



Web Dev Mastery

"React.js Most Important Interview Questions"

Basic Level:

1. What is React?
 2. What is JSX?
 3. What are Components in React?
 4. What is the difference between Functional and Class Components?
 5. What is the Virtual DOM?
 6. What are Props?
 7. What Is the State in React?
 8. What is the difference between State and Props?
 9. What is the use of the useState hook?
 10. What is the use of the useEffect hook?
-

Intermediate Level:

1. What is the purpose of Keys in React?
 2. What are React Fragments?
 3. What is the Context API in React?
 4. What is Higher-Order Component (HOC)?
 5. What are React Hooks?
 6. What is useMemo and when would you use it?
 7. What is the use of useCallback?
 8. What are controlled and uncontrolled components in React?
 9. What are Pure Components?
 10. What is Prop Drilling and how can you avoid it?
-

Advanced Level:

1. What is Redux and how is it used in React?
 2. What are React Portals?
 3. What is React Suspense?
 4. How does React handle performance optimization?
 5. What is Server-Side Rendering (SSR) in React?
 6. What is the difference between useReducer and useState?
 7. What is React Fiber?
 8. What is Code Splitting and how is it implemented in React?
 9. What is Reconciliation in React?
 10. How does React.memo work?
 11. What is Lazy Loading in React?
 12. How do you handle form validation in React?
 13. What is Strict Mode in React?
 14. What are the Life Cycle Methods In React?
-

Basic Level:

1. What is React?
 - React is a JavaScript library used for building user interfaces, primarily for single-page applications. It's maintained by Facebook and focuses on component-based architecture.

```
import React from 'react';

const App = () => {
  return <h1>Hello, React!</h1>;
};

export default App;
```

2. What is JSX?

- JSX stands for JavaScript XML. It is a syntax extension for JavaScript that looks similar to HTML and is used to describe what the UI should look like in React components.

```
const Greeting = () => {  
  return <h1>Hello, World!</h1>;  
};
```

3. What are Components in React?

- Components are the building blocks of any React application, allowing you to break the UI into reusable and isolated pieces. Components can be either class-based or functional.

```
// Functional Component  
const MyComponent = () => <h2>This is a functional component</h2>;  
  
// Class Component  
class MyClassComponent extends React.Component {  
  render() {  
    return <h2>This is a class component</h2>;  
  }  
}
```

