React is a technique used to improve the performance of your application by loading certain parts of your application only when they are needed, rather than loading everything upfront. This is particularly useful when dealing with large applications with many components.

In React, you can achieve lazy loading using React's React.lazy function and the Suspense component. Here's a basic example:

**1. Create a dynamic import for the component you want to lazy load:**

// MyComponent.js

import React from 'react';

const MyComponent = () => {

return (

<div>

<h2>This is the lazy-loaded component!</h2>

</div>

);

};

export default MyComponent;

**2. Use React.lazy to lazily load the component:**

// App.js

import React, { lazy, Suspense } from 'react';

// Use React.lazy to dynamically import the component

const MyComponent = lazy(() => import('./MyComponent'));

const App = () => {

return (

<div>

<h1>My App</h1>

{/\* Wrap the lazily loaded component with Suspense \*/}

<Suspense fallback={<div>Loading...</div>}>

{/\* Render the lazily loaded component \*/}

<MyComponent />

</Suspense>

</div>

);

};

export default App;

In the code above:

React.lazy is used to dynamically import MyComponent. The argument to React.lazy is a function that returns the dynamic import.

Suspense is used to wrap the lazily loaded component. It takes a fallback prop, which is displayed while the lazily loaded component is being loaded.

When the MyComponent is rendered, React will automatically load the component code in the background. If there is a delay in loading, the fallback content will be displayed.

**pros and cons in react js**

It seems like you may have meant "pros and cons" instead of "props and cons." Assuming you're asking about the advantages (pros) and disadvantages (cons) of using React.js, let's explore both aspects:

**Pros of using React.js:**

1. **Component-Based Architecture:** React follows a component-based architecture, making it easier to manage and reuse code. Components encapsulate their own logic and state, promoting a modular and maintainable codebase.
2. **Virtual DOM:** React uses a virtual DOM to optimize rendering performance. Instead of updating the entire DOM when changes occur, React updates a virtual representation first and then efficiently updates only the necessary parts of the actual DOM, reducing unnecessary re-rendering.
3. **Declarative Syntax:** React uses a declarative syntax, allowing developers to describe the desired UI state and let React handle how to achieve it. This makes code more readable and easier to understand.
4. **React Native:** React can be used to develop not only web applications but also mobile applications using React Native. This allows for code reuse between web and mobile platforms, saving development time.
5. **Strong Community Support:** React has a large and active community, providing a wealth of resources, libraries, and tools. This makes it easier to find solutions to common issues and stay up-to-date with best practices.
6. **One-Way Data Binding:** React's one-way data binding simplifies the management of state and ensures a more predictable flow of data within the application.
7. **JSX:** JSX, a syntax extension for JavaScript, enables the mixing of HTML-like code within JavaScript. This makes the code more readable and helps in creating dynamic UI elements.

**Cons of using React.js:**

1. **Learning Curve:** For developers new to React or modern JavaScript concepts, there might be a learning curve. Understanding concepts like JSX, components, state, and props might take some time.
2. **Boilerplate Code:** React applications can sometimes require more boilerplate code compared to other libraries or frameworks. This may lead to more initial setup and potentially increased development time.
3. **Fast-Paced Ecosystem:** The React ecosystem evolves rapidly, which can be both an advantage and a disadvantage. While updates bring new features and improvements, it may also require developers to keep up with the latest changes.
4. **SEO Challenges:** React applications are often single-page applications (SPAs) that dynamically load content. While there are solutions for server-side rendering (SSR), handling SEO requirements may require additional effort.
5. **Complexity for Small Projects:** For simple or static websites, using React might introduce unnecessary complexity. Other frameworks or libraries may be more suitable for small projects with straightforward requirements.
6. **Tooling Choices:** The React ecosystem offers a variety of tools, which can sometimes lead to decision fatigue. Developers need to choose among different state management solutions, routing libraries, and other tools based on project needs.
7. **Verbosity:** JSX, while beneficial, can make code look more verbose, especially for those unfamiliar with the syntax. Some developers may find the mix of HTML and JavaScript challenging at first.

