AIMERZ.ai
Aim.Act.Achieve

React

INTERVIEW

Questions & Answers

1. What is React?

React is a popular open-source JavaScript library used for building user interfaces, particularly single-page applications where data changes over time. Developed and maintained by Facebook, React allows developers to create large web applications that can update and render efficiently in response to data changes.

2. What are the advantages of using React?

- **Reusable Components:** You can build small pieces of a website (components) and reuse them, making it easier to manage and build your site.
- **Virtual DOM:** React uses a virtual DOM to efficiently update the actual DOM. When the state of an object changes, React updates the virtual DOM first and then calculates the most efficient way to update the real DOM. This results in improved performance and a smoother user experience.
- **Easy to Understand:** You write how you want your web page to look, and React makes it happen. It's straightforward and predictable.
- **JSX:** JSX, a syntax extension for JavaScript, allows developers to write HTML-like code within JavaScript. This makes the code more readable and easier to understand. JSX also provides the full power of JavaScript, allowing for dynamic content generation.
- **Unidirectional Data Flow:** React follows a unidirectional data flow, meaning data flows in one direction from parent to child components.This makes it easier to understand and debug applications.

3. What are the disadvantages of using React?

- **Steep Learning Curve:** React's ecosystem is vast, with many concepts like JSX, state management, and component lifecycle to learn, which can be challenging for beginners.
- **Rapid Changes:** React and its ecosystem change quickly, with frequent updates and new tools emerging. Keeping up with these changes can be time-consuming.

- **JSX Complexity:** While JSX is powerful, it can be confusing for new developers who are not familiar with the mix of HTML and JavaScript.
- **Library, Not a Framework:** React is just a library for building UIs. You may need to integrate additional libraries for state management (like Redux), routing (like React Router), and other functionalities, which can complicate the setup.
- **SEO Challenges:** React applications can face challenges with search engine optimization (SEO) because content is dynamically rendered. Techniques like server-side rendering (SSR) or static site generation (SSG) can help but add complexity.
- **Boilerplate Code:** Setting up a React project often requires a lot of initial configuration and boilerplate code, which can be cumbersome.
- **Performance Issues:** While React is generally fast, improper handling of state and re-renders can lead to performance issues if not managed carefully.

4. What is a React component?

A React component is a fundamental building block in a React application. Components are self-contained modules that represent a part of the user interface. They can be as simple as a button or as complex as an entire page. Components allow you to break down the UI into smaller, reusable pieces, making it easier to build and maintain complex applications.

There are two main types of React components:

- **Functional Components:** These are JavaScript functions that return JSX. They are simple and primarily used for rendering UI.

```javascript
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- **Class Components :** These are ES6 classes that extend from React.Component and have a render method that returns JSX. Class components can manage their own state and lifecycle methods.

```javascript
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

5. Explain the Virtual DOM in React ?

The Virtual DOM in React acts like a special helper that makes updating the web page faster and more efficient.

- When you first build your web page with React, it creates a virtual copy of the real web page structure, but this copy is just in memory, not actually displayed on the screen.
- When something changes on your webpage (like a button is clicked, or new data arrives), React updates the virtual copy first, not the real page.
- React then looks at the old virtual copy and the new one, and figures out exactly what has changed.
- Instead of updating the whole real webpage, React only updates the parts that have changed. This makes the updates faster and smoother.

6. What is JSX ?

JSX is a way to write HTML directly inside JavaScript in React. It makes it easier to design your web page by letting you mix HTML and JavaScript together. JSX looks like HTML, but it's actually JavaScript, and it helps you build user interfaces more easily.

```javascript
const name = 'Prabir';
const element = <h1>Hello, {name}!</h1>;
```

7. How do you handle events in React ?

Handling events in React is similar to handling events in regular HTML, but with some differences due to React's syntax and approach.

a. **Use CamelCase for Event Names:** In React, event names are written in camelCase, so instead of onclick, you use onClick.

```javascript
<button onClick={handleClick}>Click Me</button>
```

b. **Pass a Function as the Event Handler:** You provide a function as the event handler. This function gets called whenever the event occurs.

```javascript
function handleClick() {
  alert('Button was clicked!');
}
```

c. **Using Inline Event Handlers:** You can also define the event handler directly in the JSX if it's a simple function.

```javascript
<button onClick={() => alert('Button was clicked!')}>Click Me</button>
```

d. **Accessing Event Object:** In the event handler, you can access the event object just like in plain JavaScript. React passes the event object as the first argument to the function.

```javascript
function handleClick(event) {
  console.log(event.target);
}
```

8. What are the differences between functional and class components?

Functional and class components are two ways to define components in React, each with its own characteristics.

