# **React interview questions- Ebook**

# **Disclaimer:**

COPYRIGHT NOTICE

©All content, materials, and intellectual property contained within this document, including but not limited to text, images, diagrams, graphics, code, and any other original works, are protected by copyright laws and international treaties. This intellectual property is the sole and exclusive property of SRIKANTH TEKUMUDI

Unauthorized use, reproduction, distribution, modification, or transmission of any portion of this document, in any form or by any means, electronic or mechanical, without the prior written permission of SRIKANTH TEKUMUDI, is strictly prohibited and may result in severe civil and criminal penalties.

By accessing, viewing, or using this document, you acknowledge and agree to abide by all applicable copyright laws and the terms and conditions outlined herein. This copyright notice serves as a legal warning and reminder that the unauthorized use of this intellectual property is strictly prohibited and will be vigorously enforced.

Thank you for respecting the intellectual property rights of SRIKANTH TEKUMUD

# **1)What are the different lifecycle methods in React?**

There are three main phases in the lifecycle of a React component: mounting, updating, and unmounting.

Mounting: This phase occurs when a component is inserted into the DOM for the first time.

Updating: This phase is triggered whenever the component's state or props change, causing the component to re-render.

Unmounting: This is the final stage of a component's lifecycle. It occurs when the component is being removed from the DOM.

During these phases, different lifecycle methods are used:

1. componentDidMount: This method is invoked immediately after the component is inserted into the DOM.

2. render: This method is responsible for rendering the component's UI, and it is invoked every time the state or props are updated.

3. shouldComponentUpdate: This method is called immediately after a state or prop change. It determines whether the component should update or not.

4. componentDidUpdate: This method is invoked after the component has re-rendered due to an update. It provides information about the previous state or props.

5. componentWillUnmount: This method is called right before the component is removed from the DOM. It is typically used for performing any necessary cleanup activities.

Please note that the lifecycle methods mentioned above are based on the class component syntax. With the introduction of React Hooks, functional components use different lifecycle methods such as useEffect for accomplishing similar tasks.

# **What is the distinction between virtual DOM and real DOM, and how does the virtual DOM function?**

During the initial load, ReactDOM.render() creates both the virtual DOM and the real DOM.

Whenever there is a keypress, API response, or any other relevant event, the virtual DOM nodes are notified of the property changes. If the properties are updated, the respective node in the virtual DOM is updated accordingly. If there are no property changes, the node remains the same.

React then compares the real DOM with the virtual DOM and updates the real DOM accordingly. This process is known as reconciliation.

Once the real DOM is updated, it is repainted on the browser to reflect the changes.

In React, shallow comparison of prop values is performed by default. If deep comparison is needed, it has to be handled separately. Immutable data structures are commonly used to facilitate deep comparison.

Manipulating the real DOM directly can be costly in terms of performance. To mitigate this, React utilizes the concept of the virtual DOM. The virtual DOM is essentially a JavaScript object that serves as a copy of the real DOM but does not have the ability to directly write on the screen. Manipulating the virtual DOM is more efficient than manipulating the real DOM. React uses a diffing algorithm to determine which components require updates and applies those changes to the real DOM, minimizing the number of manipulations needed.

# **3)Why is it important to use keys in lists?**

keys lets virtual dom to know that which item changed in the list and updates that item only otherwise it render whole list

keys should be unique and predictable

# **4)What are the different types of components in React?**

There are two main types of components in React: class-based components and functional components. The key differences between these two types are as follows:

1. Syntax: Class components are defined using ES6 class syntax, while functional components are defined as regular functions.

2. Lifecycle methods: Class components have access to lifecycle methods, which allow you to perform certain actions at specific points in the component's lifecycle. Functional components do not have lifecycle methods of their own, but they can use React hooks, such as useEffect, to achieve similar functionality.

3. State: Class components can maintain state using the setState method, allowing them to store and update data within the component. Functional components traditionally did not have state, but with the introduction of React hooks, such as useState, functional components can now also manage state in a similar way to class components.

