🧩 JSX & Components: The Building Blocks of React

> **Master the foundation: JSX syntax, component creation, and the mental model that powers React applications**

🎯 What You'll Learn

- JSX syntax and how it transforms into JavaScript
- Creating functional components with modern React
- Component composition and the component tree
- Props flow and component communication
- Common JSX patterns and best practices

🤔 What is JSX?

JSX (JavaScript XML) is a syntax extension that lets you write HTML-like code inside JavaScript. It's **not** HTML - it's syntactic sugar that gets compiled to `React.createElement()` calls.

```
// This JSX...
const element = <h1>Hello, World!</h1>;

// ...compiles to this JavaScript:
const element = React.createElement("h1", null, "Hello, World!");
```

JSX Rules You Must Know

```
// ✅ CORRECT: JSX must return a single parent element
function ValidComponent() {
  return (
    <div>
      <h1>Title</h1>
      <p>Content</p>
    </div>
  );
}

// ✅ CORRECT: Use React.Fragment or <> for multiple elements
function AlsoValid() {
  return (
    <>
      <h1>Title</h1>
      <p>Content</p>
    </>
  );
}

// ❌ WRONG: Multiple root elements
```

```
function InvalidComponent() {
  return (
    <h1>Title</h1>
    <p>Content</p> // This will cause an error!
  );
}
```

JSX vs HTML: Key Differences

```
function JSXDifferences() {
  return (
    <div>
      {/* Use className instead of class */}
      <div className="container">
        {/* Use camelCase for attributes */}
        <input
          type="text"
          onChange={handleChange} // not onchange
          autoComplete="off" // not autocomplete
        />

        {/* Self-closing tags must have closing slash */}
        <img src="image.jpg" alt="Description" />
        <br />

        {/* Use curly braces for JavaScript expressions */}
        <p>Current time: {new Date().toLocaleTimeString()}</p>

        {/* Comments use JavaScript syntax */}
        {/* This is a comment in JSX */}
      </div>
    </div>
  );
}
```

🏗️ Creating Components

Functional Components (Modern React)

```
// Simple component
function Welcome() {
  return <h1>Welcome to React!</h1>;
}

// Component with props
function Greeting({ name, age }) {
  return (
    <div>
      <h2>Hello, {name}!</h2>
```

```
      <p>You are {age} years old.</p>
    </div>
  );
}

// Arrow function component
const Button = ({ children, onClick, variant = "primary" }) => {
  return (
    <button className={`btn btn-${variant}`} onClick={onClick}>
      {children}
    </button>
  );
};

// Using the components
function App() {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return (
    <div>
      <Welcome />
      <Greeting name="Alice" age={25} />
      <Button onClick={handleClick} variant="success">
        Click me!
      </Button>
    </div>
  );
}
```

Component Naming and File Structure

```
// ✅ CORRECT: PascalCase for component names
function UserProfile() {
  /* ... */
}
const NavigationBar = () => {
  /* ... */
};

// ❌ WRONG: lowercase or camelCase
function userProfile() {
  /* ... */
} // Won't work as JSX element
const navigationBar = () => {
  /* ... */
};

// File structure best practices:
// components/
```

```
//   ├── UserProfile.jsx
//   ├── NavigationBar.jsx
//   └── Button/
//       ├── Button.jsx
//       ├── Button.css
//       └── index.js  // exports Button
```

🔄 Component Composition

The Power of Composition

```jsx
// Base components
function Card({ children, className = "" }) {
  return <div className={`card ${className}`}>{children}</div>;
}

function CardHeader({ children }) {
  return <div className="card-header">{children}</div>;
}

function CardBody({ children }) {
  return <div className="card-body">{children}</div>;
}

function CardFooter({ children }) {
  return <div className="card-footer">{children}</div>;
}

// Composed component
function UserCard({ user }) {
  return (
    <Card className="user-card">
      <CardHeader>
        <h3>{user.name}</h3>
      </CardHeader>
      <CardBody>
        <p>Email: {user.email}</p>
        <p>Role: {user.role}</p>
      </CardBody>
      <CardFooter>
        <button>Edit Profile</button>
      </CardFooter>
    </Card>
  );
}
```

Conditional Rendering Patterns

```jsx
function ConditionalExample({ user, isLoading, error }) {
  // Pattern 1: Logical AND (&&)
  const showWelcome = user && !isLoading;

  return (
    <div>
      {/* Simple conditional rendering */}
      {isLoading && <div>Loading...</div>}

      {/* Conditional with logical AND */}
      {showWelcome && (
        <div>
          <h2>Welcome back, {user.name}!</h2>
        </div>
      )}

      {/* Ternary operator for if-else */}
      {error ? (
        <div className="error">Error: {error.message}</div>
      ) : (
        <div className="success">Everything looks good!</div>
      )}

      {/* Complex conditional logic */}
      {(() => {
        if (isLoading) return <div>Loading...</div>;
        if (error) return <div>Error occurred</div>;
        if (user) return <div>Welcome, {user.name}</div>;
        return <div>Please log in</div>;
      })()}
    </div>
  );
}
```

📊 Visual Component Tree

```
App
├── Header
│   ├── Logo
│   └── Navigation
│       ├── NavItem ("Home")
│       ├── NavItem ("About")
│       └── NavItem ("Contact")
├── Main
│   ├── UserCard
│   │   ├── CardHeader
│   │   ├── CardBody
│   │   └── CardFooter
│   └── PostList
│       ├── Post
```

```
│       ├── Post
│       └── Post
└── Footer
    └── Copyright
```

⚠ Common Mistakes & Anti-Patterns

1. Mutating Props

```
// ❌ WRONG: Never mutate props
function BadComponent({ user }) {
  user.name = "Modified"; // DON'T DO THIS!
  return <div>{user.name}</div>;
}

// ✅ CORRECT: Props are read-only
function GoodComponent({ user }) {
  const displayName = user.name.toUpperCase();
  return <div>{displayName}</div>;
}
```

2. Incorrect JSX Expressions

```
function JSXMistakes({ items, showTitle }) {
  return (
    <div>
      {/* ❌ WRONG: Will render "false" as text */}
      {showTitle && true && <h1>Title</h1>}

      {/* ✅ CORRECT: Use boolean conversion */}
      {!!showTitle && <h1>Title</h1>}
      {showTitle === true && <h1>Title</h1>}

      {/* ❌ WRONG: Will render "0" when items is empty */}
      {items.length && <div>Items: {items.length}</div>}

      {/* ✅ CORRECT: Explicit boolean check */}
      {items.length > 0 && <div>Items: {items.length}</div>}
    </div>
  );
}
```

3. Inline Object Creation

```
// ❌ WRONG: Creates new object on every render
function BadStyling() {
```

```
  return (
    <div style={{ color: "red", fontSize: "16px" }}>
      This creates a new object every render!
    </div>
  );
}

// ✅ CORRECT: Define styles outside or use CSS classes
const styles = {
  error: {
    color: "red",
    fontSize: "16px",
  },
};

function GoodStyling() {
  return <div style={styles.error}>This reuses the same object!</div>;
}
```

🎯 When and Why to Use Components

Component Creation Guidelines

```
// ✅ GOOD: Create components for reusability
function Button({ children, variant, onClick }) {
  return (
    <button className={`btn btn-${variant}`} onClick={onClick}>
      {children}
    </button>
  );
}

// ✅ GOOD: Create components for complex logic
function UserAvatar({ user, size = "medium" }) {
  const getInitials = (name) => {
    return name
      .split(" ")
      .map((n) => n[0])
      .join("")
      .toUpperCase();
  };

  const sizeClasses = {
    small: "w-8 h-8 text-sm",
    medium: "w-12 h-12 text-base",
    large: "w-16 h-16 text-lg",
  };

  return (
    <div className={`avatar ${sizeClasses[size]}`}>
      {user.avatar ? (
```

```
          <img src={user.avatar} alt={user.name} />
        ) : (
          <span>{getInitials(user.name)}</span>
        )}
      </div>
    );
}

// ❌ AVOID: Over-componentizing simple elements
// Don't create a component for every single div or span
function OverEngineered() {
  return <div>Just use a div here</div>;
}
```

🧪 Mini Challenges

Challenge 1: Build a Product Card

Create a `ProductCard` component that displays:

- Product image
- Product name
- Price (with currency formatting)
- "Add to Cart" button
- Sale badge (if on sale)

```
// Your solution here:
function ProductCard({ product }) {
  // Implement this component
}

// Test data:
const sampleProduct = {
  id: 1,
  name: "Wireless Headphones",
  price: 99.99,
  image: "headphones.jpg",
  onSale: true,
  originalPrice: 149.99,
};
```

▶ 💡 Solution

```
function ProductCard({ product }) {
  const formatPrice = (price) => `$${price.toFixed(2)}`;

  return (
    <div className="product-card">
      {product.onSale && <div className="sale-badge">SALE</div>}
```

```
      <img src={product.image} alt={product.name} className="product-image" />

      <div className="product-info">
        <h3 className="product-name">{product.name}</h3>

        <div className="price-section">
          <span className="current-price">{formatPrice(product.price)}</span>

          {product.onSale && product.originalPrice && (
            <span className="original-price">
              {formatPrice(product.originalPrice)}
            </span>
          )}
        </div>

        <button className="add-to-cart-btn">Add to Cart</button>
      </div>
    </div>
  );
}
```

Challenge 2: Conditional Navigation

Create a `Navigation` component that shows different menu items based on user authentication status.

```
// Requirements:
// - Show "Login" and "Register" when user is null
// - Show "Dashboard", "Profile", "Logout" when user exists
// - Highlight current page

function Navigation({ user, currentPage }) {
  // Your solution here
}
```

▶ 💡 Solution

```
function Navigation({ user, currentPage }) {
  const NavItem = ({ href, children, isActive }) => (
    <a href={href} className={`nav-item ${isActive ? "active" : ""}`}>
      {children}
    </a>
  );

  return (
    <nav className="navigation">
      {user ? (
        // Authenticated user menu
        <>
```

```
            <NavItem href="/dashboard" isActive={currentPage === "dashboard"}>
              Dashboard
            </NavItem>
            <NavItem href="/profile" isActive={currentPage === "profile"}>
              Profile
            </NavItem>
            <NavItem href="/logout">Logout</NavItem>
          </>
        ) : (
          // Guest user menu
          <>
            <NavItem href="/login" isActive={currentPage === "login"}>
              Login
            </NavItem>
            <NavItem href="/register" isActive={currentPage === "register"}>
              Register
            </NavItem>
          </>
        )}
      </nav>
    );
}
```

Interview Insights

What Interviewers Look For

1. **JSX Understanding**: "Explain what happens when JSX is compiled"
    - **Good Answer**: JSX transforms into `React.createElement()` calls, creating a virtual DOM representation
2. **Component Design**: "How do you decide when to create a new component?"
    - **Good Answer**: Reusability, single responsibility, complex logic separation, and maintainability
3. **Props vs State**: "What's the difference between props and state?"
    - **Good Answer**: Props are read-only data passed from parent; state is mutable data managed within component

Common Interview Questions

```
// Q: What's wrong with this code?
function BrokenComponent({ items }) {
  return (
    <div>
      <h1>Items</h1>
      {items.map((item) => (
        <div>{item.name}</div> // Missing key prop!
      ))}
```

```
    </div>
  );
}

// Q: Fix this conditional rendering
function FixMe({ count }) {
  return (
    <div>{count && <p>Count: {count}</p>} // Will show "0" when count is 0</div>
  );
}

// A: Correct version
function Fixed({ count }) {
  return <div>{count > 0 && <p>Count: {count}</p>}</div>;
}
```

Pro Tips for Interviews

1. **Always mention keys** when discussing lists
2. **Explain the virtual DOM** concept clearly
3. **Show awareness of performance** implications
4. **Demonstrate composition** over inheritance
5. **Use modern functional components** (avoid class components unless specifically asked)

🚀 Next Steps

Now that you understand JSX and components, you're ready to dive into:

- **Props & Children** - How components communicate
- **State Management** - Making components interactive
- **Event Handling** - Responding to user interactions

> **Key Takeaway**: JSX is just syntactic sugar for `React.createElement()`. Components are functions that return JSX. Master these fundamentals, and everything else in React becomes much clearer!

🎁 Props & Children: Component Communication Mastery

> **Learn how React components talk to each other through props and unlock the power of component composition with children**

🎯 What You'll Learn

- Props: the data highway between components
- Destructuring props like a pro
- Default props and prop validation
- The special `children` prop and composition patterns

- Prop drilling and when it becomes a problem
- Advanced prop patterns for flexible components

🚀 Understanding Props

Props (short for "properties") are how parent components pass data to child components. Think of them as function arguments for your components.

```
// Parent component passing props
function App() {
  const user = {
    name: "Sarah Chen",
    email: "sarah@example.com",
    avatar: "avatar.jpg",
    isOnline: true,
  };

  return (
    <div>
      {/* Passing different types of props */}
      <UserProfile
        user={user} // Object prop
        theme="dark" // String prop
        showStatus={true} // Boolean prop
        onEdit={() => console.log("Edit")} // Function prop
        maxPosts={10} // Number prop
      />
    </div>
  );
}

// Child component receiving props
function UserProfile({ user, theme, showStatus, onEdit, maxPosts }) {
  return (
    <div className={`profile profile--${theme}`}>
      <img src={user.avatar} alt={user.name} />
      <h2>{user.name}</h2>
      <p>{user.email}</p>

      {showStatus && (
        <span className={`status ${user.isOnline ? "online" : "offline"}`}>
          {user.isOnline ? "🟢 Online" : "⚫ Offline"}
        </span>
      )}

      <button onClick={onEdit}>Edit Profile</button>
      <p>Showing last {maxPosts} posts</p>
    </div>
  );
}
```

🎯 Props Destructuring Patterns

Basic Destructuring

```
// ✅ GOOD: Destructure in function parameters
function Welcome({ name, age, city }) {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>
        Age: {age}, City: {city}
      </p>
    </div>
  );
}

// ❌ AVOID: Using props object directly
function WelcomeVerbose(props) {
  return (
    <div>
      <h1>Hello, {props.name}!</h1>
      <p>
        Age: {props.age}, City: {props.city}
      </p>
    </div>
  );
}
```

Advanced Destructuring with Defaults

```
function Button({
  children,
  variant = "primary", // Default value
  size = "medium",
  disabled = false,
  onClick = () => {}, // Default function
  ...restProps // Spread remaining props
}) {
  const baseClasses = "btn";
  const variantClass = `btn--${variant}`;
  const sizeClass = `btn--${size}`;

  return (
    <button
      className={`${baseClasses} ${variantClass} ${sizeClass}`}
      disabled={disabled}
      onClick={onClick}
      {...restProps} // Pass through any additional props
    >
      {children}
```

```
    </button>
  );
}

// Usage examples
function ButtonExamples() {
  return (
    <div>
      <Button>Default Button</Button>
      <Button variant="secondary" size="large">
        Large Secondary
      </Button>
      <Button disabled onClick={() => alert("Clicked")}>
        Disabled Button
      </Button>
      <Button type="submit" form="myForm">
        {" "}
        {/* type and form passed via ...restProps */}
        Submit
      </Button>
    </div>
  );
}
```

Nested Object Destructuring

```
function UserCard({
  user: {
    name,
    email,
    profile: { bio, location, website } = {}, // Default empty object
  },
  settings: { theme, notifications } = { theme: "light", notifications: true },
}) {
  return (
    <div className={`user-card user-card--${theme}`}>
      <h3>{name}</h3>
      <p>{email}</p>
      {bio && <p className="bio">{bio}</p>}
      {location && <p>🔑 {location}</p>}
      {website && <a href={website}>🌐 Website</a>}

      {notifications && <div className="notification-badge">🔔</div>}
    </div>
  );
}

// Usage
const userData = {
  name: "Alex Johnson",
  email: "alex@example.com",
```

```
  profile: {
    bio: "Full-stack developer passionate about React",
    location: "San Francisco, CA",
    website: "https://alexjohnson.dev",
  },
};

const userSettings = {
  theme: "dark",
  notifications: true,
};

<UserCard user={userData} settings={userSettings} />;
```

👶 The Children Prop: Composition Magic

The `children` prop is special - it represents the content between opening and closing JSX tags.

Basic Children Usage

```
// Container component that wraps children
function Card({ children, title, className = "" }) {
  return (
    <div className={`card ${className}`}>
      {title && <div className="card-header">{title}</div>}
      <div className="card-body">
        {children} {/* This is where child content goes */}
      </div>
    </div>
  );
}

// Usage - anything between <Card> tags becomes children
function App() {
  return (
    <div>
      <Card title="User Info">
        <h3>John Doe</h3>
        <p>Software Engineer</p>
        <button>Contact</button>
      </Card>

      <Card title="Statistics" className="stats-card">
        <div className="stat">
          <span className="number">1,234</span>
          <span className="label">Users</span>
        </div>
        <div className="stat">
          <span className="number">567</span>
          <span className="label">Orders</span>
        </div>
```

```
      </Card>
    </div>
  );
}
```

Advanced Children Patterns

```
// 1. Children as a function (render props pattern)
function DataFetcher({ url, children }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
  }, [url]);

  // Call children as a function, passing state
  return children({ data, loading, error });
}

// Usage
function UserList() {
  return (
    <DataFetcher url="/api/users">
      {({ data, loading, error }) => {
        if (loading) return <div>Loading users...</div>;
        if (error) return <div>Error: {error.message}</div>;

        return (
          <ul>
            {data.map((user) => (
              <li key={user.id}>{user.name}</li>
            ))}
          </ul>
        );
      }}
    </DataFetcher>
  );
}

// 2. Manipulating children with React.Children
```

```
function List({ children, ordered = false }) {
  const Tag = ordered ? "ol" : "ul";

  // Add props to each child
  const enhancedChildren = React.Children.map(children, (child, index) => {
    if (React.isValidElement(child)) {
      return React.cloneElement(child, {
        key: index,
        className: `${child.props.className || ""} list-item`.trim(),
      });
    }
    return child;
  });

  return <Tag className="custom-list">{enhancedChildren}</Tag>;
}

// Usage
function TodoApp() {
  return (
    <List ordered>
      <li>Learn React fundamentals</li>
      <li>Master hooks</li>
      <li>Build awesome projects</li>
    </List>
  );
}

// 3. Conditional children rendering
function Modal({ children, isOpen, onClose }) {
  if (!isOpen) return null;

  return (
    <div className="modal-overlay" onClick={onClose}>
      <div className="modal-content" onClick={(e) => e.stopPropagation()}>
        <button className="modal-close" onClick={onClose}>
          ×
        </button>
        {children}
      </div>
    </div>
  );
}
```

🔄 Prop Flow and Communication Patterns

Parent to Child Communication

```
// Parent manages state and passes down
function ShoppingCart() {
  const [items, setItems] = useState([
```

```
    { id: 1, name: "Laptop", price: 999, quantity: 1 },
    { id: 2, name: "Mouse", price: 25, quantity: 2 },
  ]);

  const updateQuantity = (id, newQuantity) => {
    setItems(
      items.map((item) =>
        item.id === id ? { ...item, quantity: newQuantity } : item
      )
    );
  };

  const removeItem = (id) => {
    setItems(items.filter((item) => item.id !== id));
  };

  return (
    <div className="shopping-cart">
      <h2>Shopping Cart</h2>
      {items.map((item) => (
        <CartItem
          key={item.id}
          item={item}
          onUpdateQuantity={updateQuantity} // Function prop
          onRemove={removeItem} // Function prop
        />
      ))}
      <CartTotal items={items} /> {/* Data prop */}
    </div>
  );
}

// Child receives props and calls parent functions
function CartItem({ item, onUpdateQuantity, onRemove }) {
  const handleQuantityChange = (e) => {
    const newQuantity = parseInt(e.target.value);
    onUpdateQuantity(item.id, newQuantity);
  };

  return (
    <div className="cart-item">
      <span>{item.name}</span>
      <span>${item.price}</span>
      <input
        type="number"
        value={item.quantity}
        onChange={handleQuantityChange}
        min="1"
      />
      <button onClick={() => onRemove(item.id)}>Remove</button>
    </div>
  );
}
```

```
function CartTotal({ items }) {
  const total = items.reduce(
    (sum, item) => sum + item.price * item.quantity,
    0
  );

  return (
    <div className="cart-total">
      <strong>Total: ${total.toFixed(2)}</strong>
    </div>
  );
}
```

Prop Drilling Problem

```
// ❌ PROBLEM: Prop drilling - passing props through multiple levels
function App() {
  const [user, setUser] = useState({ name: "John", theme: "dark" });

  return (
    <Layout user={user} setUser={setUser}>
      <Dashboard user={user} setUser={setUser} />
    </Layout>
  );
}

function Layout({ children, user, setUser }) {
  return (
    <div>
      <Header user={user} setUser={setUser} /> {/* Passing through */}
      <main>{children}</main>
    </div>
  );
}

function Header({ user, setUser }) {
  return (
    <header>
      <Navigation user={user} /> {/* Still passing through */}
      <UserMenu user={user} setUser={setUser} />
    </header>
  );
}

function Navigation({ user }) {
  return (
    <nav>
      <ThemeToggle user={user} /> {/* Finally used here */}
    </nav>
  );
}
```

```
function ThemeToggle({ user }) {
  // This component is 4 levels deep but needs user data!
  return <button>Current theme: {user.theme}</button>;
}
```

⚠ Common Mistakes & Anti-Patterns

1. Mutating Props

```
// ❌ WRONG: Never mutate props
function BadComponent({ user, items }) {
  user.name = "Modified"; // DON'T DO THIS!
  items.push({ id: 4, name: "New" }); // DON'T DO THIS!

  return <div>{user.name}</div>;
}

// ✅ CORRECT: Create new objects/arrays
function GoodComponent({ user, items }) {
  const updatedUser = { ...user, name: "Modified" };
  const updatedItems = [...items, { id: 4, name: "New" }];

  return <div>{updatedUser.name}</div>;
}
```

2. Unnecessary Prop Passing

```
// ❌ WRONG: Passing all props when only some are needed
function OverpassingParent() {
  const massiveUserObject = {
    id: 1,
    name: "John",
    email: "john@example.com",
    preferences: {
      /* huge object */
    },
    history: {
      /* huge array */
    },
    // ... 50 more properties
  };

  return <SimpleGreeting user={massiveUserObject} />;
}

function SimpleGreeting({ user }) {
  return <h1>Hello, {user.name}!</h1>; // Only needs name!
}
```

```
// ✅ CORRECT: Pass only what's needed
function EfficientParent() {
  const user = {
    /* massive object */
  };

  return <SimpleGreeting name={user.name} />;
}

function SimpleGreeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

3. Boolean Prop Mistakes

```
// ❌ WRONG: Explicit true values
<Button disabled={true} loading={true} />

// ✅ CORRECT: Shorthand for true
<Button disabled loading />

// ❌ WRONG: String booleans
<Button disabled="true" />  // This is always truthy!

// ✅ CORRECT: Actual booleans
<Button disabled={isDisabled} />
```

🎯 When and Why to Use Different Prop Patterns

Component Flexibility Spectrum

```
// 1. RIGID: Specific, limited use cases
function UserAvatar({ user }) {
  return <img src={user.avatar} alt={user.name} className="user-avatar" />;
}

// 2. FLEXIBLE: Configurable behavior
function Avatar({
  src,
  alt,
  size = "medium",
  shape = "circle",
  fallback = "👤",
}) {
  const sizeClasses = {
    small: "w-8 h-8",
    medium: "w-12 h-12",
    large: "w-16 h-16",
```

```
  };

  return (
    <div className={`avatar avatar--${shape} ${sizeClasses[size]}`}>
      {src ? (
        <img src={src} alt={alt} />
      ) : (
        <span className="avatar-fallback">{fallback}</span>
      )}
    </div>
  );
}

// 3. HIGHLY FLEXIBLE: Composition-based
function FlexibleCard({ children, ...props }) {
  return (
    <div className="card" {...props}>
      {children}
    </div>
  );
}

// Usage shows the flexibility
function Examples() {
  return (
    <div>
      {/* Rigid - only works with user objects */}
      <UserAvatar user={currentUser} />

      {/* Flexible - works with any image data */}
      <Avatar src="profile.jpg" alt="Profile" size="large" shape="square" />

      {/* Highly flexible - can contain anything */}
      <FlexibleCard className="special" onClick={handleClick}>
        <h3>Custom Content</h3>
        <p>Any JSX can go here</p>
        <Avatar src="user.jpg" alt="User" />
      </FlexibleCard>
    </div>
  );
}
```

Mini Challenges

Challenge 1: Smart Button Component

Create a `Button` component that:

- Accepts `variant` (primary, secondary, danger)
- Accepts `size` (small, medium, large)
- Shows loading state with spinner
- Handles disabled state

- Accepts any additional HTML button props

```
// Your solution:
function Button({ children, variant, size, loading, disabled, ...props }) {
  // Implement this component
}

// Test cases:
<Button variant="primary" size="large">Save</Button>
<Button variant="danger" loading disabled>Delete</Button>
<Button onClick={handleClick} type="submit">Submit</Button>
```

▶ 💡 Solution

```
function Button({
  children,
  variant = "primary",
  size = "medium",
  loading = false,
  disabled = false,
  className = "",
  ...props
}) {
  const baseClasses = "btn";
  const variantClass = `btn--${variant}`;
  const sizeClass = `btn--${size}`;
  const loadingClass = loading ? "btn--loading" : "";

  const allClasses = [
    baseClasses,
    variantClass,
    sizeClass,
    loadingClass,
    className,
  ]
    .filter(Boolean)
    .join(" ");

  return (
    <button className={allClasses} disabled={disabled || loading} {...props}>
      {loading && <span className="spinner">⏳</span>}
      {children}
    </button>
  );
}
```

Challenge 2: Flexible List Component

Create a `List` component that:

- Renders items from an array
- Accepts a render function for custom item rendering
- Shows empty state when no items
- Supports loading state

```
// Your solution:
function List({ items, renderItem, loading, emptyMessage }) {
  // Implement this component
}

// Test cases:
const users = [
  { id: 1, name: "John" },
  { id: 2, name: "Jane" },
];

<List
  items={users}
  renderItem={(user) => <div key={user.id}>{user.name}</div>}
  emptyMessage="No users found"
/>;
```

▶ 💡 Solution

```
function List({
  items = [],
  renderItem,
  loading = false,
  emptyMessage = "No items found",
  className = "",
}) {
  if (loading) {
    return <div className="list-loading">Loading...</div>;
  }

  if (items.length === 0) {
    return <div className="list-empty">{emptyMessage}</div>;
  }

  return (
    <div className={`list ${className}`}>
      {items.map((item, index) => {
        // Support both render function and default rendering
        if (renderItem) {
          return renderItem(item, index);
        }

        // Default rendering assumes item has id and can be stringified
        return (
          <div key={item.id || index} className="list-item">
```

```
            {typeof item === "object" ? JSON.stringify(item) : item}
          </div>
        );
      })}
    </div>
  );
}
```

Interview Insights

What Interviewers Ask

1. **"Explain the difference between props and state"**
   - **Good Answer**: Props are read-only data passed from parent to child. State is mutable data managed within a component. Props flow down, events flow up.
2. **"What is prop drilling and how do you solve it?"**
   - **Good Answer**: Prop drilling is passing props through multiple component layers. Solutions include Context API, state management libraries, or component composition.
3. **"How do you make components reusable?"**
   - **Good Answer**: Use props for configuration, provide sensible defaults, use children for composition, and follow single responsibility principle.

Code Review Red Flags

```
// ❌ RED FLAGS in interviews:

// 1. Mutating props
function Bad({ user }) {
  user.name = "Changed"; // NEVER!
  return <div>{user.name}</div>;
}

// 2. Not using destructuring
function Verbose(props) {
  return (
    <div>
      {props.user.name} - {props.user.email}
    </div>
  );
}

// 3. Inline object creation
function Inefficient() {
  return (
    <Component
      style={{ color: "red" }} // New object every render!
```

```
        data={{ items: [] }} // New object every render!
      />
    );
  }

  // 4. Missing key props in lists
  function MissingKeys({ items }) {
    return (
      <div>
        {items.map((item) => (
          <div>{item.name}</div>
        ))}{" "}
        // No key!
      </div>
    );
  }
```

Pro Interview Tips

1. **Always destructure props** in function parameters
2. **Mention performance implications** of prop changes
3. **Show awareness of prop drilling** and solutions
4. **Demonstrate composition patterns** with children
5. **Use TypeScript prop types** if asked about type safety

🚀 Next Steps

Now that you understand props and children, you're ready for:

- **useState and Functional State** - Making components interactive
- **Event Handling** - Responding to user actions
- **Controlled Components** - Managing form inputs

> **Key Takeaway**: Props are the communication system of React. Master prop patterns and component composition, and you'll build flexible, reusable components that scale with your application!

🔄 useState & Functional State: Making Components Interactive

> **Master React's most fundamental hook: useState. Learn how to manage component state, trigger re-renders, and build truly interactive applications**

🎯 What You'll Learn

- useState fundamentals and the state update cycle
- State initialization patterns and lazy initialization
- Functional updates and avoiding stale closures

- Managing different types of state (primitives, objects, arrays)
- State batching and performance considerations
- Common useState patterns and anti-patterns

🚀 useState Fundamentals

The Anatomy of useState

```
import { useState } from "react";

function Counter() {
  // useState returns an array with exactly 2 elements:
  // [currentValue, setterFunction]
  const [count, setCount] = useState(0); // 0 is the initial value

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

State Update Cycle

```
function StateUpdateDemo() {
  const [message, setMessage] = useState("Initial");

  console.log("Component rendering with message:", message);

  const handleClick = () => {
    console.log("Before setState:", message);
    setMessage("Updated");
    console.log("After setState (still old):", message); // Still "Initial"!

    // State updates are asynchronous and trigger re-render
    // The new value will be available in the next render
  };

  return (
    <div>
      <p>Message: {message}</p>
      <button onClick={handleClick}>Update Message</button>
    </div>
  );
}

// Console output when button is clicked:
```

```
// "Component rendering with message: Initial"
// "Before setState: Initial"
// "After setState (still old): Initial"
// "Component rendering with message: Updated"
```

🎯 State Initialization Patterns

Simple Initialization

```
function SimpleState() {
  // Primitive values
  const [name, setName] = useState("");
  const [age, setAge] = useState(0);
  const [isActive, setIsActive] = useState(false);

  // Objects and arrays
  const [user, setUser] = useState({ name: "", email: "" });
  const [items, setItems] = useState([]);

  return (
    <div>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Name"
      />
      <input
        type="number"
        value={age}
        onChange={(e) => setAge(Number(e.target.value))}
        placeholder="Age"
      />
      <label>
        <input
          type="checkbox"
          checked={isActive}
          onChange={(e) => setIsActive(e.target.checked)}
        />
        Active
      </label>
    </div>
  );
}
```

Lazy Initialization (Performance Optimization)

```
// ❌ EXPENSIVE: This function runs on every render
function ExpensiveInit() {
  const [data, setData] = useState(expensiveCalculation()); // Runs every render!
```

```
  return <div>{data}</div>;
}

// ✅ OPTIMIZED: Lazy initialization with function
function OptimizedInit() {
  // Pass a function to useState - it only runs once
  const [data, setData] = useState(() => {
    console.log("Expensive calculation running...");
    return expensiveCalculation(); // Only runs on initial render
  });

  return <div>{data}</div>;
}

// Real-world examples of lazy initialization
function LazyExamples() {
  // Reading from localStorage only once
  const [theme, setTheme] = useState(() => {
    return localStorage.getItem("theme") || "light";
  });

  // Complex object creation
  const [gameState, setGameState] = useState(() => {
    return {
      level: 1,
      score: 0,
      lives: 3,
      powerUps: [],
      enemies: generateEnemies(1), // Expensive function
    };
  });

  // Date/time that shouldn't change
  const [sessionStart, setSessionStart] = useState(() => new Date());

  return (
    <div>
      <p>Theme: {theme}</p>
      <p>Game Level: {gameState.level}</p>
      <p>Session started: {sessionStart.toLocaleTimeString()}</p>
    </div>
  );
}

function expensiveCalculation() {
  // Simulate expensive operation
  let result = 0;
  for (let i = 0; i < 1000000; i++) {
    result += Math.random();
  }
  return result;
}

function generateEnemies(level) {
```

```
  // Simulate expensive enemy generation
  return Array.from({ length: level * 5 }, (_, i) => ({
    id: i,
    type: "goblin",
    health: 100,
    position: { x: Math.random() * 800, y: Math.random() * 600 },
  }));
}
```

🔄 Functional Updates: Avoiding Stale Closures

The Stale Closure Problem

```
// ❌ PROBLEM: Stale closure with rapid updates
function StaleClosureProblem() {
  const [count, setCount] = useState(0);

  const handleMultipleUpdates = () => {
    // These all use the same 'count' value from when the function was created
    setCount(count + 1); // If count was 0, this sets it to 1
    setCount(count + 1); // This also sets it to 1 (not 2!)
    setCount(count + 1); // This also sets it to 1 (not 3!)

    // Result: count becomes 1, not 3
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleMultipleUpdates}>Add 3 (Broken)</button>
    </div>
  );
}

// ✅ SOLUTION: Functional updates
function FunctionalUpdatesSolution() {
  const [count, setCount] = useState(0);

  const handleMultipleUpdates = () => {
    // Use functional updates - each gets the latest value
    setCount((prevCount) => prevCount + 1); // Gets current count
    setCount((prevCount) => prevCount + 1); // Gets updated count
    setCount((prevCount) => prevCount + 1); // Gets updated count again

    // Result: count increases by 3
  };

  const handleAsyncUpdate = () => {
    // Functional updates work great with async operations
    setTimeout(() => {
      setCount((prevCount) => prevCount + 1); // Always gets latest value
```

```
    }, 1000);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleMultipleUpdates}>Add 3 (Works!)</button>
      <button onClick={handleAsyncUpdate}>Add 1 After Delay</button>
    </div>
  );
}
```

Advanced Functional Update Patterns

```
function AdvancedFunctionalUpdates() {
  const [todos, setTodos] = useState([
    { id: 1, text: "Learn React", completed: false },
    { id: 2, text: "Build a project", completed: false },
  ]);

  // Toggle todo completion
  const toggleTodo = (id) => {
    setTodos((prevTodos) =>
      prevTodos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  // Add new todo
  const addTodo = (text) => {
    setTodos((prevTodos) => [
      ...prevTodos,
      {
        id: Date.now(), // Simple ID generation
        text,
        completed: false,
      },
    ]);
  };

  // Remove todo
  const removeTodo = (id) => {
    setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== id));
  };

  // Complex state update with multiple conditions
  const updateTodoWithValidation = (id, newText) => {
    setTodos((prevTodos) => {
      // You can add complex logic in functional updates
      if (!newText.trim()) {
```

```
        // Don't update if text is empty
        return prevTodos;
      }

      return prevTodos.map((todo) => {
        if (todo.id === id) {
          return {
            ...todo,
            text: newText.trim(),
            lastModified: new Date().toISOString(),
          };
        }
        return todo;
      });
    });
  };

  return (
    <div>
      <h3>Todo List</h3>
      {todos.map((todo) => (
        <div key={todo.id} className="todo-item">
          <input
            type="checkbox"
            checked={todo.completed}
            onChange={() => toggleTodo(todo.id)}
          />
          <span className={todo.completed ? "completed" : ""}>{todo.text}</span>
          <button onClick={() => removeTodo(todo.id)}>Remove</button>
        </div>
      ))}

      <button onClick={() => addTodo("New todo")}>Add Todo</button>
    </div>
  );
}
```