**protected routes in react js**In a React.js application, you might want to implement protected routes to control access to certain parts of your application based on user authentication status. Here's a simple example of how you can implement protected routes using React Router and a basic authentication context.

Let's assume you have a **PrivateRoute** component that you want to protect:

Jsx  
**// PrivateRoute.js**

import React, { useContext } from 'react';

import { Route, Redirect } from 'react-router-dom';

import { AuthContext } from './AuthContext';

const PrivateRoute = ({ component: Component, ...rest }) => {

const { isAuthenticated } = useContext(AuthContext);

return (

<Route

{...rest}

render={(props) =>

isAuthenticated ? (

<Component {...props} />

) : (

<Redirect to="/login" />

)

}

/>

);

};

export default PrivateRoute;  
  
Here, **PrivateRoute** is a higher-order component that takes a **component** prop and some other **rest** props. It checks if the user is authenticated using the **isAuthenticated** state from the **AuthContext**. If the user is authenticated, it renders the protected component; otherwise, it redirects to the login page.

Now, let's create an **AuthContext** to manage the authentication state:

// AuthContext.js

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

export const AuthContext = createContext();

const AuthProvider = ({ children }) => {

const [isAuthenticated, setIsAuthenticated] = useState(false);

const login = () => {

// Implement your authentication logic here

// Set isAuthenticated to true if the user is authenticated

setIsAuthenticated(true);

};

const logout = () => {

// Implement your logout logic here

// Set isAuthenticated to false

setIsAuthenticated(false);

};

return (

<AuthContext.Provider value={{ isAuthenticated, login, logout }}>

{children}

</AuthContext.Provider>

);

};

export default AuthProvider;  
  
  
This context provider contains the **isAuthenticated** state and provides **login** and **logout** functions to update the authentication state.

Now, wrap your **App** component with the **AuthProvider** and use **PrivateRoute** for protected routes:

// App.js

import React from 'react';

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

import AuthProvider from './AuthContext';

import PrivateRoute from './PrivateRoute';

import Home from './Home';

import Login from './Login';

const App = () => {

return (

<AuthProvider>

<Router>

<Switch>

<Route path="/login" component={Login} />

<PrivateRoute path="/home" component={Home} />

{/\* Other public routes \*/}

</Switch>

</Router>

</AuthProvider>

);

};

export default App;  
  
This is a basic example, and you might need to adapt it based on your specific authentication requirements and application structure. Additionally, consider using more advanced state management solutions like Redux for larger applications.

**react js key prop**In React.js, the **key** prop is a special attribute that you use to uniquely identify and differentiate between elements in a collection (such as an array of components) during the process of rendering and updating. The **key** prop helps React optimize the rendering performance and efficiently update the DOM.

Here are a few important points about the **key** prop:

1. **Uniqueness**: Each **key** within a set of sibling elements should be unique. This allows React to efficiently update and reorder elements without unnecessary re-rendering.
2. **Stability**: The **key** should be stable over time. Avoid using dynamic or random values as **key** attributes unless the order or identity of the elements is likely to change. Changing keys between renders can lead to unexpected behavior and negatively impact performance.
3. **Reserved Word**: The **key** prop is reserved in React, and you should not use it for any other purpose than identifying elements in a collection.

Here's an example of using the **key** prop in a simple React component:

**import React from 'react';**

**const MyList = ({ items }) => {**

**return (**

**<ul>**

**{items.map((item) => (**

**<li key={item.id}>{item.name}</li>**

**))}**

**</ul>**

**);**

**};**

**export default MyList;**

n this example, each **li** element has a **key** attribute set to the **id** of the corresponding item. This helps React efficiently update the list when the order or content of items changes.

