React Questions / Week 2 - The Compilation (22July-27July)

Created by	Ⓚ Kumar Nayan

❓ Q #1 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , ⚙️ Performance

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Explain the difference between `useMemo` and `useCallback`. When would you choose one over the other?

✅ **Answer:**

`useMemo` and `useCallback` are both React Hooks for performance optimization, but they memoize different things. `useMemo` caches the *result* of a function (a value), preventing expensive calculations from re-running on every render unless its dependencies change. You use it for computationally intensive operations like filtering a large list.

`useCallback` caches the *function definition* itself, ensuring the same function instance is returned between renders. This is crucial when passing callbacks as props to child components that are optimized with `React.memo`, as it prevents the child from re-rendering just because the parent created a new function

reference. In essence, `useCallback(fn, deps)` is a specialized version of `useMemo(() => fn, deps)`.

```jsx
// useMemo caches the calculated value
const filteredList = useMemo(() => {
  return largeList.filter(item => item.includes(search));
}, [search]);

// useCallback caches the function itself
const handleClick = useCallback(() => {
  setCount(count + 1);
}, [count]);
```

❓ Q #2 (🔴 Advanced)

Tags: 🚀 MAANG-level, 📦 State Management, ⚙️ Performance, 🏗️ Architecture

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Compare React Hook Form and Formik, focusing on their core philosophy, performance implications, and how they handle component re-renders.

✅ **Answer:**

React Hook Form (RHF) and Formik are both powerful form libraries, but they follow different philosophies. RHF is hook-based and prioritizes performance by using **uncontrolled components** and `refs`, which results in minimal re-renders. Formik is component-driven and declarative, using **controlled components**, which can cause more frequent re-renders as form state changes.

Feature	Formik	React Hook Form (RHF)
Philosophy	Declarative, component-driven	Hook-based, minimal re-renders
Performance	Slower with large forms	Faster due to fewer re-renders
Inputs	Controlled	Uncontrolled
Bundle Size	Bigger	Smaller

You should choose RHF for performance-critical applications with large or dynamic forms. Formik is a good choice if you prefer a declarative, component-based syntax and are working with forms with complex validation logic.

❓ Q #3 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , 🧱 `Components` , ⚙️ `Performance` , 🔁 `Rendering`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is `React.memo` , and how do `useCallback` and `React.memo` work together to prevent unnecessary re-renders?

✅ **Answer:**

`React.memo` is a Higher-Order Component (HOC) that memoizes a component. It prevents the component from re-rendering if its props have not changed. This is a shallow comparison of props.

However, if a parent component passes a function as a prop to a `React.memo` -wrapped child, the child will still re-render every time the parent does. This is because the function is redefined on each render of the parent, creating a new reference.

This is where `useCallback` becomes essential. By wrapping the function in `useCallback` in the parent component, you ensure that the function reference remains stable across re-renders (as long as its dependencies don't change). When this stable function reference is passed to the `React.memo` child, `React.memo` sees that the prop hasn't changed and correctly skips the re-render.

```jsx
// Parent Component
function Parent() {
  const [count, setCount] = useState(0);

  // Without useCallback, this function is a new reference on every render// With useCallback, the reference is stable
  const memoizedHandleClick = useCallback(() => {
    console.log('Button clicked!');
  }, []);

  return <Child onClick={memoizedHandleClick} />;
}

// Child Component wrapped in React.memo
const Child = React.memo(({ onClick }) => {
  console.log('Child rendered');
  // This logs only when props actually change
  return <button onClick={onClick}>Click Me</button>;
});
```

❓ Q #4 (🔴 Advanced)

Tags: 🧩 `General Interview` , 🌐 `Routing` , 🏗️ `Architecture`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Compare `BrowserRouter`, `HashRouter`, and `MemoryRouter`. When is it appropriate to use each?

✅ **Answer:**

These are three different router implementations in `react-router-dom` for different environments:

- `BrowserRouter`: This is the most common router for modern web apps. It uses the HTML5 History API to keep the UI in sync with the URL, resulting in clean URLs (e.g., `/about`). It requires server-side configuration to handle client-side routes, typically by redirecting all requests to `index.html`.
- `HashRouter`: This router uses the URL hash (`#`) to manage routing (e.g., `/#/about`). It's ideal for static hosting environments (like GitHub Pages) where you cannot configure the server. Since everything after the `#` is handled client-side, it works out-of-the-box without server setup but results in less clean URLs and is not ideal for SEO.
- `MemoryRouter`: This router stores the URL history in memory and does not interact with the browser's address bar. Its primary use case is for testing components in isolation or in non-browser environments like React Native.

❓ Q #5 (🟢 Basic)

Tags: 🧩 General Interview, 🔧 Hooks

🧠 **Question:**

What are the two fundamental "Rules of Hooks"?