📊 Managing Different State Types

Object State Management

```
function ObjectStateManagement() {
  const [user, setUser] = useState({
    name: "",
    email: "",
    preferences: {
      theme: "light",
      notifications: true,
      language: "en",
    },
    profile: {
```

```
    bio: "",
    avatar: null,
    socialLinks: [],
  },
});

// ❌ WRONG: Mutating state directly
const updateNameWrong = (newName) => {
  user.name = newName; // DON'T DO THIS!
  setUser(user);
};

// ✅ CORRECT: Creating new object
const updateName = (newName) => {
  setUser((prevUser) => ({
    ...prevUser,
    name: newName,
  }));
};

// ✅ CORRECT: Updating nested objects
const updateTheme = (newTheme) => {
  setUser((prevUser) => ({
    ...prevUser,
    preferences: {
      ...prevUser.preferences,
      theme: newTheme,
    },
  }));
};

// ✅ CORRECT: Updating deeply nested arrays
const addSocialLink = (platform, url) => {
  setUser((prevUser) => ({
    ...prevUser,
    profile: {
      ...prevUser.profile,
      socialLinks: [
        ...prevUser.profile.socialLinks,
        { platform, url, id: Date.now() },
      ],
    },
  }));
};

// Helper function for complex nested updates
const updateUserField = (path, value) => {
  setUser((prevUser) => {
    const newUser = { ...prevUser };

    // Simple path handling (for demo purposes)
    if (path === "preferences.theme") {
      newUser.preferences = { ...newUser.preferences, theme: value };
    } else if (path === "profile.bio") {
```

```
        newUser.profile = { ...newUser.profile, bio: value };
      }

      return newUser;
    });
  };

  return (
    <div>
      <input
        value={user.name}
        onChange={(e) => updateName(e.target.value)}
        placeholder="Name"
      />

      <select
        value={user.preferences.theme}
        onChange={(e) => updateTheme(e.target.value)}
      >
        <option value="light">Light</option>
        <option value="dark">Dark</option>
      </select>

      <textarea
        value={user.profile.bio}
        onChange={(e) => updateUserField("profile.bio", e.target.value)}
        placeholder="Bio"
      />

      <button onClick={() => addSocialLink("twitter", "@username")}>
        Add Twitter
      </button>

      <pre>{JSON.stringify(user, null, 2)}</pre>
    </div>
  );
}
```

Array State Management

```
function ArrayStateManagement() {
  const [items, setItems] = useState([
    { id: 1, name: "Apple", category: "fruit", price: 1.2 },
    { id: 2, name: "Banana", category: "fruit", price: 0.8 },
    { id: 3, name: "Carrot", category: "vegetable", price: 0.5 },
  ]);

  // Add item to end
  const addItem = (item) => {
    setItems((prevItems) => [...prevItems, { ...item, id: Date.now() }]);
  };
```

```
// Add item to beginning
const addItemToStart = (item) => {
  setItems((prevItems) => [{ ...item, id: Date.now() }, ...prevItems]);
};

// Remove item by id
const removeItem = (id) => {
  setItems((prevItems) => prevItems.filter((item) => item.id !== id));
};

// Update item by id
const updateItem = (id, updates) => {
  setItems((prevItems) =>
    prevItems.map((item) => (item.id === id ? { ...item, ...updates } : item))
  );
};

// Insert item at specific index
const insertItemAt = (index, item) => {
  setItems((prevItems) => {
    const newItems = [...prevItems];
    newItems.splice(index, 0, { ...item, id: Date.now() });
    return newItems;
  });
};

// Move item (reorder)
const moveItem = (fromIndex, toIndex) => {
  setItems((prevItems) => {
    const newItems = [...prevItems];
    const [movedItem] = newItems.splice(fromIndex, 1);
    newItems.splice(toIndex, 0, movedItem);
    return newItems;
  });
};

// Sort items
const sortItems = (sortBy) => {
  setItems((prevItems) => {
    const sorted = [...prevItems].sort((a, b) => {
      if (sortBy === "name") {
        return a.name.localeCompare(b.name);
      } else if (sortBy === "price") {
        return a.price - b.price;
      }
      return 0;
    });
    return sorted;
  });
};

// Filter and update (complex operation)
const updateItemsInCategory = (category, updates) => {
```

```
      setItems((prevItems) =>
        prevItems.map((item) =>
          item.category === category ? { ...item, ...updates } : item
        )
      );
    };

    return (
      <div>
        <h3>Items ({items.length})</h3>

        <div className="controls">
          <button
            onClick={() =>
              addItem({ name: "Orange", category: "fruit", price: 1.0 })
            }
          >
            Add Orange
          </button>
          <button onClick={() => sortItems("name")}>Sort by Name</button>
          <button onClick={() => sortItems("price")}>Sort by Price</button>
          <button
            onClick={() => updateItemsInCategory("fruit", { onSale: true })}
          >
            Put Fruits on Sale
          </button>
        </div>

        <ul>
          {items.map((item, index) => (
            <li key={item.id}>
              <span>
                {item.name} - ${item.price}
              </span>
              {item.onSale && <span className="sale-badge">ON SALE</span>}
              <button
                onClick={() => updateItem(item.id, { price: item.price * 0.9 })}
              >
                10% Off
              </button>
              <button onClick={() => removeItem(item.id)}>Remove</button>
              <button onClick={() => moveItem(index, 0)}>Move to Top</button>
            </li>
          ))}
        </ul>
      </div>
    );
  }
```

⚡ State Batching and Performance

Understanding State Batching

```jsx
function StateBatchingDemo() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  console.log("Component rendered with:", { count, name, email });

  // React 18: Automatic batching in all scenarios
  const handleBatchedUpdates = () => {
    console.log("Before updates");

    // These will be batched together - only one re-render
    setCount((c) => c + 1);
    setName("John");
    setEmail("john@example.com");

    console.log("After updates (but before re-render)");
    // Component will re-render once with all updates
  };

  // Even in async operations (React 18+)
  const handleAsyncBatchedUpdates = () => {
    setTimeout(() => {
      // These are also batched in React 18
      setCount((c) => c + 1);
      setName("Jane");
      setEmail("jane@example.com");
    }, 1000);
  };

  // Force separate updates (rarely needed)
  const handleUnbatchedUpdates = () => {
    import("react-dom").then(({ flushSync }) => {
      flushSync(() => {
        setCount((c) => c + 1);
      });
      // Re-render happens here

      flushSync(() => {
        setName("Bob");
      });
      // Another re-render happens here
    });
  };

  return (
    <div>
      <p>Count: {count}</p>
      <p>Name: {name}</p>
      <p>Email: {email}</p>

      <button onClick={handleBatchedUpdates}>Batched Updates (1 render)</button>
```

```
      <button onClick={handleAsyncBatchedUpdates}>Async Batched Updates</button>

      <button onClick={handleUnbatchedUpdates}>
        Unbatched Updates (multiple renders)
      </button>
    </div>
  );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Direct State Mutation

```
// ❌ WRONG: Mutating state directly
function MutationMistakes() {
  const [user, setUser] = useState({ name: "John", hobbies: ["reading"] });
  const [items, setItems] = useState([1, 2, 3]);

  const badUpdate = () => {
    // DON'T DO THESE:
    user.name = "Jane"; // Mutating object
    user.hobbies.push("coding"); // Mutating nested array
    items.push(4); // Mutating array
    setUser(user); // React won't detect the change!
    setItems(items);
  };

  // ✅ CORRECT: Create new objects/arrays
  const goodUpdate = () => {
    setUser((prevUser) => ({
      ...prevUser,
      name: "Jane",
      hobbies: [...prevUser.hobbies, "coding"],
    }));

    setItems((prevItems) => [...prevItems, 4]);
  };

  return (
    <div>
      <p>User: {user.name}</p>
      <p>Hobbies: {user.hobbies.join(", ")}</p>
      <p>Items: {items.join(", ")}</p>
      <button onClick={badUpdate}>Bad Update</button>
      <button onClick={goodUpdate}>Good Update</button>
    </div>
  );
}
```

2. Stale State in Event Handlers

```
// ❌ PROBLEM: Stale state in closures
function StaleStateProblems() {
  const [count, setCount] = useState(0);

  // This creates a closure that captures the current count value
  const handleDelayedIncrement = () => {
    setTimeout(() => {
      setCount(count + 1); // Uses stale count value!
    }, 3000);
  };

  // ✅ SOLUTION: Use functional updates
  const handleCorrectDelayedIncrement = () => {
    setTimeout(() => {
      setCount((prevCount) => prevCount + 1); // Always gets latest value
    }, 3000);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment Now</button>
      <button onClick={handleDelayedIncrement}>Increment in 3s (Broken)</button>
      <button onClick={handleCorrectDelayedIncrement}>
        Increment in 3s (Fixed)
      </button>
    </div>
  );
}
```

3. Unnecessary State

```
// ❌ WRONG: Storing derived values in state
function UnnecessaryState({ items }) {
  const [itemCount, setItemCount] = useState(0);
  const [expensiveItems, setExpensiveItems] = useState([]);

  // DON'T DO THIS - these should be computed values!
  useEffect(() => {
    setItemCount(items.length);
    setExpensiveItems(items.filter((item) => item.price > 100));
  }, [items]);

  return (
    <div>
      <p>Item count: {itemCount}</p>
      <p>Expensive items: {expensiveItems.length}</p>
    </div>
```

```
  );
}

// ✅ CORRECT: Compute derived values during render
function ComputedValues({ items }) {
  // These are computed on every render (which is fine for simple calculations)
  const itemCount = items.length;
  const expensiveItems = items.filter((item) => item.price > 100);

  // For expensive calculations, use useMemo (covered in later chapters)
  const expensiveCalculation = useMemo(() => {
    return items.reduce((sum, item) => sum + item.price, 0);
  }, [items]);

  return (
    <div>
      <p>Item count: {itemCount}</p>
      <p>Expensive items: {expensiveItems.length}</p>
      <p>Total value: ${expensiveCalculation}</p>
    </div>
  );
}
```

🎯 When and Why to Use useState

useState Decision Tree

```
Do you need to store data that:
├── Changes over time? ──────────────────── YES → useState
├── Triggers re-renders when it changes? ─── YES → useState
├── Is derived from props or other state? ── NO → Computed value
├── Persists between renders? ────────────── YES → useState
├── Is only used in event handlers? ──────── MAYBE → useRef
└── Is shared between components? ────────── MAYBE → Context/Props
```

Real-World useState Patterns

```
// 1. Form state management
function ContactForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    message: "",
  });
  const [errors, setErrors] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  const handleInputChange = (field, value) => {
    setFormData((prev) => ({ ...prev, [field]: value }));
```

```
    // Clear error when user starts typing
    if (errors[field]) {
      setErrors((prev) => ({ ...prev, [field]: null }));
    }
  };

  const validateForm = () => {
    const newErrors = {};
    if (!formData.name.trim()) newErrors.name = "Name is required";
    if (!formData.email.includes("@")) newErrors.email = "Valid email required";
    if (!formData.message.trim()) newErrors.message = "Message is required";

    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  const handleSubmit = async (e) => {
    e.preventDefault();

    if (!validateForm()) return;

    setIsSubmitting(true);
    try {
      await submitForm(formData);
      setFormData({ name: "", email: "", message: "" }); // Reset form
    } catch (error) {
      setErrors({ submit: "Failed to submit form" });
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={formData.name}
        onChange={(e) => handleInputChange("name", e.target.value)}
        placeholder="Name"
      />
      {errors.name && <span className="error">{errors.name}</span>}

      <input
        type="email"
        value={formData.email}
        onChange={(e) => handleInputChange("email", e.target.value)}
        placeholder="Email"
      />
      {errors.email && <span className="error">{errors.email}</span>}

      <textarea
        value={formData.message}
        onChange={(e) => handleInputChange("message", e.target.value)}
        placeholder="Message"
```

```
        />
        {errors.message && <span className="error">{errors.message}</span>}

        <button type="submit" disabled={isSubmitting}>
          {isSubmitting ? "Submitting..." : "Submit"}
        </button>

        {errors.submit && <div className="error">{errors.submit}</div>}
      </form>
    );
}

// 2. Toggle and UI state
function ToggleExamples() {
  const [isMenuOpen, setIsMenuOpen] = useState(false);
  const [activeTab, setActiveTab] = useState("home");
  const [isLoading, setIsLoading] = useState(false);
  const [showModal, setShowModal] = useState(false);

  return (
    <div>
      <button onClick={() => setIsMenuOpen(!isMenuOpen)}>
        {isMenuOpen ? "Close" : "Open"} Menu
      </button>

      <div className="tabs">
        {["home", "about", "contact"].map((tab) => (
          <button
            key={tab}
            className={activeTab === tab ? "active" : ""}
            onClick={() => setActiveTab(tab)}
          >
            {tab}
          </button>
        ))}
      </div>

      <button
        onClick={() => {
          setIsLoading(true);
          setTimeout(() => setIsLoading(false), 2000);
        }}
        disabled={isLoading}
      >
        {isLoading ? "Loading..." : "Load Data"}
      </button>

      <button onClick={() => setShowModal(true)}>Show Modal</button>

      {showModal && (
        <div className="modal" onClick={() => setShowModal(false)}>
          <div className="modal-content" onClick={(e) => e.stopPropagation()}>
            <h3>Modal Content</h3>
            <button onClick={() => setShowModal(false)}>Close</button>
```

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

🧪 Mini Challenges

Challenge 1: Shopping Cart

Build a shopping cart component with:

- Add/remove items
- Update quantities
- Calculate total price
- Clear cart functionality

```
// Your solution:
function ShoppingCart() {
  // Implement cart state and functions
}

// Test data:
const products = [
  { id: 1, name: "Laptop", price: 999 },
  { id: 2, name: "Mouse", price: 25 },
  { id: 3, name: "Keyboard", price: 75 },
];
```

▶ 💡 Solution

```
function ShoppingCart() {
  const [cartItems, setCartItems] = useState([]);

  const addToCart = (product) => {
    setCartItems((prevItems) => {
      const existingItem = prevItems.find((item) => item.id === product.id);

      if (existingItem) {
        return prevItems.map((item) =>
          item.id === product.id
            ? { ...item, quantity: item.quantity + 1 }
            : item
        );
      } else {
        return [...prevItems, { ...product, quantity: 1 }];
      }
    });
```

```
  };

  const removeFromCart = (productId) => {
    setCartItems((prevItems) =>
      prevItems.filter((item) => item.id !== productId)
    );
  };

  const updateQuantity = (productId, newQuantity) => {
    if (newQuantity <= 0) {
      removeFromCart(productId);
      return;
    }

    setCartItems((prevItems) =>
      prevItems.map((item) =>
        item.id === productId ? { ...item, quantity: newQuantity } : item
      )
    );
  };

  const clearCart = () => {
    setCartItems([]);
  };

  const totalPrice = cartItems.reduce(
    (sum, item) => sum + item.price * item.quantity,
    0
  );

  const totalItems = cartItems.reduce((sum, item) => sum + item.quantity, 0);

  return (
    <div className="shopping-cart">
      <h2>Shopping Cart ({totalItems} items)</h2>

      {cartItems.length === 0 ? (
        <p>Your cart is empty</p>
      ) : (
        <>
          {cartItems.map((item) => (
            <div key={item.id} className="cart-item">
              <span>{item.name}</span>
              <span>${item.price}</span>
              <input
                type="number"
                value={item.quantity}
                onChange={(e) =>
                  updateQuantity(item.id, parseInt(e.target.value))
                }
                min="1"
              />
              <span>${(item.price * item.quantity).toFixed(2)}</span>
              <button onClick={() => removeFromCart(item.id)}>Remove</button>
```

```
            </div>
          ))}

          <div className="cart-total">
            <strong>Total: ${totalPrice.toFixed(2)}</strong>
          </div>

          <button onClick={clearCart}>Clear Cart</button>
        </>
      )}

      <div className="products">
        <h3>Products</h3>
        {products.map((product) => (
          <div key={product.id} className="product">
            <span>
              {product.name} - ${product.price}
            </span>
            <button onClick={() => addToCart(product)}>Add to Cart</button>
          </div>
        ))}
      </div>
    </div>
  );
}
```

Challenge 2: Multi-Step Form

Create a multi-step form with:

- Step navigation (next/previous)
- Form validation
- Progress indicator
- Data persistence between steps

▶ 💡 Solution

```
function MultiStepForm() {
  const [currentStep, setCurrentStep] = useState(1);
  const [formData, setFormData] = useState({
    // Step 1
    firstName: "",
    lastName: "",
    email: "",
    // Step 2
    address: "",
    city: "",
    zipCode: "",
    // Step 3
    cardNumber: "",
    expiryDate: "",
```

```
    cvv: "",
  });
  const [errors, setErrors] = useState({});

  const totalSteps = 3;

  const updateField = (field, value) => {
    setFormData((prev) => ({ ...prev, [field]: value }));
    if (errors[field]) {
      setErrors((prev) => ({ ...prev, [field]: null }));
    }
  };

  const validateStep = (step) => {
    const newErrors = {};

    if (step === 1) {
      if (!formData.firstName.trim()) newErrors.firstName = "Required";
      if (!formData.lastName.trim()) newErrors.lastName = "Required";
      if (!formData.email.includes("@"))
        newErrors.email = "Valid email required";
    } else if (step === 2) {
      if (!formData.address.trim()) newErrors.address = "Required";
      if (!formData.city.trim()) newErrors.city = "Required";
      if (!formData.zipCode.trim()) newErrors.zipCode = "Required";
    } else if (step === 3) {
      if (!formData.cardNumber.trim()) newErrors.cardNumber = "Required";
      if (!formData.expiryDate.trim()) newErrors.expiryDate = "Required";
      if (!formData.cvv.trim()) newErrors.cvv = "Required";
    }

    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  const nextStep = () => {
    if (validateStep(currentStep)) {
      setCurrentStep((prev) => Math.min(prev + 1, totalSteps));
    }
  };

  const prevStep = () => {
    setCurrentStep((prev) => Math.max(prev - 1, 1));
  };

  const handleSubmit = () => {
    if (validateStep(currentStep)) {
      console.log("Form submitted:", formData);
      alert("Form submitted successfully!");
    }
  };

  const renderStep = () => {
    switch (currentStep) {
```

```
        case 1:
          return (
            <div>
              <h3>Personal Information</h3>
              <input
                value={formData.firstName}
                onChange={(e) => updateField("firstName", e.target.value)}
                placeholder="First Name"
              />
              {errors.firstName && (
                <span className="error">{errors.firstName}</span>
              )}

              <input
                value={formData.lastName}
                onChange={(e) => updateField("lastName", e.target.value)}
                placeholder="Last Name"
              />
              {errors.lastName && (
                <span className="error">{errors.lastName}</span>
              )}

              <input
                type="email"
                value={formData.email}
                onChange={(e) => updateField("email", e.target.value)}
                placeholder="Email"
              />
              {errors.email && <span className="error">{errors.email}</span>}
            </div>
          );

        case 2:
          return (
            <div>
              <h3>Address Information</h3>
              <input
                value={formData.address}
                onChange={(e) => updateField("address", e.target.value)}
                placeholder="Address"
              />
              {errors.address && <span className="error">{errors.address}</span>}

              <input
                value={formData.city}
                onChange={(e) => updateField("city", e.target.value)}
                placeholder="City"
              />
              {errors.city && <span className="error">{errors.city}</span>}

              <input
                value={formData.zipCode}
                onChange={(e) => updateField("zipCode", e.target.value)}
                placeholder="Zip Code"
```

```
            />
            {errors.zipCode && <span className="error">{errors.zipCode}</span>}
          </div>
        );

      case 3:
        return (
          <div>
            <h3>Payment Information</h3>
            <input
              value={formData.cardNumber}
              onChange={(e) => updateField("cardNumber", e.target.value)}
              placeholder="Card Number"
            />
            {errors.cardNumber && (
              <span className="error">{errors.cardNumber}</span>
            )}

            <input
              value={formData.expiryDate}
              onChange={(e) => updateField("expiryDate", e.target.value)}
              placeholder="MM/YY"
            />
            {errors.expiryDate && (
              <span className="error">{errors.expiryDate}</span>
            )}

            <input
              value={formData.cvv}
              onChange={(e) => updateField("cvv", e.target.value)}
              placeholder="CVV"
            />
            {errors.cvv && <span className="error">{errors.cvv}</span>}
          </div>
        );

      default:
        return null;
    }
  };

  return (
    <div className="multi-step-form">
      <div className="progress-bar">
        {Array.from({ length: totalSteps }, (_, i) => (
          <div
            key={i + 1}
            className={`step ${currentStep >= i + 1 ? "active" : ""}`}
          >
            {i + 1}
          </div>
        ))}
      </div>
```

```
      {renderStep()}

      <div className="form-navigation">
        {currentStep > 1 && <button onClick={prevStep}>Previous</button>}

        {currentStep < totalSteps ? (
          <button onClick={nextStep}>Next</button>
        ) : (
          <button onClick={handleSubmit}>Submit</button>
        )}
      </div>

      <div className="step-indicator">
        Step {currentStep} of {totalSteps}
      </div>
    </div>
  );
}
```

Interview Insights

What Interviewers Look For

1. **State Update Understanding**: "What happens when you call setState?"
    - **Good Answer**: setState schedules a re-render, state updates are asynchronous, React may batch multiple updates
2. **Functional Updates**: "When would you use a function in setState?"
    - **Good Answer**: When the new state depends on the previous state, to avoid stale closures, especially in async operations
3. **State Structure**: "How do you decide what to put in state?"
    - **Good Answer**: Only data that changes over time and affects rendering. Avoid derived values, prefer computed values.

Common Interview Questions

```
// Q: What's wrong with this code?
function BrokenCounter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
    setCount(count + 1); // Won't work as expected!
  };

  return <button onClick={increment}>Count: {count}</button>;
}
```

```
// Q: Fix this async state update
function AsyncProblem() {
  const [count, setCount] = useState(0);

  const delayedIncrement = () => {
    setTimeout(() => {
      setCount(count + 1); // Stale closure!
    }, 1000);
  };

  return <button onClick={delayedIncrement}>Count: {count}</button>;
}

// Q: Optimize this expensive initialization
function ExpensiveInit() {
  const [data, setData] = useState(expensiveCalculation()); // Runs every render!
  return <div>{data}</div>;
}
```

Pro Interview Tips

1. **Always mention functional updates** when discussing state that depends on previous state
2. **Explain the difference** between state and props clearly
3. **Show awareness of performance** implications (unnecessary re-renders)
4. **Demonstrate immutability** principles with objects and arrays
5. **Know when NOT to use state** (derived values, refs for non-rendering data)

🚀 Next Steps

Now that you've mastered useState, you're ready for:

- **Rendering Logic, Lists, and Keys** - Efficiently rendering dynamic content
- **Handling DOM Events** - User interactions and event handling
- **useEffect** - Side effects and lifecycle management

> **Key Takeaway**: useState is React's fundamental state primitive. Master functional updates, immutability, and when to use state vs computed values. Remember: state should only contain data that changes over time and affects what's rendered!

📋 Rendering Logic, Lists, and Keys: Dynamic Content Mastery

> **Master React's rendering patterns: conditional rendering, list rendering, and the crucial role of keys in React's reconciliation process**

🎯 What You'll Learn

- Conditional rendering patterns and best practices
- List rendering with map() and performance considerations
- The critical importance of keys in React's reconciliation
- Advanced rendering patterns and optimization techniques
- Common pitfalls and how to avoid them
- When and how to use React.Fragment

🔀 Conditional Rendering Patterns

Basic Conditional Rendering

```
function ConditionalBasics({ user, isLoading, error }) {
  // 1. Simple if statement (early return)
  if (isLoading) {
    return <div className="spinner">Loading...</div>;
  }

  if (error) {
    return <div className="error">Error: {error.message}</div>;
  }

  if (!user) {
    return <div className="empty">No user found</div>;
  }

  // 2. Ternary operator for inline conditions
  return (
    <div className="user-profile">
      <h1>{user.name}</h1>

      {/* Simple ternary */}
      <p>{user.isOnline ? "🟢 Online" : "⚫ Offline"}</p>

      {/* Ternary with components */}
      {user.isPremium ? (
        <PremiumBadge />
      ) : (
        <button onClick={() => upgradeToPremium(user.id)}>
          Upgrade to Premium
        </button>
      )}

      {/* Logical AND for optional rendering */}
      {user.bio && <p className="bio">{user.bio}</p>}

      {/* Multiple conditions */}
      {user.isAdmin && user.permissions.includes("delete") && (
        <button className="danger" onClick={() => deleteUser(user.id)}>
          Delete User
        </button>
      )}
```

```
    </div>
  );
}

// Helper components
function PremiumBadge() {
  return <span className="premium-badge">☆ Premium</span>;
}
```

Advanced Conditional Patterns

```
function AdvancedConditionals({ userRole, permissions, features }) {
  // 1. Switch-like pattern with object mapping
  const roleComponents = {
    admin: <AdminDashboard />,
    moderator: <ModeratorPanel />,
    user: <UserDashboard />,
    guest: <GuestWelcome />,
  };

  // 2. Complex conditional logic with helper functions
  const canAccessFeature = (feature) => {
    return permissions.includes(feature) || userRole === "admin";
  };

  const renderFeatureSection = () => {
    if (!features || features.length === 0) {
      return <EmptyState message="No features available" />;
    }

    const availableFeatures = features.filter((feature) =>
      canAccessFeature(feature.permission)
    );

    if (availableFeatures.length === 0) {
      return <AccessDenied />;
    }

    return (
      <div className="features">
        {availableFeatures.map((feature) => (
          <FeatureCard key={feature.id} feature={feature} />
        ))}
      </div>
    );
  };

  // 3. Nested conditional rendering
  const renderUserActions = () => {
    if (userRole === "guest") {
      return (
```

```
        <div className="guest-actions">
          <button>Sign In</button>
          <button>Sign Up</button>
        </div>
      );
    }

    return (
      <div className="user-actions">
        {userRole === "admin" && (
          <>
            <button>Manage Users</button>
            <button>System Settings</button>
          </>
        )}

        {(userRole === "admin" || userRole === "moderator") && (
          <button>Moderate Content</button>
        )}

        <button>Profile Settings</button>
        <button>Sign Out</button>
      </div>
    );
  };

  return (
    <div className="dashboard">
      {/* Render role-specific component */}
      {roleComponents[userRole] || <div>Unknown role</div>}

      {/* Conditional sections */}
      {renderFeatureSection()}
      {renderUserActions()}

      {/* Complex conditional with multiple factors */}
      {userRole !== "guest" && permissions.includes("notifications") && (
        <NotificationCenter />
      )}
    </div>
  );
}

// Helper components
function EmptyState({ message }) {
  return (
    <div className="empty-state">
      <p>{message}</p>
    </div>
  );
}

function AccessDenied() {
  return (
```

```
    <div className="access-denied">
      <p>You don't have permission to access these features.</p>
    </div>
  );
}
```

Conditional Rendering Anti-Patterns

```
// ❌ WRONG: These patterns can cause issues
function ConditionalAntiPatterns({ items, user }) {
  return (
    <div>
      {/* ❌ DON'T: Conditional rendering with numbers (0 is falsy!) */}
      {items.length && <div>You have {items.length} items</div>}
      {/* If items.length is 0, this renders "0" instead of nothing! */}

      {/* ❌ DON'T: Complex expressions in JSX */}
      {user &&
        user.preferences &&
        user.preferences.theme === "dark" &&
        user.isActive && <DarkThemeComponent />}

      {/* ❌ DON'T: Nested ternaries (hard to read) */}
      {user ? (
        user.isAdmin ? (
          user.permissions.includes("delete") ? (
            <DeleteButton />
          ) : (
            <span>No delete permission</span>
          )
        ) : (
          <span>Not an admin</span>
        )
      ) : (
        <span>No user</span>
      )}
    </div>
  );
}

// ✅ CORRECT: Better patterns
function ConditionalBestPractices({ items, user }) {
  // Helper functions for complex logic
  const hasItems = items && items.length > 0;
  const canDelete = user?.isAdmin && user?.permissions?.includes("delete");
  const isDarkThemeUser = user?.preferences?.theme === "dark" && user?.isActive;

  const renderUserStatus = () => {
    if (!user) return <span>No user</span>;
    if (!user.isAdmin) return <span>Not an admin</span>;
    if (!user.permissions.includes("delete"))
```

```
      return <span>No delete permission</span>;
    return <DeleteButton />;
  };

  return (
    <div>
      {/* ✅ DO: Explicit boolean conversion */}
      {hasItems && <div>You have {items.length} items</div>}

      {/* ✅ DO: Extract complex conditions */}
      {isDarkThemeUser && <DarkThemeComponent />}

      {/* ✅ DO: Use helper functions for complex logic */}
      {renderUserStatus()}

      {/* ✅ DO: Use early returns in helper functions */}
      {canDelete && <DeleteButton />}
    </div>
  );
}
```

📋 List Rendering Mastery

Basic List Rendering

```
function BasicListRendering() {
  const fruits = ["apple", "banana", "orange", "grape"];
  const users = [
    { id: 1, name: "John", email: "john@example.com", isActive: true },
    { id: 2, name: "Jane", email: "jane@example.com", isActive: false },
    { id: 3, name: "Bob", email: "bob@example.com", isActive: true },
  ];

  return (
    <div>
      {/* Simple array of strings */}
      <h3>Fruits</h3>
      <ul>
        {fruits.map((fruit, index) => (
          <li key={index}>{fruit}</li>
        ))}
      </ul>

      {/* Array of objects */}
      <h3>Users</h3>
      <div className="user-list">
        {users.map((user) => (
          <div key={user.id} className="user-card">
            <h4>{user.name}</h4>
            <p>{user.email}</p>
            <span className={`status ${user.isActive ? "active" : "inactive"}`}>
```

```
              {user.isActive ? "Active" : "Inactive"}
            </span>
          </div>
        ))}
      </div>

      {/* Conditional rendering within lists */}
      <h3>Active Users Only</h3>
      <div className="active-users">
        {users
          .filter((user) => user.isActive)
          .map((user) => (
            <UserCard key={user.id} user={user} />
          ))}
      </div>
    </div>
  );
}

function UserCard({ user }) {
  return (
    <div className="user-card">
      <h4>{user.name}</h4>
      <p>{user.email}</p>
    </div>
  );
}
```

Advanced List Patterns

```
function AdvancedListPatterns() {
  const [products, setProducts] = useState([
    {
      id: 1,
      name: "Laptop",
      category: "electronics",
      price: 999,
      inStock: true,
    },
    { id: 2, name: "Book", category: "education", price: 25, inStock: false },
    {
      id: 3,
      name: "Headphones",
      category: "electronics",
      price: 199,
      inStock: true,
    },
    { id: 4, name: "Notebook", category: "education", price: 5, inStock: true },
  ]);

  const [sortBy, setSortBy] = useState("name");
```

```
  const [filterCategory, setFilterCategory] = useState("all");
  const [showOutOfStock, setShowOutOfStock] = useState(true);

  // Complex filtering and sorting
  const processedProducts = useMemo(() => {
    let filtered = products;

    // Filter by category
    if (filterCategory !== "all") {
      filtered = filtered.filter(
        (product) => product.category === filterCategory
      );
    }

    // Filter by stock status
    if (!showOutOfStock) {
      filtered = filtered.filter((product) => product.inStock);
    }

    // Sort products
    filtered.sort((a, b) => {
      switch (sortBy) {
        case "name":
          return a.name.localeCompare(b.name);
        case "price":
          return a.price - b.price;
        case "category":
          return a.category.localeCompare(b.category);
        default:
          return 0;
      }
    });

    return filtered;
  }, [products, sortBy, filterCategory, showOutOfStock]);

  // Group products by category
  const groupedProducts = useMemo(() => {
    return processedProducts.reduce((groups, product) => {
      const category = product.category;
      if (!groups[category]) {
        groups[category] = [];
      }
      groups[category].push(product);
      return groups;
    }, {});
  }, [processedProducts]);

  const renderProductList = () => {
    if (processedProducts.length === 0) {
      return (
        <div className="empty-state">
          <p>No products found matching your criteria.</p>
        </div>
```

```
      );
    }

    return (
      <div className="product-list">
        {processedProducts.map((product) => (
          <ProductCard
            key={product.id}
            product={product}
            onToggleStock={() => toggleStock(product.id)}
          />
        ))}
      </div>
    );
  };

  const renderGroupedProducts = () => {
    return Object.entries(groupedProducts).map(
      ([category, categoryProducts]) => (
        <div key={category} className="category-group">
          <h3 className="category-title">
            {category.charAt(0).toUpperCase() + category.slice(1)}(
            {categoryProducts.length})
          </h3>
          <div className="category-products">
            {categoryProducts.map((product) => (
              <ProductCard
                key={product.id}
                product={product}
                onToggleStock={() => toggleStock(product.id)}
              />
            ))}
          </div>
        </div>
      )
    );
  };

  const toggleStock = (productId) => {
    setProducts((prevProducts) =>
      prevProducts.map((product) =>
        product.id === productId
          ? { ...product, inStock: !product.inStock }
          : product
      )
    );
  };

  return (
    <div className="product-manager">
      {/* Controls */}
      <div className="controls">
        <select value={sortBy} onChange={(e) => setSortBy(e.target.value)}>
          <option value="name">Sort by Name</option>
```

```
            <option value="price">Sort by Price</option>
            <option value="category">Sort by Category</option>
          </select>

          <select
            value={filterCategory}
            onChange={(e) => setFilterCategory(e.target.value)}
          >
            <option value="all">All Categories</option>
            <option value="electronics">Electronics</option>
            <option value="education">Education</option>
          </select>

          <label>
            <input
              type="checkbox"
              checked={showOutOfStock}
              onChange={(e) => setShowOutOfStock(e.target.checked)}
            />
            Show out of stock
          </label>
        </div>

        {/* View toggle */}
        <div className="view-toggle">
          <button onClick={() => setViewMode("list")}>List View</button>
          <button onClick={() => setViewMode("grouped")}>Grouped View</button>
        </div>

        {/* Render products */}
        {viewMode === "grouped" ? renderGroupedProducts() : renderProductList()}
      </div>
    );
  }

  function ProductCard({ product, onToggleStock }) {
    return (
      <div className={`product-card ${!product.inStock ? "out-of-stock" : ""}`}>
        <h4>{product.name}</h4>
        <p className="category">{product.category}</p>
        <p className="price">${product.price}</p>
        <p className="stock">
          {product.inStock ? "✅ In Stock" : "❌ Out of Stock"}
        </p>
        <button onClick={onToggleStock}>
          {product.inStock ? "Mark Out of Stock" : "Mark In Stock"}
        </button>
      </div>
    );
  }
```

🔑 Keys: The Heart of React's Reconciliation

Understanding Keys and Reconciliation

```
// React's reconciliation process:
// 1. Compare new virtual DOM with previous virtual DOM
// 2. Identify what changed
// 3. Update only the changed parts in real DOM
// 4. Keys help React identify which items changed, moved, or were added/removed

function KeysExplanation() {
  const [items, setItems] = useState([
    { id: 1, name: "Apple", color: "red" },
    { id: 2, name: "Banana", color: "yellow" },
    { id: 3, name: "Orange", color: "orange" },
  ]);

  const addItem = () => {
    const newItem = {
      id: Date.now(),
      name: "New Fruit",
      color: "green",
    };
    setItems((prevItems) => [newItem, ...prevItems]); // Add to beginning
  };

  const removeItem = (id) => {
    setItems((prevItems) => prevItems.filter((item) => item.id !== id));
  };

  const shuffleItems = () => {
    setItems((prevItems) => [...prevItems].sort(() => Math.random() - 0.5));
  };

  return (
    <div>
      <div className="controls">
        <button onClick={addItem}>Add Item to Beginning</button>
        <button onClick={shuffleItems}>Shuffle Items</button>
      </div>

      {/* ❌ BAD: Using index as key */}
      <div className="bad-example">
        <h3>❌ Bad: Index as Key</h3>
        {items.map((item, index) => (
          <ItemComponent
            key={index} // DON'T DO THIS!
            item={item}
            onRemove={() => removeItem(item.id)}
          />
        ))}
      </div>

      {/* ✅ GOOD: Using stable unique ID as key */}
```

```
      <div className="good-example">
        <h3>✅ Good: Stable ID as Key</h3>
        {items.map((item) => (
          <ItemComponent
            key={item.id} // DO THIS!
            item={item}
            onRemove={() => removeItem(item.id)}
          />
        ))}
      </div>
    </div>
  );
}

