Advanced Questions:

1. What are React Hooks, and how do they improve React development?

- **Expected Answer**: React hooks allow developers to manage state and side-effects in functional components, making them more powerful. Hooks like `useState`, `useEffect`, `useReducer`, `useContext`, and `useRef` provide the ability to manage component logic without relying on class components. This reduces boilerplate code and simplifies component structure. It also promotes a functional programming style, which is easier to maintain and test.

2. What is `useCallback` and how does it differ from `useMemo`?

- **Expected Answer**:
  - `useCallback` is used to memoize functions, preventing them from being recreated on every render unless their dependencies change. It's useful when passing functions down to child components to avoid unnecessary re-renders.
  - `useMemo` is used to memoize values (like expensive calculations or objects) and will recompute the result only if one of its dependencies has changed. `useMemo` is used when you need to optimize performance by memoizing data, while `useCallback` is specifically for memoizing functions.

Example:

```
jsx
Copy code
const memoizedValue = useMemo(() => expensiveCalculation(),
[dependency]);
const memoizedCallback = useCallback(() => { doSomething() },
```

```
[dependency]);
```

3. What is the Context API in React, and when would you use it?

- **Expected Answer**: The Context API allows you to share state or data between components without having to pass props through every level of the component tree (i.e., "prop drilling"). It is useful for managing global state, such as user authentication status or theme settings, where the state needs to be accessed by many components at different levels of the component tree.

Example usage:

```
jsx
Copy code
const ThemeContext = React.createContext();

function App() {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={theme}>
      <ChildComponent />
    </ThemeContext.Provider>
  );
}

function ChildComponent() {
  const theme = useContext(ThemeContext);
  return <div>{theme === 'light' ? 'Light Mode' : 'Dark Mod
e'}</div>;
}
```

4. Explain how `useReducer` works and when you should use it instead of `useState`.

- **Expected Answer**: `useReducer` is used when state logic is complex, or when the next state depends on the previous state. It's similar to how reducers work in Redux. You use `useReducer` when:
  - You have complex state that involves multiple sub-values.
  - You want to manage state transitions in a centralized way.
  - You need to avoid multiple `setState` calls that might become difficult to track.

Example:

```
jsx
Copy code
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialStat
e);

  return (
    <div>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}
>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}
```

```
>Decrement</button>
    </div>
  );
}
```

5. How does React handle component re-renders?

- **Expected Answer**: React re-renders components when their state or props change. During the render, React compares the new JSX returned by the component with the previous JSX (using the Virtual DOM). If there is a difference, React updates the actual DOM. However, React will optimize re-renders:
  - If a component's state hasn't changed, React won't re-render it.
  - If props haven't changed, the child component won't re-render (unless it's forced via `forceUpdate`).
  - React uses techniques like "reconciliation" and "diffing" to make the update process faster.

React also provides tools like `shouldComponentUpdate`, `React.memo`, and `PureComponent` to fine-tune the performance and prevent unnecessary renders.

6. What is `React.memo` and how does it work?

- **Expected Answer**: `React.memo` is a higher-order component that memoizes a functional component and prevents unnecessary re-renders when the props have not changed. It works by shallowly comparing the previous and next props. If they are equal, React skips re-rendering the component.

Example:

```
jsx
Copy code
const MyComponent = React.memo(function MyComponent(props) {
  return <div>{props.text}</div>;
```

```
});
```

7. What is the `useRef` hook, and how is it different from `useState`?

- **Expected Answer**: `useRef` is used to persist values across renders without causing a re-render. It returns a mutable `ref` object which doesn't trigger a re-render when it is changed. It is commonly used to access DOM elements directly or store mutable values that don't need to trigger re-renders.

Example:

```
jsx
Copy code
function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

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

8. What are some common performance optimization techniques in React?

- **Expected Answer**:

- **React.memo**: Memoizes components to prevent re-renders when props don't change.
- **Lazy Loading**: Using `React.lazy` and `Suspense` to load components only when they are needed, improving the initial load time.
- **Code Splitting**: Dividing the app into smaller chunks to load only necessary code.
- **useMemo & useCallback**: Memoizing values and functions to prevent unnecessary recalculations and re-renders.
- **Virtualization**: Using libraries like `react-window` or `react-virtualized` to render only the visible elements in large lists or tables.
- **Debouncing and Throttling**: Reducing the number of API calls or event handler calls by limiting how often they are fired.

9. What is the difference between `useEffect` and `useLayoutEffect`?

- **Expected Answer**:
  - `useEffect` runs after the paint, meaning it does not block the browser's painting process. It is suitable for operations like fetching data, subscriptions, or DOM manipulations.
  - `useLayoutEffect` runs synchronously before the browser paints, which can be useful when you need to measure DOM elements or make DOM changes before the paint, to avoid flickering or layout shifts.

Example: If you need to measure the size of an element before the page renders, you should use `useLayoutEffect`.

10. What are some challenges or limitations you might face when working with React?

- **Expected Answer**:
  - **SEO issues**: Since React is a client-side library, SEO can be challenging for content that's dynamically generated. Server-side rendering (SSR) or static site generation (SSG) with tools like Next.js can help.

- **State Management Complexity**: As your app grows, managing state can become difficult. Tools like Redux or Context API help, but they also introduce complexity.
- **Performance Optimization**: React's re-rendering process can be expensive if not managed well. Techniques like memoization, avoiding unnecessary renders, and code splitting are crucial for maintaining performance in large apps.
- **Learning Curve**: React's learning curve can be steep, especially when understanding concepts like hooks, state management, and component lifecycle.

Bonus Practical Problem

Task: Build a **Login Form** with the following requirements:

1. Include a text field for the username and password.
2. Validate the form when submitted:
   - The username should not be empty.
   - The password should be at least 6 characters long.
3. Show error messages if the form validation fails.

Expected Answer:

```
jsx
Copy code
function LoginForm() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!username) {
      setError('Username is required');
    } else if (password.length < 6) {
```

```
      setError('Password must be at least 6 characters');
    } else {
      setError('');
      // Handle successful login
      console.log('Logging in', { username, password });
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
        placeholder="
```