✅ **Answer:**

There are two main rules for using Hooks that must be followed:

1. **Only Call Hooks at the Top Level**: Hooks must be called at the top level of a React function or a custom hook. They cannot be called inside loops, conditions, or nested functions. This ensures that hooks are called in the same order on every render, which is how React preserves state between calls.
2. **Only Call Hooks from React Functions**: Hooks can only be called from React function components or from custom hooks. They cannot be called

from regular JavaScript functions or class components.

❓ Q #6 (🟡 Intermediate)

Tags: 🚀 MAANG-level , ⚙️ Performance , 🔍 Real-World Scenario

🧠 Question:

Describe a real-world scenario where `useMemo` is crucial for performance, and provide a code example.

✅ Answer:

A crucial scenario for `useMemo` is when a component renders a large list of data that needs to be filtered or transformed based on user input. Without `useMemo`, this expensive filtering operation would run on every single re-render of the component, even if the re-render was caused by an unrelated state change.

By wrapping the filtering logic in `useMemo`, we ensure the calculation only runs when the source list or the filter criteria (the dependencies) actually change, significantly improving performance.

```
jsxfunction FilteredFruitList({ allFruits }) {
 const [search, setSearch] = useState('');
 const [unrelatedState, setUnrelatedState] = useState(0);

 // This expensive filtering only re-runs when 'search' or 'allFruits' changes,// not when 'unrelatedState' changes.
 const filteredList = useMemo(() => {
   console.log("Filtering list...");
   return allFruits.filter(fruit =>
     fruit.toLowerCase().includes(search.toLowerCase())
   );
 }, [search, allFruits]);

 return (
   <div>
     <input
       type="text"
       placeholder="Search fruits..."
       value={search}
       onChange={(e) => setSearch(e.target.value)}
     />
     {
/* This button causes a re-render, but filtering is skipped */ }
     <button onClick={() => setUnrelatedState(c => c + 1)}>Re-render</button>
     <ul>
       {filteredList.map((item) => (
         <li key={item}>{item}</li>
       ))}
     </ul>
   </div>
```

```
);
}
```

❓ Q #7 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🔧 `Hooks`

🧠 **Question:**

What are the two primary use cases for the `useRef` hook in React?

✅ **Answer:**

The `useRef` hook serves two main purposes:

1. **Accessing DOM Elements**: Its primary and most common use is to get a direct reference to a DOM element. You attach the ref to an element in your JSX, and you can then access and manipulate that DOM node directly (e.g., to manage focus, trigger animations, or integrate with third-party DOM libraries).
2. **Persisting Mutable Values Across Renders**: `useRef` can hold a mutable value in its `.current` property that persists across component re-renders without causing a re-render itself when the value is changed. This is useful for storing values like timers, subscription IDs, or previous state/props that you don't want to trigger a UI update when they change.

❓ Q #8 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🌐 `Routing` , ⚙️ `Performance`

🧠 **Question:**

What are `React.lazy()` and `<Suspense>` , and how do they work together to implement code-splitting?

✅ **Answer:**

`React.lazy()` and `<Suspense>` are React's built-in tools for code-splitting, a technique to improve initial load performance.

- `React.lazy()` is a function that lets you render a dynamically imported component as a regular component. It takes a function that must call a dynamic `import()` and returns a promise that resolves to a module with a `default` export containing a React component.
- `Suspense` is a component that lets you specify a loading indicator (a "fallback" UI) if the components in its tree are not yet ready to render.

They work together by allowing you to wrap a lazy-loaded component in a `Suspense` boundary. While React waits for the lazy component's code to be downloaded and loaded, it will render the `fallback` UI provided to `Suspense`. This prevents the user from seeing a blank screen or an error.

```
jsximport React, { Suspense } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const HomePage = React.lazy(() => import('./pages/HomePage'));
const AboutPage = React.lazy(() => import('./pages/AboutPage'));

function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading page...</div>}>
        <Routes>
          <Route path="/" element={<HomePage />} />
          <Route path="/about" element={<AboutPage />} />
        </Routes>
      </Suspense>
    </Router>
  );
}
```

❓ Q #9 (🟢 Basic)

Tags: 🧩 General Interview , 🌐 Routing

🧠 Question:

What is the difference between using a standard `<a>` tag and React Router's `<Link>` component for navigation?

✅ Answer:

The key difference is how they handle navigation. A standard `<a>` tag, when clicked, will trigger a full-page refresh, causing the browser to request a new HTML document from the server. This reloads all scripts and styles, losing any client-side state.

React Router's `<Link>` component, on the other hand, enables client-side routing. When clicked, it intercepts the navigation event, updates the URL in the browser's address bar using the History API, and re-renders only the necessary components without a full page reload. This creates the fast, seamless experience of a Single-Page Application (SPA).

❓ Q #10 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 📦 State Management , ⚙️ Performance , 💥 Pitfall

🧠 **Question:**

How does React Hook Form achieve better performance than many other form libraries?