// Component with internal state to demonstrate key importance
function ItemComponent({ item, onRemove }) {
  const [isExpanded, setIsExpanded] = useState(false);
  const [inputValue, setInputValue] = useState("");

  // This component maintains its own state
  // Wrong keys can cause state to "stick" to wrong items

  return (
    <div className="item-component">
      <div className="item-header" onClick={() => setIsExpanded(!isExpanded)}>
        <span style={{ color: item.color }}>🍎</span>
        <span>{item.name}</span>
        <button
          onClick={(e) => {
            e.stopPropagation();
            onRemove();
          }}
        >
          Remove
        </button>
      </div>

      {isExpanded && (
        <div className="item-details">
          <p>Color: {item.color}</p>
          <input
            value={inputValue}
            onChange={(e) => setInputValue(e.target.value)}
            placeholder="Type something..."
          />
          <p>You typed: {inputValue}</p>
        </div>
      )}
    </div>
  );
}
```

Key Selection Strategies

```
function KeyStrategies() {
  // Different scenarios for choosing keys

  // 1. ✅ BEST: Stable unique IDs from data
  const usersWithIds = [
    { id: "user_123", name: "John", email: "john@example.com" },
    { id: "user_456", name: "Jane", email: "jane@example.com" },
  ];

  // 2. ✅ GOOD: Composite keys when no single unique field
  const userPosts = [
    { userId: 1, postId: 101, title: "First Post" },
    { userId: 1, postId: 102, title: "Second Post" },
    { userId: 2, postId: 103, title: "Another Post" },
  ];

  // 3. ⚠️ ACCEPTABLE: Index when list is static and never reordered
  const staticMenuItems = ["Home", "About", "Contact"]; // Never changes

  // 4. ✅ GOOD: Generated stable IDs
  const [dynamicItems, setDynamicItems] = useState([
    { tempId: generateId(), text: "Item 1" },
    { tempId: generateId(), text: "Item 2" },
  ]);

  return (
    <div>
      {/* ✅ Using stable IDs */}
      <section>
        <h3>Users (Stable IDs)</h3>
        {usersWithIds.map((user) => (
          <div key={user.id}>
            {user.name} - {user.email}
          </div>
        ))}
      </section>

      {/* ✅ Using composite keys */}
      <section>
        <h3>User Posts (Composite Keys)</h3>
        {userPosts.map((post) => (
          <div key={`${post.userId}-${post.postId}`}>
            User {post.userId}: {post.title}
          </div>
        ))}
      </section>

      {/* ⚠️ Index acceptable for static lists */}
      <section>
        <h3>Static Menu (Index OK)</h3>
```

```
        {staticMenuItems.map((item, index) => (
          <div key={index}>{item}</div>
        ))}
      </section>

      {/* ✅ Generated stable IDs */}
      <section>
        <h3>Dynamic Items (Generated IDs)</h3>
        {dynamicItems.map((item) => (
          <div key={item.tempId}>
            {item.text}
            <button onClick={() => removeItem(item.tempId)}>Remove</button>
          </div>
        ))}
      </section>
    </div>
  );
}

// Helper function to generate stable IDs
function generateId() {
  return `${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;
}
```

Key Anti-Patterns and Solutions

```
function KeyAntiPatterns() {
  const [todos, setTodos] = useState([
    { text: "Learn React", completed: false },
    { text: "Build a project", completed: false },
    { text: "Get a job", completed: false },
  ]);

  // ❌ WRONG: Random keys (breaks reconciliation)
  const renderWithRandomKeys = () => {
    return todos.map((todo) => (
      <div key={Math.random()}>
        {" "}
        {/* DON'T DO THIS! */}
        <TodoItem todo={todo} />
      </div>
    ));
  };

  // ❌ WRONG: Using array index when list can be reordered
  const renderWithIndexKeys = () => {
    return todos.map((todo, index) => (
      <div key={index}>
        {" "}
        {/* PROBLEMATIC for dynamic lists */}
        <TodoItem todo={todo} />
```

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

  // ❌ WRONG: Non-unique keys
  const renderWithNonUniqueKeys = () => {
    return todos.map((todo) => (
      <div key={todo.completed}>
        {" "}
        {/* NOT UNIQUE! */}
        <TodoItem todo={todo} />
      </div>
    ));
  };

  // ✅ CORRECT: Add stable IDs to data
  const [todosWithIds, setTodosWithIds] = useState([
    { id: 1, text: "Learn React", completed: false },
    { id: 2, text: "Build a project", completed: false },
    { id: 3, text: "Get a job", completed: false },
  ]);

  const renderCorrectly = () => {
    return todosWithIds.map((todo) => (
      <div key={todo.id}>
        {" "}
        {/* CORRECT! */}
        <TodoItem todo={todo} />
      </div>
    ));
  };

  // ✅ ALTERNATIVE: Use content-based keys when appropriate
  const renderWithContentKeys = () => {
    // Only if todo text is guaranteed to be unique and stable
    return todos.map((todo) => (
      <div key={todo.text}>
        {" "}
        {/* OK if text is unique */}
        <TodoItem todo={todo} />
      </div>
    ));
  };

  return (
    <div>
      <h3>Todo List</h3>
      {renderCorrectly()}
    </div>
  );
}

function TodoItem({ todo }) {
  const [isEditing, setIsEditing] = useState(false);
```

```
  const [editText, setEditText] = useState(todo.text);

  return (
    <div className="todo-item">
      {isEditing ? (
        <input
          value={editText}
          onChange={(e) => setEditText(e.target.value)}
          onBlur={() => setIsEditing(false)}
          autoFocus
        />
      ) : (
        <span onClick={() => setIsEditing(true)}>{todo.text}</span>
      )}

      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => {
          /* toggle todo */
        }}
      />
    </div>
  );
}
```

React.Fragment and Multiple Elements

Using React.Fragment

```
import React, { Fragment } from "react";

function FragmentExamples() {
  const items = ["apple", "banana", "orange"];

  return (
    <div>
      {/* ❌ PROBLEM: Extra wrapper div */}
      <div className="wrapper">
        <h3>Fruits</h3>
        <p>Here are some fruits:</p>
      </div>

      {/* ✅ SOLUTION 1: React.Fragment */}
      <React.Fragment>
        <h3>Fruits</h3>
        <p>Here are some fruits:</p>
      </React.Fragment>

      {/* ✅ SOLUTION 2: Fragment shorthand */}
      <>
```

```
        <h3>Fruits</h3>
        <p>Here are some fruits:</p>
      </>

      {/* ✅ SOLUTION 3: Fragment with key (when needed in lists) */}
      {items.map((item) => (
        <Fragment key={item}>
          <h4>{item}</h4>
          <p>Description of {item}</p>
        </Fragment>
      ))}

      {/* ✅ SOLUTION 4: Array of elements (less common) */}
      {[
        <h3 key="title">Fruits</h3>,
        <p key="description">Here are some fruits:</p>,
      ]}
    </div>
  );
}

// Real-world Fragment usage
function TableRowFragment({ user }) {
  // Fragment allows returning multiple <td> elements
  // without wrapping them in a <div> (which would break table structure)
  return (
    <>
      <td>{user.name}</td>
      <td>{user.email}</td>
      <td>{user.role}</td>
      <td>
        <button>Edit</button>
        <button>Delete</button>
      </td>
    </>
  );
}

function UserTable({ users }) {
  return (
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Email</th>
          <th>Role</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        {users.map((user) => (
          <tr key={user.id}>
            <TableRowFragment user={user} />
          </tr>
```

```
          ))}
        </tbody>
      </table>
    );
}
```

⚡ Performance Considerations

Optimizing List Rendering

```
import { memo, useMemo, useCallback } from "react";

// ✅ Memoized list item to prevent unnecessary re-renders
const ListItem = memo(function ListItem({ item, onUpdate, onDelete }) {
  console.log(`Rendering item: ${item.name}`);

  return (
    <div className="list-item">
      <span>{item.name}</span>
      <span>${item.price}</span>
      <button onClick={() => onUpdate(item.id)}>Update</button>
      <button onClick={() => onDelete(item.id)}>Delete</button>
    </div>
  );
});

function OptimizedList() {
  const [items, setItems] = useState([
    { id: 1, name: "Laptop", price: 999 },
    { id: 2, name: "Mouse", price: 25 },
    { id: 3, name: "Keyboard", price: 75 },
  ]);
  const [filter, setFilter] = useState("");

  // ✅ Memoize filtered items to avoid recalculation
  const filteredItems = useMemo(() => {
    console.log("Filtering items...");
    return items.filter((item) =>
      item.name.toLowerCase().includes(filter.toLowerCase())
    );
  }, [items, filter]);

  // ✅ Memoize callbacks to prevent child re-renders
  const handleUpdate = useCallback((id) => {
    setItems((prevItems) =>
      prevItems.map((item) =>
        item.id === id ? { ...item, price: item.price * 1.1 } : item
      )
    );
  }, []);
```

```
  const handleDelete = useCallback((id) => {
    setItems((prevItems) => prevItems.filter((item) => item.id !== id));
  }, []);

  return (
    <div>
      <input
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        placeholder="Filter items..."
      />

      <div className="item-list">
        {filteredItems.map((item) => (
          <ListItem
            key={item.id}
            item={item}
            onUpdate={handleUpdate}
            onDelete={handleDelete}
          />
        ))}
      </div>

      <p>
        Showing {filteredItems.length} of {items.length} items
      </p>
    </div>
  );
}
```

Virtual Scrolling for Large Lists

```
// For very large lists (1000+ items), consider virtual scrolling
function VirtualScrollExample() {
  const [items] = useState(() =>
    Array.from({ length: 10000 }, (_, i) => ({
      id: i,
      name: `Item ${i}`,
      value: Math.random() * 100,
    }))
  );

  // This is a simplified example - use libraries like react-window or react-
virtualized
  const [visibleRange, setVisibleRange] = useState({ start: 0, end: 50 });
  const itemHeight = 50;
  const containerHeight = 400;

  const visibleItems = useMemo(() => {
    return items.slice(visibleRange.start, visibleRange.end);
  }, [items, visibleRange]);
```

```
  const handleScroll = (e) => {
    const scrollTop = e.target.scrollTop;
    const start = Math.floor(scrollTop / itemHeight);
    const end = Math.min(
      start + Math.ceil(containerHeight / itemHeight) + 5,
      items.length
    );

    setVisibleRange({ start, end });
  };

  return (
    <div
      className="virtual-scroll-container"
      style={{ height: containerHeight, overflow: "auto" }}
      onScroll={handleScroll}
    >
      <div style={{ height: items.length * itemHeight, position: "relative" }}>
        {visibleItems.map((item, index) => (
          <div
            key={item.id}
            style={{
              position: "absolute",
              top: (visibleRange.start + index) * itemHeight,
              height: itemHeight,
              width: "100%",
            }}
          >
            {item.name} - {item.value.toFixed(2)}
          </div>
        ))}
      </div>
    </div>
  );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Index as Key in Dynamic Lists

```
// ❌ WRONG: Index as key with dynamic operations
function BrokenTodoList() {
  const [todos, setTodos] = useState([
    "Learn React",
    "Build a project",
    "Get a job",
  ]);

  const addTodo = () => {
    setTodos(["New todo", ...todos]); // Adding to beginning
```

```
  };

  return (
    <div>
      <button onClick={addTodo}>Add Todo</button>
      {todos.map((todo, index) => (
        <TodoInput key={index} initialValue={todo} /> // BROKEN!
      ))}
    </div>
  );
}

function TodoInput({ initialValue }) {
  const [value, setValue] = useState(initialValue);

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

// ✅ CORRECT: Stable keys
function FixedTodoList() {
  const [todos, setTodos] = useState([
    { id: 1, text: "Learn React" },
    { id: 2, text: "Build a project" },
    { id: 3, text: "Get a job" },
  ]);

  const addTodo = () => {
    const newTodo = { id: Date.now(), text: "New todo" };
    setTodos([newTodo, ...todos]);
  };

  return (
    <div>
      <button onClick={addTodo}>Add Todo</button>
      {todos.map((todo) => (
        <TodoInput key={todo.id} initialValue={todo.text} /> // FIXED!
      ))}
    </div>
  );
}
```

2. Conditional Rendering Gotchas

```
// ❌ WRONG: Falsy values rendering unexpectedly
function ConditionalGotchas({ items, count }) {
  return (
    <div>
      {/* ❌ Renders "0" when count is 0 */}
      {count && <div>Count: {count}</div>}

      {/* ❌ Renders "0" when items array is empty */}
```

```
      {items.length && <div>Items: {items.length}</div>}

      {/* ❌ Can render "NaN" or unexpected values */}
      {items.length && items.length > 0 && (
        <div>Average: {items.reduce((a, b) => a + b, 0) / items.length}</div>
      )}
    </div>
  );
}

// ✅ CORRECT: Explicit boolean conversion
function ConditionalFixed({ items, count }) {
  return (
    <div>
      {/* ✅ Explicit boolean conversion */}
      {count > 0 && <div>Count: {count}</div>}

      {/* ✅ Explicit length check */}
      {items.length > 0 && <div>Items: {items.length}</div>}

      {/* ✅ Safe calculation with proper checks */}
      {items.length > 0 && (
        <div>
          Average:{" "}
          {(items.reduce((a, b) => a + b, 0) / items.length).toFixed(2)}
        </div>
      )}
    </div>
  );
}
```

🧪 Mini Challenges

Challenge 1: Dynamic Table with Sorting

Build a sortable table component with:

- Click column headers to sort
- Visual indicators for sort direction
- Multiple data types (string, number, date)

```
// Your solution:
function SortableTable({ data, columns }) {
  // Implement sorting logic
}

// Test data:
const userData = [
  {
    id: 1,
    name: "John",
```

```
    age: 30,
    email: "john@example.com",
    joinDate: "2023-01-15",
  },
  {
    id: 2,
    name: "Jane",
    age: 25,
    email: "jane@example.com",
    joinDate: "2023-03-20",
  },
  {
    id: 3,
    name: "Bob",
    age: 35,
    email: "bob@example.com",
    joinDate: "2023-02-10",
  },
];

const columns = [
  { key: "name", label: "Name", type: "string" },
  { key: "age", label: "Age", type: "number" },
  { key: "email", label: "Email", type: "string" },
  { key: "joinDate", label: "Join Date", type: "date" },
];
```

▶ 💡 Solution

```
function SortableTable({ data, columns }) {
  const [sortConfig, setSortConfig] = useState({ key: null, direction: "asc" });

  const sortedData = useMemo(() => {
    if (!sortConfig.key) return data;

    return [...data].sort((a, b) => {
      const aValue = a[sortConfig.key];
      const bValue = b[sortConfig.key];

      // Find column type
      const column = columns.find((col) => col.key === sortConfig.key);
      const type = column?.type || "string";

      let comparison = 0;

      switch (type) {
        case "number":
          comparison = aValue - bValue;
          break;
        case "date":
          comparison = new Date(aValue) - new Date(bValue);
          break;
```

```
          case "string":
          default:
            comparison = aValue.localeCompare(bValue);
            break;
        }

        return sortConfig.direction === "desc" ? -comparison : comparison;
      });
    }, [data, sortConfig, columns]);

    const handleSort = (key) => {
      setSortConfig((prevConfig) => ({
        key,
        direction:
          prevConfig.key === key && prevConfig.direction === "asc"
            ? "desc"
            : "asc",
      }));
    };

    const getSortIcon = (columnKey) => {
      if (sortConfig.key !== columnKey) return "↕";
      return sortConfig.direction === "asc" ? "↑" : "↓";
    };

    return (
      <table className="sortable-table">
        <thead>
          <tr>
            {columns.map((column) => (
              <th
                key={column.key}
                onClick={() => handleSort(column.key)}
                className="sortable-header"
              >
                {column.label} {getSortIcon(column.key)}
              </th>
            ))}
          </tr>
        </thead>
        <tbody>
          {sortedData.map((row) => (
            <tr key={row.id}>
              {columns.map((column) => (
                <td key={column.key}>{row[column.key]}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
```

Challenge 2: Nested Comment System

Create a nested comment component with:

- Recursive rendering of replies
- Expand/collapse functionality
- Add reply functionality

▶ 💡 Solution

```
function CommentSystem({ comments }) {
  return (
    <div className="comment-system">
      {comments.map((comment) => (
        <Comment key={comment.id} comment={comment} />
      ))}
    </div>
  );
}

function Comment({ comment, depth = 0 }) {
  const [isExpanded, setIsExpanded] = useState(true);
  const [showReplyForm, setShowReplyForm] = useState(false);
  const [replies, setReplies] = useState(comment.replies || []);

  const addReply = (replyText) => {
    const newReply = {
      id: Date.now(),
      author: "Current User",
      text: replyText,
      timestamp: new Date().toISOString(),
      replies: [],
    };

    setReplies((prevReplies) => [...prevReplies, newReply]);
    setShowReplyForm(false);
  };

  return (
    <div className="comment" style={{ marginLeft: `${depth * 20}px` }}>
      <div className="comment-header">
        <strong>{comment.author}</strong>
        <span className="timestamp">{comment.timestamp}</span>
        {replies.length > 0 && (
          <button
            onClick={() => setIsExpanded(!isExpanded)}
            className="toggle-replies"
          >
            {isExpanded ? "▼" : "▶"} {replies.length} replies
          </button>
        )}
      </div>
```

```
      <div className="comment-text">{comment.text}</div>

      <div className="comment-actions">
        <button onClick={() => setShowReplyForm(!showReplyForm)}>Reply</button>
      </div>

      {showReplyForm && (
        <ReplyForm
          onSubmit={addReply}
          onCancel={() => setShowReplyForm(false)}
        />
      )}

      {isExpanded && replies.length > 0 && (
        <div className="replies">
          {replies.map((reply) => (
            <Comment key={reply.id} comment={reply} depth={depth + 1} />
          ))}
        </div>
      )}
    </div>
  );
}

function ReplyForm({ onSubmit, onCancel }) {
  const [text, setText] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    if (text.trim()) {
      onSubmit(text.trim());
      setText("");
    }
  };

  return (
    <form onSubmit={handleSubmit} className="reply-form">
      <textarea
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Write a reply..."
        rows={3}
      />
      <div className="form-actions">
        <button type="submit" disabled={!text.trim()}>
          Post Reply
        </button>
        <button type="button" onClick={onCancel}>
          Cancel
        </button>
      </div>
    </form>
```

```
  );
}
```

Interview Insights

What Interviewers Look For

1. **Key Understanding**: "Why are keys important in React?"
   - **Good Answer**: Keys help React identify which items have changed, been added, or removed. They're crucial for efficient reconciliation and maintaining component state correctly.
2. **Conditional Rendering**: "What's wrong with `{count && <div>{count}</div>}`?"
   - **Good Answer**: If count is 0, it will render "0" instead of nothing. Should use `{count > 0 && <div>{count}</div>}`
3. **Performance**: "How would you optimize a large list?"
   - **Good Answer**: Use React.memo for list items, useMemo for filtering/sorting, useCallback for event handlers, consider virtual scrolling for very large lists.

Common Interview Questions

```jsx
// Q: What will this render when items is an empty array?
function Question1({ items }) {
  return <div>{items.length && <div>Items: {items.length}</div>}</div>;
}
// Answer: It will render "0" because items.length is 0 (falsy)

// Q: What's the problem with this key usage?
function Question2({ todos }) {
  return (
    <div>
      {todos.map((todo, index) => (
        <TodoItem key={index} todo={todo} />
      ))}
    </div>
  );
}
// Answer: Index as key can cause issues when list is reordered or items are
added/removed

// Q: How would you render this list efficiently?
function Question3({ largeList }) {
  return (
    <div>
      {largeList.map((item) => (
        <ExpensiveComponent key={item.id} item={item} />
      ))}
    </div>
```

```
    );
}
// Answer: Memoize ExpensiveComponent with React.memo, use virtual scrolling for
very large lists
```

Pro Interview Tips

1. **Always mention keys** when discussing list rendering
2. **Explain reconciliation** - show you understand how React works under the hood
3. **Discuss performance** implications of different patterns
4. **Show awareness of edge cases** (falsy values, empty arrays, etc.)
5. **Demonstrate optimization knowledge** (memo, useMemo, virtual scrolling)

🚀 Next Steps

Now that you've mastered rendering patterns, you're ready for:

- **Handling DOM Events** - User interactions and event handling
- **Controlled vs Uncontrolled Inputs** - Form handling patterns
- **useEffect** - Side effects and lifecycle management

> **Key Takeaway**: Master conditional rendering, list rendering with proper keys, and React's reconciliation process. Keys are crucial for performance and correctness - always use stable, unique identifiers when possible!

🎯 Handling DOM Events: User Interactions Mastery

> **Master React's event system: SyntheticEvents, event handling patterns, and building interactive user interfaces**

🎯 What You'll Learn

- React's SyntheticEvent system and how it differs from native events
- Event handling patterns and best practices
- Event delegation and performance considerations
- Form events and input handling
- Mouse, keyboard, and touch events
- Event propagation, bubbling, and capturing
- Custom event handlers and event composition
- Common event handling mistakes and solutions

⚡ React's SyntheticEvent System

Understanding SyntheticEvents

```
// React wraps native DOM events in SyntheticEvent objects
// This provides cross-browser compatibility and consistent behavior

function SyntheticEventDemo() {
  const handleClick = (event) => {
    console.log("Event type:", event.type); // 'click'
    console.log("Is SyntheticEvent:", event.nativeEvent !== undefined); // true
    console.log("Target element:", event.target); // The clicked element
    console.log("Current target:", event.currentTarget); // The element with the
event handler
    console.log("Native event:", event.nativeEvent); // Original DOM event

    // SyntheticEvent properties (cross-browser compatible)
    console.log("Timestamp:", event.timeStamp);
    console.log("Bubbles:", event.bubbles);
    console.log("Cancelable:", event.cancelable);

    // Prevent default behavior
    event.preventDefault();

    // Stop event propagation
    event.stopPropagation();
  };

  const handleMouseEvent = (event) => {
    // Mouse-specific properties
    console.log("Mouse position:", { x: event.clientX, y: event.clientY });
    console.log("Button pressed:", event.button); // 0: left, 1: middle, 2: right
    console.log("Modifier keys:", {
      ctrl: event.ctrlKey,
      shift: event.shiftKey,
      alt: event.altKey,
      meta: event.metaKey, // Cmd on Mac, Windows key on PC
    });
  };

  const handleKeyEvent = (event) => {
    // Keyboard-specific properties
    console.log("Key pressed:", event.key); // 'a', 'Enter', 'ArrowUp', etc.
    console.log("Key code:", event.keyCode); // Deprecated but still used
    console.log("Which:", event.which); // Deprecated
    console.log("Code:", event.code); // 'KeyA', 'Enter', 'ArrowUp', etc.

    // Modern way to check for specific keys
    if (event.key === "Enter") {
      console.log("Enter key pressed!");
    }

    if (event.key === "Escape") {
      console.log("Escape key pressed!");
    }
  };
```

```
  return (
    <div>
      <button onClick={handleClick}>Click me (check console)</button>

      <div
        onMouseMove={handleMouseEvent}
        style={{
          width: 200,
          height: 100,
          border: "1px solid #ccc",
          margin: "10px 0",
        }}
      >
        Move mouse here
      </div>

      <input
        type="text"
        onKeyDown={handleKeyEvent}
        placeholder="Type here and check console"
      />
    </div>
  );
}
```

Event Handler Patterns

```
function EventHandlerPatterns() {
  const [message, setMessage] = useState("");
  const [count, setCount] = useState(0);

  // 1. ✅ Inline arrow function (good for simple handlers)
  const inlineHandler = (
    <button onClick={() => setCount(count + 1)}>Count: {count}</button>
  );

  // 2. ✅ Function declaration (good for complex logic)
  function handleComplexClick(event) {
    console.log("Complex click handler");
    setCount((prevCount) => prevCount + 1);
    setMessage(`Button clicked at ${new Date().toLocaleTimeString()}`);
  }

  // 3. ✅ Arrow function with useCallback (for performance)
  const handleOptimizedClick = useCallback((event) => {
    setCount((prevCount) => prevCount + 1);
  }, []); // Empty dependency array since we use functional update

  // 4. ✅ Parameterized event handlers
  const handleParameterizedClick = (increment) => (event) => {
    setCount((prevCount) => prevCount + increment);
```

```
  };

  // 5. ✅ Event handler with additional parameters
  const handleClickWithParams = (id, action) => (event) => {
    console.log(`Performing ${action} on item ${id}`);
    // Handle the action
  };

  // 6. ❌ WRONG: Calling function immediately
  const wrongHandler = (
    <button onClick={handleComplexClick()}>
      {" "}
      {/* DON'T DO THIS! */}
      Wrong way
    </button>
  );

  return (
    <div>
      <h3>Event Handler Patterns</h3>

      {/* Pattern 1: Inline */}
      {inlineHandler}

      {/* Pattern 2: Function reference */}
      <button onClick={handleComplexClick}>Complex Handler</button>

      {/* Pattern 3: Optimized with useCallback */}
      <button onClick={handleOptimizedClick}>Optimized Handler</button>

      {/* Pattern 4: Parameterized */}
      <button onClick={handleParameterizedClick(5)}>Add 5</button>
      <button onClick={handleParameterizedClick(-1)}>Subtract 1</button>

      {/* Pattern 5: With additional parameters */}
      <button onClick={handleClickWithParams("user123", "delete")}>
        Delete User
      </button>

      <p>Message: {message}</p>
    </div>
  );
}
```

🖱️ Mouse Events

Comprehensive Mouse Event Handling

```
function MouseEventHandling() {
  const [mouseState, setMouseState] = useState({
    position: { x: 0, y: 0 },
```

```
    isDown: false,
    button: null,
    clickCount: 0,
    dragStart: null,
    isDragging: false,
  });

  const [hoverState, setHoverState] = useState({
    isHovered: false,
    hoverTime: 0,
  });

  // Mouse movement tracking
  const handleMouseMove = (event) => {
    setMouseState((prev) => ({
      ...prev,
      position: {
        x: event.clientX,
        y: event.clientY,
      },
    }));

    // If dragging, update drag position
    if (mouseState.isDragging && mouseState.dragStart) {
      const deltaX = event.clientX - mouseState.dragStart.x;
      const deltaY = event.clientY - mouseState.dragStart.y;
      console.log("Drag delta:", { deltaX, deltaY });
    }
  };

  // Mouse button events
  const handleMouseDown = (event) => {
    setMouseState((prev) => ({
      ...prev,
      isDown: true,
      button: event.button,
      dragStart: {
        x: event.clientX,
        y: event.clientY,
      },
      isDragging: true,
    }));

    // Prevent text selection during drag
    event.preventDefault();
  };

  const handleMouseUp = (event) => {
    setMouseState((prev) => ({
      ...prev,
      isDown: false,
      button: null,
      dragStart: null,
      isDragging: false,
```

```
    }));
  };

  // Click events
  const handleClick = (event) => {
    setMouseState((prev) => ({
      ...prev,
      clickCount: prev.clickCount + 1,
    }));

    // Different actions based on button
    switch (event.button) {
      case 0: // Left click
        console.log("Left click");
        break;
      case 1: // Middle click
        console.log("Middle click");
        event.preventDefault(); // Prevent default middle-click behavior
        break;
      case 2: // Right click
        console.log("Right click");
        break;
    }
  };

  const handleDoubleClick = (event) => {
    console.log("Double click detected");
    setMouseState((prev) => ({
      ...prev,
      clickCount: prev.clickCount + 1,
    }));
  };

  // Context menu (right-click menu)
  const handleContextMenu = (event) => {
    event.preventDefault(); // Prevent default context menu
    console.log("Custom context menu would appear here");
  };

  // Hover events
  const handleMouseEnter = (event) => {
    setHoverState({
      isHovered: true,
      hoverTime: Date.now(),
    });
  };

  const handleMouseLeave = (event) => {
    const hoverDuration = Date.now() - hoverState.hoverTime;
    console.log(`Hovered for ${hoverDuration}ms`);

    setHoverState({
      isHovered: false,
      hoverTime: 0,
```

```
    });
  };

  // Mouse wheel events
  const handleWheel = (event) => {
    console.log("Wheel delta:", {
      deltaX: event.deltaX,
      deltaY: event.deltaY,
      deltaZ: event.deltaZ,
    });

    // Prevent page scroll if needed
    // event.preventDefault();
  };

  return (
    <div className="mouse-demo">
      <h3>Mouse Event Demo</h3>

      {/* Mouse tracking area */}
      <div
        className="mouse-area"
        style={{
          width: 400,
          height: 200,
          border: "2px solid #333",
          position: "relative",
          backgroundColor: hoverState.isHovered ? "#f0f0f0" : "#fff",
          cursor: mouseState.isDragging ? "grabbing" : "grab",
        }}
        onMouseMove={handleMouseMove}
        onMouseDown={handleMouseDown}
        onMouseUp={handleMouseUp}
        onClick={handleClick}
        onDoubleClick={handleDoubleClick}
        onContextMenu={handleContextMenu}
        onMouseEnter={handleMouseEnter}
        onMouseLeave={handleMouseLeave}
        onWheel={handleWheel}
      >
        <div className="instructions">
          <p>Try different mouse interactions:</p>
          <ul>
            <li>Move mouse to track position</li>
            <li>Click (left, middle, right)</li>
            <li>Double-click</li>
            <li>Right-click for context menu</li>
            <li>Drag to see drag detection</li>
            <li>Scroll wheel</li>
          </ul>
        </div>

        {/* Mouse position indicator */}
        <div
```

```
          style={{
            position: "absolute",
            left: mouseState.position.x - 5,
            top: mouseState.position.y - 5,
            width: 10,
            height: 10,
            backgroundColor: "red",
            borderRadius: "50%",
            pointerEvents: "none",
          }}
        />
      </div>

      {/* Mouse state display */}
      <div className="mouse-state">
        <h4>Mouse State:</h4>
        <p>
          Position: ({mouseState.position.x}, {mouseState.position.y})
        </p>
        <p>Mouse Down: {mouseState.isDown ? "Yes" : "No"}</p>
        <p>Button: {mouseState.button !== null ? mouseState.button : "None"}</p>
        <p>Click Count: {mouseState.clickCount}</p>
        <p>Dragging: {mouseState.isDragging ? "Yes" : "No"}</p>
        <p>Hovered: {hoverState.isHovered ? "Yes" : "No"}</p>
      </div>
    </div>
  );
}
```

⌨ Keyboard Events

Advanced Keyboard Event Handling

```
function KeyboardEventHandling() {
  const [keyState, setKeyState] = useState({
    lastKey: "",
    keysPressed: new Set(),
    keySequence: [],
    shortcuts: [],
  });

  const [textInput, setTextInput] = useState("");
  const [isComposing, setIsComposing] = useState(false);

  // Key down event
  const handleKeyDown = (event) => {
    const key = event.key;

    setKeyState((prev) => ({
      ...prev,
      lastKey: key,
```

```
      keysPressed: new Set([...prev.keysPressed, key]),
      keySequence: [...prev.keySequence, key].slice(-10), // Keep last 10 keys
    }));

    // Handle special key combinations
    if (event.ctrlKey && event.key === "s") {
      event.preventDefault();
      console.log("Save shortcut triggered");
      setKeyState((prev) => ({
        ...prev,
        shortcuts: [...prev.shortcuts, "Ctrl+S"],
      }));
    }

    if (event.ctrlKey && event.key === "z") {
      event.preventDefault();
      console.log("Undo shortcut triggered");
      setKeyState((prev) => ({
        ...prev,
        shortcuts: [...prev.shortcuts, "Ctrl+Z"],
      }));
    }

    // Handle arrow keys
    if (["ArrowUp", "ArrowDown", "ArrowLeft", "ArrowRight"].includes(key)) {
      console.log(`Arrow key pressed: ${key}`);
      // Prevent default scrolling if needed
      // event.preventDefault();
    }

    // Handle escape key
    if (key === "Escape") {
      console.log("Escape pressed - could close modal, clear input, etc.");
      setTextInput("");
    }

    // Handle enter key
    if (key === "Enter") {
      if (event.shiftKey) {
        console.log("Shift+Enter: New line");
      } else {
        console.log("Enter: Submit form");
        event.preventDefault();
      }
    }
  };

  // Key up event
  const handleKeyUp = (event) => {
    setKeyState((prev) => {
      const newKeysPressed = new Set(prev.keysPressed);
      newKeysPressed.delete(event.key);
      return {
        ...prev,
```

```
        keysPressed: newKeysPressed,
      };
    });
  };

  // Input composition events (for IME support)
  const handleCompositionStart = (event) => {
    setIsComposing(true);
    console.log("Composition started (IME input)");
  };

  const handleCompositionEnd = (event) => {
    setIsComposing(false);
    console.log("Composition ended:", event.data);
  };

  // Global keyboard shortcuts
  useEffect(() => {
    const handleGlobalKeyDown = (event) => {
      // Global shortcuts that work anywhere on the page
      if (event.ctrlKey && event.shiftKey && event.key === "K") {
        event.preventDefault();
        console.log("Global shortcut: Ctrl+Shift+K");
      }
    };

    document.addEventListener("keydown", handleGlobalKeyDown);

    return () => {
      document.removeEventListener("keydown", handleGlobalKeyDown);
    };
  }, []);

  // Custom hook for keyboard shortcuts
  const useKeyboardShortcut = (keys, callback) => {
    useEffect(() => {
      const handleKeyDown = (event) => {
        const pressedKeys = [];
        if (event.ctrlKey) pressedKeys.push("ctrl");
        if (event.shiftKey) pressedKeys.push("shift");
        if (event.altKey) pressedKeys.push("alt");
        if (event.metaKey) pressedKeys.push("meta");
        pressedKeys.push(event.key.toLowerCase());

        const shortcut = pressedKeys.join("+");
        if (keys.includes(shortcut)) {
          event.preventDefault();
          callback(event);
        }
      };

      document.addEventListener("keydown", handleKeyDown);
      return () => document.removeEventListener("keydown", handleKeyDown);
    }, [keys, callback]);
```

```
  };

  // Use the custom hook
  useKeyboardShortcut(["ctrl+k"], () => {
    console.log("Quick search triggered");
  });

  return (
    <div className="keyboard-demo">
      <h3>Keyboard Event Demo</h3>

      <div className="input-section">
        <textarea
          value={textInput}
          onChange={(e) => setTextInput(e.target.value)}
          onKeyDown={handleKeyDown}
          onKeyUp={handleKeyUp}
          onCompositionStart={handleCompositionStart}
          onCompositionEnd={handleCompositionEnd}
          placeholder="Type here to test keyboard events..."
          rows={4}
          cols={50}
        />

        <p>Composing: {isComposing ? "Yes" : "No"}</p>
      </div>

      <div className="keyboard-state">
        <h4>Keyboard State:</h4>
        <p>Last Key: {keyState.lastKey}</p>
        <p>
          Keys Currently Pressed: {Array.from(keyState.keysPressed).join(", ")}
        </p>
        <p>Key Sequence: {keyState.keySequence.join(" → ")}</p>
        <p>Shortcuts Used: {keyState.shortcuts.join(", ")}</p>
      </div>

      <div className="shortcuts-help">
        <h4>Try These Shortcuts:</h4>
        <ul>
          <li>Ctrl+S - Save</li>
          <li>Ctrl+Z - Undo</li>
          <li>Ctrl+K - Quick search</li>
          <li>Ctrl+Shift+K - Global shortcut</li>
          <li>Escape - Clear input</li>
          <li>Enter - Submit</li>
          <li>Shift+Enter - New line</li>
        </ul>
      </div>
    </div>
  );
}
```

Form Events

Comprehensive Form Event Handling

```
function FormEventHandling() {
  const [formData, setFormData] = useState({
    username: "",
    email: "",
    password: "",
    confirmPassword: "",
    age: "",
    country: "",
    interests: [],
    newsletter: false,
    bio: "",
  });

  const [formState, setFormState] = useState({
    isDirty: false,
    isValid: false,
    errors: {},
    touchedFields: new Set(),
    isSubmitting: false,
  });