Functional Components:

- Functional components are plain JavaScript functions that return JSX.

```javascript
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- Originally, functional components were stateless and didn't have access to lifecycle methods. However, with the introduction of React Hooks (e.g., useState, useEffect), functional components can now manage state and handle side effects.
- Functional components are simpler and easier to write, especially for components that don't need state or lifecycle methods.
- Functional components are generally more lightweight and can be more performant, especially when using React's memoization features like React.memo to prevent unnecessary re-renders.
- In functional components, you don't need to worry about this keyword, which simplifies the code.

Class Components

- Class components are ES6 classes that extend React.Component and must include a render method that returns JSX.

```javascript
class Welcome extends React.Component {
```

```
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- Class components have built-in support for state and lifecycle methods (e.g., componentDidMount, componentDidUpdate), making them a go-to for more complex components that need these features.
- Class components can be more complex to write and understand, particularly because of the need to manage this binding for event handlers and other methods.
- While class components can be more feature-rich, they may be slightly heavier in terms of performance compared to functional components, especially for simple, stateless components.
- In class components, you often need to handle the this keyword, which can lead to more boilerplate code (e.g., binding this in constructors).

9. What are props in React?

In React, props (short for "properties") are a way to pass data from one component to another. They are used to give components information they need to render content or perform actions.

- **Passed from Parent to Child:** Props are passed from a parent component to a child component. The parent component provides the data, and the child component receives it as an argument.

JavaScript

```
function Parent() {
  return <Child name="Prabir" />;
}

function Child(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

- **Read-Only:** Props are immutable, meaning that once they are passed to a child component, they cannot be modified by that component. This helps keep the data flow in one direction and makes the app easier to manage.

```javascript
function Child(props) {
  // props.name = "Pradeep"; // This will cause an error
  return <h1>Hello, {props.name}!</h1>;
}
```

- **Dynamic Content**: Props can be used to pass dynamic data, functions, or even other components, making your UI flexible and reusable.

```javascript
function Parent() {
  const name = "Prabir";
  return <Child name={name} />;
}
```

- **Default Props:** You can define default values for props in case they are not provided by the parent component.

```javascript
function Child(props) {
  return <h1>Hello, {props.name}!</h1>;
}

Child.defaultProps = {
  name: "Guest"
};
```

10. What is the difference between React state vs props

State	Props
State is used to manage data that can change over time within a component. It is primarily used for dynamic, interactive components where the data or the UI needs to change based on user actions or other events.	Props (short for properties) are used to pass data and event handlers from a parent component to a child component. Props are used to provide information to a component, but they are not intended to be changed by the component receiving them.
State is mutable, meaning it can be	Props are immutable. Once passed to a

changed by the component that owns it. This is done using the **useState hook**	component, they cannot be modified by that component. This immutability ensures a one-way data flow and helps keep components predictable.
State is owned and managed within the component itself. Each component can have its own state, and it controls how and when that state is updated.	Props are passed to a component by its parent. The child component has no control over the props it receives and can only use them as provided.
State is used for data that is local to the component and may change over time, such as form input values, toggles, or counters.	Props are used to pass data and functions from parent to child components, making them a mechanism for communication between components.

Ex:

```javascript
function Parent() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <Child count={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

function Child(props) {
  return <h1>{props.count}</h1>;
}
```

11. What is a stateful component in React?

A stateful component in React is a component that maintains and manages its own internal state. This means the component can track data that can change over time and use that data to control what gets rendered on the screen.

- A stateful component holds data in its state object and updates this data using functions like the useState hook .
- When the state changes, the component automatically re-renders to reflect the updated data.

```javascript
function Counter() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

12. How useState works?

useState is a React hook that lets you add state (like variables) to functional components.

- You use useState to create a piece of state, starting with an initial value. It gives you two things: **the current state value** and **a function to change it.**

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

Here, count is the state, and it starts at 0. setCount is the function to update count.

- When you want to change the state, you call the update function (setCount) with the new value.

```javascript
setCount(count + 1);
```

- When you update the state, React re-renders the component so the changes show up on the screen.
- You can use useState as many times as needed to manage different pieces of state in the same component.

13. What is lifting state up in React?

In React, lifting state up refers to the pattern of moving state from a child component to a common parent component, so that multiple child components can share and synchronize the same state.

Why use Lift State Up?

When you have two or more components that need to share the same state or need to interact based on a common state, you lift the state up to the closest common ancestor (parent component) of those components.
This allows the parent component to manage the state, and it can then pass the relevant state data and functions to update that state down to the child components via props.