4. Performance: Functional components are generally considered more performant than class components. This is because functional components are simpler and do not require the overhead of a class definition. However, in some cases, class components may still be necessary to achieve certain advanced functionality.

There are two types of components in React: functional components and class components.

Functional components are simple JavaScript functions that return JSX (JavaScript XML) to define the component's structure and content. They don't have built-in state or lifecycle methods by default, but you can add state and lifecycle functionality to functional components using React hooks. Here's an example of a functional component:

| function MyFunctionalComponent() {  return (  <div>  <p>Hello</p>  </div>  ); } |
| --- |

On the other hand, class components were more popular in the past. They are defined as ES6 classes that extend the base React.Component class. Class components have built-in state, props, and lifecycle methods. State can be set using this.state, and you can modify the state using this.setState to notify React which components to re-render. Here's an example of a class component:

class MyClassComponent extends React.Component { render() { return ( <div> <p>Hello</p> </div> ); } }

# **5)What are pure components, and how can we achieve them in functional components?**

Pure components are special components in React that only re-render when there are changes in their state or props. This optimization can improve performance by preventing unnecessary re-renders when the component's data hasn't changed.

In functional components, we can achieve the behavior of pure components by using the `React.memo` higher-order component (HOC).

`React.memo` is a higher-order component that takes a functional component as its input and returns a memoized version of the component. The memoized component will only re-render when there are changes in its props. The comparison of props is done using a shallow equality check by default, but you can provide a custom comparison function if needed.

Here's an example of how to use `React.memo` in a functional component:

| import React from 'react';  const MyFunctionalComponent = React.memo((props) => {  *// Component logic and JSX rendering* }); |
| --- |

export default MyFunctionalComponent;

By wrapping the functional component with React.memo, the component will only re-render when the props change. This can be beneficial in scenarios where the component's rendering is computationally expensive or when there are frequent prop updates.

# **6)What are the differences between props and state in React?**

The main difference between props and state in React lies in their control and mutability.

Props (short for properties) are external inputs to a component that are passed from a parent component. They are read-only and cannot be modified within the component. Props are used to provide data or configuration to the component, allowing it to render and behave accordingly.

State, on the other hand, is internal to a component and managed by the component itself. It represents the current state of the component and can be modified using the setState method or this.setState in class components. State allows the component to keep track of and update its data over time. Unlike props, state can be mutated within the component.

* Props are external inputs passed from a parent component to a child component.
* Props are read-only and cannot be modified within the component receiving them.
* State is internal to a component and managed by the component itself.
* State represents the current state of the component and can be modified using setState.
* State allows the component to keep track of and update its data over time.

Understanding the distinction between props and state is crucial in React, as it helps in managing data flow and maintaining a clear separation of concerns between components.

# **7)What is JSX in React?**

JSX stands for JavaScript XML. It is a syntax extension used in React that allows us to write HTML-like code within JavaScript. With JSX, we can define the structure and content of components in a more concise and intuitive manner, combining markup and logic into a single file.

Here are some key points about JSX:

1. JSX is not mandatory in React, but it is commonly used because of its advantages.

2. It enables us to write HTML-like syntax directly within JavaScript code.

3. Expressions can be embedded inside curly braces {} in JSX. This allows us to dynamically insert values, variables, or JavaScript expressions.

4. To include a block of logic or multiple elements in JSX, we enclose them in parentheses ().

5. JSX uses camelCase notation for HTML attributes. For example, `class` becomes `className`, and `for` becomes `htmlFor`, to avoid conflicts with JavaScript keywords.

6. JSX gets transformed into JavaScript function calls that create React elements. This conversion is done by a tool like Babel during the build process.

Advantages of JSX:

1. JSX allows us to seamlessly integrate HTML-like syntax into React components, making the code more readable and maintainable.

2. It simplifies the process of converting HTML tags into React elements.

3. JSX is an expression, so we can use it inside loops, conditional statements, or assign it to variables.

4. JSX facilitates the use of React lifecycle methods and hooks by allowing us to include the necessary code alongside the markup.