  // Input change handler
  const handleInputChange = (event) => {
    const { name, value, type, checked } = event.target;

    setFormData((prev) => ({
      ...prev,
      [name]: type === "checkbox" ? checked : value,
    }));

    setFormState((prev) => ({
      ...prev,
      isDirty: true,
      touchedFields: new Set([...prev.touchedFields, name]),
    }));

    // Real-time validation
    validateField(name, type === "checkbox" ? checked : value);
  };

  // Multi-select handler
  const handleMultiSelectChange = (event) => {
    const { name, value } = event.target;

    setFormData((prev) => {
      const currentValues = prev[name] || [];
      const newValues = currentValues.includes(value)
        ? currentValues.filter((v) => v !== value)
```

```
        : [...currentValues, value];

      return {
        ...prev,
        [name]: newValues,
      };
    });
  };

  // Field validation
  const validateField = (fieldName, value) => {
    const errors = { ...formState.errors };

    switch (fieldName) {
      case "username":
        if (!value.trim()) {
          errors.username = "Username is required";
        } else if (value.length < 3) {
          errors.username = "Username must be at least 3 characters";
        } else {
          delete errors.username;
        }
        break;

      case "email":
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        if (!value.trim()) {
          errors.email = "Email is required";
        } else if (!emailRegex.test(value)) {
          errors.email = "Please enter a valid email";
        } else {
          delete errors.email;
        }
        break;

      case "password":
        if (!value) {
          errors.password = "Password is required";
        } else if (value.length < 8) {
          errors.password = "Password must be at least 8 characters";
        } else {
          delete errors.password;
        }
        break;

      case "confirmPassword":
        if (value !== formData.password) {
          errors.confirmPassword = "Passwords do not match";
        } else {
          delete errors.confirmPassword;
        }
        break;

      case "age":
```

```
        const ageNum = parseInt(value);
        if (!value) {
          errors.age = "Age is required";
        } else if (isNaN(ageNum) || ageNum < 13 || ageNum > 120) {
          errors.age = "Please enter a valid age (13-120)";
        } else {
          delete errors.age;
        }
        break;
    }

    setFormState((prev) => ({
      ...prev,
      errors,
      isValid: Object.keys(errors).length === 0,
    }));
  };

  // Focus and blur events
  const handleFocus = (event) => {
    const { name } = event.target;
    console.log(`Field focused: ${name}`);

    // Could show help text, highlight field, etc.
  };

  const handleBlur = (event) => {
    const { name, value } = event.target;
    console.log(`Field blurred: ${name}`);

    setFormState((prev) => ({
      ...prev,
      touchedFields: new Set([...prev.touchedFields, name]),
    }));

    // Validate on blur
    validateField(name, value);
  };

  // Form submission
  const handleSubmit = async (event) => {
    event.preventDefault();

    setFormState((prev) => ({ ...prev, isSubmitting: true }));

    // Validate all fields
    Object.keys(formData).forEach((field) => {
      validateField(field, formData[field]);
    });

    if (formState.isValid) {
      try {
        // Simulate API call
        await new Promise((resolve) => setTimeout(resolve, 2000));
```

```
        console.log("Form submitted:", formData);
        alert("Form submitted successfully!");

        // Reset form
        setFormData({
          username: "",
          email: "",
          password: "",
          confirmPassword: "",
          age: "",
          country: "",
          interests: [],
          newsletter: false,
          bio: "",
        });

        setFormState({
          isDirty: false,
          isValid: false,
          errors: {},
          touchedFields: new Set(),
          isSubmitting: false,
        });
      } catch (error) {
        console.error("Submission error:", error);
        setFormState((prev) => ({ ...prev, isSubmitting: false }));
      }
    } else {
      setFormState((prev) => ({ ...prev, isSubmitting: false }));
      console.log("Form has errors:", formState.errors);
    }
  };

  // Form reset
  const handleReset = (event) => {
    // event.preventDefault(); // Uncomment to prevent default reset
    console.log("Form reset");

    setFormData({
      username: "",
      email: "",
      password: "",
      confirmPassword: "",
      age: "",
      country: "",
      interests: [],
      newsletter: false,
      bio: "",
    });

    setFormState({
      isDirty: false,
      isValid: false,
      errors: {},
```

```
      touchedFields: new Set(),
      isSubmitting: false,
    });
  };

  return (
    <div className="form-demo">
      <h3>Form Event Handling Demo</h3>

      <form onSubmit={handleSubmit} onReset={handleReset}>
        {/* Text input */}
        <div className="form-group">
          <label htmlFor="username">Username:</label>
          <input
            type="text"
            id="username"
            name="username"
            value={formData.username}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            className={formState.errors.username ? "error" : ""}
          />
          {formState.errors.username && (
            <span className="error-message">{formState.errors.username}</span>
          )}
        </div>

        {/* Email input */}
        <div className="form-group">
          <label htmlFor="email">Email:</label>
          <input
            type="email"
            id="email"
            name="email"
            value={formData.email}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            className={formState.errors.email ? "error" : ""}
          />
          {formState.errors.email && (
            <span className="error-message">{formState.errors.email}</span>
          )}
        </div>

        {/* Password input */}
        <div className="form-group">
          <label htmlFor="password">Password:</label>
          <input
            type="password"
            id="password"
            name="password"
            value={formData.password}
```

```
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            className={formState.errors.password ? "error" : ""}
          />
          {formState.errors.password && (
            <span className="error-message">{formState.errors.password}</span>
          )}
        </div>

        {/* Confirm password */}
        <div className="form-group">
          <label htmlFor="confirmPassword">Confirm Password:</label>
          <input
            type="password"
            id="confirmPassword"
            name="confirmPassword"
            value={formData.confirmPassword}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            className={formState.errors.confirmPassword ? "error" : ""}
          />
          {formState.errors.confirmPassword && (
            <span className="error-message">
              {formState.errors.confirmPassword}
            </span>
          )}
        </div>

        {/* Number input */}
        <div className="form-group">
          <label htmlFor="age">Age:</label>
          <input
            type="number"
            id="age"
            name="age"
            value={formData.age}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            min="13"
            max="120"
            className={formState.errors.age ? "error" : ""}
          />
          {formState.errors.age && (
            <span className="error-message">{formState.errors.age}</span>
          )}
        </div>

        {/* Select dropdown */}
        <div className="form-group">
          <label htmlFor="country">Country:</label>
          <select
```

```
            id="country"
            name="country"
            value={formData.country}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
          >
            <option value="">Select a country</option>
            <option value="us">United States</option>
            <option value="ca">Canada</option>
            <option value="uk">United Kingdom</option>
            <option value="de">Germany</option>
            <option value="fr">France</option>
          </select>
        </div>

        {/* Checkboxes */}
        <div className="form-group">
          <label>Interests:</label>
          {["Technology", "Sports", "Music", "Travel", "Cooking"].map(
            (interest) => (
              <label key={interest} className="checkbox-label">
                <input
                  type="checkbox"
                  name="interests"
                  value={interest}
                  checked={formData.interests.includes(interest)}
                  onChange={handleMultiSelectChange}
                />
                {interest}
              </label>
            )
          )}
        </div>

        {/* Single checkbox */}
        <div className="form-group">
          <label className="checkbox-label">
            <input
              type="checkbox"
              name="newsletter"
              checked={formData.newsletter}
              onChange={handleInputChange}
            />
            Subscribe to newsletter
          </label>
        </div>

        {/* Textarea */}
        <div className="form-group">
          <label htmlFor="bio">Bio:</label>
          <textarea
            id="bio"
            name="bio"
```

```
            value={formData.bio}
            onChange={handleInputChange}
            onFocus={handleFocus}
            onBlur={handleBlur}
            rows={4}
            placeholder="Tell us about yourself..."
          />
        </div>

        {/* Form actions */}
        <div className="form-actions">
          <button
            type="submit"
            disabled={!formState.isValid || formState.isSubmitting}
          >
            {formState.isSubmitting ? "Submitting..." : "Submit"}
          </button>
          <button type="reset">Reset</button>
        </div>
      </form>

      {/* Form state display */}
      <div className="form-state">
        <h4>Form State:</h4>
        <p>Dirty: {formState.isDirty ? "Yes" : "No"}</p>
        <p>Valid: {formState.isValid ? "Yes" : "No"}</p>
        <p>Submitting: {formState.isSubmitting ? "Yes" : "No"}</p>
        <p>Touched Fields: {Array.from(formState.touchedFields).join(", ")}</p>
        <p>Errors: {Object.keys(formState.errors).length}</p>
      </div>
    </div>
  );
}
```

Event Propagation and Delegation

Understanding Event Flow

```
function EventPropagationDemo() {
  const [eventLog, setEventLog] = useState([]);

  const logEvent = (phase, element, event) => {
    setEventLog((prev) => [
      ...prev,
      {
        id: Date.now() + Math.random(),
        phase,
        element,
        timestamp: new Date().toLocaleTimeString(),
      },
    ]);
```

```
  };

  const clearLog = () => setEventLog([]);

  // Event handlers for different phases
  const handleGrandparentCapture = (event) => {
    logEvent("Capture", "Grandparent", event);
  };

  const handleGrandparentBubble = (event) => {
    logEvent("Bubble", "Grandparent", event);
  };

  const handleParentCapture = (event) => {
    logEvent("Capture", "Parent", event);
  };

  const handleParentBubble = (event) => {
    logEvent("Bubble", "Parent", event);
  };

  const handleChildClick = (event) => {
    logEvent("Target", "Child", event);

    // Uncomment to stop propagation
    // event.stopPropagation();
  };

  const handleChildStopPropagation = (event) => {
    logEvent("Target", "Child (Stop Propagation)", event);
    event.stopPropagation(); // Stops bubbling
  };

  const handleChildPreventDefault = (event) => {
    logEvent("Target", "Child (Prevent Default)", event);
    event.preventDefault(); // Prevents default behavior
  };

  return (
    <div className="propagation-demo">
      <h3>Event Propagation Demo</h3>

      <div className="controls">
        <button onClick={clearLog}>Clear Log</button>
      </div>

      {/* Event propagation example */}
      <div
        className="grandparent"
        style={{
          padding: "20px",
          border: "3px solid red",
          backgroundColor: "#ffe6e6",
        }}
```

```
          onClickCapture={handleGrandparentCapture}
          onClick={handleGrandparentBubble}
        >
          <strong>Grandparent (Red)</strong>

          <div
            className="parent"
            style={{
              padding: "20px",
              border: "3px solid blue",
              backgroundColor: "#e6f3ff",
              margin: "10px",
            }}
            onClickCapture={handleParentCapture}
            onClick={handleParentBubble}
          >
            <strong>Parent (Blue)</strong>

            <div style={{ margin: "10px" }}>
              <button
                className="child"
                style={{
                  padding: "10px",
                  border: "3px solid green",
                  backgroundColor: "#e6ffe6",
                  margin: "5px",
                }}
                onClick={handleChildClick}
              >
                Child (Normal)
              </button>

              <button
                className="child"
                style={{
                  padding: "10px",
                  border: "3px solid green",
                  backgroundColor: "#e6ffe6",
                  margin: "5px",
                }}
                onClick={handleChildStopPropagation}
              >
                Child (Stop Propagation)
              </button>

              <a
                href="#"
                style={{
                  display: "inline-block",
                  padding: "10px",
                  border: "3px solid green",
                  backgroundColor: "#e6ffe6",
                  margin: "5px",
                  textDecoration: "none",
```

```
              }}
              onClick={handleChildPreventDefault}
            >
              Link (Prevent Default)
            </a>
          </div>
        </div>
      </div>

      {/* Event log */}
      <div className="event-log">
        <h4>Event Log (Order of Execution):</h4>
        <div style={{ maxHeight: "200px", overflowY: "auto" }}>
          {eventLog.map((log) => (
            <div key={log.id} className="log-entry">
              <span className="timestamp">{log.timestamp}</span>
              <span className="phase">{log.phase}</span>
              <span className="element">{log.element}</span>
            </div>
          ))}
        </div>
      </div>

      <div className="explanation">
        <h4>Event Flow Explanation:</h4>
        <ol>
          <li>
            <strong>Capture Phase:</strong> Event travels from root to target
          </li>
          <li>
            <strong>Target Phase:</strong> Event reaches the target element
          </li>
          <li>
            <strong>Bubble Phase:</strong> Event bubbles back up to root
          </li>
        </ol>
        <p>
          Click different buttons to see how stopPropagation() and
          preventDefault() affect the flow.
        </p>
      </div>
    </div>
  );
}
```

Event Delegation Pattern

```
function EventDelegationDemo() {
  const [items, setItems] = useState([
    { id: 1, name: "Item 1", type: "task" },
    { id: 2, name: "Item 2", type: "note" },
```

```
    { id: 3, name: "Item 3", type: "task" },
    { id: 4, name: "Item 4", type: "note" },
  ]);

  // ❌ INEFFICIENT: Individual event handlers for each item
  const renderWithIndividualHandlers = () => {
    return items.map((item) => (
      <div key={item.id} className="item">
        <span>{item.name}</span>
        <button onClick={() => editItem(item.id)}>Edit</button>
        <button onClick={() => deleteItem(item.id)}>Delete</button>
        <button onClick={() => duplicateItem(item.id)}>Duplicate</button>
      </div>
    ));
  };

  // ✅ EFFICIENT: Event delegation with single handler
  const handleItemAction = (event) => {
    const action = event.target.dataset.action;
    const itemId = parseInt(event.target.dataset.itemId);

    if (!action || !itemId) return;

    switch (action) {
      case "edit":
        editItem(itemId);
        break;
      case "delete":
        deleteItem(itemId);
        break;
      case "duplicate":
        duplicateItem(itemId);
        break;
      case "toggle":
        toggleItem(itemId);
        break;
    }
  };

  const renderWithDelegation = () => {
    return (
      <div className="items-container" onClick={handleItemAction}>
        {items.map((item) => (
          <div key={item.id} className="item">
            <input
              type="checkbox"
              data-action="toggle"
              data-item-id={item.id}
            />
            <span>
              {item.name} ({item.type})
            </span>
            <button data-action="edit" data-item-id={item.id}>
              Edit
```

```
            </button>
            <button data-action="delete" data-item-id={item.id}>
              Delete
            </button>
            <button data-action="duplicate" data-item-id={item.id}>
              Duplicate
            </button>
          </div>
        ))}
      </div>
    );
  };

  // Action handlers
  const editItem = (id) => {
    console.log(`Editing item ${id}`);
    const newName = prompt("Enter new name:");
    if (newName) {
      setItems((prev) =>
        prev.map((item) => (item.id === id ? { ...item, name: newName } : item))
      );
    }
  };

  const deleteItem = (id) => {
    console.log(`Deleting item ${id}`);
    setItems((prev) => prev.filter((item) => item.id !== id));
  };

  const duplicateItem = (id) => {
    console.log(`Duplicating item ${id}`);
    const item = items.find((item) => item.id === id);
    if (item) {
      const newItem = {
        ...item,
        id: Math.max(...items.map((i) => i.id)) + 1,
        name: `${item.name} (Copy)`,
      };
      setItems((prev) => [...prev, newItem]);
    }
  };

  const toggleItem = (id) => {
    console.log(`Toggling item ${id}`);
    // Toggle logic here
  };

  const addItem = () => {
    const newItem = {
      id: Math.max(...items.map((i) => i.id)) + 1,
      name: `Item ${items.length + 1}`,
      type: Math.random() > 0.5 ? "task" : "note",
    };
    setItems((prev) => [...prev, newItem]);
```

```
  };

  return (
    <div className="delegation-demo">
      <h3>Event Delegation Demo</h3>

      <button onClick={addItem}>Add Item</button>

      <h4>✅ With Event Delegation (Efficient):</h4>
      {renderWithDelegation()}

      <div className="performance-note">
        <p>
          <strong>Performance Benefits:</strong>
        </p>
        <ul>
          <li>Single event listener instead of multiple</li>
          <li>Better memory usage</li>
          <li>Automatically handles dynamically added elements</li>
          <li>Easier to manage and maintain</li>
        </ul>
      </div>
    </div>
  );
}
```

📱 Touch Events

Mobile Touch Event Handling

```
function TouchEventHandling() {
  const [touchState, setTouchState] = useState({
    touches: [],
    gesture: null,
    swipeDirection: null,
    pinchScale: 1,
    rotation: 0,
  });

  const [dragState, setDragState] = useState({
    isDragging: false,
    startPosition: null,
    currentPosition: null,
  });

  // Touch start
  const handleTouchStart = (event) => {
    const touches = Array.from(event.touches).map((touch) => ({
      id: touch.identifier,
      x: touch.clientX,
      y: touch.clientY,
```

```
    }));

    setTouchState((prev) => ({
      ...prev,
      touches,
    }));

    if (touches.length === 1) {
      setDragState({
        isDragging: true,
        startPosition: { x: touches[0].x, y: touches[0].y },
        currentPosition: { x: touches[0].x, y: touches[0].y },
      });
    }

    console.log(`Touch start: ${touches.length} touches`);
  };

  // Touch move
  const handleTouchMove = (event) => {
    event.preventDefault(); // Prevent scrolling

    const touches = Array.from(event.touches).map((touch) => ({
      id: touch.identifier,
      x: touch.clientX,
      y: touch.clientY,
    }));

    setTouchState((prev) => ({
      ...prev,
      touches,
    }));

    // Single finger drag
    if (touches.length === 1 && dragState.isDragging) {
      setDragState((prev) => ({
        ...prev,
        currentPosition: { x: touches[0].x, y: touches[0].y },
      }));
    }

    // Two finger gestures (pinch/zoom)
    if (touches.length === 2) {
      const [touch1, touch2] = touches;
      const distance = Math.sqrt(
        Math.pow(touch2.x - touch1.x, 2) + Math.pow(touch2.y - touch1.y, 2)
      );

      // Calculate rotation
      const angle =
        (Math.atan2(touch2.y - touch1.y, touch2.x - touch1.x) * 180) / Math.PI;

      setTouchState((prev) => ({
        ...prev,
```

```
          gesture: "pinch",
          pinchScale: distance / 100, // Normalize
          rotation: angle,
        }));
      }
    };

    // Touch end
    const handleTouchEnd = (event) => {
      const remainingTouches = Array.from(event.touches).map((touch) => ({
        id: touch.identifier,
        x: touch.clientX,
        y: touch.clientY,
      }));

      // Detect swipe gesture
      if (
        dragState.isDragging &&
        dragState.startPosition &&
        dragState.currentPosition
      ) {
        const deltaX = dragState.currentPosition.x - dragState.startPosition.x;
        const deltaY = dragState.currentPosition.y - dragState.startPosition.y;
        const distance = Math.sqrt(deltaX * deltaX + deltaY * deltaY);

        if (distance > 50) {
          // Minimum swipe distance
          let direction;
          if (Math.abs(deltaX) > Math.abs(deltaY)) {
            direction = deltaX > 0 ? "right" : "left";
          } else {
            direction = deltaY > 0 ? "down" : "up";
          }

          setTouchState((prev) => ({
            ...prev,
            swipeDirection: direction,
          }));

          console.log(`Swipe detected: ${direction}`);

          // Clear swipe direction after a delay
          setTimeout(() => {
            setTouchState((prev) => ({
              ...prev,
              swipeDirection: null,
            }));
          }, 1000);
        }
      }

      setTouchState((prev) => ({
        ...prev,
        touches: remainingTouches,
```

```
        gesture: remainingTouches.length > 1 ? prev.gesture : null,
      }));

      if (remainingTouches.length === 0) {
        setDragState({
          isDragging: false,
          startPosition: null,
          currentPosition: null,
        });
      }

      console.log(`Touch end: ${remainingTouches.length} touches remaining`);
    };

    // Touch cancel (when touch is interrupted)
    const handleTouchCancel = (event) => {
      console.log("Touch cancelled");
      setTouchState({
        touches: [],
        gesture: null,
        swipeDirection: null,
        pinchScale: 1,
        rotation: 0,
      });

      setDragState({
        isDragging: false,
        startPosition: null,
        currentPosition: null,
      });
    };

    return (
      <div className="touch-demo">
        <h3>Touch Event Handling Demo</h3>

        <div
          className="touch-area"
          style={{
            width: 300,
            height: 200,
            border: "2px solid #333",
            backgroundColor: "#f0f0f0",
            position: "relative",
            touchAction: "none", // Disable default touch behaviors
            transform: `scale(${touchState.pinchScale})
rotate(${touchState.rotation}deg)`,
            transition: touchState.gesture ? "none" : "transform 0.2s",
          }}
          onTouchStart={handleTouchStart}
          onTouchMove={handleTouchMove}
          onTouchEnd={handleTouchEnd}
          onTouchCancel={handleTouchCancel}
        >
```

```
          <div className="instructions">
            <p>Touch Area</p>
            <p>Try:</p>
            <ul>
              <li>Single finger drag</li>
              <li>Swipe gestures</li>
              <li>Two finger pinch/zoom</li>
              <li>Two finger rotation</li>
            </ul>
          </div>

          {/* Visualize touch points */}
          {touchState.touches.map((touch) => (
            <div
              key={touch.id}
              style={{
                position: "absolute",
                left: touch.x - 10,
                top: touch.y - 10,
                width: 20,
                height: 20,
                backgroundColor: "red",
                borderRadius: "50%",
                pointerEvents: "none",
              }}
            />
          ))}

          {/* Show swipe direction */}
          {touchState.swipeDirection && (
            <div
              style={{
                position: "absolute",
                top: "50%",
                left: "50%",
                transform: "translate(-50%, -50%)",
                fontSize: "24px",
                fontWeight: "bold",
                color: "blue",
              }}
            >
              Swipe: {touchState.swipeDirection.toUpperCase()}
            </div>
          )}
        </div>

        {/* Touch state display */}
        <div className="touch-state">
          <h4>Touch State:</h4>
          <p>Active Touches: {touchState.touches.length}</p>
          <p>Gesture: {touchState.gesture || "None"}</p>
          <p>Swipe Direction: {touchState.swipeDirection || "None"}</p>
          <p>Pinch Scale: {touchState.pinchScale.toFixed(2)}</p>
          <p>Rotation: {touchState.rotation.toFixed(1)}°</p>
```

```
          <p>Dragging: {dragState.isDragging ? "Yes" : "No"}</p>
        </div>

        <div className="touch-tips">
          <h4>Touch Event Tips:</h4>
          <ul>
            <li>
              Use <code>touchAction: 'none'</code> to disable default behaviors
            </li>
            <li>
              Always call <code>preventDefault()</code> in touchmove to prevent
              scrolling
            </li>
            <li>
              Handle <code>touchcancel</code> for interrupted touches
            </li>
            <li>Use touch identifiers to track individual fingers</li>
            <li>Consider gesture libraries for complex interactions</li>
          </ul>
        </div>
      </div>
    );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Event Handler Performance Issues

```
// ❌ WRONG: Creating new functions on every render
function PerformanceIssues({ items }) {
  return (
    <div>
      {items.map((item) => (
        <div key={item.id}>
          {/* New function created on every render! */}
          <button onClick={() => console.log(item.id)}>Click</button>

          {/* Inline object creation */}
          <div style={{ color: item.active ? "green" : "red" }}>
            {item.name}
          </div>
        </div>
      ))}
    </div>
  );
}

// ✅ CORRECT: Optimized event handlers
function OptimizedEventHandlers({ items }) {
  // Memoized handler
  const handleItemClick = useCallback((itemId) => {
```

```
    console.log(itemId);
  }, []);

  // Memoized style objects
  const activeStyle = useMemo(() => ({ color: "green" }), []);
  const inactiveStyle = useMemo(() => ({ color: "red" }), []);

  return (
    <div>
      {items.map((item) => (
        <ItemComponent
          key={item.id}
          item={item}
          onClick={handleItemClick}
          activeStyle={activeStyle}
          inactiveStyle={inactiveStyle}
        />
      ))}
    </div>
  );
}

const ItemComponent = memo(({ item, onClick, activeStyle, inactiveStyle }) => {
  const handleClick = useCallback(() => {
    onClick(item.id);
  }, [item.id, onClick]);

  return (
    <div>
      <button onClick={handleClick}>Click</button>
      <div style={item.active ? activeStyle : inactiveStyle}>{item.name}</div>
    </div>
  );
});
```

2. Event Object Persistence

```
// ❌ WRONG: Accessing event after async operation
function EventPersistenceIssue() {
  const handleClick = (event) => {
    // This will cause an error!
    setTimeout(() => {
      console.log(event.target.value); // Error: event is pooled
    }, 1000);
  };

  return <button onClick={handleClick}>Click me</button>;
}

// ✅ CORRECT: Persist event data
function EventPersistenceFixed() {
```

```
  const handleClick = (event) => {
    // Extract needed data immediately
    const targetValue = event.target.value;
    const eventType = event.type;

    setTimeout(() => {
      console.log(targetValue); // Works!
      console.log(eventType); // Works!
    }, 1000);

    // Alternative: persist the entire event (React 17+)
    // event.persist();
  };

  return <button onClick={handleClick}>Click me</button>;
}
```

3. Incorrect Event Binding

```
// ❌ WRONG: Common binding mistakes
function EventBindingMistakes() {
  const [count, setCount] = useState(0);

  return (
    <div>
      {/* ❌ Calling function immediately */}
      <button onClick={setCount(count + 1)}>Wrong</button>

      {/* ❌ Missing function reference */}
      <button onClick="handleClick()">Wrong</button>

      {/* ❌ Incorrect this binding (class components) */}
      <button onClick={this.handleClick}>Might be wrong</button>
    </div>
  );
}

// ✅ CORRECT: Proper event binding
function EventBindingCorrect() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount((prev) => prev + 1);
  };

  return (
    <div>
      {/* ✅ Function reference */}
      <button onClick={handleClick}>Correct</button>

      {/* ✅ Inline arrow function */}
```

```
        <button onClick={() => setCount((prev) => prev + 1)}>Correct</button>

        {/* ✅ Functional update */}
        <button onClick={() => setCount(count + 1)}>Correct</button>
      </div>
    );
  }
```

🧪 Mini Challenges

Challenge 1: Keyboard Shortcut Manager

Build a component that:

- Registers multiple keyboard shortcuts
- Shows active shortcuts
- Handles conflicts
- Allows enabling/disabling shortcuts

▶ 💡 Solution

```
function KeyboardShortcutManager() {
  const [shortcuts, setShortcuts] = useState([
    { id: 1, keys: "ctrl+s", action: "Save", enabled: true },
    { id: 2, keys: "ctrl+z", action: "Undo", enabled: true },
    { id: 3, keys: "ctrl+y", action: "Redo", enabled: true },
    { id: 4, keys: "ctrl+k", action: "Search", enabled: true },
    { id: 5, keys: "escape", action: "Cancel", enabled: true },
  ]);

  const [activeKeys, setActiveKeys] = useState(new Set());
  const [lastTriggered, setLastTriggered] = useState(null);

  useEffect(() => {
    const handleKeyDown = (event) => {
      const pressedKeys = [];
      if (event.ctrlKey) pressedKeys.push("ctrl");
      if (event.shiftKey) pressedKeys.push("shift");
      if (event.altKey) pressedKeys.push("alt");
      if (event.metaKey) pressedKeys.push("meta");
      pressedKeys.push(event.key.toLowerCase());

      const keyCombo = pressedKeys.join("+");

      setActiveKeys(new Set(pressedKeys));

      const matchedShortcut = shortcuts.find(
        (shortcut) => shortcut.enabled && shortcut.keys === keyCombo
      );

      if (matchedShortcut) {
```

```
        event.preventDefault();
        setLastTriggered({
          shortcut: matchedShortcut,
          timestamp: new Date().toLocaleTimeString(),
        });
        console.log(`Triggered: ${matchedShortcut.action}`);
      }
    };

    const handleKeyUp = () => {
      setActiveKeys(new Set());
    };

    document.addEventListener("keydown", handleKeyDown);
    document.addEventListener("keyup", handleKeyUp);

    return () => {
      document.removeEventListener("keydown", handleKeyDown);
      document.removeEventListener("keyup", handleKeyUp);
    };
  }, [shortcuts]);

  const toggleShortcut = (id) => {
    setShortcuts((prev) =>
      prev.map((shortcut) =>
        shortcut.id === id
          ? { ...shortcut, enabled: !shortcut.enabled }
          : shortcut
      )
    );
  };

  const addShortcut = (keys, action) => {
    const newShortcut = {
      id: Math.max(...shortcuts.map((s) => s.id)) + 1,
      keys,
      action,
      enabled: true,
    };
    setShortcuts((prev) => [...prev, newShortcut]);
  };

  return (
    <div className="shortcut-manager">
      <h3>Keyboard Shortcut Manager</h3>

      <div className="active-keys">
        <h4>Currently Pressed Keys:</h4>
        <p>{Array.from(activeKeys).join(" + ") || "None"}</p>
      </div>

      {lastTriggered && (
        <div className="last-triggered">
          <h4>Last Triggered:</h4>
```

```
            <p>
              {lastTriggered.shortcut.action} at {lastTriggered.timestamp}
            </p>
          </div>
        )}

        <div className="shortcuts-list">
          <h4>Registered Shortcuts:</h4>
          {shortcuts.map((shortcut) => (
            <div key={shortcut.id} className="shortcut-item">
              <span
                className={`keys ${shortcut.enabled ? "enabled" : "disabled"}`}
              >
                {shortcut.keys}
              </span>
              <span className="action">{shortcut.action}</span>
              <button onClick={() => toggleShortcut(shortcut.id)}>
                {shortcut.enabled ? "Disable" : "Enable"}
              </button>
            </div>
          ))}
        </div>
      </div>
    );
  }
```

Challenge 2: Advanced Form Validation

Create a form with:

- Real-time validation
- Custom validation rules
- Field dependencies
- Async validation

▶ 💡 Solution

```
  function AdvancedFormValidation() {
    const [formData, setFormData] = useState({
      username: "",
      email: "",
      password: "",
      confirmPassword: "",
      age: "",
      terms: false,
    });

    const [validationState, setValidationState] = useState({
      errors: {},
      isValidating: {},
      touchedFields: new Set(),
```

```
  });

  // Validation rules
  const validationRules = {
    username: [
      {
        rule: (value) => value.length >= 3,
        message: "Username must be at least 3 characters",
      },
      {
        rule: (value) => /^[a-zA-Z0-9_]+$/.test(value),
        message: "Username can only contain letters, numbers, and underscores",
      },
    ],
    email: [
      {
        rule: (value) => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value),
        message: "Please enter a valid email",
      },
    ],
    password: [
      {
        rule: (value) => value.length >= 8,
        message: "Password must be at least 8 characters",
      },
      {
        rule: (value) => /(?=.*[a-z])/.test(value),
        message: "Password must contain lowercase letter",
      },
      {
        rule: (value) => /(?=.*[A-Z])/.test(value),
        message: "Password must contain uppercase letter",
      },
      {
        rule: (value) => /(?=.*\d)/.test(value),
        message: "Password must contain number",
      },
    ],
    confirmPassword: [
      {
        rule: (value) => value === formData.password,
        message: "Passwords do not match",
      },
    ],
    age: [
      {
        rule: (value) => parseInt(value) >= 13,
        message: "Must be at least 13 years old",
      },
      {
        rule: (value) => parseInt(value) <= 120,
        message: "Please enter a valid age",
      },
    ],
```

```
    terms: [
      { rule: (value) => value === true, message: "You must accept the terms" },
    ],
  };

  // Async validation (simulate API call)
  const asyncValidateUsername = async (username) => {
    if (username.length < 3) return;

    setValidationState((prev) => ({
      ...prev,
      isValidating: { ...prev.isValidating, username: true },
    }));

    // Simulate API call
    await new Promise((resolve) => setTimeout(resolve, 1000));

    const isAvailable = !["admin", "user", "test"].includes(
      username.toLowerCase()
    );

    setValidationState((prev) => {
      const newErrors = { ...prev.errors };
      if (!isAvailable) {
        newErrors.username = "Username is already taken";
      } else {
        delete newErrors.username;
      }

      return {
        ...prev,
        errors: newErrors,
        isValidating: { ...prev.isValidating, username: false },
      };
    });
  };

  const validateField = (fieldName, value) => {
    const rules = validationRules[fieldName] || [];
    const errors = { ...validationState.errors };

    for (const { rule, message } of rules) {
      if (!rule(value)) {
        errors[fieldName] = message;
        break;
      } else {
        delete errors[fieldName];
      }
    }

    setValidationState((prev) => ({
      ...prev,
      errors,
    }));
```

```
    // Async validation for username
    if (fieldName === "username" && !errors[fieldName]) {
      asyncValidateUsername(value);
    }
  };

  const handleInputChange = (event) => {
    const { name, value, type, checked } = event.target;
    const newValue = type === "checkbox" ? checked : value;

    setFormData((prev) => ({
      ...prev,
      [name]: newValue,
    }));

    // Validate field if it's been touched
    if (validationState.touchedFields.has(name)) {
      validateField(name, newValue);
    }
  };

  const handleBlur = (event) => {
    const { name, value } = event.target;

    setValidationState((prev) => ({
      ...prev,
      touchedFields: new Set([...prev.touchedFields, name]),
    }));

    validateField(name, value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();

    // Validate all fields
    Object.keys(formData).forEach((field) => {
      validateField(field, formData[field]);
    });

    const hasErrors = Object.keys(validationState.errors).length > 0;
    const isValidating = Object.values(validationState.isValidating).some(
      Boolean
    );

    if (!hasErrors && !isValidating) {
      console.log("Form submitted:", formData);
      alert("Form submitted successfully!");
    }
  };

  return (
    <form onSubmit={handleSubmit} className="advanced-form">
```

```
        <h3>Advanced Form Validation</h3>

        <div className="form-group">
          <label htmlFor="username">Username:</label>
          <input
            type="text"
            id="username"
            name="username"
            value={formData.username}
            onChange={handleInputChange}
            onBlur={handleBlur}
            className={validationState.errors.username ? "error" : ""}
          />
          {validationState.isValidating.username && (
            <span className="validating">Checking availability...</span>
          )}
          {validationState.errors.username && (
            <span className="error-message">
              {validationState.errors.username}
            </span>
          )}
        </div>

        <div className="form-group">
          <label htmlFor="email">Email:</label>
          <input
            type="email"
            id="email"
            name="email"
            value={formData.email}
            onChange={handleInputChange}
            onBlur={handleBlur}
            className={validationState.errors.email ? "error" : ""}
          />
          {validationState.errors.email && (
            <span className="error-message">{validationState.errors.email}</span>
          )}
        </div>

        <div className="form-group">
          <label htmlFor="password">Password:</label>
          <input
            type="password"
            id="password"
            name="password"
            value={formData.password}
            onChange={handleInputChange}
            onBlur={handleBlur}
            className={validationState.errors.password ? "error" : ""}
          />
          {validationState.errors.password && (
            <span className="error-message">
              {validationState.errors.password}
            </span>
```

```
          )}
        </div>

        <div className="form-group">
          <label htmlFor="confirmPassword">Confirm Password:</label>
          <input
            type="password"
            id="confirmPassword"
            name="confirmPassword"
            value={formData.confirmPassword}
            onChange={handleInputChange}
            onBlur={handleBlur}
            className={validationState.errors.confirmPassword ? "error" : ""}
          />
          {validationState.errors.confirmPassword && (
            <span className="error-message">
              {validationState.errors.confirmPassword}
            </span>
          )}
        </div>

        <div className="form-group">
          <label htmlFor="age">Age:</label>
          <input
            type="number"
            id="age"
            name="age"
            value={formData.age}
            onChange={handleInputChange}
            onBlur={handleBlur}
            className={validationState.errors.age ? "error" : ""}
          />
          {validationState.errors.age && (
            <span className="error-message">{validationState.errors.age}</span>
          )}
        </div>

        <div className="form-group">
          <label>
            <input
              type="checkbox"
              name="terms"
              checked={formData.terms}
              onChange={handleInputChange}
              onBlur={handleBlur}
            />
            I accept the terms and conditions
          </label>
          {validationState.errors.terms && (
            <span className="error-message">{validationState.errors.terms}</span>
          )}
        </div>

        <button
```

```
          type="submit"
          disabled={
            Object.keys(validationState.errors).length > 0 ||
            Object.values(validationState.isValidating).some(Boolean)
          }
        >
          Submit
        </button>
      </form>
    );
  }
```

🎯 When and Why to Use Different Event Patterns

Event Handler Patterns Decision Tree

```
📊 Choosing the Right Event Handler Pattern:

1. **Simple, one-time actions** → Inline arrow functions
   Example: onClick={() => setCount(count + 1)}

2. **Complex logic or reusable handlers** → Function declarations
   Example: const handleComplexSubmit = (event) => { /* complex logic */ }

3. **Performance-critical components** → useCallback + memo
   Example: const optimizedHandler = useCallback(() => {}, [deps])

4. **Dynamic parameters** → Higher-order functions
   Example: const handleClick = (id) => (event) => { /* use id */ }

5. **Many similar elements** → Event delegation
   Example: Single handler on parent with event.target checks

6. **Global shortcuts** → useEffect with document listeners
   Example: Global keyboard shortcuts, escape key handling
```

Performance Considerations

```
// 🎯 Performance Guidelines

// ✅ Good for small lists or simple interactions
const SimpleList = ({ items }) => (
  <div>
    {items.map((item) => (
      <button key={item.id} onClick={() => handleClick(item.id)}>
        {item.name}
      </button>
    ))}
  </div>
```

```
);

// ✅ Better for large lists or complex components
const OptimizedList = ({ items }) => {
  const handleClick = useCallback((event) => {
    const itemId = event.target.dataset.itemId;
    // Handle click
  }, []);

  return (
    <div onClick={handleClick}>
      {items.map((item) => (
        <button key={item.id} data-item-id={item.id}>
          {item.name}
        </button>
      ))}
    </div>
  );
};
```

🔦 Interview Insights

Common Interview Questions

1. **"Explain React's SyntheticEvent system"**
   - Cross-browser compatibility wrapper
   - Event pooling (React 16 and earlier)
   - Access to native event via `nativeEvent`
   - Consistent API across browsers
2. **"How do you handle performance with many event handlers?"**
   - Event delegation pattern
   - useCallback for expensive handlers
   - React.memo for component optimization
   - Avoid inline functions in render-heavy components
3. **"What's the difference between onClick and onClickCapture?"**
   - onClick: Bubble phase (default)
   - onClickCapture: Capture phase
   - Event flow: Capture → Target → Bubble
4. **"How do you prevent event bubbling?"**
   - `event.stopPropagation()` - Stops bubbling
   - `event.preventDefault()` - Prevents default behavior
   - `event.stopImmediatePropagation()` - Stops all handlers
5. **"Explain event delegation and when to use it"**

- Single event listener on parent element
- Use `event.target` to determine actual clicked element
- Better performance for large lists
- Automatically handles dynamically added elements

Code Review Red Flags

```
// 🚩 Red flags interviewers look for:

// 1. Calling functions immediately
<button onClick={handleClick()}> // ❌ Wrong

// 2. Not handling async operations properly
const handleClick = (event) => {
  setTimeout(() => {
    console.log(event.target); // ❌ Event is pooled
  }, 1000);
};

// 3. Memory leaks with global listeners
useEffect(() => {
  document.addEventListener('click', handler);
  // ❌ Missing cleanup
}, []);

// 4. Inefficient re-renders
{items.map(item => (
  <div onClick={() => expensive(item)}> // ❌ New function every render
))}

// 5. Not preventing default when needed
const handleSubmit = (event) => {
  // ❌ Missing event.preventDefault()
  // Form will submit and page will reload
};
```

🎯 Key Takeaways

1. **SyntheticEvents** provide cross-browser compatibility and consistent behavior
2. **Event delegation** is crucial for performance with large lists
3. **useCallback** and **memo** help optimize event handler performance
4. **Event propagation** understanding is essential for complex UIs
5. **Touch events** require special handling for mobile experiences
6. **Form events** need careful validation and state management
7. **Global event listeners** must be cleaned up to prevent memory leaks
8. **Event persistence** is important for async operations

Next up: 06-controlled-vs-uncontrolled-inputs.md - Master form input patterns and when to use each approach!

