PW SKILLS

Cognizant- Frontend Developer

Interview Process

Round 1: HR Round
Round 2: Assignment
Round 3: Coding Round
Round 4: HR Round

Interview Questions

1. What are Higher Order Components?
2. Why is redux needed?
3. What is a pure component?
4. How is the store connected to react routes?
5. What are react Hooks?
6. Explain any other hooks except UseState and UseEffect.
7. Context api
8. Why are functional components better than class based components?
9. What is props drilling?
10. How objects can be copied. (Shallow copy and Deep copy)
11. What is closure?
12. What is lexical scoping?
13. What is prototype inheritance
14. Do u know server side rendering
15. What is PWA
16. Any idea about Node Js
17. Any idea on unit test cases
18. Can you explain your experience with HTML, CSS, JavaScript, and Angular?
19. What is the main goal of React Fiber?
20. What are controlled components?
21. What are uncontrolled components?
22. What is Lifting State Up in React?
23. What are the lifecycle methods of React?
24. What are portals in React?
25. What are the advantages of React?
26. What are the limitations of React?
27. What is the use of the react-dom package?
28. How to use InnerHtml in React?
29. How are events different in React?
30. How do you use decorators in React?
31. How to enable production mode in React?
32. What is strict mode in React?
33. What are React Mixins?

SOLUTIONS

Q1: What are Higher Order Components?
A1: Higher-Order Components (HOCs) are functions in React that take a component as an argument and return a new component with enhanced functionality. HOCs allow for code reuse, logic, and behavior across multiple components without modifying the original component. Common use cases include adding conditional rendering, handling authentication, and injecting additional props into components. An example of an HOC might be a function that adds logging functionality to a component, logging every time the component renders.

Q2: Why is Redux needed?
A2: Redux is needed to manage the state of an application in a predictable and centralized manner, especially in complex applications where multiple components need to share and update the same state. By using a single source of truth (the Redux store), Redux ensures that state changes are consistent, making debugging and testing easier. It also simplifies the state management by decoupling state logic from components, which enhances maintainability and scalability.

Q3: What is a pure component?
A3: A pure component in React is a component that only re-renders when its props or state change. React's `PureComponent` implements `shouldComponentUpdate` with a shallow comparison of props and state, preventing unnecessary re-renders and improving performance. Pure components are ideal for optimizing performance in scenarios where the component renders the same output given the same input.

Q4: How is the store connected to React routes?
A4: In a React application using Redux for state management, the Redux store is connected to React routes by wrapping the entire application or specific components in the `Provider` component from `react-redux`. This makes the Redux store available to all components in the application, including those rendered by React Router. Components connected to routes can then access the store using the `connect` function or `useSelector` and `useDispatch` hooks.

Q5: What are React Hooks?
A5: React Hooks are functions that let you use state and other React features in functional components. Introduced in React 16.8, Hooks enable functional components to manage state

(`useState`), run side effects (`useEffect`), and utilize other React features like context, refs, and reducers without writing class components. Hooks promote code reuse, improve readability, and simplify component logic by eliminating the need for lifecycle methods.

Q6: Explain any other hooks except useState and useEffect.
A6: **`useContext`**: This hook allows you to access values from a React Context directly within a functional component, without needing to use the `Context.Consumer` component. It's useful for managing global state, such as theme or user authentication data, across the application. **`useRef`**: This hook returns a mutable ref object that persists across renders. It's commonly used to access DOM elements directly, manage focus, or store mutable values that do not trigger re-renders. **`useMemo`**: This hook memoizes a computed value, recomputing it only when its dependencies change. It's useful for optimizing performance by avoiding expensive calculations on every render. **`useCallback`**: Similar to `useMemo`, this hook memoizes a function, ensuring that it only gets recreated when its dependencies change, which can prevent unnecessary re-renders when passing callbacks to child components.

Q7: Context API
A7: The Context API in React allows you to share values (such as themes, user information, or configuration data) across components without passing props down through every level of the component tree. It consists of two main components: `Context.Provider`, which provides the value to be shared, and `Context.Consumer`, which allows components to consume the value. The `useContext` hook simplifies accessing the context in functional components.

Q8: Why are functional components better than class-based components?
A8: Functional components are considered better than class-based components for several reasons:

- **Simplicity**: Functional components are simpler and easier to read, focusing only on rendering UI without lifecycle methods or state management.
- **Hooks**: Hooks bring the power of state and lifecycle features to functional components, eliminating the need for classes.
- **Performance**: Functional components avoid the overhead associated with class instances, resulting in better performance.
- **Testability**: Functional components are easier to test as they are more predictable and follow a pure function model.

Q9: What is props drilling?
A9: Props drilling refers to the process of passing data from a parent component to deeply nested child components through multiple layers of intermediary components. This can make the code harder to manage and maintain, as intermediary components may receive and pass down props that they do not directly use. The Context API or state management libraries like Redux are often used to avoid props drilling.