# **8)What are React hooks, specifically useState and useEffect?**

React hooks are a feature introduced in React 16.8 that allow functional components to have state and lifecycle methods, previously only available in class components. Hooks are built-in functions provided by React, such as useState, useEffect, useRef, useCallback, useMemo, and more. They offer a more flexible and concise way to manage state and perform side effects in functional components.

The useState hook is used to add state to functional components. It takes an initial value as an argument and returns an array with two elements: the current state value and a function to update the state. Here's an example:

React hooks are a feature introduced in React 16.8 that allow functional components to have state and lifecycle methods, previously only available in class components. Hooks are built-in functions provided by React, such as useState, useEffect, useRef, useCallback, useMemo, and more. They offer a more flexible and concise way to manage state and perform side effects in functional components.

The useState hook is used to add state to functional components. It takes an initial value as an argument and returns an array with two elements: the current state value and a function to update the state. Here's an example:

| const [state, setState] = useState(initialValue); |
| --- |

The state can be of any data type, including primitive values like numbers or strings, as well as objects or arrays.

To update the state asynchronously when the new state depends on the previous state, you can pass a function to the setState function. This function receives the previous state value as an argument and returns the updated state. Here's an example:

| setState((prev) => {  return {  ...prev,  name: 'satish'  }; }); |
| --- |

In the example above, if the state is an object, the setState function updates the name property to 'satish' while preserving the other properties by spreading the previous state.

The useEffect hook is used to perform side effects in functional components. It accepts a function as its first argument, which will be executed after the component renders. The function can perform tasks like fetching data, subscribing to events, or modifying the DOM. useEffect can also return a cleanup function to handle any necessary cleanup operations when the component unmounts or when the dependencies change. This helps prevent memory leaks and ensures proper cleanup of resources.

.

# **9)What is the Context API in React?**

The Context API in React provides a way to share data between components without explicitly passing props through all intermediate components. It acts as a global variable or data layer that any component can access, making it easier to manage and share data across the application. The Context API can be seen as an alternative to Redux for state management, especially for simpler use cases.

Here's how to use the Context API in React:

| Creating a Context: Use the React.createContext() function to create a Context object.   const MyContext = React.createContext(); |
| --- |

Providing the Context: Wrap the components that need access to the shared data with the Context Provider component. Provide the value that you want to share.

| <MyContext.Provider value={sharedData}>  {*/\* Components that consume the context \*/*} </MyContext.Provider> |
| --- |

Consuming the Context: Use the Context Consumer component to access the shared data within components that need it.

| <MyContext.Consumer>  {value => (  {*/\* Render components that consume the shared data \*/*}  )} </MyContext.Consumer> |
| --- |

Alternatively, you can use the useContext hook to consume the context within functional components:

| const value = useContext(MyContext); |
| --- |

The value represents the shared data that is passed through the Context Provider and accessed by the consuming components.

By utilizing the Context API, you can eliminate the need for prop drilling and make the sharing of data more efficient and manageable. It allows components to subscribe to context changes and access the shared data as needed.

# **10)What is Redux?**

Redux is a predictable state container for JavaScript applications. It provides a centralized way of managing the application state by storing all the states in one place, commonly referred to as the store.

State management becomes necessary when there is a need to share data across different components. While internal component state can be controlled using the useState hook, state management libraries like Redux are used when state needs to be managed across multiple components.

How Redux Works: There are three fundamental building blocks in Redux:

Actions: Actions are plain JavaScript objects that serve as the only way to send data from the application to the store. They contain information about the type of action being performed and can have additional data called payload. Actions are dispatched using the store.dispatch() method.

| Example:   {  type: "LOGIN",  payload: {  username: "foo",  password: "bar"  } } |
| --- |

Action creators are functions that create actions and return them. They help in maintaining a consistent structure for actions throughout the application.

