Advanced Questions 2:

Explain the React reconciliation algorithm. How does React optimize re-rendering?

- **Expected Answer**: The reconciliation algorithm in React is responsible for efficiently updating the DOM when state or props change. React uses a *Virtual DOM* (a lightweight in-memory representation of the real DOM) to minimize costly direct DOM manipulations. The algorithm works by comparing the current Virtual DOM with the new one and updating only the parts that have changed (diffing). Key optimizations include:
  - **Key Prop**: React uses the `key` prop to identify which items in a list are changed, added, or removed, thus optimizing the re-rendering process.
  - **Batched Updates**: React batches multiple state updates and re-renders to avoid unnecessary reflows.
  - **Fiber Architecture**: React's Fiber architecture enables more granular control over rendering, allowing React to split rendering work into chunks and prioritize important updates (such as animations or user input).

React's approach results in faster updates, minimal DOM manipulations, and better performance.

2. How does React handle error boundaries?

- **Expected Answer**: React provides **Error Boundaries** to catch JavaScript errors in any part of the component tree and display a fallback UI instead of crashing the entire app. Error boundaries are implemented using the `componentDidCatch` lifecycle method in class components or the `static getDerivedStateFromError` method.
  - **Example**:

```
jsx
Copy code
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    logErrorToMyService(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

- Error boundaries are crucial in production apps to avoid the entire app crashing if one component fails.

3. What is the difference between `componentWillMount`, `componentDidMount`, and `getDerivedStateFromProps`?

- **Expected Answer**: These are lifecycle methods used in **class components**. However, most of the methods have been deprecated in React 16.3+ in favor of the newer `useEffect` and `getDerivedStateFromProps`.

- `componentWillMount`: This method is called before the component is mounted. It is often used for setup tasks, but it is now **deprecated**.
- `componentDidMount`: This is called immediately after the component is mounted (i.e., inserted into the DOM). It is typically used for making AJAX requests, subscribing to external data sources, or triggering animations.
- `getDerivedStateFromProps`: This method is called before every render, both when the component is mounted and updated. It allows the component to update its state based on changes in props. This is a **static method** and is useful when you want to derive state from props.

4. What are controlled and uncontrolled components in React?

- **Expected Answer**:
  - **Controlled Components**: In a controlled component, the form element (e.g., `<input>`) is controlled by React state. The value of the form field is driven by React state, and updates to the field are handled by event handlers like `onChange`.
  - **Uncontrolled Components**: In an uncontrolled component, the form element's value is not controlled by React state. Instead, the value is stored in the DOM, and React only interacts with the element when needed, often using `ref` to access the value.

  Example of controlled vs uncontrolled components:

```
jsx
Copy code
// Controlled Component
function ControlledInput() {
  const [value, setValue] = useState('');

  return <input value={value} onChange={(e) => setValue(e.
target.value)} />;
}

// Uncontrolled Component
```

```jsx
function UncontrolledInput() {
  const inputRef = useRef(null);

  const handleSubmit = () => {
    alert(inputRef.current.value);
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

5. What is the purpose of `useEffect` in React, and how can you optimize it?

- **Expected Answer**: `useEffect` is a hook that allows you to run side effects in your function components. These side effects could include data fetching, subscriptions, or manual DOM manipulation. `useEffect` is called after every render, but you can optimize it by specifying dependencies to run the effect conditionally.

  Optimization:

  - **Empty Dependency Array (`[]`)**: Run the effect only once, like `componentDidMount`.
  - **Dependency Array**: Provide a list of dependencies to run the effect only when those values change.
  - **Cleanup Function**: Return a function from `useEffect` to clean up subscriptions, timers, or other side effects when the component unmounts or the effect is re-executed.

  Example:

```jsx
Copy code
useEffect(() => {
  const timer = setTimeout(() => console.log('Timer expire
d'), 1000);
  return () => clearTimeout(timer); // Cleanup
}, []); // Effect runs once when the component mounts
```

6. Explain React's Suspense and how it works.

- **Expected Answer**: React Suspense is a feature designed for **data fetching** and **code splitting** in React apps. Suspense allows you to "suspend" rendering of a component until some asynchronous task (like fetching data or loading a chunk of code) has completed.
- **Usage**: Suspense is often used with **React.lazy** to dynamically import components only when they are needed. Suspense can also be used for asynchronous data fetching with libraries like `React Query` or `Relay`.

  Example:

```jsx
Copy code
const OtherComponent = React.lazy(() => import('./OtherCom
ponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </Suspense>
  );
}
```

Fallback is shown while the component is being loaded.

7. What is Server-Side Rendering (SSR) and how is it different from Client-Side Rendering (CSR)?

- **Expected Answer**:
  - **SSR**: Server-Side Rendering is the process of rendering your React components on the server and sending the fully rendered HTML to the client. This approach improves **SEO** and **initial load performance** because the browser doesn't need to execute JavaScript to render the content.
  - **CSR**: Client-Side Rendering relies on the browser to render the React app. The browser loads an empty HTML shell and then fetches the necessary JavaScript files to render the UI. The app can be faster after the initial load because it doesn't need to make requests for full HTML pages, but the first load might be slower, and SEO can be harder.

Tools for SSR: Next.js is a popular React framework for SSR, which can also handle static site generation (SSG) and API routes.

8. How does React handle accessibility?

- **Expected Answer**: React encourages developers to follow accessibility best practices. Some strategies include:
  - **Semantic HTML**: Using HTML elements that convey meaning (e.g., `<button>`, `<form>`, `<input>`).
  - *aria- attributes*: Providing additional information to assistive technologies for better screen reader support (e.g., `aria-label`, `aria-live`).
  - **Focus management**: Using the `useRef` hook to manage focus programmatically.
  - **Keyboard events**: Ensuring interactive components are accessible via keyboard (e.g., `onKeyDown` for custom buttons).

React's built-in accessibility checks (like warnings for invalid DOM attributes) and libraries like `react-aria` help developers make their apps more accessible.

9. How would you implement lazy loading in React?

- **Expected Answer**: Lazy loading is the practice of loading components only when they are needed, which improves the app's initial loading performance.

In React, you can use `React.lazy` to dynamically import components and `Suspense` to handle the loading state.

Example:

```
jsx
Copy code
const LazyComponent = React.lazy(() => import('./LazyComponen
t'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

Practical Tasks

Task 1: Build a Pagination component with React

Create a pagination component that:

- Displays a list of items (e.g., 100 items).
- Shows 10 items per page.
- Allows navigating between pages (next, previous, and direct page numbers).
- Implements proper state handling and performance optimization to handle large lists efficiently.

Task 2: Build a Debounced Search Component

Create a search component where:

- The user types in a search box.
- A search is only triggered after 500ms of inactivity (debouncing).
- Show loading state while waiting for results.

Final Thoughts

To **stand out** in your React interview, demonstrate not just your knowledge of concepts but also:

- How you **optimize performance**.
- Your ability to **handle edge cases**.
- How you approach **scaling applications** (both in terms of code and performance).

Also, demonstrate that you know when to use **class components** vs. **functional components**, when to apply **Redux** or **Context API**, and how to handle real-world **complex state management** scenarios.