✅ **Answer:**

React Hook Form achieves superior performance primarily by minimizing re-renders. It does this by using **uncontrolled components** and leveraging `refs` to subscribe to input changes instead of using component state.

When you `register` an input, RHF attaches a `ref` to the DOM element and listens for changes directly. The component managing the form does not re-render on every keystroke. It only re-renders when necessary, such as when there's a validation error or during form submission. This subscription-based model is much more efficient than the controlled component approach used by libraries like Formik, where the component's state is updated on every input change, triggering a re-render.

❓ Q #11 (🟢 Basic)

Tags: 🧩 `General Interview` , 🔧 `Hooks`

🧠 **Question:**

What is a custom hook in React, and what is the primary motivation for creating one?

✅ **Answer:**

A custom hook is a reusable JavaScript function whose name starts with "use" and that can call other hooks. The primary motivation for creating custom hooks is to **extract and share stateful logic** between multiple components.

If you find yourself writing the same logic (like managing state, subscriptions, or side effects with `useState` and `useEffect`) in several different components, you can extract that logic into a custom hook. This keeps your components clean, reduces code duplication, and makes the logic modular and reusable across your application.

❓ Q #12 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🌐 `Routing`

🧠 **Question:**

What is the purpose of the `<Outlet>` component in React Router v6?

✅ **Answer:**

The `<Outlet>` component is used within parent route components to render their child route components. It acts as a placeholder. When you define nested routes, the parent route renders its own layout, and the `<Outlet>` component within that layout determines where the matched child route's element should be rendered.

For example, if you have a `/dashboard` route with nested routes like `/dashboard/profile` and `/dashboard/settings`, the `Dashboard` component would contain the shared layout (e.g., a sidebar) and an `<Outlet>` to render either the `Profile` or `Settings` component based on the current URL.

```jsx
// Route Configuration
<Routes>
 <Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<Profile />} />
  <Route path="settings" element={<Settings />} />
 </Route>
</Routes>

// Dashboard Component
function Dashboard() {
 return (
  <div>
   <h1>Dashboard</h1>
   <nav>...</nav>
   {
/* Child route component will be rendered here */ }
   <Outlet />
  </div>
 );
}
```

❓ Q #13 (🟢 Basic)

Tags: 🧩 General Interview , 📦 State Management

🧠 **Question:**

What does the `register` function in React Hook Form do?

✅ **Answer:**

The `register` function is a core method from the `useForm` hook that connects an input element to the form's state. When you spread `{...register("inputName")}` onto an input, it provides the necessary props (`ref` , `name` , `onChange` , `onBlur`) to track the input's value and validation status without causing re-renders on every change.

You can also pass a second argument to `register` to define validation rules directly on the input, such as `required`, `minLength`, or a `pattern`.

```jsx
<form onSubmit={handleSubmit(onSubmit)}>
 {
 /* Basic registration */ }
 <input {...register("firstName")} />

 {
 /* Registration with validation */ }
 <input {...register("email", { required: "Email is required" })} />

 <input type="submit" />
</form>
```

❓ Q #14 (🟡 Intermediate)

Tags: 🧩 General Interview, 🌐 Routing

🧠 **Question:**

What is a "Route Guard" in React, and how can one be implemented conceptually?

✅ **Answer:**

A Route Guard is not a built-in feature of React or React Router but a logical pattern you implement to protect certain routes from unauthorized access. The core idea is to create a wrapper component that checks for an authentication condition (e.g., if a user is logged in) before rendering the intended component.

If the condition is met, it renders the protected component. If not, it typically redirects the user to a login page or another public route. This can be implemented using a wrapper component that conditionally renders the `children` or a `<Navigate>` component from React Router.

❓ Q #15 (🟢 Basic)

Tags: 🧩 General Interview, 🌐 Routing

🧠 **Question:**

How do you access dynamic URL parameters in a component, such as the `productId` from `/product/:productId`?

✅ **Answer:**

You use the `useParams` hook from React Router. This hook returns an object containing key-value pairs of the dynamic segments of the URL. The key

corresponds to the dynamic parameter name you defined in your `<Route>` path.

```jsximport { useParams } from 'react-router-dom';

// In your route config: <Route path="/product/:productId" element={<ProductDetail />} />

function ProductDetail() {
  // useParams will return an object like { productId: '123' }
  const { productId } = useParams();

  return <div>Showing details for product ID: {productId}</div>;
}
```

❓ Q #16 (🔴 Advanced)

Tags: 🚀 MAANG-level , ⚙️ Performance , 💥 Pitfall

🧠 Question:

When should you *avoid* using memoization like `React.memo` or `useMemo`?