Reducers: Reducers are pure functions that take the current state and an action as input and return the new state. They are responsible for handling specific actions and updating the state accordingly. Redux state is stored as objects, and reducers ensure that the state remains immutable.

| Example:   const loginReducer = (state, action) => {  switch (action.type) {  case "LOGIN":  *// Perform state update based on the action*  break;  default:  return state;  } } |
| --- |

Store: The store is the central hub that holds the entire application state. It is the single source of truth for the application's data. There is only one store in an application, but it can have multiple reducers combined using the combineReducers function.  
  
 Example:

| const store = createStore(loginReducer); |
| --- |

Redux Middleware: Middleware in Redux sits between actions and reducers, allowing developers to intercept actions before they reach the reducer. It provides a way to add additional functionality to the dispatch process. Middleware is commonly used for tasks like logging, asynchronous operations, or manipulating actions before they reach the reducer.

Advantages of Redux:

* Redux makes the application state predictable by enforcing a consistent flow of data.
* Redux promotes immutable state, ensuring that the state is not directly mutated.
* Redux enables powerful debugging capabilities, such as logging and time travel, using developer tools.

# **11)Explain react-redux**

React-Redux:

Redux itself is not directly related to React. To establish a connection between Redux and React, the React-Redux library is used. React-Redux provides bindings between Redux and React components, making it easier to integrate Redux into a React application.

Here are some important features and functions provided by React-Redux:

useSelector: The useSelector hook is used to extract data from the Redux store within a React component. It allows you to select and access specific data from the store's state.  
  
 Example:

| import { useSelector } from 'react-redux';  const MyComponent = () => {  const data = useSelector(state => state.myReducer.data);  *// Use the selected data from the Redux store*  return <div>{data}</div>; } |
| --- |

useDispatch: The useDispatch hook is used to access the dispatch function from the Redux store. It allows you to dispatch actions to update the Redux state.  
  
 Example:

| import { useDispatch } from 'react-redux';  const MyComponent = () => {  const dispatch = useDispatch();  *// Dispatch an action*  dispatch({ type: 'ACTION\_TYPE' });  *// ...* } |
| --- |

Provider: The Provider component is a wrapper component from React-Redux that makes the Redux store available to all components in the React application. It needs to be placed at the top level of the component hierarchy.  
  
 Example:

| import { Provider } from 'react-redux'; import store from './store';  const App = () => {  return (  <Provider store={store}>  {/\* Components that need access to the Redux store \*/}  </Provider>  ); } |
| --- |

Additionally, React-Redux works well with the Redux Toolkit, which provides a set of utilities and functions to simplify Redux development. Some of the important functions provided by the Redux Toolkit include configureStore, createSlice, createAction, createReducer, and more. These functions help in creating and managing Redux store, reducers, actions, and state.

Example:

| import { configureStore, createSlice } from "@reduxjs/toolkit";  const counterSlice = createSlice({  name: "counter",  initialState: 0,  reducers: {  increment: (state) => state + 1,  decrement: (state) => state - 1  } });  const store = configureStore({  reducer: counterSlice.reducer }); |
| --- |

export default store;

# **12)What is the difference between controlled and uncontrolled inputs in React?**

Controlled components and uncontrolled components are terms used to describe how React components handle form elements, such as input fields.

Uncontrolled components: An uncontrolled component is a component that renders form elements where the data is handled by the DOM rather than by React. In other words, the state of the input field is managed by the browser's native form handling mechanisms. To access the data entered in an uncontrolled input field, you can use the ref hook to obtain its value directly from the DOM.

Example:

<input type="text" ref={inputRef} />

Controlled components: A controlled component is a component in which the form data is handled by React using component state. The value of the input field is controlled and managed by React, and any changes to the input value are reflected in the component state. To update the value of a controlled input field, you need to provide an onChange event handler that updates the component state accordingly.

Example:

<input type="text" value={value} onChange={handleChange} />

In the controlled component example, value is the state value used to control the input field, and handleChange is the event handler that updates the value state whenever the input value changes.