How It Works:

a. **Identify the Shared State:**
   First, determine which piece of state needs to be shared across multiple components.
b. **Move State to Parent Component:**
   Move the state and the function to update the state to the nearest common parent component of the components that need to share the state.
c. **Pass State and Update Function via Props:**
   Pass the state and the update function down to the child components as props. The child components can then use the props to display the state or trigger changes.

```javascript
function ParentComponent() {
  const [sharedValue, setSharedValue] = useState('');

  return (
    <div>
      <ChildA value={sharedValue} onChange={setSharedValue} />
      <ChildB value={sharedValue} />
    </div>
  );
}

function ChildA({ value, onChange }) {
  return (
    <input
      type="text"
      value={value}
      onChange={(e) => onChange(e.target.value)}
    />
  );
}

function ChildB({ value }) {
  return <p>Shared Value: {value}</p>;
}
```

14. What is the use of hooks in React?

Hooks in React are special functions that allow you to use state and other React features in functional components, which were previously only available in class components. Hooks simplify and enhance how you write React components.

Main Uses of Hooks:

- **State Management:**
  useState: Allows you to add state to functional components.

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

- **useEffect:** Handles side effects like data fetching, subscriptions, or manually changing the DOM. It runs after the component renders.

```javascript
useEffect(() => {
  document.title = `Count: ${count}`;
}, [count]);
```

- **Context:**
  useContext: Allows you to use context to share data across components without prop drilling (passing props down manually through every level of the component tree).

```javascript
const user = useContext(UserContext);
```

- **Memoization:**

  useMemo: Memoizes expensive calculations, re-computing them only when necessary.
  useCallback: Memoizes functions, preventing unnecessary re-creations.

```javascript
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- **Custom Hooks:**

You can create your own custom hooks to extract and reuse logic across different components.

```javascript
function useCustomHook() {
  // Custom logic here
}
```

15. What is the use of the useEffect hook in react?

The useEffect hook in React is used to handle side effects, such as fetching data, updating the DOM, or setting up subscriptions, in functional components.

a. **Fetching data:** Runs code to fetch data from an API when the component mounts.

```javascript
useEffect(() => {
  fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => setData(data));
}, []);
```

b. **Updating the DOM:** Updates the webpage title or other DOM elements when state or props change.

```javascript
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);
```

16. What are props and how does it work ?

Props in React are short for "properties." They are a way to pass data from one component to another, specifically from a parent component to a child component. Props make components more dynamic and reusable by allowing them to receive different data inputs.

How it Works

a. **Passing props:** In a parent component, you pass props to a child component by including them as attributes.

```javascript
function Parent() {
  return <Child name="Prabir " age={30} />;
}
```

b. **Receiving Props:**The child component receives these props as a single object and can use them to display dynamic content or perform actions.

```javascript
function Child(props) {
  return <p>Name: {props.name}, Age: {props.age}</p>;
}
```

c. **Using Props:**
Inside the child component, you access the props via props.propName. Props are read-only in the child component, meaning they cannot be modified by the component receiving them.

d. **Props in Functional Components:**
Props can be passed directly as arguments to the function, which is a common pattern.

```javascript
function Child({ name, age }) {
  return <p>Name: {name}, Age: {age}</p>;
}
```

e. **Default Props**:You can set default values for props in case they are not provided by the parent component.

```javascript
Child.defaultProps = {
  name: 'Unknown',
  age: 0
};
```

17. What is prop drilling?

Prop drilling occurs when data is passed from a parent component to its children, and then from those children to their own children, and so on, until the data reaches the component that needs it. This can result in a long chain of component dependencies that can be difficult to manage and maintain.

Problems with Prop Drilling

- **Code Complexity:** As the number of components increases, the code becomes more complex and harder to follow.
- **Reduced Maintainability:** Making changes to the data flow requires updates in many components, making the code harder to maintain.
- **Performance Overhead:** Passing props through unnecessary intermediary components can affect performance.
- **Decreased Component Reusability:** Components become tightly coupled to the structure of the parent components, making them harder to reuse.
- **Increased Development Time:** Implementing and managing prop drilling can slow down development, especially in complex component hierarchies.

Let us take an example to understand better

```javascript
import React, { useState } from 'react';

function Parent() {
  const [data, setData] = useState("Hello from Parent");

  return (
    <div>
      <h1>Parent Component</h1>
      <ChildA data={data} />
```