4. What is the difference between Functional and Class Components?

- **Class components** are ES6 classes that extend from `React.Component`, whereas **Functional components** are simpler functions that take props and return JSX. Functional components can use hooks to manage state and side effects.

```
// Functional Component
const FuncComponent = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

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

5. What is the Virtual DOM?

- The **Virtual DOM** (VDOM) is a lightweight, in-memory representation of the actual DOM (Document Object Model) that React uses to optimize the rendering process. Instead of directly manipulating the real DOM, React creates a virtual copy of the DOM, updates the virtual DOM when changes occur, and then compares it to the previous version using a process called **Reconciliation**. After identifying the differences, it efficiently updates only the parts of the real DOM that changed, minimizing the performance impact.

How Does the Virtual DOM Work?

1. **Initial Rendering:** When a React application is rendered for the first time, a virtual representation of the DOM is created. This is an exact copy of the actual DOM.
2. **State/Prop Changes:** When there is a change in state or props, React creates a new version of the virtual DOM.

3. **Reconciliation:** React compares the new virtual DOM with the previous one to determine what has changed. This process is called **Diffing**.
4. **Minimal DOM Updates:** After identifying the changes, React updates only the specific parts of the actual DOM that have changed, rather than re-rendering the entire DOM. This makes the process faster and more efficient.

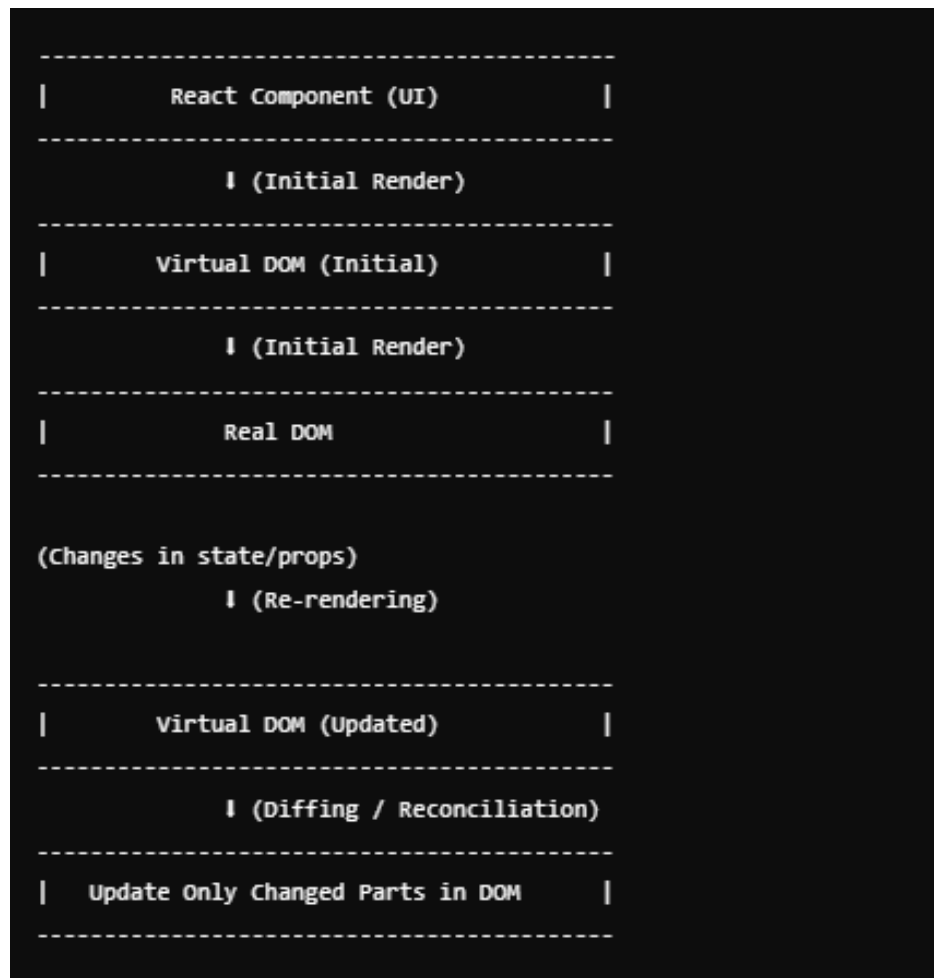
Virtual DOM Flow with Diagram:

Let's illustrate this with a diagram and an example.

```
Initial Render -> Virtual DOM created -> Render real DOM from Virtual DOM

User Interacts or State Changes ->
New Virtual DOM Created ->
Diffing (Comparison between old and new Virtual DOM) ->
Patch only the changes in the Real DOM
```

Diagram of Virtual DOM Flow:



Example:

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

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

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};

export default Counter;
```

1. **Initial Render:** When the app is first rendered, React creates a virtual DOM representing the structure of the counter.

`<div>`, `<h1>`, and `<button>` are part of this virtual DOM.

The initial count value is `0`, so the `h1` displays `0`.

2. **Click Event:** When the user clicks the button, the `count` state is updated to `1`.

React creates a new virtual DOM with the updated count value (`1`).

3. **Diffing:** React compares the old virtual DOM (where the count was `0`) with the new one (where the count is `1`).

It identifies that only the text content inside the `<h1>` tag has changed.

4. **DOM Update:** React efficiently updates only the `h1` element in the real DOM, without touching the rest of the DOM (like the `button`).

6. What are Props?

- Props (short for properties) are used to pass data from one component to another. They are read-only and cannot be modified by the receiving component.

```
const Child = (props) => {  
  return <h1>{props.message}</h1>;  
};  
  
const Parent = () => {  
  return <Child message="Hello from Parent!" />;  
};
```

7. What is the State in React?

- State is an object that determines how a component renders and behaves. Unlike props, state is mutable and is managed within the component.

```
import { useState } from 'react';  
  
const Counter = () => {  
  const [count, setCount] = useState(0);  
  
  return (  
    <div>  
      <p>Count: {count}</p>  
      <button onClick={() => setCount(count + 1)}>Increment</button>  
    </div>  
  );  
};
```


8. What is the difference between **State** and **Props**?

- Props are passed from parent to child and are immutable, whereas State is managed within the component and can change over time.