The key differences between controlled and uncontrolled inputs are:

* Uncontrolled inputs rely on the DOM to manage the input data, while controlled inputs handle the input data through React component state.
* Uncontrolled inputs are typically used for simple forms where you only need to access the final input value when the form is submitted.
* Controlled inputs offer more control and flexibility as you can validate, manipulate, or react to input changes in real time using React's state and event handling mechanisms.

# **13)What is the forwardRef method in React?**

The forwardRef method in React allows parent components to pass down refs to their child components. It enables child components to access and interact with the parent component's references. To understand the purpose of forwardRef, let's first understand the concept of refs in React.

In React, refs provide access to the underlying DOM node or React component instance. With refs, we can directly modify the DOM node without changing the component's state or triggering a re-render. Refs are typically used when we need to interact with a specific element, such as focusing an input field or measuring its dimensions.

In some cases, a child component may need a reference to a parent component's current node. This is where forwardRef comes into play. By using forwardRef, the parent component can pass down its refs to the child component, allowing the child component to access and utilize those references.

ForwardRef is particularly useful when developing reusable components that need to interact with elements outside of their own scope.

Here's an example of using forwardRef:

| const InputText = React.forwardRef((props, ref) => {  return <input type="text" ref={ref} {...props} />; }); |
| --- |

In the example above, the InputText component is a child component that receives a ref from its parent component. The ref is passed to the input element, allowing the parent component to reference and interact with the input field through the ref.

By utilizing forwardRef, we can create more flexible and reusable components that can seamlessly work with references from their parent components.

In summary, the forwardRef method in React allows parent components to pass down refs to child components. It enables child components to access and utilize the parent's references, facilitating interaction with elements outside of their own scope. This technique is especially beneficial when developing reusable components that need to work with references provided by their parent components.

# **14)What are Higher-Order Components (HOCs)?**

Higher-Order Components (HOCs) are functions that take a component as input and return a modified version of that component. HOCs allow us to enhance and extend the functionality of existing components in a reusable way. They are a pattern in React that promotes code reuse and separation of concerns.

Here's a simplified explanation of HOCs:

1. Input Component: An input component is the component that we want to enhance or modify. It serves as the base component that we pass to the HOC.
2. Higher-Order Component: The HOC is a function that takes the input component as an argument and returns a new component. The HOC can add additional props, modify behavior, or wrap the input component with other components.
3. Output Component: The output component is the modified component that is returned by the HOC. It incorporates the changes or enhancements made by the HOC and can be used in the application.

| Example: *// Higher-Order Component* const withLogging = (WrappedComponent) => {  const EnhancedComponent = (props) => {  console.log('Component is rendered:', WrappedComponent.name);  return <WrappedComponent {...props} />;  };  return EnhancedComponent; };  // Input Component const MyComponent = (props) => {  return <div>{props.text}</div>; };  // Output Component after applying the HOC const EnhancedMyComponent = withLogging(MyComponent); |
| --- |

In the example above, withLogging is the HOC that takes MyComponent as input. It returns a new component, EnhancedMyComponent, which logs a message before rendering MyComponent. The EnhancedMyComponent component incorporates the logging behavior added by the HOC.

By using HOCs, we can apply common functionalities, such as logging, authentication, or data fetching, to multiple components without repeating the code in each component. HOCs enable code reusability and separation of concerns, making it easier to manage and maintain components.

# **15)What are Portals in React?**

Portals in React provide a first-class way to render child components into a DOM node that exists outside the normal DOM hierarchy of the parent component. With portals, you can render components into a different part of the DOM tree, even outside the parent component's root node.

Here's a simplified explanation of portals:

1. Normal Rendering: By default, when you render a component in React, it is rendered as a child within the parent component's DOM hierarchy. The rendered component is placed in the DOM structure at the location defined by its parent.
2. Portal Rendering: Portals allow you to render a component into a different DOM node that is outside the normal DOM hierarchy of the parent component. This means that the rendered component can be placed in a different part of the DOM tree, such as a sibling or ancestor of the parent component.