Q10: How objects can be copied. (Shallow copy and Deep copy)
A10:

- **Shallow Copy**: Copies an object's top-level properties, but nested objects or arrays are copied by reference, meaning changes to the nested objects affect both the original and the copy. In JavaScript, shallow copies can be made using `Object.assign({}, obj)` or the spread operator `{...obj}`.
- **Deep Copy**: Creates a copy of an object and all of its nested objects, ensuring that changes to the copy do not affect the original object. Deep copies can be made using methods like `JSON.parse(JSON.stringify(obj))`, although this method has limitations, or using libraries like Lodash with `_.cloneDeep(obj)`.

Q11: What is closure?
A11: A closure is a feature in JavaScript where an inner function has access to the variables and functions of its outer function, even after the outer function has returned. Closures allow for the creation of private variables and functions, as the inner function can "close over" the variables from the outer function's scope. This concept is essential for callbacks, event handlers, and creating factories in JavaScript.

Q12: What is lexical scoping?
A12: Lexical scoping refers to the scope of a variable being determined by its location within the source code at the time of writing, not at runtime. In JavaScript, functions have access to variables from their own scope and the scope of their parent functions, creating a chain of scopes that dictates how variables are resolved. This chain is known as the scope chain.

Q13: What is prototype inheritance?
A13: Prototype inheritance is a feature in JavaScript that allows objects to inherit properties and methods from other objects. Every JavaScript object has a prototype, which is another object that it inherits from. The `__proto__` property or `Object.create()` is used to set an object's

prototype. This inheritance model allows for the reuse of methods across multiple objects, leading to more efficient memory usage and code reuse.

Q14: Do you know server-side rendering?
A14: Yes, Server-Side Rendering (SSR) is a technique used to render React components on the server instead of the client. SSR improves the performance of web applications by sending a fully-rendered HTML page to the client, which can be displayed while the JavaScript is still loading. This technique enhances SEO and provides a better user experience on slow networks or devices.

Q15: What is PWA?
A15: PWA (Progressive Web Application) is a type of web application that uses modern web capabilities to deliver an app-like experience to users. PWAs are fast, reliable, and can work offline using service workers. They are installable on a user's device, similar to native apps, and can send push notifications. PWAs aim to provide a seamless user experience across different devices and network conditions.

Q16: Any idea about Node.js?
A16: Yes, Node.js is a JavaScript runtime built on Chrome's V8 engine that allows developers to run JavaScript on the server side. It is event-driven, non-blocking, and asynchronous, making it ideal for building scalable network applications. Node.js is commonly used for building REST APIs, real-time applications like chat apps, and server-side rendering for web applications. It has a rich ecosystem of modules and packages available via npm.

Q17: Any idea on unit test cases?
A17: Yes, unit testing involves testing individual units or components of a software application in isolation to ensure they work as expected. In JavaScript, frameworks like Jest, Mocha, and Jasmine are used for writing and running unit tests. Unit tests validate the behavior of functions, methods, or classes, helping to catch bugs early in development. They are a key part of Test-Driven Development (TDD) and continuous integration practices.

Q18: Can you explain your experience with HTML, CSS, JavaScript, and Angular?
A18: I have extensive experience in front-end development using HTML, CSS, and JavaScript to build responsive and interactive web applications. My experience with Angular includes developing dynamic single-page applications (SPAs), utilizing Angular's component-based architecture, services, and dependency injection. I have also worked with Angular's powerful

tools like Reactive Forms, Angular CLI for project scaffolding, and RxJS for handling asynchronous data streams.

Q19: What is the main goal of React Fiber?
A19: The main goal of React Fiber is to improve the rendering performance of React applications by enabling incremental rendering. React Fiber introduces a more flexible and efficient reconciliation algorithm that allows React to break down rendering work into smaller chunks and prioritize updates based on their importance. This results in smoother user experiences, especially in applications with complex UIs or animations.

Q20: What are controlled components?
A20: Controlled components in React are form elements (like input fields, text areas, or selects) whose values are controlled by the React state. In a controlled component, the form element's value is set by the state, and every time the user interacts with the form, an event handler updates the state. This gives you full control over the form data and makes it easier to enforce validation and other business logic.

Q21: What are uncontrolled components?
A21: Uncontrolled components are form elements in React where the data is handled by the DOM itself, not by React. Instead of using state to control the value of form elements, uncontrolled components rely on refs to access form values when needed. This approach is similar to traditional HTML form handling but gives you less control over form inputs compared to controlled components.

Q22: What is Lifting State Up in React?
A22: Lifting state up refers to the process of moving state from child components to a common parent component so that the state can be shared among multiple child components. This is often done to synchronize the state across components that need to share the same data or to enable a parent component to manage the logic that affects multiple child components. By lifting state up, React ensures a single source of truth for state management.