```
    </div>
  );
}

function ChildA({ data }) {
  return (
    <div>
      <h2>Child A Component</h2>
      <ChildB data={data} />
    </div>
  );
}

function ChildB({ data }) {
  return (
    <div>
      <h3>Child B Component</h3>
      <ChildC data={data} />
    </div>
  );
}

function ChildC({ data }) {
  return (
    <div>
      <h4>Child C Component</h4>
      <p>{data}</p>
    </div>
  );
}

export default Parent;
```

JavaScript

```
Parent
  └── ChildA
        └── ChildB
              └── ChildC
```

In this example, the data prop is passed from the Parent component down through ChildA and ChildB to ChildC, even though ChildA and ChildB do not use the data prop themselves.

18. What is Context API?

The **Context API** in React is a tool that allows you to share data (state, functions, etc.) across multiple components without having to pass props manually at every level of the component tree. It helps solve the problem of **prop drilling**, where props need to be passed through several layers of components even if only one deeply nested component needs the data.

When to Use Context API

The Context API is particularly useful in situations where:

- You have data that needs to be accessed by many components at different levels of the component tree.
- You want to avoid prop drilling, which is the process of passing data through multiple layers of components that do not need the data themselves.

How to use the Context API

Using the Context API involves three main steps: creating a context, providing the context, and consuming the context.

a. **Creating a context:**
   First, you create a context using the createContext function from React. This context will hold the data you want to share.

Mycontext.jsx

```
JavaScript
import React, { createContext } from 'react';
```

```
// Create a Context
const MyContext = createContext();
```

b. Providing the Context

Next, you use the Provider component to wrap the part of your component tree that needs access to the context. The Provider component takes a value prop, which is the data you want to share.

JavaScript

```
import React, { useState } from 'react';
import { MyContext } from './MyContext';

const MyProvider = ({ children }) => {
  const [value, setValue] = useState('Hello, World!');

  return (
    <MyContext.Provider value={{ value, setValue }}>
      {children}
    </MyContext.Provider>
  );
};

export default MyProvider;
```

c. Consuming the Context

Finally, you consume the context in any component that needs access to the shared data. This can be done using the useContext hook in functional components

JavaScript

```
import React, { useContext } from 'react';
import { MyContext } from './MyContext';

const MyComponent = () => {
  const { value, setValue } = useContext(MyContext);

  return (
    <div>
```

```
      <p>{value}</p>
      <button onClick={() => setValue('New Value')}>Change Value</button>
    </div>
  );
};

export default MyComponent;
```

Let us take an example to understand better

Create a theme toggler application where the user can switch between light and dark themes.

- **Create the Context**

```javascript
// ThemeContext.jsx
import React, { createContext, useState } from 'react';

const ThemeContext = createContext();

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export { ThemeProvider, ThemeContext };
```

- **Create a Theme Toggle Button**

Next, we'll create a button component that toggles the theme when clicked.

```javascript
// ThemeToggleButton.jsx
import React, { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

const ThemeToggleButton = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button onClick={toggleTheme}>
      Switch to {theme === 'light' ? 'dark' : 'light'} theme
    </button>
  );
};

export default ThemeToggleButton;
```

- **Create a Themed Component**

We'll create a component that changes its style based on the current theme.

```javascript
// ThemedComponent.jsx
import React, { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

const ThemedComponent = () => {
  const { theme } = useContext(ThemeContext);

  const styles = {
    backgroundColor: theme === 'light' ? '#fff' : '#333',
    color: theme === 'light' ? '#000' : '#fff',
    padding: '20px',
    textAlign: 'center',
  };

  return <div style={styles}>This is a {theme} themed
component!</div>;
};

export default ThemedComponent;
```

Finally, we'll combine everything in the App component and wrap it with the ThemeProvider.

```JavaScript
// App.jsx
import React from 'react';
import { ThemeProvider } from './ThemeContext';
import ThemeToggleButton from './ThemeToggleButton';
import ThemedComponent from './ThemedComponent';

const App = () => {
  return (
    <ThemeProvider>
      <div>
        <h1>Theme Toggler App</h1>
        <ThemeToggleButton />
        <ThemedComponent />
      </div>
    </ThemeProvider>
  );
};