✅ Answer:

While memoization is a powerful optimization tool, it's not always beneficial and can sometimes harm performance if overused. You should avoid it in these situations:

1. **For Simple, Lightweight Components**: If a component is simple and its re-rendering process is very fast (e.g., a simple button or icon), the overhead of performing the props comparison in `React.memo` can be more expensive than just re-rendering the component.
2. **When Props Always Change**: If a component almost always receives new props on every re-render, memoization is pointless. `React.memo` will run the comparison, find the props are different, and re-render anyway, adding unnecessary overhead.

The key is to use memoization strategically on components that are computationally expensive to render or that are rendered frequently with the same props.

❓ Q #17 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , 📦 State Management

🧠 Question:

Why would you use `useRef` to store a value instead of a state variable from `useState`?

✅ **Answer:**

You would choose `useRef` over `useState` when you need a value that **persists across re-renders** but **does not need to trigger a re-render** when it changes.

- `useState` is for data that is part of your component's render output. When you update it with the setter function, React schedules a re-render to reflect the change in the UI.
- `useRef` holds a value in its `.current` property. Mutating this property does *not* trigger a re-render. This makes it perfect for managing "side-data" that isn't directly involved in the visual rendering, such as timer IDs from `setInterval`, DOM node references, or storing a previous state value for comparison.

❓ Q #18 (🟢 Basic)

Tags: 🧩 General Interview , 📦 State Management

🧠 **Question:**

What is the purpose of Yup in the context of React forms?

✅ **Answer:**

Yup is a JavaScript schema builder used for value parsing and validation. In the context of React forms, particularly with libraries like Formik and React Hook Form, Yup allows you to define a centralized validation schema for your form's data.

Instead of writing validation logic inside your components, you create a Yup schema object that defines the shape of your form data and the validation rules for each field (e.g., a field is a required string, a valid email, or a number within a certain range). This approach keeps validation logic separate, organized, and reusable.

❓ Q #19 (🟢 Basic)

Tags: 🧩 General Interview , 🌐 Routing

🧠 **Question:**

How does `<NavLink>` differ from `<Link>`, and what is its primary use case?

✅ **Answer:**

`<NavLink>` is a special version of the `<Link>` component that knows whether or not it is "active." Its primary use case is for building navigation menus (like sidebars or navbars) where you want to apply a specific style (e.g., a different color or font weight) to the link corresponding to the currently active page. It provides extra props, such as a `className` or `style` function, that let you conditionally apply styles based on its `isActive` status.

```jsx
<NavLink
 to="/messages"
 className={({ isActive }) => (isActive ? 'active-link' : 'inactive-link')}
>
 Messages
</NavLink>
```

❓ Q #20 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 📦 State Management , 🔍 Real-World Scenario

🧠 **Question:**

In which specific scenarios would you recommend a team use React Hook Form over Formik?

✅ **Answer:**

Based on the provided notes, you should strongly recommend React Hook Form over Formik in the following scenarios:

1. **Performance is a Top Priority**: If the application contains large, complex forms with many fields, RHF's minimal re-render strategy will provide a significantly faster and smoother user experience.
2. **Forms with Dynamic Fields**: For forms where fields are frequently added or removed, RHF's performance benefits become even more pronounced.
3. **Uncontrolled Component Preference**: If the team's philosophy leans towards uncontrolled components to isolate form logic from component state, RHF is the natural choice.
4. **Minimizing Bundle Size**: In projects where every kilobyte matters, RHF's smaller bundle size is a clear advantage over Formik.

❓ Q #21 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔁 Rendering , 💥 Pitfall

🧠 **Question:**

What are the three main reasons a React component re-renders, and which one often leads to performance issues in deeply nested component trees?

✅ **Answer:**

A React component typically re-renders due to one of three reasons:

1. A change in its own **state** (e.g., via `useState` setter).
2. A change in its **props**.
3. The **re-rendering of its parent component**.

The third reason—a parent component re-rendering—is often the cause of performance bottlenecks in large or deeply nested applications. When a parent re-renders, it causes all of its children to re-render by default, even if their props have not changed. This cascading effect can lead to many unnecessary render cycles, which is precisely the problem that optimization tools like `React.memo`, `useCallback`, and `useMemo` are designed to solve.

❓ Q #22 (🟢 Basic)

Tags: 🧩 General Interview, 📦 State Management, 🔧 Hooks

🧠 **Question:**

What is the `formState` object in React Hook Form, and what are some of its useful properties?

✅ **Answer:**

The `formState` object, returned by the `useForm` hook, contains information about the current state of the form. It's read-only and is used to track metadata without causing unnecessary re-renders unless a specific property is subscribed to.