9. What is the use of the **useState** hook?

- **useState** is a hook that allows you to add state to functional components. It returns an array with the current state and a function to update it.

```
import { useState } from 'react';

const MyComponent = () => {
  const [value, setValue] = useState('Hello');

  return (
    <div>
      <p>{value}</p>
      <button onClick={() => setValue('Hello, World!')}>Change Text</button>
    </div>
  );
};
```

10. What is the use of the **useEffect** hook?

- **useEffect** is used to perform side effects in functional components. It can be used for tasks such as fetching data, subscribing to events, or directly interacting with the DOM.

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

const DataFetching = () => {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => setData(data));
  }, []); // Empty array means it runs once after the component mounts.

  return (
    <ul>
      {data.map(item => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
};
```

Intermediate Level:

1. What is the purpose of Keys in React?

- Keys help React identify which items have changed, are added, or are removed in a list. They improve the performance of rendering lists by ensuring each element has a unique identity.

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

2. What are React Fragments?

- Fragments allow you to return multiple elements from a component without adding an extra node to the DOM.

```
const FragmentExample = () => {  
  return (  
    <>  
      <h1>Heading</h1>  
      <p>Paragraph inside a fragment.</p>  
    </>  
  );  
};
```

3. What is the **Context API** in React?

- The Context API is used for prop drilling, which allows you to pass data through the component tree without having to pass props manually at every level.

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

const MyContext = createContext();

const Child = () => {
  const value = useContext(MyContext);
  return <h1>{value}</h1>;
};

const Parent = () => {
  return (
    <MyContext.Provider value="Hello from Context!">
      <Child />
    </MyContext.Provider>
  );
};
```

4. What is Higher-Order Component (HOC)?

- A Higher-Order Component is a pattern that takes a component and returns a new component with enhanced functionality.

Project Structure:

```
/src
├─ App.js
├─ components
│   └─ Hello.js
│   └─ withLogger.js
└─ index.js
```

1. withLogger.jsx (HOC)

This file contains the Higher Order Component that enhances any component it wraps by logging props.

```
// src/components/withLogger.js
import React from 'react';

const withLogger = (WrappedComponent) => {
  return (props) => {
    console.log('Props:', props); // Logging props to console
    return <WrappedComponent {...props} />;
  };
};

export default withLogger;
```

2. Hello.js

This is a simple functional component that will be wrapped by our HOC.

```
// src/components/Hello.js
import React from 'react';

const Hello = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default Hello;
```

3. App.js

This is the main entry point where we apply the HOC to the `Hello` component and render it.

```
// src/App.js
import React from 'react';
import Hello from './components/Hello';
import withLogger from './components/withLogger';

const HelloWithLogger = withLogger(Hello); // Wrapping Hello with HOC

function App() {
  return (
    <div className="App">
      <HelloWithLogger name="Suman Malakar" /> {/* Using enhanced component */}
    </div>
  );
}

export default App;
```

5. What are React Hooks?

- Hooks are functions that let you “hook into” React state and lifecycle features in functional components. Examples include `useState`, `useEffect`, and `useContext`.

6. What is `useMemo` and when would you use it?

- `useMemo` is used to memoize expensive calculations so that they are only recalculated when necessary, improving performance.

```

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

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

  // useMemo remembers the result until count changes
  const doubleCount = useMemo(() => {
    console.log('Running calculation...');
    return count * 2;
  }, [count]); // Only runs when count changes

  return (
    <div>
      <h1>With useMemo</h1>
      <p>Count: {count}</p>
      <p>Double: {doubleCount}</p> { /* Only recalculates when needed */ }
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
    </div>
  );
}

export default App;