export default App;
```


19. What are React hooks?

React Hooks are functions that let you use React features like state and lifecycle methods in functional components. Before hooks were introduced, these features were only available in class components. Hooks make functional components more powerful and allow you to write cleaner, more reusable code.

Ex: useState, useeffect, usecontext

20. What is React router ?

React Router is a standard library for routing in React applications. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL. This is essential for creating single-page applications (SPAs) where different views are rendered based on the URL without requiring a full page reload.

Features

- **Client-Side Routing:**
  React Router allows your app to update the URL and render new UI components without making a request to the server for a new document. This results in faster transitions and a more dynamic user experience
- **Nested Routing:**
  It supports nested routes, which means you can define routes within routes. This helps in organizing the application better by coupling URL segments with component hierarchy and data.
- **Dynamic Routing**
  React Router allows for dynamic routing, meaning routes can be created based on the data or user actions, making the application more flexible and powerful.
- **Declarative Routing:**
  Routes are defined declaratively using JSX, making the code more readable and easier to manage.

And many more.

21. What are higher-order-components(HOCs)

Higher-Order Components (HOCs) are a pattern in React used for reusing component logic. They are functions that take a component as an argument and return a new component with additional props or behavior.

Imagine you have a cake, and you want to add some frosting and decorations on it to make it look better. You don't want to change the cake itself, but you want to enhance it. Similarly, HOCs "decorate" or "enhance" a component by adding extra functionality without changing the original component.

How It Works

- **Original Component:** You start with a simple React component.

- **HOC Function:** You pass this component to a function (HOC) that adds some extra features.
- **Enhanced Component:** The HOC returns a new component with the added features.

Example:
Let's say you have a component that displays a user's name:

```javascript
function UserProfile({ name }) {
  return <div>User: {name}</div>;
}
```

Now, you want to add a feature that logs the user's name whenever the component renders. Instead of modifying the UserProfile component, you can create an HOC to handle this:

```javascript
function withLogging(WrappedComponent) {
  return function EnhancedComponent(props) {
    console.log(`Rendering user: ${props.name}`);
    return <WrappedComponent {...props} />;
  };
}
```

Here's how you would use this HOC:

```javascript
const UserProfileWithLogging = withLogging(UserProfile);
```

Now, when you use UserProfileWithLogging, it will log the user's name every time it renders, while still displaying the user's name as before:

```javascript
<UserProfileWithLogging name="Vishwa" />
```

Why use HOC

- **Code Reusability:** HOCs allow you to reuse logic across multiple components.
- **Separation of Concerns:** They help keep components clean and focused on their core responsibility, while additional features are added through HOCs.
- **Scalability:** As your application grows, HOCs make it easier to manage and maintain shared logic.

22. What are react Fragments?

React Fragments are a feature in React that allows you to group multiple elements without adding extra nodes to the DOM. This is useful when you want to return multiple elements from a component but don't want to clutter the DOM with unnecessary wrappers like .

Imagine you have a box, and inside that box, you have three smaller items. If you use a regular wrapper, it's like putting those three items in another smaller box before putting them in the big box. With React Fragments, it's like placing the items directly in the big box without any extra wrapping, making it neater

Why Use Fragments?

Avoid Extra Markup: Often, you might need to return multiple elements, and without Fragments, you'd have to wrap them in a `div` or another container. This adds unnecessary elements to the DOM, which can lead to layout issues and makes your code less clean.

Cleaner DOM Structure: Fragments help maintain a cleaner, more efficient DOM.

How It Works

You can use React Fragments in two main ways:

1. **Using the `<React.Fragment>` Tag:**

```javascript
function MyComponent() {
  return (
    <React.Fragment>
      <h1>Title</h1>
      <p>This is a paragraph.</p>
    </React.Fragment>
  );
}
```

2. Using the Short Syntax:

React also provides a shorthand syntax using empty tags <> and </>:

```javascript
function MyComponent() {
  return (
    <>
      <h1>Title</h1>
      <p>This is a paragraph.</p>
    </>
  );
}
```

Both examples will render the <h1> and <p> elements side by side in the DOM without any additional wrapping element.

When to use fragments

- **Returning Multiple Elements:** When your component needs to return multiple elements at the same level.
- **List Rendering:** When mapping over an array of items to render a list, Fragments can be handy to avoid extra `div`s or `li`s.
- **Avoiding Layout Issues:** If adding extra elements would cause layout or styling issues, Fragments are a good solution.

Example Without Fragments

```javascript
function MyComponent() {
  return (
    <div>
      <h1>Title</h1>
      <p>This is a paragraph.</p>
    </div>
  );
}
```

This would add an extra div to the DOM, which may not be needed.

Example With Fragments

```javascript
function MyComponent() {
  return (
    <>
      <h1>Title</h1>
      <p>This is a paragraph.</p>
    </>
  );
}
```

This achieves the same visual output without adding any unnecessary elements to the DOM.

React Fragments are a useful tool to keep your component structure clean and efficient, especially when dealing with multiple elements.

23. Differentiate between stateless and stateful components?

In React, components can be classified as either stateless or stateful based on whether they manage state.

Stateless Components (Functional Components)

- Stateless components, also known as functional components, do not manage any state internally. They are primarily used for rendering UI and rely on the props passed to them.
- They focus on receiving data through props and rendering it accordingly. They are simple, predictable, and easier to test because they don't have side effects or manage state.
- **Syntax:** Stateless components are typically written as functions.

Example:

```javascript
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Here, Greeting is a stateless component because it doesn't hold or manage any state. It just displays the name prop passed to it.