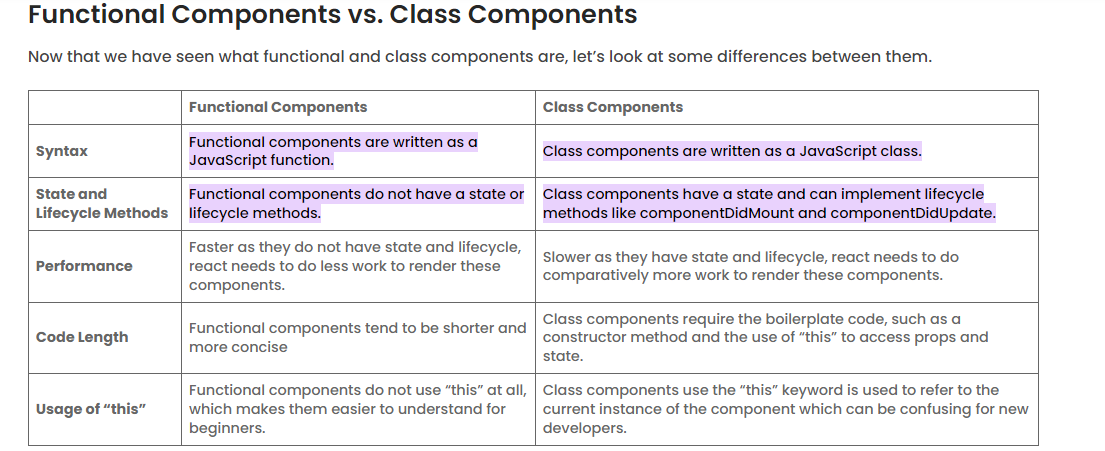
Remember that the **key** prop is primarily used when working with dynamic lists or collections of components. It is not necessary for static content or when rendering a single element.

**in react js why we use classsName not class**In React.js, we use **className** instead of **class** for specifying CSS classes on elements. This is because **class** is a reserved keyword in JavaScript for defining classes in object-oriented programming, so using it directly in JSX would cause conflicts.

JSX is a syntax extension that allows writing HTML-like code within JavaScript, used primarily in React for defining UI components. Since JSX resembles HTML, it uses attributes like **class** to specify CSS classes on elements. However, to avoid conflicts with JavaScript's **class** keyword, React uses **className** instead.

So, when you're writing JSX in a React component, you would write:

<div className="my-class">Hello, World!</div>  
  
  
<div class="my-class">Hello, World!</div>  
  
This helps maintain consistency with HTML syntax while preventing any issues related to JavaScript's



Top of Form

**what is a conditional rendering for functional component**

Conditional rendering in functional components in React allows you to render different JSX based on certain conditions. There are various ways to achieve conditional rendering in functional components:

**Using if-else statements:**

const MyComponent = ({ condition }) => {

if (condition) {

return <div>Condition is true</div>;

} else {

return <div>Condition is false</div>;

}

};

**Using ternary operator:**

const MyComponent = ({ condition }) => {

return (

<div>

{condition ? <div>Condition is true</div> : <div>Condition is false</div>}

</div>

);

};

**Using logical && operator:**

const MyComponent = ({ condition }) => {

return (

<div>

{condition && <div>Condition is true</div>}

</div>

);

};

**Using logical || operator:**

const MyComponent = ({ condition }) => {

return (

<div>

{condition || <div>Condition is false</div>}

</div>

);

};

**Using the Switch statement:**

const MyComponent = ({ value }) => {

switch (value) {

case 'A':

return <div>Value is A</div>;

case 'B':

return <div>Value is B</div>;

default:

return <div>Value is neither A nor B</div>;

}

};

Choose the method that suits your use case best in terms of readability and maintainability. These methods allow you to conditionally render JSX based on the state, props, or any other conditions within your functional components.

**Using the map function for conditional rendering of arrays:**

const MyComponent = ({ items }) => {

return (

<ul>

{items.map(item => (

<li key={item.id}>

{item.condition ? <span>{item.name}</span> : <span>{item.alternativeName}</span>}

</li>

))}

</ul>

);

};

**Using an external function for conditional logic:**

const MyComponent = ({ condition }) => {

const renderContent = () => {

if (condition) {

return <div>Condition is true</div>;

} else {

return <div>Condition is false</div>;

}

};

return <div>{renderContent()}</div>;

};

**Using object literals for conditional rendering:**

const MyComponent = ({ status }) => {

const statusMap = {

'pending': <div>Pending</div>,

'approved': <div>Approved</div>,

'rejected': <div>Rejected</div>

};

return (

<div>

{statusMap[status]}

</div>

);

};

These techniques offer flexibility in how you conditionally render content within your functional components based on different criteria. Depending on your specific use case and preferences, you can choose the most suitable approach for your application.  
  