🎛️ Controlled vs Uncontrolled Inputs: Form Mastery

> **Master React's input patterns: When to control state, when to let the DOM handle it, and how to choose the right approach for your forms**

🎯 What You'll Learn

- The fundamental difference between controlled and uncontrolled components
- When and why to use each pattern
- Form validation strategies for both approaches
- Performance implications and optimization techniques
- Hybrid approaches and advanced patterns
- Real-world form handling scenarios
- Common pitfalls and how to avoid them

🎛️ Controlled Components: React Takes Control

Understanding Controlled Components

In controlled components, React state is the "single source of truth" for form inputs. The input's value is always driven by React state, and changes flow through event handlers.

```
function ControlledInputBasics() {
  const [inputValue, setInputValue] = useState("");
  const [textareaValue, setTextareaValue] = useState("");
  const [selectValue, setSelectValue] = useState("");
  const [checkboxValue, setCheckboxValue] = useState(false);
  const [radioValue, setRadioValue] = useState("");

  // ✅ Controlled input - value comes from state
  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  const handleTextareaChange = (event) => {
    setTextareaValue(event.target.value);
  };

  const handleSelectChange = (event) => {
    setSelectValue(event.target.value);
  };

  const handleCheckboxChange = (event) => {
    setCheckboxValue(event.target.checked);
  };
```

```
    const handleRadioChange = (event) => {
      setRadioValue(event.target.value);
    };

    return (
      <div className="controlled-demo">
        <h3>Controlled Components Demo</h3>

        {/* Text Input */}
        <div className="form-group">
          <label htmlFor="controlled-input">Text Input:</label>
          <input
            type="text"
            id="controlled-input"
            value={inputValue} // ✅ Value controlled by React state
            onChange={handleInputChange} // ✅ Changes update state
            placeholder="Type something..."
          />
          <p>Current value: "{inputValue}"</p>
          <p>Length: {inputValue.length}</p>
        </div>

        {/* Textarea */}
        <div className="form-group">
          <label htmlFor="controlled-textarea">Textarea:</label>
          <textarea
            id="controlled-textarea"
            value={textareaValue}
            onChange={handleTextareaChange}
            placeholder="Enter multiple lines..."
            rows={4}
          />
          <p>Lines: {textareaValue.split("\n").length}</p>
          <p>
            Words:{" "}
            {textareaValue.trim() ? textareaValue.trim().split(/\s+/).length : 0}
          </p>
        </div>

        {/* Select Dropdown */}
        <div className="form-group">
          <label htmlFor="controlled-select">Select:</label>
          <select
            id="controlled-select"
            value={selectValue}
            onChange={handleSelectChange}
          >
            <option value="">Choose an option...</option>
            <option value="option1">Option 1</option>
            <option value="option2">Option 2</option>
            <option value="option3">Option 3</option>
          </select>
          <p>Selected: {selectValue || "None"}</p>
        </div>
```

```
      {/* Checkbox */}
      <div className="form-group">
        <label>
          <input
            type="checkbox"
            checked={checkboxValue} // ✅ Use 'checked' for checkboxes
            onChange={handleCheckboxChange}
          />
          Checkbox Option
        </label>
        <p>Checked: {checkboxValue ? "Yes" : "No"}</p>
      </div>

      {/* Radio Buttons */}
      <div className="form-group">
        <fieldset>
          <legend>Radio Options:</legend>
          {["radio1", "radio2", "radio3"].map((option) => (
            <label key={option}>
              <input
                type="radio"
                name="controlled-radio"
                value={option}
                checked={radioValue === option} // ✅ Compare with current state
                onChange={handleRadioChange}
              />
              {option.charAt(0).toUpperCase() + option.slice(1)}
            </label>
          ))}
        </fieldset>
        <p>Selected: {radioValue || "None"}</p>
      </div>

      {/* Form State Summary */}
      <div className="state-summary">
        <h4>Current Form State:</h4>
        <pre>
          {JSON.stringify(
            {
              input: inputValue,
              textarea: textareaValue,
              select: selectValue,
              checkbox: checkboxValue,
              radio: radioValue,
            },
            null,
            2
          )}
        </pre>
      </div>
    </div>
  );
}
```

Advanced Controlled Component Patterns

```
function AdvancedControlledPatterns() {
  const [formData, setFormData] = useState({
    username: "",
    email: "",
    password: "",
    confirmPassword: "",
    bio: "",
    skills: [],
    experience: "",
    newsletter: false,
  });

  const [formState, setFormState] = useState({
    errors: {},
    touched: {},
    isSubmitting: false,
    isDirty: false,
  });

  // ✅ Generic handler for all form fields
  const handleFieldChange = (fieldName) => (event) => {
    const { type, value, checked } = event.target;
    const newValue = type === "checkbox" ? checked : value;

    setFormData((prev) => ({
      ...prev,
      [fieldName]: newValue,
    }));

    setFormState((prev) => ({
      ...prev,
      isDirty: true,
      touched: { ...prev.touched, [fieldName]: true },
    }));

    // Real-time validation
    validateField(fieldName, newValue);
  };

  // ✅ Multi-select handler
  const handleMultiSelectChange = (fieldName) => (event) => {
    const { value, checked } = event.target;

    setFormData((prev) => {
      const currentArray = prev[fieldName] || [];
      const newArray = checked
        ? [...currentArray, value]
        : currentArray.filter((item) => item !== value);
```

```
      return {
        ...prev,
        [fieldName]: newArray,
      };
    });
  };

  // ✅ Input transformation (e.g., formatting)
  const handleFormattedChange = (fieldName, formatter) => (event) => {
    const rawValue = event.target.value;
    const formattedValue = formatter(rawValue);

    setFormData((prev) => ({
      ...prev,
      [fieldName]: formattedValue,
    }));
  };

  // Formatters
  const formatters = {
    phone: (value) => {
      // Format as (XXX) XXX-XXXX
      const numbers = value.replace(/\D/g, "");
      if (numbers.length <= 3) return numbers;
      if (numbers.length <= 6)
        return `(${numbers.slice(0, 3)}) ${numbers.slice(3)}`;
      return `(${numbers.slice(0, 3)}) ${numbers.slice(3, 6)}-${numbers.slice(
        6,
        10
      )}`;
    },

    currency: (value) => {
      // Format as currency
      const numbers = value.replace(/[^\d.]/g, "");
      const parts = numbers.split(".");
      if (parts.length > 2) return parts[0] + "." + parts[1];
      return numbers;
    },

    uppercase: (value) => value.toUpperCase(),

    noSpaces: (value) => value.replace(/\s/g, ""),
  };

  // Validation
  const validateField = (fieldName, value) => {
    const errors = { ...formState.errors };

    switch (fieldName) {
      case "username":
        if (!value.trim()) {
          errors.username = "Username is required";
```

```
        } else if (value.length < 3) {
          errors.username = "Username must be at least 3 characters";
        } else if (!/^[a-zA-Z0-9_]+$/.test(value)) {
          errors.username =
            "Username can only contain letters, numbers, and underscores";
        } else {
          delete errors.username;
        }
        break;

      case "email":
        if (!value.trim()) {
          errors.email = "Email is required";
        } else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value)) {
          errors.email = "Please enter a valid email address";
        } else {
          delete errors.email;
        }
        break;

      case "password":
        if (!value) {
          errors.password = "Password is required";
        } else if (value.length < 8) {
          errors.password = "Password must be at least 8 characters";
        } else if (!/(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/.test(value)) {
          errors.password =
            "Password must contain uppercase, lowercase, and number";
        } else {
          delete errors.password;
        }
        break;

      case "confirmPassword":
        if (value !== formData.password) {
          errors.confirmPassword = "Passwords do not match";
        } else {
          delete errors.confirmPassword;
        }
        break;
    }

    setFormState((prev) => ({
      ...prev,
      errors,
    }));
  };

  const handleSubmit = (event) => {
    event.preventDefault();

    // Validate all fields
    Object.keys(formData).forEach((field) => {
      validateField(field, formData[field]);
```

```
    });

    const hasErrors = Object.keys(formState.errors).length > 0;

    if (!hasErrors) {
      setFormState((prev) => ({ ...prev, isSubmitting: true }));

      // Simulate API call
      setTimeout(() => {
        console.log("Form submitted:", formData);
        alert("Form submitted successfully!");
        setFormState((prev) => ({ ...prev, isSubmitting: false }));
      }, 2000);
    }
  };

  return (
    <form onSubmit={handleSubmit} className="advanced-controlled-form">
      <h3>Advanced Controlled Form Patterns</h3>

      {/* Basic text input with validation */}
      <div className="form-group">
        <label htmlFor="username">Username:</label>
        <input
          type="text"
          id="username"
          value={formData.username}
          onChange={handleFieldChange("username")}
          className={formState.errors.username ? "error" : ""}
        />
        {formState.errors.username && (
          <span className="error-message">{formState.errors.username}</span>
        )}
      </div>

      {/* Email with validation */}
      <div className="form-group">
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          value={formData.email}
          onChange={handleFieldChange("email")}
          className={formState.errors.email ? "error" : ""}
        />
        {formState.errors.email && (
          <span className="error-message">{formState.errors.email}</span>
        )}
      </div>

      {/* Password with strength indicator */}
      <div className="form-group">
        <label htmlFor="password">Password:</label>
        <input
```

```
            type="password"
            id="password"
            value={formData.password}
            onChange={handleFieldChange("password")}
            className={formState.errors.password ? "error" : ""}
          />
          {formState.errors.password && (
            <span className="error-message">{formState.errors.password}</span>
          )}

          {/* Password strength indicator */}
          {formData.password && (
            <div className="password-strength">
              <div className="strength-bar">
                <div
                  className={`strength-fill ${
                    formData.password.length >= 8 &&
                    /(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/.test(formData.password)
                      ? "strong"
                      : formData.password.length >= 6
                      ? "medium"
                      : "weak"
                  }`}
                  style={{
                    width: `${Math.min(
                      (formData.password.length / 12) * 100,
                      100
                    )}%`,
                  }}
                />
              </div>
            </div>
          )}
        </div>

        {/* Confirm password */}
        <div className="form-group">
          <label htmlFor="confirmPassword">Confirm Password:</label>
          <input
            type="password"
            id="confirmPassword"
            value={formData.confirmPassword}
            onChange={handleFieldChange("confirmPassword")}
            className={formState.errors.confirmPassword ? "error" : ""}
          />
          {formState.errors.confirmPassword && (
            <span className="error-message">
              {formState.errors.confirmPassword}
            </span>
          )}
        </div>

        {/* Formatted phone input */}
        <div className="form-group">
```

```
          <label htmlFor="phone">Phone:</label>
          <input
            type="tel"
            id="phone"
            value={formData.phone || ""}
            onChange={handleFormattedChange("phone", formatters.phone)}
            placeholder="(555) 123-4567"
          />
        </div>

        {/* Multi-select checkboxes */}
        <div className="form-group">
          <fieldset>
            <legend>Skills:</legend>
            {["JavaScript", "React", "Node.js", "Python", "SQL"].map((skill) => (
              <label key={skill}>
                <input
                  type="checkbox"
                  value={skill}
                  checked={formData.skills.includes(skill)}
                  onChange={handleMultiSelectChange("skills")}
                />
                {skill}
              </label>
            ))}
          </fieldset>
          <p>Selected: {formData.skills.join(", ") || "None"}</p>
        </div>

        {/* Character-limited textarea */}
        <div className="form-group">
          <label htmlFor="bio">Bio (max 500 characters):</label>
          <textarea
            id="bio"
            value={formData.bio}
            onChange={(e) => {
              if (e.target.value.length <= 500) {
                handleFieldChange("bio")(e);
              }
            }}
            rows={4}
            placeholder="Tell us about yourself..."
          />
          <div className="character-count">
            {formData.bio.length}/500 characters
          </div>
        </div>

        {/* Newsletter checkbox */}
        <div className="form-group">
          <label>
            <input
              type="checkbox"
              checked={formData.newsletter}
```

```
            onChange={handleFieldChange("newsletter")}
          />
          Subscribe to newsletter
        </label>
      </div>

      {/* Submit button */}
      <button
        type="submit"
        disabled={
          formState.isSubmitting || Object.keys(formState.errors).length > 0
        }
        className="submit-button"
      >
        {formState.isSubmitting ? "Submitting..." : "Submit"}
      </button>

      {/* Form state debug */}
      <details className="form-debug">
        <summary>Form State Debug</summary>
        <pre>{JSON.stringify({ formData, formState }, null, 2)}</pre>
      </details>
    </form>
  );
}
```

🆓 Uncontrolled Components: Let the DOM Handle It

Understanding Uncontrolled Components

Uncontrolled components let the DOM maintain its own state. You access values using refs when needed (like on form submission).

```
function UncontrolledInputBasics() {
  // ✅ Refs to access DOM values
  const inputRef = useRef(null);
  const textareaRef = useRef(null);
  const selectRef = useRef(null);
  const checkboxRef = useRef(null);
  const fileRef = useRef(null);

  const [submittedData, setSubmittedData] = useState(null);

  const handleSubmit = (event) => {
    event.preventDefault();

    // ✅ Access values through refs
    const formData = {
      input: inputRef.current.value,
      textarea: textareaRef.current.value,
      select: selectRef.current.value,
```

```
      checkbox: checkboxRef.current.checked,
      file: fileRef.current.files[0]?.name || "No file selected",
    };

    setSubmittedData(formData);
    console.log("Uncontrolled form data:", formData);
  };

  const handleReset = () => {
    // ✅ Reset form using DOM methods
    inputRef.current.value = "";
    textareaRef.current.value = "";
    selectRef.current.value = "";
    checkboxRef.current.checked = false;
    fileRef.current.value = "";
    setSubmittedData(null);
  };

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

  return (
    <div className="uncontrolled-demo">
      <h3>Uncontrolled Components Demo</h3>

      <form onSubmit={handleSubmit}>
        {/* Text Input with default value */}
        <div className="form-group">
          <label htmlFor="uncontrolled-input">Text Input:</label>
          <input
            type="text"
            id="uncontrolled-input"
            ref={inputRef}
            defaultValue="Default text" // ✅ Use defaultValue, not value
            placeholder="Type something..."
          />
        </div>

        {/* Textarea */}
        <div className="form-group">
          <label htmlFor="uncontrolled-textarea">Textarea:</label>
          <textarea
            id="uncontrolled-textarea"
            ref={textareaRef}
            defaultValue="Default textarea content"
            rows={4}
          />
        </div>

        {/* Select */}
        <div className="form-group">
          <label htmlFor="uncontrolled-select">Select:</label>
          <select
```

```
            id="uncontrolled-select"
            ref={selectRef}
            defaultValue="option2" // ✅ Default selection
          >
            <option value="">Choose an option...</option>
            <option value="option1">Option 1</option>
            <option value="option2">Option 2</option>
            <option value="option3">Option 3</option>
          </select>
        </div>

        {/* Checkbox */}
        <div className="form-group">
          <label>
            <input
              type="checkbox"
              ref={checkboxRef}
              defaultChecked={true} // ✅ Use defaultChecked
            />
            Checkbox Option
          </label>
        </div>

        {/* File Input (always uncontrolled) */}
        <div className="form-group">
          <label htmlFor="file-input">File:</label>
          <input
            type="file"
            id="file-input"
            ref={fileRef}
            accept=".txt,.pdf,.doc,.docx"
          />
        </div>

        <div className="form-actions">
          <button type="submit">Submit</button>
          <button type="button" onClick={handleReset}>
            Reset
          </button>
          <button type="button" onClick={focusInput}>
            Focus Input
          </button>
        </div>
      </form>

      {/* Display submitted data */}
      {submittedData && (
        <div className="submitted-data">
          <h4>Submitted Data:</h4>
          <pre>{JSON.stringify(submittedData, null, 2)}</pre>
        </div>
      )}
    </div>
```

```
  );
}
```

Advanced Uncontrolled Patterns

```
function AdvancedUncontrolledPatterns() {
  const formRef = useRef(null);
  const [validationErrors, setValidationErrors] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  // ✅ Validation using FormData API
  const validateForm = (formData) => {
    const errors = {};

    const username = formData.get("username");
    const email = formData.get("email");
    const password = formData.get("password");
    const confirmPassword = formData.get("confirmPassword");
    const age = formData.get("age");

    if (!username || username.length < 3) {
      errors.username = "Username must be at least 3 characters";
    }

    if (!email || !/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
      errors.email = "Please enter a valid email";
    }

    if (!password || password.length < 8) {
      errors.password = "Password must be at least 8 characters";
    }

    if (password !== confirmPassword) {
      errors.confirmPassword = "Passwords do not match";
    }

    if (!age || parseInt(age) < 13 || parseInt(age) > 120) {
      errors.age = "Please enter a valid age (13-120)";
    }

    return errors;
  };

  const handleSubmit = async (event) => {
    event.preventDefault();

    // ✅ Use FormData API for uncontrolled forms
    const formData = new FormData(formRef.current);

    // Convert FormData to regular object for easier handling
    const data = Object.fromEntries(formData.entries());
```

```
    // Handle checkboxes (they won't appear in FormData if unchecked)
    data.newsletter = formData.has("newsletter");

    // Handle multiple checkboxes
    data.skills = formData.getAll("skills");

    console.log("Form data:", data);

    // Validate
    const errors = validateForm(formData);
    setValidationErrors(errors);

    if (Object.keys(errors).length === 0) {
      setIsSubmitting(true);

      try {
        // Simulate API call
        await new Promise((resolve) => setTimeout(resolve, 2000));

        alert("Form submitted successfully!");
        formRef.current.reset(); // ✅ Reset entire form
        setValidationErrors({});
      } catch (error) {
        console.error("Submission error:", error);
      } finally {
        setIsSubmitting(false);
      }
    }
  };

  // ✅ Real-time validation on blur
  const handleFieldBlur = (event) => {
    const { name, value } = event.target;
    const tempFormData = new FormData();
    tempFormData.set(name, value);

    // Get all current form values for cross-field validation
    const currentFormData = new FormData(formRef.current);
    const errors = validateForm(currentFormData);

    setValidationErrors((prev) => ({
      ...prev,
      [name]: errors[name],
    }));
  };

  return (
    <div className="advanced-uncontrolled-demo">
      <h3>Advanced Uncontrolled Form Patterns</h3>

      <form ref={formRef} onSubmit={handleSubmit}>
        {/* Username */}
        <div className="form-group">
```

```
            <label htmlFor="username">Username:</label>
            <input
              type="text"
              id="username"
              name="username" // ✅ Name attribute is crucial
              onBlur={handleFieldBlur}
              className={validationErrors.username ? "error" : ""}
            />
            {validationErrors.username && (
              <span className="error-message">{validationErrors.username}</span>
            )}
          </div>

          {/* Email */}
          <div className="form-group">
            <label htmlFor="email">Email:</label>
            <input
              type="email"
              id="email"
              name="email"
              onBlur={handleFieldBlur}
              className={validationErrors.email ? "error" : ""}
            />
            {validationErrors.email && (
              <span className="error-message">{validationErrors.email}</span>
            )}
          </div>

          {/* Password */}
          <div className="form-group">
            <label htmlFor="password">Password:</label>
            <input
              type="password"
              id="password"
              name="password"
              onBlur={handleFieldBlur}
              className={validationErrors.password ? "error" : ""}
            />
            {validationErrors.password && (
              <span className="error-message">{validationErrors.password}</span>
            )}
          </div>

          {/* Confirm Password */}
          <div className="form-group">
            <label htmlFor="confirmPassword">Confirm Password:</label>
            <input
              type="password"
              id="confirmPassword"
              name="confirmPassword"
              onBlur={handleFieldBlur}
              className={validationErrors.confirmPassword ? "error" : ""}
            />
            {validationErrors.confirmPassword && (
```

```
            <span className="error-message">
              {validationErrors.confirmPassword}
            </span>
          )}
        </div>

        {/* Age */}
        <div className="form-group">
          <label htmlFor="age">Age:</label>
          <input
            type="number"
            id="age"
            name="age"
            min="13"
            max="120"
            onBlur={handleFieldBlur}
            className={validationErrors.age ? "error" : ""}
          />
          {validationErrors.age && (
            <span className="error-message">{validationErrors.age}</span>
          )}
        </div>

        {/* Country Select */}
        <div className="form-group">
          <label htmlFor="country">Country:</label>
          <select id="country" name="country" defaultValue="">
            <option value="">Select a country</option>
            <option value="us">United States</option>
            <option value="ca">Canada</option>
            <option value="uk">United Kingdom</option>
            <option value="de">Germany</option>
          </select>
        </div>

        {/* Skills (Multiple Checkboxes) */}
        <div className="form-group">
          <fieldset>
            <legend>Skills:</legend>
            {["JavaScript", "React", "Node.js", "Python"].map((skill) => (
              <label key={skill}>
                <input
                  type="checkbox"
                  name="skills" // ✅ Same name for multiple values
                  value={skill}
                />
                {skill}
              </label>
            ))}
          </fieldset>
        </div>

        {/* Newsletter */}
        <div className="form-group">
```

```
            <label>
              <input type="checkbox" name="newsletter" value="yes" />
              Subscribe to newsletter
            </label>
          </div>

          {/* Bio */}
          <div className="form-group">
            <label htmlFor="bio">Bio:</label>
            <textarea
              id="bio"
              name="bio"
              rows={4}
              placeholder="Tell us about yourself..."
            />
          </div>

          {/* Hidden field */}
          <input type="hidden" name="formVersion" value="2.0" />

          <button type="submit" disabled={isSubmitting} className="submit-button">
            {isSubmitting ? "Submitting..." : "Submit"}
          </button>
        </form>
      </div>
    );
}
```

⚖️ Controlled vs Uncontrolled: When to Use Each

Decision Matrix

```
// 📊 Decision Guide:

// ✅ Use CONTROLLED when:
// - Real-time validation needed
// - Input formatting required
// - Conditional field visibility
// - Complex form interactions
// - State needs to be shared with other components
// - You need to prevent certain inputs

function ControlledUseCase() {
  const [creditCard, setCreditCard] = useState("");
  const [expiryDate, setExpiryDate] = useState("");
  const [showCVV, setShowCVV] = useState(false);

  // ✅ Real-time formatting
  const handleCreditCardChange = (event) => {
    let value = event.target.value.replace(/\D/g, ""); // Remove non-digits
    value = value.replace(/(\d{4})(?=\d)/g, "$1 "); // Add spaces
```

```
    if (value.length <= 19) {
      // Limit length
      setCreditCard(value);
    }
  };

  // ✅ Conditional field visibility
  useEffect(() => {
    setShowCVV(creditCard.length >= 19); // Show CVV when card number is complete
  }, [creditCard]);

  return (
    <div>
      <input
        type="text"
        value={creditCard}
        onChange={handleCreditCardChange}
        placeholder="1234 5678 9012 3456"
      />

      {showCVV && <input type="text" placeholder="CVV" maxLength={3} />}
    </div>
  );
}

// ✅ Use UNCONTROLLED when:
// - Simple forms with basic validation
// - Performance is critical (large forms)
// - Working with third-party form libraries
// - File uploads
// - You don't need real-time updates

function UncontrolledUseCase() {
  const formRef = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    const formData = new FormData(formRef.current);

    // ✅ Simple validation on submit
    const email = formData.get("email");
    if (!email.includes("@")) {
      alert("Please enter a valid email");
      return;
    }

    // Process form...
  };

  return (
    <form ref={formRef} onSubmit={handleSubmit}>
      <input type="email" name="email" required />
      <input type="file" name="resume" accept=".pdf,.doc" />
      <button type="submit">Submit</button>
```

```
      </form>
    );
  }
```

Performance Comparison

```
function PerformanceComparison() {
  const [renderCount, setRenderCount] = useState(0);
  const [controlledValue, setControlledValue] = useState("");
  const uncontrolledRef = useRef();

  // Track renders
  useEffect(() => {
    setRenderCount((prev) => prev + 1);
  });

  return (
    <div className="performance-demo">
      <h3>Performance Comparison</h3>
      <p>Component renders: {renderCount}</p>

      {/* ❌ Controlled - causes re-render on every keystroke */}
      <div className="controlled-section">
        <h4>Controlled Input (Re-renders on every change)</h4>
        <input
          type="text"
          value={controlledValue}
          onChange={(e) => setControlledValue(e.target.value)}
          placeholder="Type here - watch render count"
        />
        <p>Current value: {controlledValue}</p>
      </div>

      {/* ✅ Uncontrolled - no re-renders during typing */}
      <div className="uncontrolled-section">
        <h4>Uncontrolled Input (No re-renders during typing)</h4>
        <input
          ref={uncontrolledRef}
          type="text"
          placeholder="Type here - render count stays same"
        />
        <button
          onClick={() => {
            alert(`Value: ${uncontrolledRef.current.value}`);
          }}
        >
          Get Value
        </button>
      </div>
    </div>
```

```
  );
}
```

🔄 Hybrid Approaches

Mixing Controlled and Uncontrolled

```
function HybridFormApproach() {
  // ✅ Controlled for fields that need real-time updates
  const [searchQuery, setSearchQuery] = useState("");
  const [selectedCategory, setSelectedCategory] = useState("");

  // ✅ Uncontrolled for simple fields
  const formRef = useRef();

  // ✅ Filtered results based on controlled inputs
  const filteredResults = useMemo(() => {
    // Simulate filtering logic
    return mockData.filter(
      (item) =>
        item.name.toLowerCase().includes(searchQuery.toLowerCase()) &&
        (selectedCategory === "" || item.category === selectedCategory)
    );
  }, [searchQuery, selectedCategory]);

  const handleSubmit = (event) => {
    event.preventDefault();

    // Get uncontrolled values
    const formData = new FormData(formRef.current);

    // Combine with controlled values
    const submissionData = {
      searchQuery,
      selectedCategory,
      name: formData.get("name"),
      email: formData.get("email"),
      message: formData.get("message"),
    };

    console.log("Hybrid form submission:", submissionData);
  };

  return (
    <div className="hybrid-form">
      <h3>Hybrid Form Approach</h3>

      {/* Controlled section for real-time features */}
      <div className="search-section">
        <h4>Search & Filter (Controlled)</h4>
```

```
        <input
          type="text"
          value={searchQuery}
          onChange={(e) => setSearchQuery(e.target.value)}
          placeholder="Search..."
        />

        <select
          value={selectedCategory}
          onChange={(e) => setSelectedCategory(e.target.value)}
        >
          <option value="">All Categories</option>
          <option value="tech">Technology</option>
          <option value="design">Design</option>
          <option value="business">Business</option>
        </select>

        <div className="results">
          <p>Found {filteredResults.length} results</p>
          {/* Render filtered results */}
        </div>
      </div>

      {/* Uncontrolled section for simple form */}
      <form ref={formRef} onSubmit={handleSubmit} className="contact-form">
        <h4>Contact Form (Uncontrolled)</h4>

        <input type="text" name="name" placeholder="Your name" required />

        <input type="email" name="email" placeholder="Your email" required />

        <textarea name="message" placeholder="Your message" rows={4} required />

        <button type="submit">Send Message</button>
      </form>
    </div>
  );
}
```

Converting Between Patterns

```
function ConvertibleForm() {
  const [isControlled, setIsControlled] = useState(false);
  const [controlledValue, setControlledValue] = useState("");
  const uncontrolledRef = useRef();

  const toggleMode = () => {
    if (isControlled) {
      // Converting from controlled to uncontrolled
      // Set the DOM value to match current state
      uncontrolledRef.current.value = controlledValue;
```

```
    } else {
      // Converting from uncontrolled to controlled
      // Set state to match current DOM value
      setControlledValue(uncontrolledRef.current.value);
    }
    setIsControlled(!isControlled);
  };

  return (
    <div className="convertible-form">
      <h3>Convertible Form Pattern</h3>

      <button onClick={toggleMode}>
        Switch to {isControlled ? "Uncontrolled" : "Controlled"} Mode
      </button>

      <p>
        Current mode:{" "}
        <strong>{isControlled ? "Controlled" : "Uncontrolled"}</strong>
      </p>

      {isControlled ? (
        <input
          type="text"
          value={controlledValue}
          onChange={(e) => setControlledValue(e.target.value)}
          placeholder="Controlled input"
        />
      ) : (
        <input
          ref={uncontrolledRef}
          type="text"
          defaultValue={controlledValue}
          placeholder="Uncontrolled input"
        />
      )}

      <p>Current value: {isControlled ? controlledValue : "Check on submit"}</p>
    </div>
  );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Mixing value and defaultValue

```
// ❌ WRONG: Don't mix controlled and uncontrolled props
function MixedPropsError() {
  const [value, setValue] = useState("");

  return (
```

```
      <input
        value={value} // Controlled
        defaultValue="initial" // ❌ This will be ignored!
        onChange={(e) => setValue(e.target.value)}
      />
    );
}

// ✅ CORRECT: Choose one pattern
function CorrectPatterns() {
  const [value, setValue] = useState("initial"); // Start with initial value

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

      {/* OR Uncontrolled */}
      <input defaultValue="initial" ref={someRef} />
    </div>
  );
}
```

2. Forgetting to handle null/undefined values

```
// ❌ WRONG: Can cause "uncontrolled to controlled" warning
function NullValueError() {
  const [value, setValue] = useState(null); // ❌ null makes it uncontrolled
initially

  return (
    <input
      value={value} // ❌ null -> string causes warning
      onChange={(e) => setValue(e.target.value)}
    />
  );
}

// ✅ CORRECT: Always use string for controlled inputs
function CorrectNullHandling() {
  const [value, setValue] = useState(""); // ✅ Always string

  return (
    <input
      value={value || ""} // ✅ Ensure it's always a string
      onChange={(e) => setValue(e.target.value)}
    />
  );
}
```

3. Inefficient controlled components

```
// ❌ WRONG: Expensive operations in render
function ExpensiveControlled() {
  const [value, setValue] = useState("");

  // ❌ Expensive operation on every keystroke
  const expensiveValidation = (val) => {
    // Simulate expensive operation
    for (let i = 0; i < 1000000; i++) {
      // Heavy computation
    }
    return val.length > 5;
  };

  return (
    <div>
      <input value={value} onChange={(e) => setValue(e.target.value)} />
      <p>Valid: {expensiveValidation(value) ? "Yes" : "No"}</p>
    </div>
  );
}

// ✅ CORRECT: Debounce expensive operations
function OptimizedControlled() {
  const [value, setValue] = useState("");
  const [isValid, setIsValid] = useState(false);

  // ✅ Debounced validation
  const debouncedValidation = useCallback(
    debounce((val) => {
      // Expensive operation only after user stops typing
      const valid = expensiveValidation(val);
      setIsValid(valid);
    }, 300),
    []
  );

  useEffect(() => {
    debouncedValidation(value);
  }, [value, debouncedValidation]);

  return (
    <div>
      <input value={value} onChange={(e) => setValue(e.target.value)} />
      <p>Valid: {isValid ? "Yes" : "No"}</p>
    </div>
  );
}
```

🧪 Mini Challenges

Challenge 1: Smart Form Component

Build a form component that:

- Automatically chooses controlled vs uncontrolled based on props
- Supports both real-time and submit-time validation
- Handles file uploads properly
- Provides a clean API for both patterns

▶ 💡 Solution

```
function SmartForm({
  fields,
  onSubmit,
  realTimeValidation = false,
  initialValues = {},
}) {
  const [controlledValues, setControlledValues] = useState(initialValues);
  const [errors, setErrors] = useState({});
  const formRef = useRef();

  const isControlled =
    realTimeValidation || Object.keys(initialValues).length > 0;

  const validateField = (name, value, fieldConfig) => {
    if (!fieldConfig.validation) return null;

    for (const rule of fieldConfig.validation) {
      if (!rule.test(value)) {
        return rule.message;
      }
    }
    return null;
  };

  const handleControlledChange = (name, fieldConfig) => (event) => {
    const value =
      event.target.type === "checkbox"
        ? event.target.checked
        : event.target.value;

    setControlledValues((prev) => ({ ...prev, [name]: value }));

    if (realTimeValidation) {
      const error = validateField(name, value, fieldConfig);
      setErrors((prev) => ({ ...prev, [name]: error }));
    }
  };

  const handleSubmit = (event) => {
    event.preventDefault();
```

```
    let formData;
    if (isControlled) {
      formData = controlledValues;
    } else {
      const formDataObj = new FormData(formRef.current);
      formData = Object.fromEntries(formDataObj.entries());
    }