Some of its most useful properties include:

- `errors`: An object containing validation errors for each field.
- `isDirty`: A boolean that is `true` if the user has modified any of the inputs.
- `isSubmitting`: A boolean that is `true` while the form submission handler is executing.

```jsx
const { formState: { errors, isSubmitting } } = useForm();

return (
  <form>
    {
```

```
      /* ... inputs ... */ }
        {errors.email && <p>Email is required</p>}
        <button type="submit" disabled={isSubmitting}>
          {isSubmitting ? 'Submitting...' : 'Submit'}
        </button>
      </form>
    );
```

❓ Q #23 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 📦 State Management , 🔍 Real-World Scenario

🧠 Question:

How would you implement schema-based validation using Yup in React Hook Form?

✅ Answer:

To use Yup for schema validation with React Hook Form, you need to install the `@hookform/resolvers` package. Then, you create your Yup validation schema and pass it to the `useForm` hook inside the `resolver` option, using `yupResolver`.

This approach cleanly separates your validation logic from your component and lets RHF use your schema to validate the form data automatically upon submission.

```
jsximport { useForm } from "react-hook-form";
import { yupResolver } from "@hookform/resolvers/yup";
import * as Yup from "yup";

// 1. Define the validation schema with Yup
const schema = Yup.object().shape({
  name: Yup.string().required("Name is required"),
  email: Yup.string().email("Must be a valid email").required("Email is required"),
});

function MyForm() {
  const { register, handleSubmit, formState: { errors } } = useForm({
// 2. Connect the schema to the form using the resolver
    resolver: yupResolver(schema),
  });

  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("name")} />
      <p>{errors.name?.message}</p>

      <input {...register("email")} />
```

```
    <p>{errors.email?.message}</p>

    <button type="submit">Submit</button>
  </form>
 );
}
```

❓ Q #24 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , 💥 Pitfall

🧠 **Question:**

Why can't you call a Hook inside a conditional statement? What is the underlying reason for this rule?

✅ **Answer:**

You cannot call a Hook inside a conditional statement because React relies on the **order of Hook calls** to be consistent on every render of a component. Internally, React maintains an array of state and other hook data for each component. When you call `useState` , `useEffect` , etc., React retrieves the corresponding data from its internal array based on the order in which the hook was called.

If you place a hook inside a condition, the number and order of hook calls could change between renders. For example, if the condition is `true` on the first render but `false` on the second, the order of subsequent hooks would be shifted, causing React to retrieve the wrong state and leading to unpredictable bugs. Enforcing that hooks are always called at the top level ensures this order is preserved.

❓ Q #25 (🟢 Basic)

Tags: 🧩 General Interview , 🌐 Routing , 🔧 Hooks

🧠 **Question:**

What is the `useNavigate` hook used for in React Router?

✅ **Answer:**

The `useNavigate` hook gives you a function that you can use to **programmatically navigate** to different routes within your application. It is the replacement for the `useHistory` hook from earlier versions of React Router.

You would use it inside event handlers, `useEffect` hooks, or other functions where you need to trigger a route change based on some logic, such as after a form

submission is successful or a user clicks a button that isn't a standard `<Link>`.

```
jsximport { useNavigate } from 'react-router-dom';

function LoginForm() {
 const navigate = useNavigate();

 const handleLogin = async () => {
  const success = await loginUser();
  if (success) {

// Programmatically navigate to the dashboard after successful login
    navigate('/dashboard');
  }
 };

 return <button onClick={handleLogin}>Log In</button>;
}
```

❓ Q #26 (🟡 Intermediate)

Tags: 🧩 General Interview , 🌐 Routing

🧠 Question:

When would you choose to use the `useNavigate` hook over the `<Link>` component for routing?

✅ Answer:

You use `<Link>` for standard, user-driven navigation where a user clicks on a visual element to go to a new page. It's the declarative way to handle routing in JSX.

You use the `useNavigate` hook for programmatic or imperative navigation, where you need to change routes based on logic within a function. This is common in scenarios like:

- Redirecting a user after a successful form submission.
- Navigating away from a page after a certain action is completed inside a `useEffect`.
- Handling navigation from a non-link UI element, like a button in a modal.

❓ Q #27 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🔧 Hooks , 🔍 Real-World Scenario

🧠 **Question:**

How can you use the `useRef` hook to track the previous value of a state variable, and why is this useful?

✅ **Answer:**

You can track a state's previous value by updating a `useRef` hook's `.current` property *after* the component has rendered. This is typically done inside a `useEffect` hook. The effect runs after the render, so you can store the current state value in the ref, and in the *next* render, the ref will still hold the value from the *previous* render.

This is useful for making comparisons, such as detecting when a specific prop or state value has changed, or for creating dependencies in effects that need to compare the current and previous values.

```
jsxfunction StateChangeTracker({ value }) {
 const [currentValue, setCurrentValue] = useState(value);
 const prevValueRef = useRef();

 useEffect(() => {

// This runs *after* the render, so it updates the ref for the *next* render
   prevValueRef.current = currentValue;
 }, [currentValue]);

 const previousValue = prevValueRef.current;

 return (
   <div>
     <p>Current Value: {currentValue}</p>
     <p>Previous Value: {previousValue}</p>
     <button onClick={() => setCurrentValue(val => val + 1)}>Increment</button>
   </div>
 );
}
```

❓ Q #28 (🔴 Advanced)

Tags: 🚀 MAANG-level , 📦 State Management , 🏗️ Architecture

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is the purpose of the `control` object from the `useForm` hook in React Hook Form, and when is it necessary?