Stateful Components (Class Components or Functional Components with Hooks)

- Stateful components are components that manage their own state internally. They can change their state over time, typically in response to user interactions, and re-render themselves based on the updated state.
- They are used when a component needs to manage data or behavior that changes over time, such as user input, toggling UI elements, or fetching data.
- Stateful components were traditionally written as class components, but with the introduction of React Hooks, functional components can also be stateful.

Example with Class Component:

```javascript
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Example with Functional Component (Using Hooks):

```javascript
import React, { useState } from 'react';

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

feature	Stateless Component	Stateful Component
State Management	Does not hold or manage state.	Can hold or manage state.
State Awareness	Lacks knowledge of past, current, and future state changes.	Contains knowledge of past, current, and future state changes.
Component Type	Also known as a functional component.	Also known as a class component.
Complexity	Simple and easy to understand.	More complex compared to stateless components.
Lifecycle Methods	Does not work with React lifecycle methods.	Can work with all React lifecycle methods
Reusability	Cannot be reused as effectively.	Can be reused more effectively.

24. What is “**key” in React?**

In React, the key is a special attribute used when rendering lists of elements. It helps React identify which items have changed, been added, or been removed, improving the performance of rendering updates. Keys should be given to the elements inside the array to give them a stable identity:

- Each key must be unique among siblings. It's often a good practice to use a unique identifier such as a database ID or a unique value in your data set.
- Keys help React optimize the rendering process by enabling it to efficiently update and reorder items without re-rendering the entire list.
- Keys should be assigned to the parent element in a list.

For example:

JavaScript

```
import React from 'react';

// Sample data: an array of to-do items
const todos = [
  { id: 1, task: 'Buy groceries' },
  { id: 2, task: 'Walk the dog' },
  { id: 3, task: 'Read a book' },
  { id: 4, task: 'Learn React' },
];

const TodoList = () => {
  return (
    <div>
      <h3>To-Do List</h3>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.task}</li>
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

25. What is reconciliation in React ?

Reconciliation is a key concept in React that refers to the process of updating the user interface (UI) in response to changes in state or props. React uses a virtual DOM (a lightweight representation of the actual DOM) to efficiently determine what has changed in the UI and apply those changes to the real DOM in a performant way.

Lets now understand how reconciliation works

- **Virtual DOM representation:** When you create or update a component in React, React builds a virtual DOM tree that represents the UI structure. This virtual DOM is a snapshot of the UI at a particular point in time.
- **Rendering Changes:** When state or props change, React creates a new virtual DOM tree that reflects the updated UI.
- **Diffing Algorithm:** React compares the new virtual DOM tree with the previous one to identify the differences (this process is known as "diffing"). React's diffing algorithm is highly optimized to detect what exactly has changed.
- **Applying Changes:** After identifying the differences, React updates only the parts of the real DOM that have changed, instead of re-rendering the entire UI. This selective update is what makes React efficient and fast.

Example:

Let's say you have a component that displays a list of items:

```javascript
function ItemList({ items }) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

If the items array changes (e.g., an item is added or removed), React will:

- Generate a new virtual DOM for the updated ItemList component.
- Compare it with the previous virtual DOM.
- Identify what has changed (e.g., a new <li> was added or removed).
- Update only the necessary parts of the real DOM.

Why Reconciliation is important

- **Performance Optimization:** By updating only the necessary parts of the DOM, React minimizes the performance impact of frequent updates, which is particularly important for large, dynamic applications.
- **Smooth User Experience:** Users experience a smooth and responsive UI because React efficiently manages updates behind the scenes.
- **Declarative UI:** React allows developers to write declarative UI code without worrying about the underlying DOM manipulations, as React handles the updates intelligently.

26. What is React Fiber ?

React Fiber is like a new engine under the hood of React, introduced in React 16. It was designed to make React better at handling complex and dynamic user interfaces, especially when there are many things happening at once. Think of it as an upgrade that makes React more powerful and efficient.

Before Fiber, React had a straightforward way of updating the UI: it would process all changes in one go. This worked fine for small updates but could cause problems in more complex applications, leading to slow or unresponsive UIs.

React Fiber was introduced to address these issues by making React smarter about how it handles updates.