    // Validate all fields
    const newErrors = {};
    fields.forEach((field) => {
      const error = validateField(field.name, formData[field.name], field);
      if (error) newErrors[field.name] = error;
    });

    setErrors(newErrors);

    if (Object.keys(newErrors).length === 0) {
      onSubmit(formData);
    }
  };

  const renderField = (field) => {
    const commonProps = {
      id: field.name,
      name: field.name,
      type: field.type || "text",
      placeholder: field.placeholder,
      required: field.required,
    };

    if (isControlled) {
      return (
        <input
          {...commonProps}
          value={controlledValues[field.name] || ""}
          onChange={handleControlledChange(field.name, field)}
        />
      );
    } else {
      return (
        <input
          {...commonProps}
          defaultValue={initialValues[field.name] || ""}
        />
      );
    }
  };

  return (
    <form ref={formRef} onSubmit={handleSubmit}>
      <h3>Smart Form ({isControlled ? "Controlled" : "Uncontrolled"})</h3>

      {fields.map((field) => (
```

```
          <div key={field.name} className="form-group">
            <label htmlFor={field.name}>{field.label}:</label>
            {renderField(field)}
            {errors[field.name] && (
              <span className="error">{errors[field.name]}</span>
            )}
          </div>
        ))}

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

  // Usage examples
  function SmartFormExamples() {
    const fields = [
      {
        name: "username",
        label: "Username",
        validation: [{ test: (v) => v.length >= 3, message: "Min 3 characters" }],
      },
      {
        name: "email",
        label: "Email",
        type: "email",
        validation: [{ test: (v) => v.includes("@"), message: "Invalid email" }],
      },
    ];

    return (
      <div>
        {/* Uncontrolled form */}
        <SmartForm
          fields={fields}
          onSubmit={(data) => console.log("Uncontrolled:", data)}
        />

        {/* Controlled form with real-time validation */}
        <SmartForm
          fields={fields}
          realTimeValidation={true}
          initialValues={{ username: "john" }}
          onSubmit={(data) => console.log("Controlled:", data)}
        />
      </div>
    );
  }
```

Challenge 2: Form State Manager Hook

Create a custom hook that:

- Manages both controlled and uncontrolled forms
- Provides validation utilities
- Handles form reset and dirty state
- Supports field-level and form-level validation

▶ 💡 Solution

```
function useFormState(initialValues = {}, validationRules = {}) {
  const [values, setValues] = useState(initialValues);
  const [errors, setErrors] = useState({});
  const [touched, setTouched] = useState({});
  const [isDirty, setIsDirty] = useState(false);
  const [isSubmitting, setIsSubmitting] = useState(false);

  const validateField = useCallback(
    (name, value) => {
      const rules = validationRules[name];
      if (!rules) return null;

      for (const rule of rules) {
        if (typeof rule === "function") {
          const result = rule(value, values);
          if (result !== true) return result;
        } else if (rule.test && !rule.test(value, values)) {
          return rule.message;
        }
      }
      return null;
    },
    [validationRules, values]
  );

  const validateForm = useCallback(() => {
    const newErrors = {};
    Object.keys(validationRules).forEach((field) => {
      const error = validateField(field, values[field]);
      if (error) newErrors[field] = error;
    });
    return newErrors;
  }, [validateField, values, validationRules]);

  const setValue = useCallback(
    (name, value) => {
      setValues((prev) => ({ ...prev, [name]: value }));
      setIsDirty(true);

      // Clear error when user starts typing
      if (errors[name]) {
        setErrors((prev) => ({ ...prev, [name]: null }));
      }
    },
    [errors]
```

```
  );

  const setFieldTouched = useCallback((name, isTouched = true) => {
    setTouched((prev) => ({ ...prev, [name]: isTouched }));
  }, []);

  const setFieldError = useCallback((name, error) => {
    setErrors((prev) => ({ ...prev, [name]: error }));
  }, []);

  const handleChange = useCallback(
    (name) => (event) => {
      const { type, value, checked } = event.target;
      const newValue = type === "checkbox" ? checked : value;
      setValue(name, newValue);
    },
    [setValue]
  );

  const handleBlur = useCallback(
    (name) => (event) => {
      setFieldTouched(name, true);
      const error = validateField(name, event.target.value);
      setFieldError(name, error);
    },
    [validateField, setFieldTouched, setFieldError]
  );

  const handleSubmit = useCallback(
    (onSubmit) => async (event) => {
      event.preventDefault();

      setIsSubmitting(true);

      // Mark all fields as touched
      const allTouched = Object.keys(validationRules).reduce((acc, field) => {
        acc[field] = true;
        return acc;
      }, {});
      setTouched(allTouched);

      // Validate entire form
      const formErrors = validateForm();
      setErrors(formErrors);

      if (Object.keys(formErrors).length === 0) {
        try {
          await onSubmit(values);
        } catch (error) {
          console.error("Form submission error:", error);
        }
      }

      setIsSubmitting(false);
```

```
    },
    [values, validateForm, validationRules]
  );

  const reset = useCallback(() => {
    setValues(initialValues);
    setErrors({});
    setTouched({});
    setIsDirty(false);
    setIsSubmitting(false);
  }, [initialValues]);

  const getFieldProps = useCallback(
    (name) => ({
      value: values[name] || "",
      onChange: handleChange(name),
      onBlur: handleBlur(name),
      error: touched[name] ? errors[name] : null,
    }),
    [values, handleChange, handleBlur, touched, errors]
  );

  return {
    values,
    errors,
    touched,
    isDirty,
    isSubmitting,
    setValue,
    setFieldTouched,
    setFieldError,
    handleChange,
    handleBlur,
    handleSubmit,
    reset,
    validateForm,
    getFieldProps,
    isValid: Object.keys(errors).length === 0,
  };
}

// Usage example
function FormWithCustomHook() {
  const validationRules = {
    username: [
      {
        test: (v) => v.length >= 3,
        message: "Username must be at least 3 characters",
      },
      {
        test: (v) => /^[a-zA-Z0-9_]+$/.test(v),
        message: "Username can only contain letters, numbers, and underscores",
      },
    ],
```

```
    email: [
      {
        test: (v) => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(v),
        message: "Please enter a valid email",
      },
    ],
    password: [
      {
        test: (v) => v.length >= 8,
        message: "Password must be at least 8 characters",
      },
    ],
    confirmPassword: [
      (value, allValues) => {
        return value === allValues.password || "Passwords do not match";
      },
    ],
  };

  const form = useFormState(
    { username: "", email: "", password: "", confirmPassword: "" },
    validationRules
  );

  const onSubmit = async (data) => {
    console.log("Form submitted:", data);
    // Simulate API call
    await new Promise((resolve) => setTimeout(resolve, 1000));
    alert("Form submitted successfully!");
    form.reset();
  };

  return (
    <form onSubmit={form.handleSubmit(onSubmit)}>
      <h3>Form with Custom Hook</h3>

      <div className="form-group">
        <label>Username:</label>
        <input
          type="text"
          {...form.getFieldProps("username")}
          className={form.getFieldProps("username").error ? "error" : ""}
        />
        {form.getFieldProps("username").error && (
          <span className="error-message">
            {form.getFieldProps("username").error}
          </span>
        )}
      </div>

      <div className="form-group">
        <label>Email:</label>
        <input
          type="email"
```

```
            {...form.getFieldProps("email")}
            className={form.getFieldProps("email").error ? "error" : ""}
          />
          {form.getFieldProps("email").error && (
            <span className="error-message">
              {form.getFieldProps("email").error}
            </span>
          )}
        </div>

        <div className="form-group">
          <label>Password:</label>
          <input
            type="password"
            {...form.getFieldProps("password")}
            className={form.getFieldProps("password").error ? "error" : ""}
          />
          {form.getFieldProps("password").error && (
            <span className="error-message">
              {form.getFieldProps("password").error}
            </span>
          )}
        </div>

        <div className="form-group">
          <label>Confirm Password:</label>
          <input
            type="password"
            {...form.getFieldProps("confirmPassword")}
            className={form.getFieldProps("confirmPassword").error ? "error" : ""}
          />
          {form.getFieldProps("confirmPassword").error && (
            <span className="error-message">
              {form.getFieldProps("confirmPassword").error}
            </span>
          )}
        </div>

        <div className="form-actions">
          <button type="submit" disabled={!form.isValid || form.isSubmitting}>
            {form.isSubmitting ? "Submitting..." : "Submit"}
          </button>
          <button type="button" onClick={form.reset}>
            Reset
          </button>
        </div>

        <div className="form-state">
          <p>Dirty: {form.isDirty ? "Yes" : "No"}</p>
          <p>Valid: {form.isValid ? "Yes" : "No"}</p>
          <p>Submitting: {form.isSubmitting ? "Yes" : "No"}</p>
        </div>
      </form>
```

```
  );
}
```

🎯 When and Why: Decision Framework

Quick Decision Tree

```
🤔 Should I use Controlled or Uncontrolled?

├── Do you need real-time validation?
│   ├── Yes → Controlled ✅
│   └── No → Continue...
│
├── Do you need to format input as user types?
│   ├── Yes → Controlled ✅
│   └── No → Continue...
│
├── Do you need conditional field visibility?
│   ├── Yes → Controlled ✅
│   └── No → Continue...
│
├── Is it a file input?
│   ├── Yes → Uncontrolled ✅ (files are always uncontrolled)
│   └── No → Continue...
│
├── Is performance critical (large forms)?
│   ├── Yes → Uncontrolled ✅
│   └── No → Continue...
│
├── Do you need to share form state with other components?
│   ├── Yes → Controlled ✅
│   └── No → Continue...
│
└── Simple form with submit-only validation?
    └── Yes → Uncontrolled ✅
```

Performance Guidelines

```
// 📊 Performance Considerations:

// ✅ Controlled: Good for
// - Small to medium forms (< 20 fields)
// - Forms with complex interactions
// - Real-time features

// ✅ Uncontrolled: Good for
// - Large forms (> 20 fields)
// - Simple forms
// - Performance-critical applications
```

```
// - Third-party form libraries

// 🎯 Hybrid: Best of both worlds
// - Controlled for interactive fields
// - Uncontrolled for simple fields
// - Use refs for occasional access
```

🎤 Interview Insights

Common Interview Questions

1. **"What's the difference between controlled and uncontrolled components?"**
    - Controlled: React state manages the value
    - Uncontrolled: DOM manages the value, accessed via refs
    - Show code examples of both patterns
2. **"When would you choose controlled over uncontrolled?"**
    - Real-time validation
    - Input formatting
    - Conditional rendering
    - State sharing between components
3. **"What's the 'uncontrolled to controlled' warning?"**
    - Happens when value changes from null/undefined to string
    - Always initialize controlled inputs with empty string
    - Show how to fix the warning
4. **"How do you handle file uploads in React?"**
    - File inputs are always uncontrolled
    - Use refs to access FileList
    - Show proper file handling patterns
5. **"What are the performance implications?"**
    - Controlled: Re-renders on every change
    - Uncontrolled: No re-renders during typing
    - When to optimize with debouncing

Code Review Red Flags

```
// 🚩 Red Flags in Interviews:

// ❌ Mixing controlled and uncontrolled
<input value={value} defaultValue="bad" />

// ❌ Not handling null values
```

```
const [value, setValue] = useState(null);
<input value={value} /> // Warning!

// ❌ Expensive operations in render
<input onChange={() => expensiveValidation()} />

// ❌ Not using proper form submission
<button onClick={() => getValue()}> // Should use form onSubmit

// ❌ Forgetting name attributes for uncontrolled
<input ref={ref} /> // Missing name for FormData
```

Best Practices to Mention

```
// ✅ Interview-worthy patterns:

// 1. Proper controlled initialization
const [value, setValue] = useState(""); // Always string

// 2. Debounced validation
const debouncedValidate = useCallback(debounce(validateField, 300), []);

// 3. Proper form submission
const handleSubmit = (event) => {
  event.preventDefault();
  const formData = new FormData(event.target);
};

// 4. Hybrid approach for performance
const useHybridForm = () => {
  // Controlled for interactive fields
  // Uncontrolled for simple fields
};

// 5. Proper error handling
const [errors, setErrors] = useState({});
const validateField = (name, value) => {
  // Return error or null
};
```

🎯 Key Takeaways

Decision Matrix Summary

Feature	Controlled	Uncontrolled
Real-time validation	✅ Excellent	❌ Limited
Input formatting	✅ Excellent	❌ Not possible

Feature	Controlled	Uncontrolled
Performance	⚠ Re-renders	✅ No re-renders
Simplicity	⚠ More code	✅ Less code
File uploads	❌ Not possible	✅ Required
Form libraries	⚠ Some support	✅ Full support
Testing	✅ Easy to test	⚠ Requires DOM
Accessibility	✅ Full control	✅ Native behavior

Mental Model

```
// 🤔 Think of it this way:

// Controlled = "React is the boss"
// - React state controls everything
// - Every change goes through React
// - More power, more responsibility

// Uncontrolled = "DOM is the boss"
// - DOM handles its own state
// - React checks in when needed
// - Less power, less responsibility

// Hybrid = "Shared responsibility"
// - React controls what matters
// - DOM handles the rest
// - Best of both worlds
```

Production Tips

1. **Start with uncontrolled** for simple forms
2. **Upgrade to controlled** when you need real-time features
3. **Use hybrid approach** for complex forms
4. **Always validate on submit** regardless of pattern
5. **Consider performance** for large forms
6. **Test both patterns** in your component library

Next up: useEffect Dependencies & Cleanup - Master React's most powerful hook for side effects and lifecycle management.

Previous: Handling DOM Events

💡 Pro tip: In interviews, always explain your choice between controlled and uncontrolled. Show that you understand the trade-offs and can make informed decisions based on requirements.

⚡ useEffect: Dependencies & Cleanup Mastery

> **Master React's most powerful hook: When effects run, how dependencies work, and why cleanup prevents memory leaks**

🎯 What You'll Learn

- How useEffect works under the hood
- The dependency array and its impact on performance
- Common dependency pitfalls and how to avoid them
- Cleanup functions and preventing memory leaks
- Advanced patterns: conditional effects, custom hooks
- Performance optimization strategies
- Real-world scenarios and best practices

⚡ Understanding useEffect Fundamentals

The Effect Lifecycle

```
function EffectLifecycleDemo() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("");

  console.log("🔄 Component render");

  // 🎯 Effect runs AFTER every render (no dependency array)
  useEffect(() => {
    console.log("🚀 Effect runs after render");
    document.title = `Count: ${count}`;

    // 🧹 Cleanup function (optional)
    return () => {
      console.log("🧹 Cleanup from previous effect");
    };
  });

  // 🎯 Effect runs only on mount (empty dependency array)
  useEffect(() => {
    console.log("🏁 Component mounted - runs once");

    return () => {
      console.log("💀 Component will unmount");
    };
  }, []); // ✅ Empty dependency array = run once

  // 🎯 Effect runs when count changes (specific dependency)
  useEffect(() => {
    console.log(`📊 Count changed to: ${count}`);

    if (count > 0) {
```

```
      const timer = setTimeout(() => {
        console.log(`⏰ Timer fired for count: ${count}`);
      }, 1000);

      // 🧹 Cleanup timer to prevent memory leaks
      return () => {
        console.log(`🧹 Cleaning up timer for count: ${count}`);
        clearTimeout(timer);
      };
    }
  }, [count]); // ✅ Only runs when count changes

  // 🎯 Effect with multiple dependencies
  useEffect(() => {
    console.log(`👤 User info: ${name}, Count: ${count}`);

    // Simulate API call
    if (name && count > 0) {
      console.log("📡 Making API call with user data");
    }
  }, [name, count]); // ✅ Runs when either name OR count changes

  return (
    <div className="effect-lifecycle-demo">
      <h3>useEffect Lifecycle Demo</h3>
      <p>Open console to see effect execution order</p>

      <div className="controls">
        <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>

        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          placeholder="Enter your name"
        />
      </div>

      <div className="info">
        <p>Current count: {count}</p>
        <p>Current name: {name || "No name"}</p>
        <p>Document title should show count</p>
      </div>
    </div>
  );
}
```

Effect Execution Order

```
function EffectExecutionOrder() {
  const [toggle, setToggle] = useState(false);
```

```
  console.log("1️⃣ Render phase");

  // Multiple effects to show execution order
  useEffect(() => {
    console.log("2️⃣ First effect (no deps)");
    return () => console.log("🧹 First effect cleanup");
  });

  useEffect(() => {
    console.log("3️⃣ Second effect (empty deps)");
    return () => console.log("🧹 Second effect cleanup");
  }, []);

  useEffect(() => {
    console.log("4️⃣ Third effect (toggle dep)");
    return () => console.log("🧹 Third effect cleanup");
  }, [toggle]);

  console.log("5️⃣ Render complete, effects will run next");

  return (
    <div>
      <h3>Effect Execution Order</h3>
      <p>Check console for execution sequence</p>
      <button onClick={() => setToggle(!toggle)}>
        Toggle: {toggle.toString()}
      </button>

      <div className="execution-info">
        <h4>Execution Order:</h4>
        <ol>
          <li>Component renders</li>
          <li>DOM is updated</li>
          <li>Cleanup functions run (for changed effects)</li>
          <li>New effects run (in order they're defined)</li>
        </ol>
      </div>
    </div>
  );
}
```

🎯 Dependency Array Deep Dive

Understanding Dependencies

```
function DependencyArrayExamples() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState({ name: "John", age: 25 });
  const [items, setItems] = useState(["apple", "banana"]);
```

```
// ❌ WRONG: Missing dependencies
useEffect(() => {
  console.log(`Count is ${count}`); // Uses count but doesn't depend on it
  document.title = `User: ${user.name}`; // Uses user but doesn't depend on it
}, []); // ❌ Missing count and user dependencies

// ✅ CORRECT: All dependencies included
useEffect(() => {
  console.log(`Count is ${count}`);
  document.title = `User: ${user.name}, Count: ${count}`;
}, [count, user.name]); // ✅ Includes all used values

// ⚠️ CAREFUL: Object dependencies
useEffect(() => {
  console.log("User object changed:", user);
  // This will run on every render if user object is recreated
}, [user]); // ⚠️ Object reference comparison

// ✅ BETTER: Specific object properties
useEffect(() => {
  console.log("User name or age changed");
}, [user.name, user.age]); // ✅ Only specific properties

// ❌ WRONG: Array dependencies without proper comparison
useEffect(() => {
  console.log("Items changed:", items);
}, [items]); // ❌ Array reference comparison

// ✅ BETTER: Use useMemo for stable references
const stableItems = useMemo(() => items, [items.join(",")]);
useEffect(() => {
  console.log("Items actually changed:", stableItems);
}, [stableItems]);

// 🎯 Function dependencies
const expensiveCalculation = useCallback((num) => {
  console.log("Expensive calculation running");
  return num * 2;
}, []); // ✅ Memoized function

useEffect(() => {
  const result = expensiveCalculation(count);
  console.log("Calculation result:", result);
}, [count, expensiveCalculation]); // ✅ Include function dependency

const updateUser = (field, value) => {
  setUser((prev) => ({ ...prev, [field]: value }));
};

const addItem = () => {
  setItems((prev) => [...prev, `item-${Date.now()}`]);
};

return (
```

```
      <div className="dependency-examples">
        <h3>Dependency Array Examples</h3>

        <div className="controls">
          <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>

          <button onClick={() => updateUser("name", `User-${Date.now()}`)}>
            Change Name
          </button>

          <button
            onClick={() => updateUser("age", Math.floor(Math.random() * 50) + 20)}
          >
            Change Age
          </button>

          <button onClick={addItem}>Add Item</button>
        </div>

        <div className="state-display">
          <p>Count: {count}</p>
          <p>User: {JSON.stringify(user)}</p>
          <p>Items: {items.join(", ")}</p>
        </div>
      </div>
    );
  }
```

Dependency Array Patterns

```
  function DependencyPatterns() {
    const [searchTerm, setSearchTerm] = useState("");
    const [results, setResults] = useState([]);
    const [loading, setLoading] = useState(false);
    const [user, setUser] = useState({ id: 1, preferences: { theme: "dark" } });

    // 🎯 Pattern 1: Debounced API calls
    useEffect(() => {
      if (!searchTerm.trim()) {
        setResults([]);
        return;
      }

      setLoading(true);

      const timeoutId = setTimeout(async () => {
        try {
          // Simulate API call
          console.log(`Searching for: ${searchTerm}`);
          const mockResults = [
            `Result 1 for ${searchTerm}`,
```

```
          `Result 2 for ${searchTerm}`,
          `Result 3 for ${searchTerm}`,
        ];
        setResults(mockResults);
      } catch (error) {
        console.error("Search failed:", error);
        setResults([]);
      } finally {
        setLoading(false);
      }
    }, 500); // 500ms debounce

    // 🧹 Cleanup: Cancel previous search
    return () => {
      clearTimeout(timeoutId);
      setLoading(false);
    };
  }, [searchTerm]); // ✅ Only depends on searchTerm

  // 🎯 Pattern 2: Nested object dependencies
  useEffect(() => {
    console.log("User theme changed:", user.preferences.theme);
    document.body.className = `theme-${user.preferences.theme}`;

    return () => {
      document.body.className = ""; // Cleanup theme
    };
  }, [user.preferences.theme]); // ✅ Specific nested property

  // 🎯 Pattern 3: Conditional effects
  useEffect(() => {
    // Only run effect if user is logged in
    if (user.id) {
      console.log("Setting up user session");

      const sessionTimer = setInterval(() => {
        console.log("Refreshing user session");
      }, 60000); // Refresh every minute

      return () => {
        console.log("Cleaning up user session");
        clearInterval(sessionTimer);
      };
    }
  }, [user.id]); // ✅ Only runs when user.id changes

  // 🎯 Pattern 4: Effect with external dependencies
  useEffect(() => {
    const handleResize = () => {
      console.log("Window resized:", window.innerWidth);
    };

    window.addEventListener("resize", handleResize);
```

```
    // 🧹 Always cleanup event listeners
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, []); // ✅ No dependencies - setup once

  // 🎯 Pattern 5: Effect with function dependencies
  const logUserActivity = useCallback(
    (activity) => {
      console.log(`User ${user.id} performed: ${activity}`);
    },
    [user.id]
  ); // ✅ Function depends on user.id

  useEffect(() => {
    logUserActivity("page_view");
  }, [logUserActivity]); // ✅ Include function dependency

  const updateUserTheme = (theme) => {
    setUser((prev) => ({
      ...prev,
      preferences: {
        ...prev.preferences,
        theme,
      },
    }));
  };

  return (
    <div className="dependency-patterns">
      <h3>Advanced Dependency Patterns</h3>

      <div className="search-section">
        <h4>Debounced Search</h4>
        <input
          type="text"
          value={searchTerm}
          onChange={(e) => setSearchTerm(e.target.value)}
          placeholder="Search..."
        />
        {loading && <p>Searching...</p>}
        <ul>
          {results.map((result, index) => (
            <li key={index}>{result}</li>
          ))}
        </ul>
      </div>

      <div className="theme-section">
        <h4>Theme Control</h4>
        <p>Current theme: {user.preferences.theme}</p>
        <button onClick={() => updateUserTheme("light")}>Light Theme</button>
        <button onClick={() => updateUserTheme("dark")}>Dark Theme</button>
        <button onClick={() => updateUserTheme("auto")}>Auto Theme</button>
```

```
        </div>

        <div className="user-section">
          <h4>User Session</h4>
          <p>User ID: {user.id}</p>
          <button
            onClick={() => setUser((prev) => ({ ...prev, id: prev.id + 1 }))}
          >
            Switch User
          </button>
          <button onClick={() => setUser((prev) => ({ ...prev, id: null }))}>
            Logout
          </button>
        </div>
      </div>
    );
  }
```

🧹 Cleanup Functions: Preventing Memory Leaks

Essential Cleanup Patterns

```
function CleanupPatterns() {
  const [isActive, setIsActive] = useState(false);
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [data, setData] = useState(null);

  // 🎯 Pattern 1: Timer cleanup
  useEffect(() => {
    if (!isActive) return;

    console.log("Starting timer");
    const intervalId = setInterval(() => {
      console.log("Timer tick:", new Date().toLocaleTimeString());
    }, 1000);

    // 🧹 CRITICAL: Clear timer to prevent memory leak
    return () => {
      console.log("Cleaning up timer");
      clearInterval(intervalId);
    };
  }, [isActive]);

  // 🎯 Pattern 2: Event listener cleanup
  useEffect(() => {
    const handleMouseMove = (event) => {
      setPosition({ x: event.clientX, y: event.clientY });
    };

    const handleKeyPress = (event) => {
      if (event.key === "Escape") {
```

```
      setIsActive(false);
    }
  };

  if (isActive) {
    console.log("Adding event listeners");
    document.addEventListener("mousemove", handleMouseMove);
    document.addEventListener("keydown", handleKeyPress);
  }

  // 🧹 CRITICAL: Remove event listeners
  return () => {
    console.log("Removing event listeners");
    document.removeEventListener("mousemove", handleMouseMove);
    document.removeEventListener("keydown", handleKeyPress);
  };
}, [isActive]);

// 🎯 Pattern 3: AbortController for fetch requests
useEffect(() => {
  if (!isActive) return;

  const abortController = new AbortController();

  const fetchData = async () => {
    try {
      console.log("Starting fetch request");

      // Simulate API call with abort signal
      const response = await fetch("/api/data", {
        signal: abortController.signal,
      });

      if (!response.ok) throw new Error("Fetch failed");

      const result = await response.json();
      setData(result);
      console.log("Fetch completed");
    } catch (error) {
      if (error.name === "AbortError") {
        console.log("Fetch was aborted");
      } else {
        console.error("Fetch error:", error);
      }
    }
  };

  fetchData();

  // 🧹 CRITICAL: Abort ongoing requests
  return () => {
    console.log("Aborting fetch request");
    abortController.abort();
  };
```

```
  }, [isActive]);

  // 🎯 Pattern 4: WebSocket cleanup
  useEffect(() => {
    if (!isActive) return;

    console.log("Connecting to WebSocket");
    const ws = new WebSocket("wss://echo.websocket.org");

    ws.onopen = () => {
      console.log("WebSocket connected");
      ws.send("Hello WebSocket!");
    };

    ws.onmessage = (event) => {
      console.log("WebSocket message:", event.data);
    };

    ws.onerror = (error) => {
      console.error("WebSocket error:", error);
    };

    ws.onclose = () => {
      console.log("WebSocket disconnected");
    };

    // 🧹 CRITICAL: Close WebSocket connection
    return () => {
      console.log("Closing WebSocket");
      if (ws.readyState === WebSocket.OPEN) {
        ws.close();
      }
    };
  }, [isActive]);

  // 🎯 Pattern 5: Subscription cleanup
  useEffect(() => {
    if (!isActive) return;

    // Simulate subscription (like Redux store)
    const subscription = {
      unsubscribe: () => console.log("Unsubscribed from store"),
    };

    console.log("Subscribing to store");

    // 🧹 CRITICAL: Unsubscribe to prevent memory leaks
    return () => {
      subscription.unsubscribe();
    };
  }, [isActive]);

  return (
    <div className="cleanup-patterns">
```

```
      <h3>Cleanup Patterns Demo</h3>

      <div className="controls">
        <button
          onClick={() => setIsActive(!isActive)}
          className={isActive ? "active" : ""}
        >
          {isActive ? "Stop" : "Start"} Effects
        </button>
      </div>

      <div className="status">
        <p>Effects active: {isActive ? "Yes" : "No"}</p>
        <p>
          Mouse position: {position.x}, {position.y}
        </p>
        <p>Data: {data ? JSON.stringify(data) : "No data"}</p>
        <p>Press Escape to deactivate</p>
      </div>

      <div className="cleanup-info">
        <h4>What gets cleaned up:</h4>
        <ul>
          <li>✅ Timers (setInterval, setTimeout)</li>
          <li>✅ Event listeners</li>
          <li>✅ Fetch requests (AbortController)</li>
          <li>✅ WebSocket connections</li>
          <li>✅ Subscriptions</li>
          <li>✅ Animation frames</li>
          <li>✅ DOM modifications</li>
        </ul>
      </div>
    </div>
  );
}
```

Memory Leak Prevention

```
function MemoryLeakPrevention() {
  const [componentCount, setComponentCount] = useState(1);
  const [showLeakyComponent, setShowLeakyComponent] = useState(true);

  return (
    <div className="memory-leak-demo">
      <h3>Memory Leak Prevention</h3>

      <div className="controls">
        <button onClick={() => setComponentCount((c) => c + 1)}>
          Create Component #{componentCount + 1}
        </button>
```

```
        <button onClick={() => setShowLeakyComponent(!showLeakyComponent)}>
          {showLeakyComponent ? "Hide" : "Show"} Leaky Component
        </button>
      </div>

      {/* Show multiple instances to demonstrate leaks */}
      {Array.from({ length: componentCount }, (_, i) => (
        <div key={i}>
          <h4>Component Instance #{i + 1}</h4>
          {showLeakyComponent ? <LeakyComponent /> : <CleanComponent />}
        </div>
      ))}
    </div>
  );
}

// ❌ BAD: Component with memory leaks
function LeakyComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // ❌ Timer without cleanup
    setInterval(() => {
      setCount((c) => c + 1);
    }, 1000);

    // ❌ Event listener without cleanup
    const handleClick = () => console.log("Clicked");
    document.addEventListener("click", handleClick);

    // ❌ No cleanup function!
  }, []);

  return (
    <div className="leaky-component">
      <p>❌ Leaky Component - Count: {count}</p>
      <p>This component creates memory leaks!</p>
    </div>
  );
}

// ✅ GOOD: Component with proper cleanup
function CleanComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // ✅ Timer with cleanup
    const intervalId = setInterval(() => {
      setCount((c) => c + 1);
    }, 1000);

    // ✅ Event listener with cleanup
    const handleClick = () => console.log("Clicked");
    document.addEventListener("click", handleClick);
```

```
    // ✅ Proper cleanup function
    return () => {
      clearInterval(intervalId);
      document.removeEventListener("click", handleClick);
    };
  }, []);

  return (
    <div className="clean-component">
      <p>✅ Clean Component - Count: {count}</p>
      <p>This component properly cleans up!</p>
    </div>
  );
}
```

🚀 Advanced useEffect Patterns

Custom Hook Patterns

```
// 🎯 Custom hook for API calls
function useApi(url, dependencies = []) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (!url) return;

    const abortController = new AbortController();

    const fetchData = async () => {
      setLoading(true);
      setError(null);

      try {
        const response = await fetch(url, {
          signal: abortController.signal,
        });

        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }

        const result = await response.json();
        setData(result);
      } catch (err) {
        if (err.name !== "AbortError") {
          setError(err.message);
        }
      } finally {
```

```
        setLoading(false);
      }
    };

    fetchData();

    return () => {
      abortController.abort();
    };
  }, [url, ...dependencies]);

  return { data, loading, error };
}

// 🎯 Custom hook for local storage
function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(`Error reading localStorage key "${key}":`, error);
      return initialValue;
    }
  });

  const setValue = useCallback(
    (value) => {
      try {
        const valueToStore =
          value instanceof Function ? value(storedValue) : value;
        setStoredValue(valueToStore);
        window.localStorage.setItem(key, JSON.stringify(valueToStore));
      } catch (error) {
        console.error(`Error setting localStorage key "${key}":`, error);
      }
    },
    [key, storedValue]
  );

  // 🎯 Listen for storage changes from other tabs
  useEffect(() => {
    const handleStorageChange = (e) => {
      if (e.key === key && e.newValue !== null) {
        try {
          setStoredValue(JSON.parse(e.newValue));
        } catch (error) {
          console.error(`Error parsing localStorage key "${key}":`, error);
        }
      }
    };

    window.addEventListener("storage", handleStorageChange);
```

```
    return () => {
      window.removeEventListener("storage", handleStorageChange);
    };
  }, [key]);

  return [storedValue, setValue];
}

// 🔗 Custom hook for window size
function useWindowSize() {
  const [windowSize, setWindowSize] = useState({
    width: undefined,
    height: undefined,
  });

  useEffect(() => {
    const handleResize = () => {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    };

    // Set initial size
    handleResize();

    window.addEventListener("resize", handleResize);

    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, []);

  return windowSize;
}

// 🔗 Custom hook for debounced values
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

// Usage examples
```

```
function CustomHookExamples() {
  const [searchTerm, setSearchTerm] = useState("");
  const [userId, setUserId] = useState(1);

  // 🎯 Using custom hooks
  const debouncedSearchTerm = useDebounce(searchTerm, 500);
  const {
    data: userData,
    loading,
    error,
  } = useApi(userId ? `/api/users/${userId}` : null, [userId]);
  const [preferences, setPreferences] = useLocalStorage("userPreferences", {
    theme: "light",
    language: "en",
  });
  const windowSize = useWindowSize();

  // 🎯 Effect that depends on debounced value
  useEffect(() => {
    if (debouncedSearchTerm) {
      console.log("Searching for:", debouncedSearchTerm);
      // Perform search...
    }
  }, [debouncedSearchTerm]);

  return (
    <div className="custom-hook-examples">
      <h3>Custom Hook Examples</h3>

      <div className="search-section">
        <h4>Debounced Search</h4>
        <input
          type="text"
          value={searchTerm}
          onChange={(e) => setSearchTerm(e.target.value)}
          placeholder="Search (debounced)..."
        />
        <p>Search term: {searchTerm}</p>
        <p>Debounced: {debouncedSearchTerm}</p>
      </div>

      <div className="api-section">
        <h4>API Data</h4>
        <button onClick={() => setUserId(userId + 1)}>
          Load User {userId + 1}
        </button>
        {loading && <p>Loading...</p>}
        {error && <p>Error: {error}</p>}
        {userData && <pre>{JSON.stringify(userData, null, 2)}</pre>}
      </div>

      <div className="storage-section">
        <h4>Local Storage Preferences</h4>
        <button
```

```
            onClick={() =>
              setPreferences((prev) => ({
                ...prev,
                theme: prev.theme === "light" ? "dark" : "light",
              }))
            }
          >
            Toggle Theme: {preferences.theme}
          </button>
          <pre>{JSON.stringify(preferences, null, 2)}</pre>
        </div>

        <div className="window-section">
          <h4>Window Size</h4>
          <p>Width: {windowSize.width}px</p>
          <p>Height: {windowSize.height}px</p>
        </div>
      </div>
    );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Missing Dependencies

```
// ❌ WRONG: Missing dependencies
function MissingDependencies() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  useEffect(() => {
    const result = count * multiplier; // Uses count and multiplier
    console.log("Result:", result);
  }, []); // ❌ Missing count and multiplier dependencies

  return (
    <div>
      <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>
      <button onClick={() => setMultiplier((m) => m + 1)}>
        Multiplier: {multiplier}
      </button>
    </div>
  );
}

// ✅ CORRECT: All dependencies included
function CorrectDependencies() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  useEffect(() => {
```

```
    const result = count * multiplier;
    console.log("Result:", result);
  }, [count, multiplier]); // ✅ All dependencies included

  return (
    <div>
      <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>
      <button onClick={() => setMultiplier((m) => m + 1)}>
        Multiplier: {multiplier}
      </button>
    </div>
  );
}
```

2. Infinite Loops

```
// ❌ WRONG: Infinite loop
function InfiniteLoop() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // This creates a new array every time
    setData([1, 2, 3]); // ❌ Causes infinite re-renders
  }, [data]); // ❌ data changes every render

  return <div>Data: {data.join(", ")}</div>;
}

// ✅ CORRECT: Stable dependencies
function NoInfiniteLoop() {
  const [data, setData] = useState([]);

  useEffect(() => {
    setData([1, 2, 3]);
  }, []); // ✅ Empty dependency array

  return <div>Data: {data.join(", ")}</div>;
}
```