  
  
  
  
  
  
  
  
**how many lifecycle phases are there in react js**

In React.js, components go through several lifecycle phases. As of my last update in January 2022, React had three main lifecycle phases: Mounting, Updating, and Unmounting. However, with the introduction of React 16.3, some lifecycle methods were deprecated in favor of newer methods based on component classes and function components with hooks. Here's a breakdown:

1. **Mounting:**
   * **constructor()**
   * **static getDerivedStateFromProps()**
   * **render()**
   * **componentDidMount()**
2. **Updating:**
   * **static getDerivedStateFromProps()**
   * **shouldComponentUpdate()**
   * **render()**
   * **getSnapshotBeforeUpdate()**
   * **componentDidUpdate()**
3. **Unmounting:**
   * **componentWillUnmount()**

However, with React 16.3 and later, some lifecycle methods were deprecated in favor of new ones:

* **componentWillMount()** is deprecated. You can use **constructor()** or **componentDidMount()** instead.
* **componentWillReceiveProps()** is deprecated. You can use **static getDerivedStateFromProps()** instead.
* **componentWillUpdate()** is deprecated. You can use **getSnapshotBeforeUpdate()** instead.

Also, with the introduction of Hooks in React 16.8, functional components gained state and lifecycle capabilities. Hooks introduced some new lifecycle methods:

* **useState()**
* **useEffect()**
* **useLayoutEffect()**
* **useContext()**
* **useReducer()**
* **useCallback()**
* **useMemo()**
* **useRef()**
* **useImperativeHandle()**
* **useDebugValue()**

Each of these hooks serves a specific purpose within the component lifecycle or state management.

**[Lifting State Up](https://legacy.reactjs.org/docs/lifting-state-up.html)**[In React, "lifting state up" refers to the practice of moving the state from a component lower in the hierarchy to a higher one. This is typically done when multiple components need to share the same state or when the state needs to be managed by a common ancestor.](https://legacy.reactjs.org/docs/lifting-state-up.html)

[Here's how you can lift state up in React:](https://legacy.reactjs.org/docs/lifting-state-up.html)

1. [Identify the shared state: Determine which components need access to the same piece of state.](https://legacy.reactjs.org/docs/lifting-state-up.html)
2. [Find the common ancestor: Identify the closest common ancestor of these components in the component hierarchy.](https://legacy.reactjs.org/docs/lifting-state-up.html)
3. [Lift the state up: Move the state and the state-modifying functions from the lower-level component to the common ancestor.](https://legacy.reactjs.org/docs/lifting-state-up.html)
4. [Pass state down as props: Pass the state and the state-modifying functions down to the child components as props.](https://legacy.reactjs.org/docs/lifting-state-up.html)

[Here's a simple example to illustrate lifting state up:](https://legacy.reactjs.org/docs/lifting-state-up.html)

[import React, { useState } from 'react';](https://legacy.reactjs.org/docs/lifting-state-up.html)

[// Child component](https://legacy.reactjs.org/docs/lifting-state-up.html)

[const Counter = ({ count, increment }) => {](https://legacy.reactjs.org/docs/lifting-state-up.html)

[return (](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<div>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<p>Count: {count}</p>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<button onClick={increment}>Increment</button>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[</div>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[);](https://legacy.reactjs.org/docs/lifting-state-up.html)

[};](https://legacy.reactjs.org/docs/lifting-state-up.html)

[// Parent component](https://legacy.reactjs.org/docs/lifting-state-up.html)

[const App = () => {](https://legacy.reactjs.org/docs/lifting-state-up.html)

[// State is lifted up to the parent component](https://legacy.reactjs.org/docs/lifting-state-up.html)

[const [count, setCount] = useState(0);](https://legacy.reactjs.org/docs/lifting-state-up.html)

[const increment = () => {](https://legacy.reactjs.org/docs/lifting-state-up.html)

[setCount(count + 1);](https://legacy.reactjs.org/docs/lifting-state-up.html)

[};](https://legacy.reactjs.org/docs/lifting-state-up.html)