```

7. What is the use of `useCallback`?

- `useCallback` returns a memoized version of a callback function and is used to prevent re-creating functions unnecessarily during re-renders.

```

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

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

  // Using useCallback to memoize the function
  const increment = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []); // This function will not change on re-renders

  return (
    <div>
      <h1>useCallback Example</h1>
      <p>Count: {count}</p>
      {/* Increment button */}
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default App;

```

8. What is **useRef** in React?

- **useRef** is a hook that returns a mutable object which persists throughout the lifecycle of the component, often used to directly manipulate DOM elements.


```

import React, { useRef } from 'react';

const ChangeBackgroundColor = () => {
  const divRef = useRef(null);

  const toggleColor = () => {
    divRef.current.style.backgroundColor =
      divRef.current.style.backgroundColor === 'lightblue' ? 'lightcoral' : 'lightblue';
  };

  return (
    <div>
      <div ref={divRef} style={{ backgroundColor: 'lightblue', padding: '20px' }}>
        Change my background color!
      </div>
      <button onClick={toggleColor}>Change Color</button>
    </div>
  );
};

export default ChangeBackgroundColor;

```

9. What are controlled and uncontrolled components in React?

- Controlled components are components whose state is controlled by React, usually through form elements like `<input>`. Uncontrolled components manage their own state within the DOM.

- **Controlled Component:** A component where form data is handled by React state. The value of the input element is controlled by the state, and any changes to the input are reflected in the state via an `onChange` handler.
- **Uncontrolled Component:** A component where form data is handled by the DOM itself. Instead of React state, a `ref` is used to directly access the input's value from the DOM.

```
// Controlled Component
const ControlledInput = () => {
  const [value, setValue] = useState('');
  return <input value={value} onChange={(e) => setValue(e.target.value)} />;
};

// Uncontrolled Component
const UncontrolledInput = () => {
  const inputRef = React.useRef();
  const handleClick = () => alert(inputRef.current.value);
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleClick}>Show Value</button>
    </div>
  );
};
```

10. What are Pure Components?

- **A Pure Component** in React is a component that extends `React.PureComponent` and implements a shallow comparison of its props and state. If there are no changes in props or state, it does not re-render, leading to performance optimization.

```

import React, { PureComponent } from 'react';

class PureComp extends PureComponent {
  render() {
    return <p>Count: {this.props.count}</p>;
  }
}

class App extends React.Component {
  state = { count: 0 };

  render() {
    return (
      <div>
        <PureComp count={this.state.count} />
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}

export default App;

```

11. What is Prop Drilling and how can you avoid it?

- Prop drilling refers to passing props down multiple levels of components. It can be avoided using the Context API or libraries like Redux.

Advanced Level:

1. What is Redux and how is it used in React?

- Redux is a state management library that allows you to manage the global state of your application. It follows a unidirectional data flow and is often used with React to manage complex states.

2. What are React Portals?

- **React Portals** provide a way to render components outside the normal React DOM hierarchy, i.e., render child components into a DOM node that exists outside of the parent component's DOM hierarchy.
- In typical React components, everything is rendered inside the same root element. But sometimes, you need to render components in a different part of the DOM, like modals, tooltips, or overlays, which should appear on top of everything else and outside the usual component structure. React Portals make this possible.

HTML (index.html):

```
<body>
  <div id="root"></div>
  <div id="modal-root"></div> <!-- Portal target -->
</body>
```

React (App.js):

```
import React, { useState } from 'react';
import ReactDOM from 'react-dom';

// Modal component
const Modal = ({ onClose }) => {
  return ReactDOM.createPortal(
    <div style={modalStyle}>
      <h2>This is a Modal</h2>
      <button onClick={onClose}>Close Modal</button>
    </div>,
    document.getElementById('modal-root') // Targeting modal-root in the DOM
  );
};
```

```
const modalStyle = {
  position: 'fixed',
  top: '50%',
  left: '50%',
  transform: 'translate(-50%, -50%)',
  padding: '20px',
  backgroundColor: 'white',
  border: '1px solid black',
  zIndex: 1000,
};
```

```
const App = () => {  
  const [showModal, setShowModal] = useState(false);  
  
  return (  
    <div>  
      <h1>React Portal Example</h1>  
      <button onClick={() => setShowModal(true)}>Open Modal</button>  
  
      {showModal && <Modal onClose={() => setShowModal(false)} />}  
    </div>  
  );  
};  
  
export default App;
```

3. What is React Suspense?

- React Suspense allows you to delay rendering components until some asynchronous data is loaded, improving the user experience by handling loading states more efficiently.