✅ **Answer:**

The `control` object in React Hook Form is essential for integrating **controlled components** from external UI libraries (like Material-UI, Ant Design, or React-Select) into your form.

While RHF primarily uses uncontrolled components via the `register` method, these UI libraries often manage their own internal state and don't expose the `ref` needed for `register`. The `control` object contains methods to properly connect these controlled components to the RHF state and validation engine. It's typically used with a wrapper component like `<Controller>` to bridge the gap, ensuring that RHF can still manage the form's state and validation without re-rendering the entire form on every change.

❓ Q #29 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔧 Hooks , 🏗️ Architecture

🧠 **Question:**

What are the key design principles and rules you must follow when creating a custom hook in React?

✅ **Answer:**

When creating a custom hook, you must follow these key principles:

1. **Name Must Start with** `use`: This is a convention that allows ESLint plugins to identify it as a hook and enforce the Rules of Hooks. For example, `useFetchData` or `useWindowSize`1.
2. **Extract Reusable Stateful Logic**: The main goal is to extract logic that is used in multiple components, not just any function. This logic must involve other hooks like `useState` or `useEffect`2.
3. **Follow the Rules of Hooks**: A custom hook must adhere to the same rules as built-in hooks. It should only call other hooks at the top level and not inside conditions, loops, or nested functions2.
4. **Be a Pure Function**: The code inside a custom hook re-runs on every render, so it should be pure. It should not contain side effects that are not managed by `useEffect`2.

❓ Q #30 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔁 Rendering , ⚙️ Performance

🧠 **Question:**

Explain why a React component might re-render even if its own state and props have not changed.

✅ **Answer:**

A component will re-render even if its own state and props are unchanged if its **parent component re-renders**2. By default, whenever a parent component re-renders (due to its own state or prop change), React will recursively re-render all of its child components.

This is a core behavior of React's rendering mechanism and a common source of performance issues in large applications. It's the primary reason for using optimization tools like `React.memo` to prevent this cascading re-render effect for children whose props haven't actually changed2.

❓ Q #31 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 📦 State Management , ⚙️ Performance , 🏗️ Architecture

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Explain the difference between controlled and uncontrolled components in the context of form management, and how this choice impacts performance.

✅ **Answer:**

The difference lies in who manages the state of the form input:

- **Controlled Components**: The component's state is the "single source of truth." The value of the input is driven by React state, and its `onChange` handler updates that state. This causes a re-render on every keystroke. Formik primarily uses this approach3.
- **Uncontrolled Components**: The DOM manages its own state. You use a `ref` to get the value from the DOM directly when you need it (e.g., on submission). This avoids re-renders on every keystroke. React Hook Form uses this approach, which is why it's generally more performant3.

Impact on Performance: Uncontrolled components offer significantly better performance for forms, especially large ones, because they prevent the component from re-rendering every time the user types a character. Controlled components provide more direct control and are easier to integrate with a declarative state management style but can lead to performance issues if not optimized3.

❓ Q #32 (🟡 Intermediate)

Tags: 🧩 General Interview , 🌐 Routing , 🏗️ Architecture

🧠 **Question:**

How do you create nested routes in React Router v6, and what role does the `<Outlet>` component play in this architecture?

✅ **Answer:**

You create nested routes by nesting `<Route>` components inside another `<Route>` component. The parent `<Route>` defines a layout or wrapper component, and the child `<Route>`s define the components that should be rendered within that layout2.

The `<Outlet>` component is a crucial part of this pattern. It acts as a **placeholder** within the parent route's component. You place `<Outlet />` in the parent's JSX to designate where the matched child route's element should be rendered. This allows you to create shared layouts (like dashboards with sidebars) where only a portion of the UI changes based on the nested URL2.

❓ Q #33 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , ⚙️ Performance

🧠 **Question:**

In what specific scenario does `useCallback` become critical for performance optimization, particularly in relation to child components?

✅ **Answer:**

`useCallback` is most critical when you are passing a function as a prop to a **child component that is wrapped in `React.memo`** 42.

Without `useCallback`, the function in the parent component is re-created on every render, resulting in a new reference. When this new function reference is passed as a prop to the memoized child, `React.memo` sees it as a "new" prop and re-renders the child unnecessarily. By wrapping the function in `useCallback`, you ensure the function reference remains stable between renders (as long as its dependencies don't change), allowing `React.memo` to correctly skip the re-render42.