[return (](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<div>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<h1>Counter App</h1>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[<Counter count={count} increment={increment} />](https://legacy.reactjs.org/docs/lifting-state-up.html)

[{/\* You can add more child components here that need access to the count state \*/}](https://legacy.reactjs.org/docs/lifting-state-up.html)

[</div>](https://legacy.reactjs.org/docs/lifting-state-up.html)

[);](https://legacy.reactjs.org/docs/lifting-state-up.html)

[};](https://legacy.reactjs.org/docs/lifting-state-up.html)

[export default App;](https://legacy.reactjs.org/docs/lifting-state-up.html)

**diff between in react js useCallback and use Memo**

useCallback and useMemo are both hooks provided by React for optimizing performance in functional components, but they serve different purposes and are used in different scenarios:

**useCallback:**

Purpose: useCallback is used to memoize functions so that they are not recreated on every render unless their dependencies change. It's particularly useful for optimizing the performance of child components that rely on function props, as it prevents unnecessary re-renders caused by passing new function instances to child components.

Syntax: const memoizedCallback = useCallback(callback, dependencies)

Example:

const memoizedCallback = useCallback(() => {

// Function body

}, [dependency1, dependency2]);

**useMemo:**

Purpose: useMemo is used to memoize the result of a computation so that it is only recalculated when its dependencies change. It's useful for optimizing expensive calculations or computations that are repeated frequently within a component.

Syntax: const memoizedValue = useMemo(() => computeExpensiveValue(), dependencies)

Example:

const memoizedValue = useMemo(() => {

// Expensive computation

return computeExpensiveValue();

}, [dependency1, dependency2]);

**React.memo and useMemo**

React.memo and useMemo are both hooks provided by React for optimizing performance in functional components, but they serve different purposes and are used in different contexts:

**React.memo:**

Purpose: React.memo is a higher-order component used to memoize the rendering of a functional component. It works similarly to PureComponent for class components, preventing unnecessary re-renders by memoizing the result of the component rendering based on its props.

Usage: Wrap a functional component with React.memo to memoize its rendering.

const MemoizedComponent = React.memo(function MyComponent(props) {

// Component logic

});

**Benefit**: React.memo improves performance by avoiding re-renders when the component's props haven't changed.

**useMemo:**

**Purpose:** useMemo is a hook used to memoize the result of a computation and recompute it only when its dependencies change. It's useful for optimizing expensive calculations or computations that are repeated within a functional component.

Usage: Call useMemo inside a functional component to memoize a value or computation.

Example:

const memoizedValue = useMemo(() => {

// Expensive computation

return computeExpensiveValue();

}, [dependency1, dependency2]);

**Benefit:** useMemo improves performance by avoiding unnecessary recalculations of a value when its dependencies haven't changed.

**Global object in react js**In React.js, there isn't a concept of a "global object" in the same way as you might find in other programming environments like the browser's window object or Node.js's global object. React.js is designed to encourage a component-based architecture where each component manages its own state and behavior.

However, you can create a global object or state that can be shared among components using various techniques:

Context API: React provides a Context API that allows you to share data across the component tree without having to pass props down manually at every level. You can create a context object and provide it at the top level of your application.

jsx

Copy code

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

// Create a context

const GlobalContext = createContext();

// Create a provider component

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

const [globalState, setGlobalState] = useState(/\* initial state \*/);

return (

<GlobalContext.Provider value={{ globalState, setGlobalState }}>

{children}

</GlobalContext.Provider>

);

};

// Custom hook to consume the context

export const useGlobalState = () => useContext(GlobalContext);

Then, you can wrap your application with the GlobalProvider component at the root level and access the global state using the useGlobalState hook in any component.

Redux: Redux is a popular state management library for JavaScript applications, including React. It provides a global state container that can be accessed by any component in your application. You can create a Redux store and dispatch actions to update the global state.

Custom global object: While not a recommended approach in most cases, you can create a JavaScript module that exports a global object and import it wherever you need it in your components. However, be cautious with this approach as it can lead to tightly coupled and harder-to-manage code.

Each of these methods has its own use cases and trade-offs, so choose the one that best fits the needs and complexity of your application.