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

// Lazy load the component
const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>Main App Component</h1>
      {/* Suspense wraps the lazy-loaded component */}
      <Suspense fallback=<div>Loading...</div>>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

```
import React from 'react';

const LazyComponent = () => {
  return (
    <div>
      <h2>This is the Lazy Loaded Component</h2>
      <p>This component is loaded dynamically using React Suspense and lazy().</p>
    </div>
  );
};

export default LazyComponent;
```

4. How does React handle performance optimization?

- React optimizes performance using techniques like memoization (`React.memo`), using keys in lists, lazy loading, code splitting, `useMemo`, `useCallback`, and avoiding unnecessary re-renders by using Pure Components.

5. What is Server-Side Rendering (SSR) in React (with Next.js) ?

- SSR involves rendering React components on the server and sending fully rendered HTML to the client. It improves initial load time and SEO. Libraries like Next.js are often used for SSR in React.

```
import React from 'react';

function Home({ data }) {
  return (
    <div>
      <h1>Server-side rendered data: {data}</h1>
    </div>
  );
}

export async function getServerSideProps() {
  return {
    props: {
      data: "Hello from server-side",
    },
  };
}

export default Home;
```

6. What is the difference between `useReducer` and `useState`?

- `useReducer` is used when you have more complex state logic and actions, similar to how you would manage state with Redux.

`useState` is simpler and suited for less complex state management.

7. What is React Fiber?

- React Fiber is the new reconciliation algorithm in React 16. It enables React to split rendering work into chunks and handle tasks more efficiently by pausing and resuming work as needed.

What is React Fiber?

1. **Reconciliation Algorithm:** Fiber is a complete rewrite of React's reconciliation algorithm, which determines how the UI should update based on changes in state and props.
2. **Incremental Rendering:** It allows React to pause and resume work on rendering, which helps manage long tasks and improves performance, especially for complex applications.
3. **Prioritization of Updates:** Fiber introduces a scheduling mechanism that enables React to prioritize updates, meaning that more critical updates (e.g., user interactions) can be processed before less important ones (e.g., background updates).
4. **Better Error Handling:** Fiber provides improved error boundaries, allowing components to catch errors during rendering, lifecycle methods, and event handlers.

8. What is Code Splitting and how is it implemented in React?

- Code splitting allows you to load parts of your application on demand. It can be implemented using React's `React.lazy` and `Suspense` to load components lazily.

Code Splitting is a technique used in modern web development to optimize the loading performance of applications. By splitting the code into smaller bundles, you can reduce the initial load time of your application.

This means that users only download the necessary code required for the page they're visiting, rather than the entire application.

How Code Splitting is Implemented in React

In React, code splitting can be achieved using **dynamic `import()`** statements along with React's `Suspense` and `lazy` functions.

Steps to Implement Code Splitting in React:

- **Using `React.lazy`:** This allows you to load a component lazily.
- **Using `Suspense`:** This wraps your lazy-loaded components and provides a fallback UI (like a loading spinner) while the component is being loaded.

1. Create the Components

Create two components: `Home.js` and `About.js`.

Home.js:

```
import React from 'react';

const Home = () => {
  return <h1>Home Page</h1>;
};

export default Home;
```

About.js:

```
import React from 'react';

const About = () => {
  return <h1>About Page</h1>;
};

export default About;
```

2. Implement Code Splitting in Your Main Component

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';

// Lazily load the components
const Home = lazy(() => import('./Home'));
const About = lazy(() => import('./About'));
```

```
function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/about">About</Link>
        </nav>
```