❓ Q #34 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 📦 `State Management` , ⚙️ `Performance`

🧠 **Question:**

How does React Hook Form use the `formState` object to optimize re-renders, and why is it better than watching the entire form state?

✅ **Answer:**

React Hook Form optimizes re-renders using a subscription-based model for `formState`. The `formState` object is a proxy that tracks which properties (like `errors`, `isDirty`, `isSubmitting`) your component is actually using. The component will only re-render if one of those specific, subscribed-to properties changes3.

This is far more performant than watching the entire form state, because your component isn't forced to re-render for every single form state update. For example, if your component only uses `formState.errors`, it won't re-render just because `isDirty` changes. This granular subscription prevents unnecessary render cycles3.

❓ Q #35 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , 🌐 `Routing` , ⚙️ `Performance`

🧠 **Question:**

Describe how you would implement route-based code-splitting in a React application to improve initial load time.

✅ **Answer:**

You can implement route-based code-splitting using `React.lazy()` and `<Suspense>`. This technique splits the JavaScript bundle, so the user only downloads the code for the specific route they are visiting, improving the initial page load time52.

The process is:

1. Import components for your routes using `React.lazy()`. This function takes a function that must call a dynamic `import()`.
2. Wrap your `<Routes>` component with a `<Suspense>` boundary.
3. Provide a `fallback` prop to `<Suspense>`, which is a loading indicator (e.g., a spinner or text) that will be displayed while the lazy-loaded component's code is being fetched.

```jsx
import React, { Suspense } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const HomePage = React.lazy(() => import('./pages/HomePage'));
const AboutPage = React.lazy(() => import('./pages/AboutPage'));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/" element={<HomePage />} />
          <Route path="/about" element={<AboutPage />} />
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}
```

❓ Q #36 (🟡 Intermediate)

Tags: 🧩 General Interview , 🌐 Routing , 🏗️ Architecture

🧠 **Question:**

What is the critical difference in server configuration requirements between `BrowserRouter` and `HashRouter`, and how does this influence your choice?

✅ **Answer:**

The critical difference is that `BrowserRouter` **requires server-side configuration, while** `HashRouter` **does not**.

- `BrowserRouter` uses clean URLs (e.g., `/about`). If a user directly accesses or refreshes a page at `/about`, the server must be configured to serve the main `index.html` file for that path. Otherwise, it will result in a 404 error62.
- `HashRouter` uses a hash (`#`) in the URL (e.g., `/#/about`). Everything after the hash is handled on the client-side, so the server only ever needs to handle requests for the root URL (`/`). This works out-of-the-box on static hosting platforms like GitHub Pages62.

You choose `BrowserRouter` for modern web apps with server access for a better user experience and SEO. You choose `HashRouter` for projects on static hosting where you cannot configure the server2.

❓ Q #37 (🟢 Basic)

Tags: 🧩 General Interview , 📦 State Management

🧠 **Question:**

What is the purpose of wrapping your form submission handler with the `handleSubmit` function from React Hook Form?

✅ **Answer:**

The `handleSubmit` function from React Hook Form acts as a middleware. Its purpose is to first **run all validation checks** associated with your form inputs. Only if all validations pass will it then call your provided submission handler function (e.g., `onSubmit`) with the collected form data32. This prevents your submission logic from running with invalid data and separates the validation logic from the submission logic.

❓ Q #38 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🔁 `Rendering` , ⚙️ `Performance` , 💥 `Pitfall`

🧠 **Question:**

Why might `React.memo` fail to prevent a re-render for a component even if the data passed in a prop appears to be visually the same?

✅ **Answer:**

`React.memo` performs a **shallow comparison** of props. It fails to prevent a re-render if a prop is a non-primitive type like an object, array, or function. Even if these props contain the exact same data, they are created as new references in memory on each render of the parent component.

Because the references are different, the shallow comparison in `React.memo` evaluates to `false`, and it triggers a re-render. This is a common pitfall that requires using `useMemo` to memoize objects/arrays and `useCallback` to memoize functions to ensure they have stable references across renders2.

❓ Q #39 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🌐 `Routing` , 🧪 `Testing`

🧠 **Question:**

When would you use `MemoryRouter` instead of `BrowserRouter` or `HashRouter`?

✅ **Answer:**

You would use `MemoryRouter` in environments where there is no browser address bar to manipulate. Its two primary use cases are:

1. **Testing**: It's ideal for testing routing logic and components in isolation in a Node.js environment (like with Jest), as it doesn't depend on a browser's history API62.
2. **Non-Browser Environments**: It is used in environments like React Native where routing is managed entirely in memory and not tied to a URL62.

❓ Q #40 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🔧 `Hooks` , ⚙️ `Performance`

🧠 **Question:**

Besides expensive calculations, what is another crucial use case for `useMemo` related to props and child component re-renders?