Portals are useful in scenarios where you need to render a component at a different location in the DOM tree, such as modals, tooltips, or popovers. These components often require rendering outside the current DOM hierarchy to avoid CSS or z-index conflicts or to achieve a specific visual effect.

| Here's an example of using portals: import React from 'react'; import ReactDOM from 'react-dom';  const Modal = ({ children }) => {  return ReactDOM.createPortal(  <div className="modal">  {children}  </div>,  document.getElementById('modal-root')  ); };  const App = () => {  return (  <div>  {/\* Normal component rendering \*/}  <h1>My App</h1>   {/\* Portal rendering \*/}  <Modal>  <p>This is a modal dialog.</p>  </Modal>  </div>  ); };  ReactDOM.render(<App />, document.getElementById('root')); |
| --- |

In the example above, the Modal component is rendered using ReactDOM.createPortal(). It renders the provided children into a separate DOM node with the ID 'modal-root', which is outside the normal DOM hierarchy of the parent component (App). This allows the modal to be rendered in a different part of the DOM tree while still being managed and controlled by the React component.

# **16)What are stateless and stateful components?**

Answer: In software development, stateless and stateful components refer to different types of components based on their handling of data and state.

Stateless components, also known as pure components, are functions that have no internal state. They receive data as input and produce an output based solely on that data. These components are focused on performing specific tasks or computations and do not maintain any memory of previous function calls. They are commonly used for tasks such as rendering UI elements or executing a specific operation.

Stateful components, on the other hand, maintain an internal state that can change over time. These components have the ability to store and modify data within their scope. When the state of a stateful component changes, it triggers a re-rendering of the component to reflect the updated state. Stateful components are often used for managing user interactions, handling form data, or maintaining complex application states.

# **17)What is the purpose of ESLint, Babel plugin, and Webpack in JavaScript development?**

ESLint: ESLint is a powerful code analysis tool used to identify problematic patterns or errors in JavaScript code. It helps enforce consistent coding styles, detect potential bugs, and promote best practices. ESLint can be configured with custom rules or presets to match specific project requirements and coding standards. It is commonly integrated into development workflows to catch errors early and improve code quality.

Babel plugin: Babel is a JavaScript transpiler that allows developers to write modern JavaScript code using features from the latest ECMAScript (ES) standards (such as ES6+) and convert it into a backward-compatible version that can run in older environments. Babel plugins are additional modules that extend Babel's capabilities by providing support for specific language features or transformations. These plugins can handle tasks like converting JSX syntax to regular JavaScript or adding polyfills for missing browser APIs.

Webpack: Webpack is a popular module bundler for JavaScript applications. It analyzes the dependencies of a project by building a dependency graph and then creates optimized bundles based on that graph. Webpack enables developers to manage and bundle various types of assets, including JavaScript, CSS, images, and more. It provides features like code splitting, lazy loading, and minification to optimize the performance of web applications. Webpack is often used alongside Babel and other tools in modern JavaScript development workflows to bundle and optimize code for production deployment.

# **18)What is lazy loading?**

Lazy loading is a technique used to optimize web page loading by only loading the necessary resources as the user needs them. Instead of loading the entire webpage upfront, lazy loading loads specific sections or assets when they become visible or required. This approach helps improve the page's loading speed and efficiency, as unnecessary code and resources are avoided until they are actually needed. One common example of lazy loading is infinite scrolling, where additional content is loaded as the user scrolls down a webpage.

Example

Suppose you have a webpage with a long list of images, but you don't want to load all the images at once to improve performance. Instead, you want to load only the images that are currently visible to the user.

1. Initially, the webpage loads without any images.
2. As the user scrolls down the page, JavaScript detects the visible area.
3. When an image comes into view, a JavaScript function is triggered to load that particular image dynamically.
4. The image is fetched from the server and rendered on the webpage.
5. This process repeats for each image as the user continues scrolling.