3. Stale Closures

```
// ❌ WRONG: Stale closure
function StaleClosure() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount(count + 1); // ❌ Always uses initial count value (0)
    }, 1000);
```

```
    return () => clearInterval(timer);
  }, []); // ❌ Empty deps but uses count

  return <div>Count: {count}</div>;
}

// ✅ CORRECT: Functional update
function NoStaleClosure() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount((prevCount) => prevCount + 1); // ✅ Uses current value
    }, 1000);

    return () => clearInterval(timer);
  }, []); // ✅ No dependencies needed

  return <div>Count: {count}</div>;
}
```

4. Unnecessary Effects

```
// ❌ WRONG: Unnecessary effect for derived state
function UnnecessaryEffect() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");
  const [fullName, setFullName] = useState("");

  // ❌ Don't use effect for derived state
  useEffect(() => {
    setFullName(`${firstName} ${lastName}`);
  }, [firstName, lastName]);

  return (
    <div>
      <input
        value={firstName}
        onChange={(e) => setFirstName(e.target.value)}
        placeholder="First name"
      />
      <input
        value={lastName}
        onChange={(e) => setLastName(e.target.value)}
        placeholder="Last name"
      />
      <p>Full name: {fullName}</p>
    </div>
  );
}
```

```
// ✅ CORRECT: Derived state during render
function DerivedState() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");

  // ✅ Calculate during render
  const fullName = `${firstName} ${lastName}`.trim();

  return (
    <div>
      <input
        value={firstName}
        onChange={(e) => setFirstName(e.target.value)}
        placeholder="First name"
      />
      <input
        value={lastName}
        onChange={(e) => setLastName(e.target.value)}
        placeholder="Last name"
      />
      <p>Full name: {fullName}</p>
    </div>
  );
}
```

🧪 Mini Challenges

Challenge 1: Data Fetcher with Cleanup

Build a component that:

- Fetches user data based on user ID
- Shows loading state
- Handles errors gracefully
- Cancels ongoing requests when user ID changes
- Implements retry functionality

▶ 💡 Solution

```
function DataFetcher({ userId }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const [retryCount, setRetryCount] = useState(0);

  useEffect(() => {
    if (!userId) {
      setData(null);
      setError(null);
      return;
```

```
    }

    const abortController = new AbortController();
    let retryTimeout;

    const fetchUserData = async (attempt = 0) => {
      setLoading(true);
      setError(null);

      try {
        const response = await fetch(`/api/users/${userId}`, {
          signal: abortController.signal,
        });

        if (!response.ok) {
          throw new Error(`HTTP ${response.status}: ${response.statusText}`);
        }

        const userData = await response.json();
        setData(userData);
        setRetryCount(0);
      } catch (err) {
        if (err.name === "AbortError") {
          console.log("Request was aborted");
          return;
        }

        setError(err.message);

        // Retry logic
        if (attempt < 3) {
          const delay = Math.pow(2, attempt) * 1000; // Exponential backoff
          retryTimeout = setTimeout(() => {
            setRetryCount(attempt + 1);
            fetchUserData(attempt + 1);
          }, delay);
        }
      } finally {
        setLoading(false);
      }
    };

    fetchUserData();

    return () => {
      abortController.abort();
      if (retryTimeout) {
        clearTimeout(retryTimeout);
      }
    };
  }, [userId]);

  const handleRetry = () => {
    setRetryCount(0);
```

```
    setError(null);
    // Trigger re-fetch by updating a dependency
  };

  if (loading) {
    return (
      <div className="data-fetcher loading">
        <p>Loading user data...</p>
        {retryCount > 0 && <p>Retry attempt: {retryCount}</p>}
      </div>
    );
  }

  if (error) {
    return (
      <div className="data-fetcher error">
        <p>Error: {error}</p>
        <button onClick={handleRetry}>Retry</button>
      </div>
    );
  }

  if (!data) {
    return (
      <div className="data-fetcher empty">
        <p>No user selected</p>
      </div>
    );
  }

  return (
    <div className="data-fetcher success">
      <h3>User Data</h3>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}
```

Challenge 2: Real-time Chat Component

Create a chat component that:

- Connects to WebSocket on mount
- Sends and receives messages
- Shows connection status
- Handles reconnection on disconnect
- Cleans up properly on unmount

▶ 💡 Solution

```
function RealTimeChat({ roomId, userId }) {
  const [messages, setMessages] = useState([]);
  const [newMessage, setNewMessage] = useState("");
  const [connectionStatus, setConnectionStatus] = useState("disconnected");
  const [ws, setWs] = useState(null);
  const reconnectTimeoutRef = useRef(null);
  const reconnectAttemptsRef = useRef(0);

  const connectWebSocket = useCallback(() => {
    if (!roomId || !userId) return;

    setConnectionStatus("connecting");

    const websocket = new WebSocket(`wss://chat.example.com/rooms/${roomId}`);

    websocket.onopen = () => {
      console.log("WebSocket connected");
      setConnectionStatus("connected");
      reconnectAttemptsRef.current = 0;

      // Send join message
      websocket.send(
        JSON.stringify({
          type: "join",
          userId,
          roomId,
        })
      );
    };

    websocket.onmessage = (event) => {
      try {
        const message = JSON.parse(event.data);

        if (message.type === "message") {
          setMessages((prev) => [
            ...prev,
            {
              id: message.id,
              text: message.text,
              userId: message.userId,
              timestamp: new Date(message.timestamp),
            },
          ]);
        }
      } catch (error) {
        console.error("Error parsing message:", error);
      }
    };

    websocket.onerror = (error) => {
      console.error("WebSocket error:", error);
      setConnectionStatus("error");
```

```
    };

    websocket.onclose = (event) => {
      console.log("WebSocket closed:", event.code, event.reason);
      setConnectionStatus("disconnected");
      setWs(null);

      // Attempt to reconnect if not a clean close
      if (event.code !== 1000 && reconnectAttemptsRef.current < 5) {
        const delay = Math.min(
          1000 * Math.pow(2, reconnectAttemptsRef.current),
          30000
        );

        reconnectTimeoutRef.current = setTimeout(() => {
          reconnectAttemptsRef.current++;
          connectWebSocket();
        }, delay);
      }
    };

    setWs(websocket);
  }, [roomId, userId]);

  // Connect on mount and when roomId/userId changes
  useEffect(() => {
    connectWebSocket();

    return () => {
      if (reconnectTimeoutRef.current) {
        clearTimeout(reconnectTimeoutRef.current);
      }

      if (ws) {
        ws.close(1000, "Component unmounting");
      }
    };
  }, [connectWebSocket]);

  // Cleanup on unmount
  useEffect(() => {
    return () => {
      if (ws && ws.readyState === WebSocket.OPEN) {
        ws.send(
          JSON.stringify({
            type: "leave",
            userId,
            roomId,
          })
        );
        ws.close(1000, "User leaving");
      }
    };
  }, []);
```

```
  const sendMessage = (e) => {
    e.preventDefault();

    if (!newMessage.trim() || !ws || ws.readyState !== WebSocket.OPEN) {
      return;
    }

    const message = {
      type: "message",
      text: newMessage.trim(),
      userId,
      roomId,
      timestamp: new Date().toISOString(),
    };

    ws.send(JSON.stringify(message));
    setNewMessage("");
  };

  const getStatusColor = () => {
    switch (connectionStatus) {
      case "connected":
        return "green";
      case "connecting":
        return "orange";
      case "error":
        return "red";
      default:
        return "gray";
    }
  };

  return (
    <div className="real-time-chat">
      <div className="chat-header">
        <h3>Chat Room: {roomId}</h3>
        <div className="connection-status">
          <span
            className="status-indicator"
            style={{ backgroundColor: getStatusColor() }}
          />
          {connectionStatus}
          {reconnectAttemptsRef.current > 0 && (
            <span> (Attempt {reconnectAttemptsRef.current})</span>
          )}
        </div>
      </div>

      <div className="messages">
        {messages.map((message) => (
          <div
            key={message.id}
            className={`message ${message.userId === userId ? "own" : "other"}`}
```

```
            >
              <div className="message-header">
                <span className="user-id">{message.userId}</span>
                <span className="timestamp">
                  {message.timestamp.toLocaleTimeString()}
                </span>
              </div>
              <div className="message-text">{message.text}</div>
            </div>
          ))}
        </div>

        <form onSubmit={sendMessage} className="message-form">
          <input
            type="text"
            value={newMessage}
            onChange={(e) => setNewMessage(e.target.value)}
            placeholder="Type a message..."
            disabled={connectionStatus !== "connected"}
          />
          <button
            type="submit"
            disabled={connectionStatus !== "connected" || !newMessage.trim()}
          >
            Send
          </button>
        </form>

        {connectionStatus === "error" && (
          <button onClick={connectWebSocket} className="reconnect-button">
            Reconnect
          </button>
        )}
      </div>
    );
}
```

🎯 When and Why: useEffect Decision Framework

Quick Decision Tree

```
🤔 Do I need useEffect?

├── Is this for side effects? (API calls, subscriptions, DOM manipulation)
│   ├── Yes → Continue...
│   └── No → Use regular variables or useMemo/useCallback
│
├── When should it run?
│   ├── Once on mount → useEffect(() => {}, [])
│   ├── On every render → useEffect(() => {})
│   ├── When specific values change → useEffect(() => {}, [dep1, dep2])
```

```
│   └── Conditionally → Add condition inside effect
│
├── Does it need cleanup?
│   ├── Timers → Yes, clear them
│   ├── Event listeners → Yes, remove them
│   ├── Subscriptions → Yes, unsubscribe
│   ├── Network requests → Yes, abort them
│   └── DOM modifications → Yes, revert them
│
└── Are dependencies correct?
    ├── Include all values from component scope used inside effect
    ├── Use ESLint plugin to catch missing dependencies
    └── Consider useCallback/useMemo for stable references
```

Interview Insights

Common Interview Questions

1. **"Explain the useEffect dependency array"**
    - No array: runs after every render
    - Empty array: runs once on mount
    - With dependencies: runs when dependencies change
    - Show examples of each pattern
2. **"What happens if you don't clean up effects?"**
    - Memory leaks from timers and event listeners
    - Stale closures and race conditions
    - Performance degradation
    - Show before/after cleanup examples
3. **"How do you handle async operations in useEffect?"**
    - Can't make useEffect async directly
    - Create async function inside effect
    - Use AbortController for cleanup
    - Handle race conditions
4. **"What's the difference between useEffect and useLayoutEffect?"**
    - useEffect: runs after DOM updates (asynchronous)
    - useLayoutEffect: runs before DOM updates (synchronous)
    - Use useLayoutEffect for DOM measurements

Code Review Red Flags

```
// 🚩 Red Flags in Interviews:

// ❌ Missing cleanup
```

```
useEffect(() => {
  setInterval(() => {}, 1000); // No cleanup!
}, []);

// ❌ Missing dependencies
useEffect(() => {
  console.log(someVariable); // Uses someVariable but not in deps
}, []);

// ❌ Infinite loop
useEffect(() => {
  setSomeState({}); // Creates new object every time
}, [someState]);

// ❌ Async useEffect
useEffect(async () => {
  // Don't do this!
  const data = await fetch("/api");
}, []);

// ❌ Effect for derived state
useEffect(() => {
  setFullName(firstName + lastName); // Should be calculated during render
}, [firstName, lastName]);
```

🎯 Key Takeaways

Mental Model

```
// 🤔 Think of useEffect as:

// "After React finishes updating the DOM,
//  run this side effect if any of these
//  dependencies have changed since last time"

useEffect(() => {
  // Side effect code here

  return () => {
    // Cleanup code here
    // Runs before next effect or unmount
  };
}, [dependency1, dependency2]); // When to run
```

Best Practices Summary

1. **Always include dependencies** that are used inside the effect
2. **Always clean up** timers, listeners, and subscriptions
3. **Use AbortController** for fetch requests

4. **Prefer functional updates** to avoid stale closures
5. **Don't use effects for derived state** - calculate during render
6. **Use custom hooks** to encapsulate complex effect logic
7. **Test your cleanup** by rapidly mounting/unmounting components

---

Next up: useRef: Refs vs Variables - Master direct DOM access and mutable values that don't trigger re-renders.

Previous: Controlled vs Uncontrolled Inputs

💡 Pro tip: In interviews, always explain your dependency choices and demonstrate understanding of cleanup. Show that you can prevent memory leaks and handle edge cases.

🎯 useRef: Refs vs Variables Deep Dive

> **Master React's escape hatch: Direct DOM access, mutable values, and when refs are better than state**

🎯 What You'll Learn

- Understanding refs vs regular variables vs state
- Direct DOM manipulation and focus management
- Storing mutable values that don't trigger re-renders
- Ref forwarding and imperative APIs
- Common ref patterns and anti-patterns
- Performance optimizations with refs
- Real-world scenarios and best practices

🎯 Understanding useRef Fundamentals

Refs vs State vs Variables

```
function RefVsStateVsVariables() {
  // 🎯 State: Triggers re-render when changed
  const [stateValue, setStateValue] = useState(0);

  // 🎯 Ref: Persists between renders, doesn't trigger re-render
  const refValue = useRef(0);

  // ❌ Regular variable: Gets reset on every render
  let regularVariable = 0;

  // 🎯 Ref for tracking render count
  const renderCount = useRef(0);

  // Increment render count on every render
  renderCount.current += 1;
```

```
  const handleStateIncrement = () => {
    setStateValue((prev) => prev + 1); // ✅ Triggers re-render
  };

  const handleRefIncrement = () => {
    refValue.current += 1; // ✅ Persists but no re-render
    console.log("Ref value:", refValue.current);
  };

  const handleVariableIncrement = () => {
    regularVariable += 1; // ❌ Will be reset on next render
    console.log("Variable value:", regularVariable);
  };

  const forceRerender = () => {
    setStateValue((prev) => prev); // Force re-render without changing state
  };

  return (
    <div className="ref-comparison">
      <h3>Refs vs State vs Variables</h3>

      <div className="render-info">
        <p>
          <strong>Render count:</strong> {renderCount.current}
        </p>
      </div>

      <div className="value-display">
        <div className="state-section">
          <h4>State Value (triggers re-render)</h4>
          <p>Current value: {stateValue}</p>
          <button onClick={handleStateIncrement}>Increment State</button>
        </div>

        <div className="ref-section">
          <h4>Ref Value (persists, no re-render)</h4>
          <p>Current value: {refValue.current}</p>
          <button onClick={handleRefIncrement}>Increment Ref</button>
          <p>
            <em>Check console for actual value</em>
          </p>
        </div>

        <div className="variable-section">
          <h4>Regular Variable (resets each render)</h4>
          <p>Current value: {regularVariable}</p>
          <button onClick={handleVariableIncrement}>Increment Variable</button>
          <p>
            <em>Always resets to 0</em>
          </p>
        </div>
      </div>
```

```
      <div className="controls">
        <button onClick={forceRerender}>Force Re-render</button>
        <button
          onClick={() => {
            console.log("Current values:");
            console.log("State:", stateValue);
            console.log("Ref:", refValue.current);
            console.log("Variable:", regularVariable);
          }}
        >
          Log All Values
        </button>
      </div>

      <div className="explanation">
        <h4>Key Differences:</h4>
        <ul>
          <li>
            <strong>State:</strong> Triggers re-render, UI updates automatically
          </li>
          <li>
            <strong>Ref:</strong> Persists between renders, no re-render, manual
            UI updates
          </li>
          <li>
            <strong>Variable:</strong> Resets on every render, loses value
          </li>
        </ul>
      </div>
    </div>
  );
}
```

DOM Refs and Element Access

```
function DOMRefExamples() {
  // 🎯 Refs for DOM elements
  const inputRef = useRef(null);
  const textareaRef = useRef(null);
  const divRef = useRef(null);
  const videoRef = useRef(null);

  // 🎯 Refs for storing DOM measurements
  const measurementsRef = useRef({ width: 0, height: 0 });

  const [measurements, setMeasurements] = useState({ width: 0, height: 0 });
  const [inputValue, setInputValue] = useState("");
  const [isPlaying, setIsPlaying] = useState(false);

  // 🎯 Focus management
```

```
  const focusInput = () => {
    if (inputRef.current) {
      inputRef.current.focus();
      inputRef.current.select(); // Select all text
    }
  };

  const clearAndFocus = () => {
    setInputValue("");
    // Focus after state update
    setTimeout(() => {
      if (inputRef.current) {
        inputRef.current.focus();
      }
    }, 0);
  };

  // 🎯 Scroll management
  const scrollToTop = () => {
    if (divRef.current) {
      divRef.current.scrollTop = 0;
    }
  };

  const scrollToBottom = () => {
    if (divRef.current) {
      divRef.current.scrollTop = divRef.current.scrollHeight;
    }
  };

  // 🎯 Measurements
  const measureElement = () => {
    if (divRef.current) {
      const rect = divRef.current.getBoundingClientRect();
      const newMeasurements = {
        width: rect.width,
        height: rect.height,
        top: rect.top,
        left: rect.left,
      };

      // Store in ref (doesn't trigger re-render)
      measurementsRef.current = newMeasurements;

      // Update state to show in UI
      setMeasurements(newMeasurements);
    }
  };

  // 🎯 Video control
  const toggleVideo = () => {
    if (videoRef.current) {
      if (isPlaying) {
        videoRef.current.pause();
```

```
      } else {
        videoRef.current.play();
      }
      setIsPlaying(!isPlaying);
    }
  };

  const setVideoTime = (seconds) => {
    if (videoRef.current) {
      videoRef.current.currentTime = seconds;
    }
  };

  // 🎯 Text manipulation
  const insertTextAtCursor = (text) => {
    if (textareaRef.current) {
      const textarea = textareaRef.current;
      const start = textarea.selectionStart;
      const end = textarea.selectionEnd;
      const currentValue = textarea.value;

      const newValue =
        currentValue.substring(0, start) + text + currentValue.substring(end);
      textarea.value = newValue;

      // Set cursor position after inserted text
      const newCursorPos = start + text.length;
      textarea.setSelectionRange(newCursorPos, newCursorPos);
      textarea.focus();
    }
  };

  return (
    <div className="dom-ref-examples">
      <h3>DOM Refs and Element Access</h3>

      {/* Input Focus Management */}
      <div className="input-section">
        <h4>Input Focus Management</h4>
        <input
          ref={inputRef}
          type="text"
          value={inputValue}
          onChange={(e) => setInputValue(e.target.value)}
          placeholder="Type something..."
        />
        <div className="input-controls">
          <button onClick={focusInput}>Focus & Select</button>
          <button onClick={clearAndFocus}>Clear & Focus</button>
          <button
            onClick={() => {
              if (inputRef.current) {
                inputRef.current.blur();
              }
```

```
            }}
          >
            Blur
          </button>
        </div>
      </div>

      {/* Textarea with Text Insertion */}
      <div className="textarea-section">
        <h4>Textarea Manipulation</h4>
        <textarea
          ref={textareaRef}
          rows={4}
          cols={50}
          placeholder="Click buttons to insert text at cursor..."
        />
        <div className="textarea-controls">
          <button onClick={() => insertTextAtCursor("Hello ")}>
            Insert "Hello "
          </button>
          <button onClick={() => insertTextAtCursor("World!")}>
            Insert "World!"
          </button>
          <button onClick={() => insertTextAtCursor("\n")}>
            Insert New Line
          </button>
          <button
            onClick={() => {
              if (textareaRef.current) {
                textareaRef.current.value = "";
                textareaRef.current.focus();
              }
            }}
          >
            Clear
          </button>
        </div>
      </div>

      {/* Scrollable Div */}
      <div className="scroll-section">
        <h4>Scroll Management</h4>
        <div
          ref={divRef}
          className="scrollable-div"
          style={{
            height: "150px",
            overflow: "auto",
            border: "1px solid #ccc",
            padding: "10px",
          }}
        >
          {Array.from({ length: 50 }, (_, i) => (
            <p key={i}>Line {i + 1} - This is some scrollable content</p>
```

```
          ))}
        </div>
        <div className="scroll-controls">
          <button onClick={scrollToTop}>Scroll to Top</button>
          <button onClick={scrollToBottom}>Scroll to Bottom</button>
          <button onClick={measureElement}>Measure Element</button>
        </div>
      </div>

      {/* Measurements Display */}
      <div className="measurements-section">
        <h4>Element Measurements</h4>
        <pre>{JSON.stringify(measurements, null, 2)}</pre>
      </div>

      {/* Video Control */}
      <div className="video-section">
        <h4>Video Control</h4>
        <video
          ref={videoRef}
          width="300"
          height="200"
          controls={false}
          style={{ border: "1px solid #ccc" }}
        >
          <source src="/sample-video.mp4" type="video/mp4" />
          Your browser does not support the video tag.
        </video>
        <div className="video-controls">
          <button onClick={toggleVideo}>{isPlaying ? "Pause" : "Play"}</button>
          <button onClick={() => setVideoTime(0)}>Reset</button>
          <button onClick={() => setVideoTime(10)}>Skip to 10s</button>
        </div>
      </div>
    </div>
  );
}
```

🔄 Mutable Values and Performance

Using Refs for Mutable Values

```
function MutableValueExamples() {
  const [count, setCount] = useState(0);
  const [messages, setMessages] = useState([]);

  // 🎯 Refs for values that don't need to trigger re-renders
  const timerIdRef = useRef(null);
  const previousCountRef = useRef(0);
  const callbackCountRef = useRef(0);
  const isComponentMountedRef = useRef(true);
```

```
// 🎯 Ref for storing expensive calculations
const expensiveValueRef = useRef(null);
const lastCalculationInputRef = useRef(null);

// 🎯 Track previous value
useEffect(() => {
  previousCountRef.current = count;
});

// 🎯 Cleanup on unmount
useEffect(() => {
  return () => {
    isComponentMountedRef.current = false;
    if (timerIdRef.current) {
      clearInterval(timerIdRef.current);
    }
  };
}, []);

const startTimer = () => {
  if (timerIdRef.current) {
    clearInterval(timerIdRef.current);
  }

  timerIdRef.current = setInterval(() => {
    // ✅ Check if component is still mounted
    if (isComponentMountedRef.current) {
      setCount((prevCount) => prevCount + 1);
    }
  }, 1000);
};

const stopTimer = () => {
  if (timerIdRef.current) {
    clearInterval(timerIdRef.current);
    timerIdRef.current = null;
  }
};

// 🎯 Expensive calculation with caching
const getExpensiveValue = (input) => {
  // Only recalculate if input changed
  if (lastCalculationInputRef.current !== input) {
    console.log("Performing expensive calculation...");

    // Simulate expensive operation
    let result = 0;
    for (let i = 0; i < input * 1000000; i++) {
      result += Math.random();
    }

    expensiveValueRef.current = result;
    lastCalculationInputRef.current = input;
```

```
    }

    return expensiveValueRef.current;
  };

  // 🎯 Callback with ref to avoid stale closures
  const handleAsyncOperation = useCallback(() => {
    callbackCountRef.current += 1;
    const currentCallCount = callbackCountRef.current;

    console.log(`Starting async operation #${currentCallCount}`);

    setTimeout(() => {
      // ✅ Check if this is still the latest call
      if (
        callbackCountRef.current === currentCallCount &&
        isComponentMountedRef.current
      ) {
        setMessages((prev) => [
          ...prev,
          {
            id: Date.now(),
            text: `Async operation #${currentCallCount} completed`,
            timestamp: new Date().toLocaleTimeString(),
          },
        ]);
      } else {
        console.log(`Async operation #${currentCallCount} was superseded`);
      }
    }, 2000);
  }, []);

  const clearMessages = () => {
    setMessages([]);
  };

  const expensiveValue = getExpensiveValue(count);

  return (
    <div className="mutable-value-examples">
      <h3>Mutable Values with Refs</h3>

      <div className="counter-section">
        <h4>Timer with Ref Management</h4>
        <p>Current count: {count}</p>
        <p>Previous count: {previousCountRef.current}</p>
        <p>Change: {count - previousCountRef.current}</p>

        <div className="timer-controls">
          <button onClick={startTimer}>Start Timer</button>
          <button onClick={stopTimer}>Stop Timer</button>
          <button onClick={() => setCount(0)}>Reset Count</button>
        </div>
      </div>
```

```
      <div className="expensive-calculation">
        <h4>Cached Expensive Calculation</h4>
        <p>Input: {count}</p>
        <p>
          Expensive result: {expensiveValue?.toFixed(2) || "Not calculated"}
        </p>
        <p>
          <em>Check console - calculation only runs when count changes</em>
        </p>
      </div>

      <div className="async-section">
        <h4>Async Operations with Race Condition Prevention</h4>
        <button onClick={handleAsyncOperation}>
          Start Async Operation (2s delay)
        </button>
        <p>Callback count: {callbackCountRef.current}</p>

        <div className="messages">
          <h5>Messages:</h5>
          {messages.length === 0 ? (
            <p>No messages yet</p>
          ) : (
            <ul>
              {messages.map((message) => (
                <li key={message.id}>
                  <strong>{message.timestamp}:</strong> {message.text}
                </li>
              ))}
            </ul>
          )}
          {messages.length > 0 && (
            <button onClick={clearMessages}>Clear Messages</button>
          )}
        </div>
      </div>

      <div className="ref-info">
        <h4>Ref Usage Summary:</h4>
        <ul>
          <li>
            <code>timerIdRef</code>: Stores timer ID for cleanup
          </li>
          <li>
            <code>previousCountRef</code>: Tracks previous state value
          </li>
          <li>
            <code>callbackCountRef</code>: Prevents race conditions
          </li>
          <li>
            <code>isComponentMountedRef</code>: Prevents state updates after
            unmount
          </li>
```

```
            <li>
              <code>expensiveValueRef</code>: Caches expensive calculations
            </li>
          </ul>
        </div>
      </div>
    );
  }
```

Performance Optimization with Refs

```
function PerformanceOptimizationWithRefs() {
  const [items, setItems] = useState(
    Array.from({ length: 1000 }, (_, i) => ({
      id: i,
      name: `Item ${i}`,
      selected: false,
    }))
  );
  const [filter, setFilter] = useState("");

  // 🎯 Refs for performance optimization
  const selectedItemsRef = useRef(new Set());
  const lastFilterRef = useRef("");
  const filteredItemsRef = useRef([]);
  const renderCountRef = useRef(0);

  renderCountRef.current += 1;

  // 🎯 Memoized filtered items with ref caching
  const filteredItems = useMemo(() => {
    // Only recalculate if filter actually changed
    if (
      lastFilterRef.current === filter &&
      filteredItemsRef.current.length > 0
    ) {
      return filteredItemsRef.current;
    }

    console.log("Filtering items...");
    const result = items.filter((item) =>
      item.name.toLowerCase().includes(filter.toLowerCase())
    );

    lastFilterRef.current = filter;
    filteredItemsRef.current = result;

    return result;
  }, [items, filter]);

  // 🎯 Optimized selection handling
```

```
  const toggleItemSelection = useCallback((itemId) => {
    const selectedItems = selectedItemsRef.current;

    if (selectedItems.has(itemId)) {
      selectedItems.delete(itemId);
    } else {
      selectedItems.add(itemId);
    }

    // Update items state
    setItems((prevItems) =>
      prevItems.map((item) =>
        item.id === itemId
          ? { ...item, selected: selectedItems.has(itemId) }
          : item
      )
    );
  }, []);

  const selectAll = () => {
    const selectedItems = selectedItemsRef.current;
    selectedItems.clear();

    filteredItems.forEach((item) => {
      selectedItems.add(item.id);
    });

    setItems((prevItems) =>
      prevItems.map((item) => ({
        ...item,
        selected: selectedItems.has(item.id),
      }))
    );
  };

  const clearSelection = () => {
    selectedItemsRef.current.clear();

    setItems((prevItems) =>
      prevItems.map((item) => ({ ...item, selected: false }))
    );
  };

  const getSelectionCount = () => {
    return selectedItemsRef.current.size;
  };

  return (
    <div className="performance-optimization">
      <h3>Performance Optimization with Refs</h3>

      <div className="stats">
        <p>Render count: {renderCountRef.current}</p>
        <p>Total items: {items.length}</p>
```

```
        <p>Filtered items: {filteredItems.length}</p>
        <p>Selected items: {getSelectionCount()}</p>
      </div>

      <div className="controls">
        <input
          type="text"
          value={filter}
          onChange={(e) => setFilter(e.target.value)}
          placeholder="Filter items..."
        />

        <div className="selection-controls">
          <button onClick={selectAll}>Select All Filtered</button>
          <button onClick={clearSelection}>Clear Selection</button>
        </div>
      </div>

      <div className="items-list" style={{ height: "300px", overflow: "auto" }}>
        {filteredItems.map((item) => (
          <div
            key={item.id}
            className={`item ${item.selected ? "selected" : ""}`}
            onClick={() => toggleItemSelection(item.id)}
            style={{
              padding: "5px",
              cursor: "pointer",
              backgroundColor: item.selected ? "#e3f2fd" : "transparent",
            }}
          >
            <input
              type="checkbox"
              checked={item.selected}
              onChange={() => {}} // Handled by parent click
            />
            {item.name}
          </div>
        ))}
      </div>

      <div className="optimization-notes">
        <h4>Optimization Techniques Used:</h4>
        <ul>
          <li>
            <strong>selectedItemsRef:</strong> Tracks selection without
            triggering re-renders
          </li>
          <li>
            <strong>filteredItemsRef:</strong> Caches filtered results
          </li>
          <li>
            <strong>lastFilterRef:</strong> Prevents unnecessary filtering
          </li>
          <li>
```

```
            <strong>useMemo:</strong> Memoizes expensive filtering operation
          </li>
          <li>
            <strong>useCallback:</strong> Prevents function recreation
          </li>
        </ul>
      </div>
    </div>
  );
}
```

🔄 Ref Forwarding and Imperative APIs

Ref Forwarding Patterns

```
// 🎯 Basic ref forwarding
const CustomInput = forwardRef((props, ref) => {
  const { label, error, ...inputProps } = props;

  return (
    <div className="custom-input">
      {label && <label>{label}</label>}
      <input ref={ref} {...inputProps} />
      {error && <span className="error">{error}</span>}
    </div>
  );
});

// 🎯 Advanced ref forwarding with imperative API
const AdvancedInput = forwardRef((props, ref) => {
  const { onValueChange, ...otherProps } = props;
  const inputRef = useRef(null);
  const [value, setValue] = useState("");

  // 🎯 Expose imperative API
  useImperativeHandle(
    ref,
    () => ({
      focus: () => {
        inputRef.current?.focus();
      },
      blur: () => {
        inputRef.current?.blur();
      },
      clear: () => {
        setValue("");
        inputRef.current?.focus();
      },
      getValue: () => value,
      setValue: (newValue) => {
        setValue(newValue);
```

```
      },
      selectAll: () => {
        inputRef.current?.select();
      },
      insertAtCursor: (text) => {
        const input = inputRef.current;
        if (input) {
          const start = input.selectionStart;
          const end = input.selectionEnd;
          const newValue =
            value.substring(0, start) + text + value.substring(end);
          setValue(newValue);

          // Set cursor position after inserted text
          setTimeout(() => {
            const newPos = start + text.length;
            input.setSelectionRange(newPos, newPos);
          }, 0);
        }
      },
    }),
    [value]
  );

  const handleChange = (e) => {
    const newValue = e.target.value;
    setValue(newValue);
    onValueChange?.(newValue);
  };

  return (
    <input
      ref={inputRef}
      value={value}
      onChange={handleChange}
      {...otherProps}
    />
  );
});

// 🔗 Modal component with ref forwarding
const Modal = forwardRef(({ children, title, onClose }, ref) => {
  const [isOpen, setIsOpen] = useState(false);
  const modalRef = useRef(null);
  const previousFocusRef = useRef(null);

  useImperativeHandle(
    ref,
    () => ({
      open: () => {
        previousFocusRef.current = document.activeElement;
        setIsOpen(true);
      },
      close: () => {
```

```
        setIsOpen(false);
        previousFocusRef.current?.focus();
      },
      isOpen: () => isOpen,
    }),
    [isOpen]
  );

  // 🎯 Focus management
  useEffect(() => {
    if (isOpen && modalRef.current) {
      modalRef.current.focus();
    }
  }, [isOpen]);

  // 🎯 Escape key handling
  useEffect(() => {
    const handleEscape = (e) => {
      if (e.key === "Escape" && isOpen) {
        setIsOpen(false);
        onClose?.();
      }
    };

    document.addEventListener("keydown", handleEscape);
    return () => document.removeEventListener("keydown", handleEscape);
  }, [isOpen, onClose]);

  if (!isOpen) return null;

  return (
    <div
      className="modal-overlay"
      onClick={() => {
        setIsOpen(false);
        onClose?.();
      }}
    >
      <div
        ref={modalRef}
        className="modal-content"
        onClick={(e) => e.stopPropagation()}
        tabIndex={-1}
      >
        <div className="modal-header">
          <h3>{title}</h3>
          <button
            onClick={() => {
              setIsOpen(false);
              onClose?.();
            }}
          >
            ×
          </button>
```

```
          </div>
          <div className="modal-body">{children}</div>
        </div>
      </div>
    );
  });

  // Usage examples
  function RefForwardingExamples() {
    const customInputRef = useRef(null);
    const advancedInputRef = useRef(null);
    const modalRef = useRef(null);

    const [inputValue, setInputValue] = useState("");

    return (
      <div className="ref-forwarding-examples">
        <h3>Ref Forwarding Examples</h3>

        {/* Basic ref forwarding */}
        <div className="basic-forwarding">
          <h4>Basic Ref Forwarding</h4>
          <CustomInput
            ref={customInputRef}
            label="Custom Input"
            placeholder="Type something..."
          />
          <button onClick={() => customInputRef.current?.focus()}>
            Focus Custom Input
          </button>
        </div>

        {/* Advanced ref forwarding with imperative API */}
        <div className="advanced-forwarding">
          <h4>Advanced Ref Forwarding (Imperative API)</h4>
          <AdvancedInput
            ref={advancedInputRef}
            placeholder="Advanced input..."
            onValueChange={setInputValue}
          />
          <p>Current value: {inputValue}</p>

          <div className="advanced-controls">
            <button onClick={() => advancedInputRef.current?.focus()}>
              Focus
            </button>
            <button onClick={() => advancedInputRef.current?.clear()}>
              Clear
            </button>
            <button onClick={() => advancedInputRef.current?.selectAll()}>
              Select All
            </button>
            <button
              onClick={() =>
```

```
              advancedInputRef.current?.insertAtCursor(" [INSERTED] ")
            }
          >
            Insert Text
          </button>
          <button
            onClick={() => {
              const value = advancedInputRef.current?.getValue();
              alert(`Current value: ${value}`);
            }}
          >
            Get Value
          </button>
        </div>
      </div>

      {/* Modal with ref forwarding */}
      <div className="modal-forwarding">
        <h4>Modal with Ref Forwarding</h4>
        <button onClick={() => modalRef.current?.open()}>Open Modal</button>

        <Modal
          ref={modalRef}
          title="Example Modal"
          onClose={() => console.log("Modal closed")}
        >
          <p>This is modal content.</p>
          <p>Press Escape or click outside to close.</p>
          <input type="text" placeholder="Focus is managed automatically" />
        </Modal>
      </div>
    </div>
  );
}
```