✅ **Answer:**

Another crucial use case for `useMemo` is to preserve **referential equality** for non-primitive props (objects or arrays) that are passed to child components optimized with `React.memo`.

Even if an object or array has the same values, creating it directly in the render body will generate a new reference on every render. This new reference will cause a memoized child component to re-render unnecessarily. By wrapping the object or array creation in `useMemo`, you ensure that the same object reference is passed down as long as the dependencies haven't changed, allowing `React.memo` to work correctly.

```jsx
// Without useMemo, styleObj is a new object on every render
const styleObj = { color: 'blue', fontSize: 16 };

// With useMemo, styleObj reference is stable
const memoizedStyleObj = useMemo(() => ({
  color: 'blue',
  fontSize: 16,
}), []);

return <ChildComponent style={memoizedStyleObj} />;
```

❓ Q #41 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , 🔁 `Rendering` , 💥 `Pitfall`

🧠 **Question:**

What is the purpose of the `key` prop in React when rendering lists of elements, and why is using an array index as a key often considered an anti-

pattern?

✅ **Answer:**

The `key` prop is a special string attribute you need to include when creating lists of elements. React uses these keys as a stable identity for each element to track changes like additions, removals, or re-ordering during its reconciliation process. This allows React to efficiently update the UI by reordering or modifying existing DOM elements rather than re-creating them from scratch.

Using an array's index as a `key` is an anti-pattern when the list can be re-ordered, filtered, or have items inserted in the middle. If the order of items changes, the index for an item will change, but the component will retain the state associated with that index, leading to incorrect data being displayed and performance issues. Keys should be unique and stable to the item they identify (e.g., a product ID).

❓ Q #42 (🟢 Basic)

Tags: 🧩 General Interview , 🔧 Hooks

🧠 **Question:**

What is the purpose of the dependency array in hooks like `useEffect` , `useMemo` , and `useCallback` ? What happens if you omit it?

✅ **Answer:**

The dependency array is the second argument to these hooks. Its purpose is to tell React **when to re-run the effect or re-calculate the memoized value/function**.

- The hook will only re-execute if one of the values in the dependency array has changed since the last render2.
- If the dependency array is empty (`[]`), the hook runs only once after the initial render.
- If you **omit the dependency array entirely**, the hook will run after **every single render** of the component, which can lead to performance issues or infinite loops in the case of `useEffect` 2.

❓ Q #43 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture , 📦 State Management

🧠 **Question:**

How does the design of React Hook Form (using refs) compare to Formik (using state) in the context of imperative versus declarative programming?

✅ **Answer:**

This comparison highlights a core philosophical difference:

- **Formik is declarative**: It uses controlled components and manages form state within React's state management system. You declare how the form should look and behave based on the current state, and Formik handles the updates. This aligns closely with React's declarative nature32.
- **React Hook Form is more imperative**: By using refs to interact directly with DOM nodes (uncontrolled components), it takes a more imperative approach. You are directly `register`ing inputs and telling the form how to manage them. This is a deliberate trade-off to gain significant performance benefits by bypassing the React state system for input changes32.

Essentially, Formik asks "what should the form look like based on this state?", while RHF says "when this input changes, let me know directly so I can handle it."

❓ Q #44 (🟢 Basic)

Tags: 🧩 `General Interview` , 📦 `State Management`

🧠 **Question:**

What is the purpose of the `reset` function in React Hook Form, and how can it be used?

✅ **Answer:**

The `reset` function in React Hook Form is used to clear the form's input values and reset its state. It can be used in a few ways:

- Calling `reset()` with no arguments will clear all fields and reset the form's state (e.g., `isDirty`, `isSubmitting`) back to its initial condition.
- You can also pass an object to `reset({ firstName: 'New Name' })` to update the form with new default values.

It's commonly used after a successful form submission to clear the form for the next entry.

❓ Q #45 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🌐 Routing , 🏗️ Architecture , 🔍 Real-World Scenario

🧠 Question:

Since React Router doesn't have a built-in "Route Guard," how would you implement one to protect routes requiring user authentication?

✅ Answer:

You implement a Route Guard by creating a custom wrapper component that performs a conditional check. This component checks for an authentication status (e.g., from a context or a token in local storage).

If the user is authenticated, the component renders its `children` or an `<Outlet />`, allowing access to the protected route. If the user is not authenticated, it uses the `<Navigate>` component from React Router to redirect them to a public page, like the login screen52.

```
jsximport { Navigate, Outlet } from 'react-router-dom';

const ProtectedRoute = () => {
 const { isAuthenticated } = useAuth();
 // Example custom hook// If authenticated, render the nested child route.// If not, redirect to the login page.
 return isAuthenticated ? <Outlet /> : <Navigate to="/login" />;
};

 // Usage in route configuration:
<Route element={<ProtectedRoute />}>
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/profile" element={<Profile />} />
</Route>
```