Features of React Fiber

- **Incremental Rendering (Time Slicing):**
  - ❖ Imagine you're working on a big task, like reading a long book. Instead of trying to read the entire book in one sitting, you break it into smaller chunks, reading a few pages at a time so you don't get overwhelmed.
  - ❖ React Fiber allows React to break down big updates into smaller pieces, which can be processed over multiple frames. This means React can pause what it's doing, respond to more urgent tasks (like user clicks), and then come back to finish the update later. This helps keep the app responsive, even during heavy processing.

- **Prioritization of Updates:**
  - ❖ Think of a busy day where you have to prioritize your tasks. If an important call comes in, you might stop what you're doing to answer it, because it's more urgent than other tasks.
  - ❖ React Fiber lets React decide which updates are more important and should be handled first. For example, if a user clicks a button, React will prioritize updating the button's state over less urgent tasks like loading background data. This ensures that the app feels snappy and responsive to the user.

- **Concurrency:**
  - ❖ Imagine you're juggling multiple tasks at once. You work on one for a bit, then switch to another, and then go back to the first one. This way, you can manage several tasks without getting stuck on just one.
  - ❖ React Fiber allows React to work on multiple updates at the same time. It can start one task, then pause it to work on something else, and then come back to the original task. This helps React handle complex applications where lots of things are happening at the same time.

27. Why does React use className over class attribute?

React uses className instead of class for assigning CSS classes to elements because class is a reserved keyword in JavaScript. To avoid conflicts and potential errors, React adopts the className attribute to specify CSS classes in JSX.

Example:

In plain HTML, you would write:

```JavaScript
<div class="container">
  Hello, world!
</div>
```

In React, you write:

```JavaScript
<div className="container">
  Hello, world!
</div>
```

28. What are synthetic events in React?

Synthetic events in React are like special versions of regular browser events (like clicks or key presses) that work the same way in all web browsers. React uses synthetic events to handle things like clicks and form submissions in a consistent way, no matter which browser you're using.

Why React Uses Synthetic Events:

- **Works Everywhere:** Different browsers sometimes handle events differently. Synthetic events make sure your React code works the same in all browsers.
- **Better Performance:** Instead of adding a separate event listener for every button or input field, React uses one event listener for the whole app and manages all the events efficiently.
- **Easy to Use:** Synthetic events give you a simple, uniform way to work with events in React, so you don't have to worry about browser differences.

Ex:

```javascript
function Button() {
  function handleClick(event) {
    console.log('Button clicked!');
  }

  return <button onClick={handleClick}>Click me</button>;
}
```

In this example, handleClick uses a synthetic event to log a message when the button is clicked. React makes sure this works the same in all browsers.

29. What is strict mode in React ?

Strict Mode in React is a tool that helps developers find potential problems in their code. It doesn't affect how your app works in production but provides helpful warnings during development. Think of it as a "safety check" that helps you write better, more reliable code.

- **Identifies Unsafe Practices:** It checks your code for practices that might cause problems in the future, like using outdated APIs or patterns that might not work well in newer versions of React.
- **Highlights Side Effects:** React Strict Mode helps spot unexpected side effects, which are pieces of code that do something extra, like changing a variable or making a network request, which might not always be obvious. This helps you ensure that your components behave predictably.
- **Warns About Legacy Code:** If you're using older React code that might be problematic or no longer recommended, Strict Mode will give you warnings so you can update your code to modern practices.

Lets understand how to use strict Mode:

You wrap parts of your app (or the whole app) in a <React.StrictMode> tag.

```
JavaScript
import React from 'react';
import App from './App';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

In this example, everything inside <React.StrictMode> will be checked for potential issues.

Note:

- Strict Mode only works in development mode, so it won't slow down your app in production.
- It doesn't change how your app looks or behaves from a user's perspective. It's just for giving you warnings in the developer console.

30. What is Ref in React?

In React, refs (short for "references") are a way to directly access and interact with DOM elements or React components. They are often used when you need to interact with elements that are outside the normal data flow of React, like focusing an input field, scrolling to a specific section, or playing a video.

Normally, in React, we control the UI and interactions through state and props, which are part of React's declarative approach. However, there are times when you might need to directly interact with a DOM element, and this is where refs come into play.

Example:

```javascript
import React, { useRef } from 'react';

function FocusInput() {
  // Create a ref using useRef
  const inputRef = useRef(null);