Q23: What are the lifecycle methods of React?
A23: React lifecycle methods are special methods that run at different stages of a component's life:

- **Mounting**: `constructor`, `render`, `componentDidMount`

- **Updating**: `shouldComponentUpdate`, `render`, `componentDidUpdate`
- **Unmounting**: `componentWillUnmount` These methods allow developers to execute code at specific points in a component's lifecycle, such as initializing data, performing clean-up tasks, or optimizing performance.

Q24: What are portals in React?
A24: Portals in React allow you to render a component's children into a different part of the DOM than where the component itself is located. This is useful for cases like modals, tooltips, or dropdowns that need to appear outside the normal DOM hierarchy for proper positioning and stacking. Portals are created using the `ReactDOM.createPortal()` method, which takes two arguments: the children to render and the DOM node where the children should be rendered.

Q25: What are the advantages of React?
A25: React offers several advantages, including:

- **Component-based architecture**: Encourages reusable and modular code.
- **Virtual DOM**: Improves performance by minimizing direct DOM manipulations.
- **One-way data binding**: Ensures predictable data flow and easier debugging.
- **Strong community and ecosystem**: Provides access to a wide range of tools, libraries, and resources.
- **Cross-platform development**: Allows for building web applications with React and mobile apps with React Native.

Q26: What are the limitations of React?
A26: Despite its advantages, React has some limitations:

- **Learning curve**: React's ecosystem, including concepts like JSX, Hooks, and state management, can be challenging for beginners.
- **Boilerplate code**: Managing state and props can lead to verbose code in larger applications.
- **SEO limitations**: React applications require server-side rendering for optimal SEO, which adds complexity.
- **Rapid changes**: The React ecosystem evolves quickly, requiring developers to stay updated with frequent changes and new best practices.

Q27: What is the use of the react-dom package?
A27: The `react-dom` package provides DOM-specific methods that enable you to render

React components to the DOM, manipulate the DOM, and manage the lifecycle of React components within the DOM. The most commonly used method is `ReactDOM.render()`, which renders a React component to a specific DOM node. `react-dom` also includes methods like `ReactDOM.createPortal()` for rendering components outside the parent component's DOM hierarchy.

Q28: How to use innerHTML in React?
A28: In React, `dangerouslySetInnerHTML` is used to set HTML content directly inside an element. It's called "dangerous" because it can expose your application to XSS (Cross-Site Scripting) attacks if not used carefully. To safely use `dangerouslySetInnerHTML`, ensure that the content you're injecting is sanitized and trusted. Example:

javascript
Copy code

```
function MyComponent({ htmlContent }) {
    return <div dangerouslySetInnerHTML={{ __html: htmlContent }} />;
}
```

This approach should be used sparingly and only when necessary.

Q29: How are events different in React?
A29: In React, events are handled differently than in traditional HTML. React uses synthetic events, which are cross-browser wrappers around the native browser events, providing a consistent API. Synthetic events also follow React's event delegation model, where events are handled at the top level rather than being attached directly to each DOM element. This improves performance and ensures that React can manage event handling efficiently.

Q30: How do you use decorators in React?
A30: Decorators are a way to modify or enhance components and other entities in React, often used with higher-order components or in state management (e.g., with MobX). Decorators are not a built-in feature of React but can be used with ES7/ES8 syntax by enabling them through Babel plugins. They are typically applied as a function preceding a class or method definition:

javascript
Copy code

```
@observer
class MyComponent extends React.Component {
```

```
    // component code
}
```

In this example, `@observer` is a decorator from MobX that automatically tracks the state and re-renders the component as needed.

Q31: How to enable production mode in React?
A31: To enable production mode in React, you need to build your application with the `NODE_ENV` environment variable set to "production." This can be done using a build tool like Webpack by running the following command:

bash
Copy code
```
NODE_ENV=production webpack
```

Alternatively, if you're using Create React App, the production build is enabled by default when you run:

bash
Copy code
```
npm run build
```

This optimizes the React build for production by minifying code, eliminating development-specific warnings, and enabling React's production optimizations.

Q32: What is strict mode in React?
A32: React's Strict Mode is a tool that helps developers identify potential issues in their applications by activating additional checks and warnings for its descendants. It does not render any visible UI and does not affect production builds, but it helps catch common problems such as deprecated APIs, side effects in component lifecycles, and unsafe coding practices. To use Strict Mode, wrap your components with `<React.StrictMode>` in the root of your application.

Q33: What are React Mixins?
A33: Mixins were a way to share behavior between components in React before the introduction of Hooks and higher-order components. They allowed you to define methods and lifecycle hooks that could be shared across multiple components. However, mixins were

removed from React in favor of other patterns like composition and Hooks, as they could lead to complex and difficult-to-maintain code due to issues with name collisions and implicit dependencies.