```
        { /* Use Suspense to show a fallback while loading */ }
        <Suspense fallback={<div>Loading...</div>}>
          <Switch>
            <Route exact path="/" component={Home} />
            <Route path="/about" component={About} />
          </Switch>
        </Suspense>
      </div>
    </Router>
  );
}

export default App;
```

9. What is Reconciliation in React?

- Reconciliation is the process React uses to determine what changes need to be made to the DOM based on differences between the Virtual DOM and the real DOM.

Reconciliation in React is the process of updating the user interface (UI) efficiently when something changes, like data or user interactions.

How it Works:

1. **Virtual DOM:** React creates a lightweight copy of the actual web page (called the **virtual DOM**).
2. **Changes Detection:** When you update something (like clicking a button to change text), React doesn't update the whole web page. Instead, it first updates the virtual DOM to see what has changed.
3. **Finding Differences:** React compares the new virtual DOM with the previous one and finds what has changed (for example, if just one button or a text changed).
4. **Updating the Actual DOM:** Based on the differences it found, React only updates the specific parts of the actual webpage (real DOM) that changed, instead of reloading the entire page.

Why It's Important:

- **Faster Performance:** Instead of refreshing the whole web page, React only updates the parts that need to be changed. This makes your app run faster.
- **Efficient Updates:** React saves time and effort by minimizing unnecessary updates to the webpage.

Example:

If you have a list of items and you add a new one, React will:

- Compare the old list and new list.
- See that only one new item was added.
- Update just that part of the webpage.

10. How does `React.memo` work?

- `React.memo` is a higher-order component that allows you to optimize functional components in React by memoizing them. It prevents unnecessary re-renders of a component by only re-rendering when its props change. This can improve the performance of your application, especially for components that render large amounts of data or complex UI structures.


```
import React from 'react';

const ChildComponent = React.memo(({ name }) => {
  console.log('Child rendered');
  return <h1>Hello, {name}!</h1>;
});

const ParentComponent = () => {
  const [name, setName] = React.useState('Alice');

  return (
    <div>
      <ChildComponent name={name} />
      <button onClick={() => setName('Bob')}>Change Name</button>
    </div>
  );
};

export default ParentComponent;
```



11. What is Lazy Loading in React?

- Lazy loading is a technique to delay loading components until they are actually needed, which improves the performance of the application by reducing the initial bundle size.

12. How do you handle form validation in React?

- Form validation in React can be handled either by custom validation logic within controlled components or by using third-party libraries such as **Formik** or **React Hook Form**.

```

import React, { useState } from 'react';

const SimpleForm = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!email || password.length < 6) {
      setError('Email required and password must be at least 6 characters');
    } else {
      alert('Form submitted!');
      setError('');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={email} onChange={(e) => setEmail(e.target.value)} />
      <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
      {error} && <p>{error}</p>
      <button type="submit">Submit</button>
    </form>
  );
};

export default SimpleForm;

```

13. What is Strict Mode in React?

- React Strict Mode is a tool for highlighting potential problems in an application. It performs additional checks and warnings for its descendants but doesn't render anything on the UI.

Key Features of Strict Mode:

- Identifies components with unsafe lifecycle methods.
- Warns about the usage of legacy API methods.
- Detects side effects within components.
- Ensures that components are resilient to future React versions.

14.) What are the Life Cycle Methods In React?

Lifecycle methods are hooks in React that allow developers to run code at specific points in a component's life cycle, such as when the component is created, updated, or removed. They are primarily used in class components and are crucial for managing side effects, fetching data, and handling component behavior throughout its lifespan.

Key Lifecycle Methods in Class Components:

1. **constructor(props):**

- Called before the component is mounted.
- Used to initialize state and bind methods.

2. **componentDidMount():**

- Invoked immediately after a component is mounted.
- Commonly used for data fetching, subscriptions, or setting up timers.

3. **shouldComponentUpdate(nextProps, nextState):**

- Determines whether the component should re-render based on changes in props or state.
- Returns **true** or **false**.

4. **componentDidUpdate(prevProps, prevState):**

- Called immediately after updating occurs.
- Can be used to perform operations after a component has updated (e.g., fetching data based on prop changes).

5. `componentWillUnmount()`:

- Invoked immediately before a component is unmounted and destroyed.
- Ideal for cleanup tasks, like canceling network requests or removing event listeners.

6. `componentDidCatch(error, info)`:

- Used for error handling in child components.
- Allows you to log errors and display fallback UI.