  // Function to focus the input field when the button is clicked
  const focusInputField = () => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Type something..." />
      <button onClick={focusInputField}>Focus the Input</button>
    </div>
  );
}

export default FocusInput;
```

you can create refs using the useRef hook, which is part of React's hooks API.

In the above example,

- We use useRef(null) to create a ref, and assign it to inputRef. The useRef hook returns a mutable object with a .current property.
- We assign inputRef to the ref attribute of the <input> element. This links the ref to the input field.
- When the button is clicked, the focusInputField function is triggered. Inside this function, inputRef.current gives us access to the DOM element, allowing us to call the .focus() method on the input field.

31. What is Redux Toolkit, and why is it used?

Redux Toolkit is a set of tools that makes working with Redux easier and less time-consuming. It helps reduce the amount of repetitive code you have to write and provides utilities to manage state more efficiently. It also includes features that make handling tasks like asynchronous actions (e.g., API calls) simpler.

A. Simplifies Redux Setup:
- Redux Toolkit makes it easier to set up a Redux store, which is where your application's state is kept.
- It provides built-in functions to configure the store, which reduces the amount of setup code you have to write.

B. Reduces Boilerplate Code:
- Traditional Redux requires you to write a lot of code for actions, reducers, and connecting them together. Redux Toolkit simplifies this by offering tools like createSlice and createAsyncThunk that combine these steps into one.

C. Developer Experience:
- It comes with useful tools for debugging, like a development version of the Redux store with good defaults.
- It also has a configureStore function that automatically sets up the Redux DevTools Extension, making it easier to inspect and debug the state changes in your application.

32. What is createSlice in Redux Toolkit?

createSlice is a key function provided by Redux Toolkit that simplifies the process of creating a slice of the Redux state. A "slice" is essentially a portion of the Redux state along with the logic to update that state.

How createSlice Works:
When you use createSlice, it helps you generate:

- **State**: The part of the Redux store that this slice will manage.
- **Reducers**: Functions that define how the state should change in response to actions.
- **Actions**: Action creators that are automatically generated based on the reducers you define.

Lets understand step by step:

1. **Define the Initial state:** Start by defining the initial state for this slice of the Redux store.
2. **Create Reducers:** Define reducer functions that specify how the state should change in response to specific actions. Each reducer function will automatically create an action with the same name.
3. **Create the slice:** Use createSlice to combine the initial state and the reducers. This will return an object that includes the slice's reducer functions and action creators.

Let's take an example of createSlice:

```javascript
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter', // Name of the slice
  initialState: { value: 0 }, // Initial state
  reducers: {
    increment: (state) => {
      state.value += 1; // Increase the value by 1
    },
    decrement: (state) => {
      state.value -= 1; // Decrease the value by 1
```

```
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload; // Increase the value by the payload
amount
    }
  }
});

// Export the actions generated by createSlice
export const { increment, decrement, incrementByAmount } =
counterSlice.actions;

// Export the reducer function to be used in the store
export default counterSlice.reducer;
```

33. Explain configureStore in Redux Toolkit.

configureStore is a function provided by Redux Toolkit that simplifies the process of creating a Redux store. It comes with sensible defaults and integrates well with the other features of Redux Toolkit, making it easier to set up your application's state management.

What configureStore Does:

- **Combines Reducers:** It allows you to combine multiple reducers into a single root reducer easily, which then manages the state for different slices of your application.
- **Sets Up Middleware:** Middleware in Redux allows you to extend Redux with custom functionality, such as handling asynchronous actions or logging actions. configureStore automatically includes some useful middleware by default, like redux-thunk for handling asynchronous actions, and you can add or customize middleware as needed.
- **Enables Redux DevTools:** configureStore automatically enables the Redux DevTools Extension, which is a powerful tool for debugging your Redux application. This makes it easy to inspect the state changes and action dispatches during development.

- **Supports Immutable State:** It includes middleware that helps you enforce immutability in your state, ensuring that your reducers are correctly updating state without directly mutating it.

Example:

```javascript
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
import userReducer from './userSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,  // Managing state slice for counter
    user: userReducer         // Managing state slice for user
  }
});

export default store;
```

In the above example, The reducer property is where you specify your application's reducers. In the example above, counterReducer and userReducer are imported from separate slice files and are combined into a single reducer object. Each key in this object (counter, user) corresponds to a slice of the Redux state.

34. What is the difference between useMemo and useCallback?

In React, useMemo and useCallback are hooks used to optimize performance by memoizing values and functions, respectively. Although they share similar syntax and purposes, they serve distinct roles in a React application:

useMemo()	**useCallback()**
Memoizes the result of a computation or value.	Memoizes a callback function.
Returns the memoized value itself.	Returns the memoized function itself.
Use when you want to cache the result of an expensive computation.	Use when you want to prevent re-creation of a function on every render.
Useful for avoiding expensive recalculations, such as data processing.	Useful for passing stable callback functions to child components.

THANK YOU

AIMERZ.ai
Aim.Act.Achieve