React Advanced Topics

Last updated by | Subramanya Dixit | May 5, 2025 at 7:59 PM GMT+5:30

📘 Why Learn Advanced React Topics?

As you build larger, more complex applications in React, you'll face challenges such as performance optimization, state management at scale, custom hooks, and handling side effects efficiently. Mastering these advanced concepts will help you write more performant, reusable, and maintainable code.

🧠 Key Advanced Concepts

✅ Code Splitting and Lazy Loading

Code splitting allows you to load only the necessary code when it's needed, reducing the initial load time. React's `React.lazy` and `Suspense` enable lazy loading of components.

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

const OtherComponent = lazy(() => import('./OtherComponent'));

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

✅ Memoization for Performance: `React.memo` and `useMemo`

Use `React.memo` to prevent unnecessary re-renders of functional components. Similarly, `useMemo` can optimize expensive calculations.

```
const ExpensiveComponent = React.memo(function ExpensiveComponent({ value }) {
  console.log("Rendering ExpensiveComponent");
  return <div>{value}</div>;
});

const App = () => {
  const [count, setCount] = useState(0);
  return (
    <>
      <ExpensiveComponent value={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </>
  );
};
```

Use `useMemo` to memoize values or functions:

```
const memoizedValue = useMemo(() => expensiveComputation(input), [input]);
```

✅ Context API with useReducer

`useReducer` is a more powerful alternative to `useState` when dealing with complex state logic. It pairs well with the **Context API** to manage global application state.

```
import React, { createContext, useReducer } from "react";

const initialState = { count: 0 };
const reducer = (state, action) => {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const StateContext = createContext();

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <StateContext.Provider value={{ state, dispatch }}>
      <Counter />
    </StateContext.Provider>
  );
}

function Counter() {
  const { state, dispatch } = useContext(StateContext);

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

💡 Best Practices for Advanced React Development

- **Keep components small and focused:** Avoid creating large, monolithic components that handle too many responsibilities.
- **Optimize performance:** Use `React.memo`, `useMemo`, and `useCallback` to prevent unnecessary re-renders.
- **Component Design:** Favor **presentational** components that are focused on rendering UI, and **container** components that handle logic.
- **State Management:** Use `useReducer` for complex state logic, and avoid prop drilling with the Context API.
- **Avoid Anti-Patterns:** Be cautious of pattern misuse, such as overusing the Context API in large applications or relying on inline functions inside JSX.

📊 Visual Overview

```
graph TD
  A[React App] --> B[useReducer]
  A --> C[useMemo]
  A --> D[React.memo]
  A --> E[React.lazy]
  B --> F[State Management]
  C --> G[Memoization]
  D --> H[Component Optimization]
```

🛠️ Practice Exercises

1. Refactor a complex `useState` pattern to use `useReducer`.
2. Use `React.memo` to optimize a large list rendering.
3. Implement code-splitting in a large app by using `React.lazy`.
4. Optimize an expensive calculation with `useMemo` in a list of items.
5. Use the **Context API** and `useReducer` together for global state management.

❓ Quiz

1. What does `React.memo` do?
a) Caches component methods
✅ **b) Prevents re-renders of functional components**
c) Caches API responses
d) Automatically re-renders components

---

2. Which hook is best suited for handling complex state logic?
a) `useState`
✅ **b) `useReducer`**
c) `useEffect`
d) `useContext`

---

3. How do you optimize expensive calculations in React?
a) By using `React.memo`
b) By using `useState`
✅ **c) By using `useMemo`**
d) By using `useEffect`

---

4. What does `React.lazy` allow you to do?
✅ **a) Lazily load components to reduce initial load time**
b) Keep components loaded on demand
c) Prevent components from rendering
d) Simplify prop passing