By lazy loading the images, you avoid unnecessary network requests and improve the initial loading time of the webpage. This technique is commonly used in image galleries, social media feeds, or any web page with a large number of resources.

# **19)What is code splitting?**

Code splitting is a technique used to improve the performance of web applications by breaking down the application's codebase into smaller, manageable chunks. By dividing the code into smaller pieces, only the necessary code is loaded for a specific page or functionality, reducing the initial load time of the application. Code splitting can be achieved using tools like Webpack or Rollup, or by utilizing dynamic imports in JavaScript. The goal is to load only the code that is required at a given moment, resulting in faster loading times, better user experiences, and improved performance.

Example

Suppose you have a JavaScript application that consists of multiple pages or features. Instead of loading the entire application code upfront, you can split the code into smaller chunks and load them on-demand.

Let's consider a simple scenario with two pages: Home and About.

1. Initially, only the common code required for both pages is loaded.
2. When the user navigates to the Home page, the specific code for that page is loaded and executed.
3. When the user navigates to the About page, the code for the About page is loaded and executed.
4. Each page loads only the necessary code, reducing the initial loading time.

This code splitting technique allows users to navigate through your application more quickly because they only load the code needed for the specific page they are accessing. It helps improve performance, especially in larger applications with complex functionality.

Both lazy loading and code splitting are powerful techniques to optimize the performance of web applications by loading resources and code selectively when needed, resulting in faster load times and a smoother user experience.

# **20) Explain of the asynchronous form of useState?**

In the asynchronous form of useState, the setState function accepts a callback function as an argument. This callback function receives the previous state as its parameter and returns the updated state.

Here's an example using the asynchronous form of useState with a simple value state:

| const [state, setState] = useState();  setState((prevState) => {  // Update the state based on the previous state  return prevState + 1; }); |
| --- |

In this example, the setState function is called with a callback that receives the previous state (prevState) as a parameter. The callback function performs some operation on the previous state and returns the updated state. The state update is asynchronous and will be applied in a subsequent render.

If the state is an object, you can use the same asynchronous form of useState to update specific properties of the object:

| const [state, setState] = useState({ name: 'sri', age: 24 });  setState((prevState) => {  // Update the 'name' property of the state  return { ...prevState, name: 'satish' }; }); |
| --- |

In this example, the callback function receives the previous state (prevState) as a parameter and returns a new object that includes all the properties from the previous state with the updated name property.

By using the asynchronous form of useState, you can update state values based on the previous state in a safe and predictable manner.

# **21)Can you explain what web workers are and how they enable running scripts in background threads while communicating with the main JavaScript code?**

Web workers are a way to run scripts in the background threads of a web browser. They allow for executing computationally intensive tasks or performing IO operations without blocking the main user interface. Once created, a worker can communicate with the JavaScript code that spawned it by exchanging messages.

Here's a simplified explanation of web workers:

To create a web worker, you can instantiate it using the Worker constructor and provide the URL of the script file that will run in the background. For example:

| let worker = new Worker('apple.js'); |
| --- |

You can send messages to the worker using the postMessage method. The worker can process these messages and perform the necessary tasks. For instance:

| worker.postMessage('do work'); |
| --- |

To receive messages from the worker, you can listen for the message event and define a callback function to handle the received data. Here's an example:

| worker.onmessage = function (e) {  console.log(e.data); }; |
| --- |

In this case, the worker sends the computed result back to the main JavaScript code by calling postMessage with the desired data.

Now, let's take a look at the apple.js worker script:

| onmessage = function (e) {  let final = 0;  for (let i = 0; i < 10 \*\* 10; i++) {  final += i;  }  postMessage(final); }; |
| --- |

The worker script receives messages from the main code using the onmessage event. In this example, it performs a computationally intensive task of summing numbers from 0 to 10 billion (10 \*\* 10). The result is then sent back to the main JavaScript code using postMessage.

—-MORE WILL BE COMING SOON—-