Custom Hooks with Refs

```
// 🎯 Custom hook for element visibility
function useIntersectionObserver(options = {}) {
  const [isIntersecting, setIsIntersecting] = useState(false);
  const [entry, setEntry] = useState(null);
  const elementRef = useRef(null);

  useEffect(() => {
    const element = elementRef.current;
    if (!element) return;

    const observer = new IntersectionObserver(([entry]) => {
      setIsIntersecting(entry.isIntersecting);
      setEntry(entry);
    }, options);
```

```
    observer.observe(element);

    return () => {
      observer.unobserve(element);
    };
  }, [options]);

  return [elementRef, isIntersecting, entry];
}

// 🎯 Custom hook for click outside
function useClickOutside(callback) {
  const ref = useRef(null);

  useEffect(() => {
    const handleClick = (event) => {
      if (ref.current && !ref.current.contains(event.target)) {
        callback();
      }
    };

    document.addEventListener("mousedown", handleClick);
    return () => document.removeEventListener("mousedown", handleClick);
  }, [callback]);

  return ref;
}

// 🎯 Custom hook for hover
function useHover() {
  const [isHovered, setIsHovered] = useState(false);
  const ref = useRef(null);

  useEffect(() => {
    const element = ref.current;
    if (!element) return;

    const handleMouseEnter = () => setIsHovered(true);
    const handleMouseLeave = () => setIsHovered(false);

    element.addEventListener("mouseenter", handleMouseEnter);
    element.addEventListener("mouseleave", handleMouseLeave);

    return () => {
      element.removeEventListener("mouseenter", handleMouseEnter);
      element.removeEventListener("mouseleave", handleMouseLeave);
    };
  }, []);

  return [ref, isHovered];
}

// 🎯 Custom hook for element size
```

```
function useElementSize() {
  const [size, setSize] = useState({ width: 0, height: 0 });
  const ref = useRef(null);

  useEffect(() => {
    const element = ref.current;
    if (!element) return;

    const resizeObserver = new ResizeObserver((entries) => {
      for (const entry of entries) {
        const { width, height } = entry.contentRect;
        setSize({ width, height });
      }
    });

    resizeObserver.observe(element);

    return () => {
      resizeObserver.unobserve(element);
    };
  }, []);

  return [ref, size];
}

// Usage examples
function CustomHookExamples() {
  const [visibilityRef, isVisible] = useIntersectionObserver({
    threshold: 0.5,
  });

  const [hoverRef, isHovered] = useHover();
  const [sizeRef, size] = useElementSize();

  const [showDropdown, setShowDropdown] = useState(false);
  const dropdownRef = useClickOutside(() => setShowDropdown(false));

  return (
    <div className="custom-hook-examples">
      <h3>Custom Hooks with Refs</h3>

      {/* Intersection Observer */}
      <div
        style={{ height: "200px", overflow: "auto", border: "1px solid #ccc" }}
      >
        <div style={{ height: "300px", padding: "20px" }}>
          <p>Scroll down to see the observed element...</p>
        </div>
        <div
          ref={visibilityRef}
          style={{
            padding: "20px",
            backgroundColor: isVisible ? "lightgreen" : "lightcoral",
            transition: "background-color 0.3s",
```

```
        }}
      >
        <p>This element is {isVisible ? "visible" : "not visible"}</p>
      </div>
      <div style={{ height: "300px", padding: "20px" }}>
        <p>More content below...</p>
      </div>
    </div>

    {/* Hover Detection */}
    <div
      ref={hoverRef}
      style={{
        padding: "20px",
        margin: "20px 0",
        backgroundColor: isHovered ? "lightblue" : "lightgray",
        transition: "background-color 0.3s",
        cursor: "pointer",
      }}
    >
      <p>Hover over me! Currently {isHovered ? "hovered" : "not hovered"}</p>
    </div>

    {/* Element Size */}
    <div
      ref={sizeRef}
      style={{
        padding: "20px",
        margin: "20px 0",
        border: "2px solid #333",
        resize: "both",
        overflow: "auto",
        minWidth: "200px",
        minHeight: "100px",
      }}
    >
      <p>Resize me!</p>
      <p>Width: {size.width.toFixed(0)}px</p>
      <p>Height: {size.height.toFixed(0)}px</p>
    </div>

    {/* Click Outside */}
    <div className="dropdown-container">
      <button onClick={() => setShowDropdown(!showDropdown)}>
        Toggle Dropdown
      </button>

      {showDropdown && (
        <div
          ref={dropdownRef}
          style={{
            position: "absolute",
            top: "100%",
            left: 0,
```

```
              backgroundColor: "white",
              border: "1px solid #ccc",
              padding: "10px",
              zIndex: 1000,
            }}
          >
            <p>Click outside to close</p>
            <button onClick={() => alert("Button clicked!")}>Action</button>
          </div>
        )}
      </div>
    </div>
  );
}
```

⚠ Common Mistakes & Anti-Patterns

1. Using Refs Instead of State

```
// ❌ WRONG: Using ref for UI state
function WrongRefUsage() {
  const countRef = useRef(0);

  const increment = () => {
    countRef.current += 1;
    // ❌ UI won't update because no re-render is triggered
  };

  return (
    <div>
      <p>Count: {countRef.current}</p> {/* ❌ Won't update */}
      <button onClick={increment}>Increment</button>
    </div>
  );
}

// ✅ CORRECT: Use state for UI values
function CorrectStateUsage() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount((prev) => prev + 1); // ✅ Triggers re-render
  };

  return (
    <div>
      <p>Count: {count}</p> {/* ✅ Updates correctly */}
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

2. Accessing Refs During Render

```
// ❌ WRONG: Accessing ref.current during render
function WrongRefAccess() {
  const inputRef = useRef(null);

  // ❌ Don't access ref.current during render
  const inputValue = inputRef.current?.value || "";

  return (
    <div>
      <input ref={inputRef} />
      <p>Value: {inputValue}</p> {/* ❌ May be stale or null */}
    </div>
  );
}

// ✅ CORRECT: Access refs in effects or event handlers
function CorrectRefAccess() {
  const inputRef = useRef(null);
  const [inputValue, setInputValue] = useState("");

  const handleGetValue = () => {
    // ✅ Access ref in event handler
    const value = inputRef.current?.value || "";
    setInputValue(value);
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleGetValue}>Get Value</button>
      <p>Value: {inputValue}</p>
    </div>
  );
}
```

3. Not Handling Null Refs

```
// ❌ WRONG: Not checking for null
function UnsafeRefAccess() {
  const elementRef = useRef(null);

  const handleClick = () => {
    elementRef.current.focus(); // ❌ May throw error if null
  };

  return (
```

```
      <div>
        <input ref={elementRef} />
        <button onClick={handleClick}>Focus</button>
      </div>
    );
}

// ✅ CORRECT: Always check for null
function SafeRefAccess() {
  const elementRef = useRef(null);

  const handleClick = () => {
    if (elementRef.current) {
      // ✅ Safe null check
      elementRef.current.focus();
    }

    // Or use optional chaining
    elementRef.current?.focus(); // ✅ Also safe
  };

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

4. Ref Callback Mistakes

```
// ❌ WRONG: Inline ref callback
function InlineRefCallback() {
  const [elements, setElements] = useState([]);

  return (
    <div>
      {[1, 2, 3].map((num) => (
        <input
          key={num}
          ref={(el) => {
            // ❌ Creates new function on every render
            if (el) {
              setElements((prev) => [...prev, el]);
            }
          }}
        />
      ))}
    </div>
  );
}
```

```
// ✅ CORRECT: Stable ref callback
function StableRefCallback() {
  const [elements, setElements] = useState([]);
  const elementsRef = useRef([]);

  const setElementRef = useCallback(
    (index) => (el) => {
      if (el) {
        elementsRef.current[index] = el;
      }
    },
    []
  );

  return (
    <div>
      {[1, 2, 3].map((num, index) => (
        <input
          key={num}
          ref={setElementRef(index)} // ✅ Stable callback
        />
      ))}
    </div>
  );
}
```

🧪 Mini Challenges

Challenge 1: Focus Management System

Build a focus management system that:

- Tracks focus history
- Provides focus navigation (next/previous)
- Handles focus trapping in modals
- Restores focus when components unmount

▶ 💡 Solution

```
function useFocusManager() {
  const focusHistoryRef = useRef([]);
  const currentFocusIndexRef = useRef(-1);
  const trapContainerRef = useRef(null);
  const isTrapActiveRef = useRef(false);

  const addToHistory = useCallback((element) => {
    if (
      element &&
      element !== focusHistoryRef.current[currentFocusIndexRef.current]
    ) {
```

```
      focusHistoryRef.current.push(element);
      currentFocusIndexRef.current = focusHistoryRef.current.length - 1;
    }
  }, []);

  const focusPrevious = useCallback(() => {
    if (currentFocusIndexRef.current > 0) {
      currentFocusIndexRef.current -= 1;
      const element = focusHistoryRef.current[currentFocusIndexRef.current];
      if (element && document.contains(element)) {
        element.focus();
      }
    }
  }, []);

  const focusNext = useCallback(() => {
    if (currentFocusIndexRef.current < focusHistoryRef.current.length - 1) {
      currentFocusIndexRef.current += 1;
      const element = focusHistoryRef.current[currentFocusIndexRef.current];
      if (element && document.contains(element)) {
        element.focus();
      }
    }
  }, []);

  const enableFocusTrap = useCallback((container) => {
    trapContainerRef.current = container;
    isTrapActiveRef.current = true;

    const handleKeyDown = (e) => {
      if (
        e.key === "Tab" &&
        isTrapActiveRef.current &&
        trapContainerRef.current
      ) {
        const focusableElements = trapContainerRef.current.querySelectorAll(
          'button, [href], input, select, textarea,
[tabindex]:not([tabindex="-1"])'
        );

        const firstElement = focusableElements[0];
        const lastElement = focusableElements[focusableElements.length - 1];

        if (e.shiftKey) {
          if (document.activeElement === firstElement) {
            e.preventDefault();
            lastElement.focus();
          }
        } else {
          if (document.activeElement === lastElement) {
            e.preventDefault();
            firstElement.focus();
          }
        }
```

```
      }
    };

    document.addEventListener("keydown", handleKeyDown);

    return () => {
      document.removeEventListener("keydown", handleKeyDown);
      isTrapActiveRef.current = false;
    };
  }, []);

  const disableFocusTrap = useCallback(() => {
    isTrapActiveRef.current = false;
    trapContainerRef.current = null;
  }, []);

  useEffect(() => {
    const handleFocus = (e) => {
      if (!isTrapActiveRef.current) {
        addToHistory(e.target);
      }
    };

    document.addEventListener("focusin", handleFocus);

    return () => {
      document.removeEventListener("focusin", handleFocus);
    };
  }, [addToHistory]);

  return {
    focusPrevious,
    focusNext,
    enableFocusTrap,
    disableFocusTrap,
    getFocusHistory: () => focusHistoryRef.current,
  };
}

function FocusManagementDemo() {
  const { focusPrevious, focusNext, enableFocusTrap, disableFocusTrap } =
    useFocusManager();
  const [showModal, setShowModal] = useState(false);
  const modalRef = useRef(null);

  useEffect(() => {
    if (showModal && modalRef.current) {
      const cleanup = enableFocusTrap(modalRef.current);
      return cleanup;
    } else {
      disableFocusTrap();
    }
  }, [showModal, enableFocusTrap, disableFocusTrap]);
```

```
  return (
    <div className="focus-management-demo">
      <h3>Focus Management System</h3>

      <div className="controls">
        <button onClick={focusPrevious}>Focus Previous</button>
        <button onClick={focusNext}>Focus Next</button>
        <button onClick={() => setShowModal(true)}>Open Modal</button>
      </div>

      <div className="form">
        <input placeholder="Input 1" />
        <input placeholder="Input 2" />
        <button>Button 1</button>
        <textarea placeholder="Textarea"></textarea>
        <button>Button 2</button>
      </div>

      {showModal && (
        <div className="modal-overlay">
          <div ref={modalRef} className="modal" tabIndex={-1}>
            <h4>Modal with Focus Trap</h4>
            <input placeholder="Modal input 1" />
            <input placeholder="Modal input 2" />
            <button onClick={() => setShowModal(false)}>Close</button>
            <button>Another Button</button>
          </div>
        </div>
      )}
    </div>
  );
}
```

Challenge 2: Virtual Scrolling with Refs

Create a virtual scrolling component that:

- Only renders visible items
- Uses refs for scroll position tracking
- Handles dynamic item heights
- Maintains scroll position during updates

▶ 💡 Solution

```
function useVirtualScroll({
  items,
  itemHeight,
  containerHeight,
  overscan = 5,
}) {
  const scrollElementRef = useRef(null);
```

```
  const [scrollTop, setScrollTop] = useState(0);
  const [isScrolling, setIsScrolling] = useState(false);
  const scrollTimeoutRef = useRef(null);

  const startIndex = Math.max(0, Math.floor(scrollTop / itemHeight) - overscan);
  const endIndex = Math.min(
    items.length - 1,
    Math.ceil((scrollTop + containerHeight) / itemHeight) + overscan
  );

  const visibleItems = items
    .slice(startIndex, endIndex + 1)
    .map((item, index) => ({
      ...item,
      index: startIndex + index,
    }));

  const totalHeight = items.length * itemHeight;
  const offsetY = startIndex * itemHeight;

  const handleScroll = useCallback((e) => {
    setScrollTop(e.target.scrollTop);
    setIsScrolling(true);

    if (scrollTimeoutRef.current) {
      clearTimeout(scrollTimeoutRef.current);
    }

    scrollTimeoutRef.current = setTimeout(() => {
      setIsScrolling(false);
    }, 150);
  }, []);

  const scrollToIndex = useCallback(
    (index) => {
      if (scrollElementRef.current) {
        const scrollTop = index * itemHeight;
        scrollElementRef.current.scrollTop = scrollTop;
        setScrollTop(scrollTop);
      }
    },
    [itemHeight]
  );

  const scrollToTop = useCallback(() => {
    scrollToIndex(0);
  }, [scrollToIndex]);

  const scrollToBottom = useCallback(() => {
    scrollToIndex(items.length - 1);
  }, [scrollToIndex, items.length]);

  return {
    scrollElementRef,
```

```
    visibleItems,
    totalHeight,
    offsetY,
    handleScroll,
    scrollToIndex,
    scrollToTop,
    scrollToBottom,
    isScrolling,
  };
}

function VirtualScrollDemo() {
  const items = useMemo(
    () =>
      Array.from({ length: 10000 }, (_, i) => ({
        id: i,
        name: `Item ${i}`,
        description: `Description for item ${i}`,
      })),
    []
  );

  const {
    scrollElementRef,
    visibleItems,
    totalHeight,
    offsetY,
    handleScroll,
    scrollToIndex,
    scrollToTop,
    scrollToBottom,
    isScrolling,
  } = useVirtualScroll({
    items,
    itemHeight: 50,
    containerHeight: 400,
  });

  const [jumpToIndex, setJumpToIndex] = useState("");

  const handleJumpTo = () => {
    const index = parseInt(jumpToIndex);
    if (!isNaN(index) && index >= 0 && index < items.length) {
      scrollToIndex(index);
    }
  };

  return (
    <div className="virtual-scroll-demo">
      <h3>Virtual Scrolling with Refs</h3>

      <div className="controls">
        <button onClick={scrollToTop}>Scroll to Top</button>
        <button onClick={scrollToBottom}>Scroll to Bottom</button>
```

```
        <div className="jump-controls">
          <input
            type="number"
            value={jumpToIndex}
            onChange={(e) => setJumpToIndex(e.target.value)}
            placeholder="Jump to index"
            min="0"
            max={items.length - 1}
          />
          <button onClick={handleJumpTo}>Jump</button>
        </div>
      </div>

      <div className="scroll-info">
        <p>Total items: {items.length}</p>
        <p>Visible items: {visibleItems.length}</p>
        <p>Scrolling: {isScrolling ? "Yes" : "No"}</p>
      </div>

      <div
        ref={scrollElementRef}
        className="virtual-scroll-container"
        style={{
          height: "400px",
          overflow: "auto",
          border: "1px solid #ccc",
        }}
        onScroll={handleScroll}
      >
        <div style={{ height: totalHeight, position: "relative" }}>
          <div
            style={{
              transform: `translateY(${offsetY}px)`,
              position: "absolute",
              top: 0,
              left: 0,
              right: 0,
            }}
          >
            {visibleItems.map((item) => (
              <div
                key={item.id}
                style={{
                  height: "50px",
                  padding: "10px",
                  borderBottom: "1px solid #eee",
                  display: "flex",
                  alignItems: "center",
                }}
              >
                <div>
                  <strong>{item.name}</strong>
                  <br />
```

```
                  <small>{item.description}</small>
                </div>
              </div>
            ))}
          </div>
        </div>
      </div>
    </div>
  );
}
```

🎯 When and Why: useRef Decision Framework

Quick Decision Tree

```
🤔 Should I use useRef?

├── Do I need to access DOM elements directly?
│   ├── Yes → useRef for DOM reference ✅
│   └── No → Continue...
│
├── Do I need a value that persists but doesn't trigger re-renders?
│   ├── Yes → useRef for mutable value ✅
│   └── No → Continue...
│
├── Do I need to store timer IDs, intervals, or subscriptions?
│   ├── Yes → useRef for cleanup references ✅
│   └── No → Continue...
│
├── Do I need to track previous values?
│   ├── Yes → useRef for previous state ✅
│   └── No → Continue...
│
├── Do I need to prevent stale closures in effects?
│   ├── Yes → useRef for current values ✅
│   └── No → Use useState instead
│
└── Is this for UI state that should trigger re-renders?
    └── Yes → Use useState, not useRef ❌
```

🎤 Interview Insights

Common Interview Questions

1. **"What's the difference between useRef and useState?"**
   - useRef: Mutable, doesn't trigger re-renders, persists between renders
   - useState: Immutable updates, triggers re-renders, for UI state
   - Show examples of when to use each

2. **"How do you access DOM elements in React?"**
    - Use useRef to create a ref
    - Attach ref to JSX element
    - Access via ref.current in effects or event handlers
    - Show focus management example
3. **"What is ref forwarding and when do you use it?"**
    - Passing refs through components to child elements
    - Use forwardRef and useImperativeHandle
    - Common in reusable component libraries
    - Show custom input component example
4. **"How do you prevent memory leaks with refs?"**
    - Store cleanup functions (timers, subscriptions)
    - Check if component is mounted before state updates
    - Clean up in useEffect return function
    - Show timer cleanup example

Code Review Red Flags

```
// 🚩 Red Flags in Interviews:

// ❌ Using ref for UI state
const countRef = useRef(0);
return <p>{countRef.current}</p>; // Won't update

// ❌ Accessing ref during render
const value = inputRef.current?.value; // May be null/stale

// ❌ Not checking for null
elementRef.current.focus(); // May throw error

// ❌ Inline ref callbacks
ref={(el) => { /* new function every render */ }}

// ❌ Forgetting cleanup
const timer = setInterval(...);
// No cleanup in useEffect return
```

🎯 Key Takeaways

Mental Model

```
// 🧠 Think of useRef as:

// "A box that holds a value that:
```

```
//  - Persists between renders
//  - Doesn't trigger re-renders when changed
//  - Can hold any mutable value
//  - Is perfect for DOM access and cleanup"

const ref = useRef(initialValue);
// ref.current = the actual value
// ref itself never changes
```

Best Practices Summary

1. **Use refs for DOM access** - focus, scroll, measurements
2. **Use refs for mutable values** - timers, counters, flags
3. **Always check for null** - refs may be null initially
4. **Don't access refs during render** - use effects or event handlers
5. **Use ref forwarding** for reusable components
6. **Store cleanup references** - prevent memory leaks
7. **Prefer useState for UI state** - don't use refs for visible data

---

Next up: useMemo: What, When, and Why - Master performance optimization and expensive calculation caching.

Previous: useEffect Dependencies & Cleanup

💡 Pro tip: In interviews, demonstrate understanding of when NOT to use refs. Show that you know the difference between mutable refs and immutable state, and always explain your choice.# 🚀 useMemo: What, When, and Why

> **Master React's performance optimization: Expensive calculations, object stability, and when memoization actually helps**

🎯 What You'll Learn

- Understanding useMemo fundamentals and when it's needed
- Expensive calculation optimization patterns
- Object and array reference stability
- Dependency array best practices
- Performance measurement and profiling
- Common memoization mistakes and anti-patterns
- Real-world optimization scenarios
- Interview insights and best practices

🎯 Understanding useMemo Fundamentals

What is useMemo?

```
// 🎯 useMemo caches the result of expensive calculations
const memoizedValue = useMemo(() => {
  // Expensive calculation here
  return expensiveCalculation(dependencies);
}, [dependencies]);

// 🎯 Basic syntax breakdown:
// useMemo(calculateValue, dependencies)
// - calculateValue: Function that returns the value to cache
// - dependencies: Array of values that trigger recalculation
```

When React Re-calculates

```
function MemoBasicsDemo() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("John");
  const [items, setItems] = useState([1, 2, 3, 4, 5]);

  // 🎯 Without useMemo - runs on every render
  const expensiveCalculationWithoutMemo = () => {
    console.log("🟢 Expensive calculation running (no memo)");
    let result = 0;
    for (let i = 0; i < 1000000; i++) {
      result += Math.random();
    }
    return result;
  };

  // 🎯 With useMemo - only runs when dependencies change
  const expensiveCalculationWithMemo = useMemo(() => {
    console.log("🟢 Expensive calculation running (with memo)");
    let result = 0;
    for (let i = 0; i < 1000000; i++) {
      result += Math.random();
    }
    return result;
  }, [count]); // Only recalculates when count changes

  // 🎯 Array processing with memo
  const processedItems = useMemo(() => {
    console.log("🟢 Processing items array");
    return items.map((item) => ({
      id: item,
      value: item * 2,
      label: `Item ${item}`,
      isEven: item % 2 === 0,
    }));
  }, [items]); // Only recalculates when items array changes

  // 🎯 Complex object creation
```

```
  const userProfile = useMemo(() => {
    console.log("🟢 Creating user profile object");
    return {
      name,
      displayName: name.toUpperCase(),
      initials: name
        .split(" ")
        .map((n) => n[0])
        .join(""),
      metadata: {
        createdAt: new Date().toISOString(),
        count,
        hasItems: items.length > 0,
      },
    };
  }, [name, count, items.length]); // Recalculates when any dependency changes

  const renderCount = useRef(0);
  renderCount.current += 1;

  return (
    <div className="memo-basics-demo">
      <h3>useMemo Basics Demo</h3>

      <div className="render-info">
        <p>
          <strong>Render count:</strong> {renderCount.current}
        </p>
        <p>
          <em>Check console to see when calculations run</em>
        </p>
      </div>

      <div className="controls">
        <div className="control-group">
          <label>Count (affects memoized calculation):</label>
          <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>
        </div>

        <div className="control-group">
          <label>Name (affects user profile):</label>
          <input
            value={name}
            onChange={(e) => setName(e.target.value)}
            placeholder="Enter name"
          />
        </div>

        <div className="control-group">
          <label>Items (affects processed items):</label>
          <button
            onClick={() => setItems((prev) => [...prev, prev.length + 1])}
          >
            Add Item
```

```
            </button>
            <button onClick={() => setItems((prev) => prev.slice(0, -1))}>
              Remove Item
            </button>
          </div>

          <div className="control-group">
            <label>Force re-render (no state change):</label>
            <button onClick={() => setCount(count)}>Force Re-render</button>
          </div>
        </div>

        <div className="results">
          <div className="result-section">
            <h4>Without Memo (runs every render)</h4>
            <p>Result: {expensiveCalculationWithoutMemo().toFixed(2)}</p>
          </div>

          <div className="result-section">
            <h4>With Memo (runs only when count changes)</h4>
            <p>Result: {expensiveCalculationWithMemo.toFixed(2)}</p>
          </div>

          <div className="result-section">
            <h4>Processed Items</h4>
            <div className="items-grid">
              {processedItems.map((item) => (
                <div
                  key={item.id}
                  className={`item ${item.isEven ? "even" : "odd"}`}
                >
                  {item.label}: {item.value}
                </div>
              ))}
            </div>
          </div>

          <div className="result-section">
            <h4>User Profile</h4>
            <pre>{JSON.stringify(userProfile, null, 2)}</pre>
          </div>
        </div>

        <div className="explanation">
          <h4>What's Happening:</h4>
          <ul>
            <li>
              <strong>Without memo:</strong> Calculation runs on every render
              (expensive!)
            </li>
            <li>
              <strong>With memo:</strong> Calculation only runs when dependencies
              change
            </li>
```

```
            <li>
              <strong>Processed items:</strong> Only recalculates when items array
              changes
            </li>
            <li>
              <strong>User profile:</strong> Only recalculates when name, count,
              or items.length changes
            </li>
          </ul>
        </div>
      </div>
    );
  }
```

💰 Expensive Calculations

Identifying Expensive Operations

```
  function ExpensiveCalculationsDemo() {
    const [data, setData] = useState(
      Array.from({ length: 10000 }, (_, i) => ({
        id: i,
        value: Math.random() * 100,
        category: ["A", "B", "C"][Math.floor(Math.random() * 3)],
      }))
    );

    const [filterCategory, setFilterCategory] = useState("all");
    const [sortBy, setSortBy] = useState("id");
    const [searchTerm, setSearchTerm] = useState("");
    const [threshold, setThreshold] = useState(50);

    // 🎯 Expensive filtering and sorting
    const processedData = useMemo(() => {
      console.time("Data processing");
      console.log("🔄 Processing data...");

      let result = data;

      // Filter by category
      if (filterCategory !== "all") {
        result = result.filter((item) => item.category === filterCategory);
      }

      // Filter by search term (expensive string operations)
      if (searchTerm) {
        result = result.filter(
          (item) =>
            item.id.toString().includes(searchTerm) ||
            item.category.toLowerCase().includes(searchTerm.toLowerCase())
        );
```

```
    }

    // Filter by threshold
    result = result.filter((item) => item.value >= threshold);

    // Sort (expensive for large arrays)
    result = result.sort((a, b) => {
      switch (sortBy) {
        case "value":
          return b.value - a.value;
        case "category":
          return a.category.localeCompare(b.category);
        default:
          return a.id - b.id;
      }
    });

    console.timeEnd("Data processing");
    return result;
  }, [data, filterCategory, searchTerm, threshold, sortBy]);

  // 🎯 Expensive statistical calculations
  const statistics = useMemo(() => {
    console.time("Statistics calculation");
    console.log("📊 Calculating statistics...");

    if (processedData.length === 0) {
      return {
        count: 0,
        average: 0,
        min: 0,
        max: 0,
        median: 0,
        standardDeviation: 0,
      };
    }

    const values = processedData.map((item) => item.value);
    const sum = values.reduce((acc, val) => acc + val, 0);
    const average = sum / values.length;

    const sortedValues = [...values].sort((a, b) => a - b);
    const median =
      sortedValues.length % 2 === 0
        ? (sortedValues[sortedValues.length / 2 - 1] +
            sortedValues[sortedValues.length / 2]) /
          2
        : sortedValues[Math.floor(sortedValues.length / 2)];

    const variance =
      values.reduce((acc, val) => acc + Math.pow(val - average, 2), 0) /
      values.length;
    const standardDeviation = Math.sqrt(variance);
```

```
    console.timeEnd("Statistics calculation");

    return {
      count: processedData.length,
      average: average.toFixed(2),
      min: Math.min(...values).toFixed(2),
      max: Math.max(...values).toFixed(2),
      median: median.toFixed(2),
      standardDeviation: standardDeviation.toFixed(2),
    };
  }, [processedData]);

  // 🎯 Expensive chart data preparation
  const chartData = useMemo(() => {
    console.time("Chart data preparation");
    console.log("📈 Preparing chart data...");

    // Group by category and calculate averages
    const categoryGroups = processedData.reduce((acc, item) => {
      if (!acc[item.category]) {
        acc[item.category] = [];
      }
      acc[item.category].push(item.value);
      return acc;
    }, {});

    const chartData = Object.entries(categoryGroups).map(
      ([category, values]) => ({
        category,
        average: values.reduce((sum, val) => sum + val, 0) / values.length,
        count: values.length,
        min: Math.min(...values),
        max: Math.max(...values),
      })
    );

    console.timeEnd("Chart data preparation");
    return chartData;
  }, [processedData]);

  const addRandomData = () => {
    const newItems = Array.from({ length: 1000 }, (_, i) => ({
      id: data.length + i,
      value: Math.random() * 100,
      category: ["A", "B", "C"][Math.floor(Math.random() * 3)],
    }));
    setData((prev) => [...prev, ...newItems]);
  };

  const resetData = () => {
    setData(
      Array.from({ length: 10000 }, (_, i) => ({
        id: i,
        value: Math.random() * 100,
```

```
        category: ["A", "B", "C"][Math.floor(Math.random() * 3)],
      }))
    );
  };

  return (
    <div className="expensive-calculations-demo">
      <h3>Expensive Calculations Demo</h3>

      <div className="performance-note">
        <p>
          <strong>📊 Performance Note:</strong> Open DevTools Console to see
          timing information
        </p>
      </div>

      <div className="controls">
        <div className="control-row">
          <div className="control-group">
            <label>Filter Category:</label>
            <select
              value={filterCategory}
              onChange={(e) => setFilterCategory(e.target.value)}
            >
              <option value="all">All Categories</option>
              <option value="A">Category A</option>
              <option value="B">Category B</option>
              <option value="C">Category C</option>
            </select>
          </div>

          <div className="control-group">
            <label>Sort By:</label>
            <select value={sortBy} onChange={(e) => setSortBy(e.target.value)}>
              <option value="id">ID</option>
              <option value="value">Value (High to Low)</option>
              <option value="category">Category</option>
            </select>
          </div>
        </div>

        <div className="control-row">
          <div className="control-group">
            <label>Search:</label>
            <input
              type="text"
              value={searchTerm}
              onChange={(e) => setSearchTerm(e.target.value)}
              placeholder="Search by ID or category"
            />
          </div>

          <div className="control-group">
            <label>Minimum Value: {threshold}</label>
```

```
            <input
              type="range"
              min="0"
              max="100"
              value={threshold}
              onChange={(e) => setThreshold(Number(e.target.value))}
            />
          </div>
        </div>

        <div className="control-row">
          <button onClick={addRandomData}>Add 1000 Random Items</button>
          <button onClick={resetData}>Reset Data</button>
        </div>
      </div>

      <div className="results">
        <div className="result-section">
          <h4>Data Summary</h4>
          <p>Total items: {data.length}</p>
          <p>Filtered items: {processedData.length}</p>
          <p>Processing time: Check console</p>
        </div>

        <div className="result-section">
          <h4>Statistics</h4>
          <div className="stats-grid">
            <div className="stat">
              <strong>Count:</strong> {statistics.count}
            </div>
            <div className="stat">
              <strong>Average:</strong> {statistics.average}
            </div>
            <div className="stat">
              <strong>Min:</strong> {statistics.min}
            </div>
            <div className="stat">
              <strong>Max:</strong> {statistics.max}
            </div>
            <div className="stat">
              <strong>Median:</strong> {statistics.median}
            </div>
            <div className="stat">
              <strong>Std Dev:</strong> {statistics.standardDeviation}
            </div>
          </div>
        </div>

        <div className="result-section">
          <h4>Chart Data by Category</h4>
          <div className="chart-data">
            {chartData.map((item) => (
              <div key={item.category} className="chart-item">
                <h5>Category {item.category}</h5>
```

```
                <p>Count: {item.count}</p>
                <p>Average: {item.average.toFixed(2)}</p>
                <p>
                  Range: {item.min.toFixed(2)} - {item.max.toFixed(2)}
                </p>
              </div>
            ))}
          </div>
        </div>

        <div className="result-section">
          <h4>Sample Data (First 10 items)</h4>
          <div className="data-preview">
            {processedData.slice(0, 10).map((item) => (
              <div key={item.id} className="data-item">
                ID: {item.id}, Value: {item.value.toFixed(2)}, Category:{" "}
                {item.category}
              </div>
            ))}
            {processedData.length > 10 && (
              <p>... and {processedData.length - 10} more items</p>
            )}
          </div>
        </div>
      </div>

      <div className="optimization-notes">
        <h4>Optimization Techniques Used:</h4>
        <ul>
          <li>
            <strong>Data Processing:</strong> Memoized filtering, searching, and
            sorting
          </li>
          <li>
            <strong>Statistics:</strong> Memoized complex mathematical
            calculations
          </li>
          <li>
            <strong>Chart Data:</strong> Memoized data transformation for
            visualization
          </li>
          <li>
            <strong>Dependency Arrays:</strong> Only recalculate when relevant
            inputs change
          </li>
        </ul>
      </div>
    </div>
  );
}
```

Reference Stability

Object and Array Reference Stability

```
function ReferenceStabilityDemo() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("John");
  const [items, setItems] = useState(["apple", "banana", "cherry"]);

  // 🎯 Without useMemo - new object/array on every render
  const unstableConfig = {
    theme: "dark",
    count,
    features: ["feature1", "feature2"],
  };

  const unstableArray = items.map((item) => item.toUpperCase());

  // 🎯 With useMemo - stable references
  const stableConfig = useMemo(
    () => ({
      theme: "dark",
      count,
      features: ["feature1", "feature2"],
    }),
    [count]
  ); // Only creates new object when count changes

  const stableArray = useMemo(
    () => items.map((item) => item.toUpperCase()),
    [items]
  ); // Only creates new array when items change

  // 🎯 Complex object with nested properties
  const complexConfig = useMemo(
    () => ({
      user: {
        name,
        preferences: {
          theme: "dark",
          notifications: true,
          layout: "grid",
        },
      },
      app: {
        version: "1.0.0",
        features: {
          search: true,
          export: count > 5,
          advanced: count > 10,
        },
      },
      data: {
        items: items.length,
```

```
        processed: items.map((item) => ({
          original: item,
          processed: item.toUpperCase(),
          length: item.length,
        })),
      },
    }),
    [name, count, items]
  );

  // 🎯 Filtered and sorted data
  const processedItems = useMemo(() => {
    return items
      .filter((item) => item.length > 3)
      .sort()
      .map((item) => ({
        id: item,
        display: item.charAt(0).toUpperCase() + item.slice(1),
        metadata: {
          length: item.length,
          vowels: (item.match(/[aeiou]/gi) || []).length,
        },
      }));
  }, [items]);

  // 🎯 Reference tracking
  const previousRefsRef = useRef({});
  const renderCountRef = useRef(0);
  renderCountRef.current += 1;

  const checkReferenceStability = (name, current) => {
    const previous = previousRefsRef.current[name];
    const isStable = previous === current;
    previousRefsRef.current[name] = current;
    return isStable;
  };

  const addItem = () => {
    const newItem = `item${items.length + 1}`;
    setItems((prev) => [...prev, newItem]);
  };

  const removeItem = () => {
    setItems((prev) => prev.slice(0, -1));
  };

  const updateName = (newName) => {
    setName(newName);
  };

  return (
    <div className="reference-stability-demo">
      <h3>Reference Stability Demo</h3>
```

```
        <div className="render-info">
          <p>
            <strong>Render #{renderCountRef.current}</strong>
          </p>
          <p>
            <em>Watch reference stability indicators below</em>
          </p>
        </div>

        <div className="controls">
          <div className="control-group">
            <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>
            <button onClick={() => setCount(count)}>
              Force Re-render (same count)
            </button>
          </div>

          <div className="control-group">
            <input
              value={name}
              onChange={(e) => updateName(e.target.value)}
              placeholder="Enter name"
            />
          </div>

          <div className="control-group">
            <button onClick={addItem}>Add Item</button>
            <button onClick={removeItem}>Remove Item</button>
            <span>Items: {items.join(", ")}</span>
          </div>
        </div>

        <div className="reference-tracking">
          <h4>Reference Stability Tracking</h4>

          <div className="stability-grid">
            <div className="stability-item">
              <h5>Unstable Config Object</h5>
              <div
                className={`stability-indicator ${
                  checkReferenceStability("unstableConfig", unstableConfig)
                    ? "stable"
                    : "unstable"
                }`}
              >
                {checkReferenceStability("unstableConfig", unstableConfig)
                  ? "✅ Stable"
                  : "❌ New Reference"}
              </div>
              <pre>{JSON.stringify(unstableConfig, null, 2)}</pre>
            </div>

            <div className="stability-item">
              <h5>Stable Config Object (useMemo)</h5>
```

```
            <div
              className={`stability-indicator ${
                checkReferenceStability("stableConfig", stableConfig)
                  ? "stable"
                  : "unstable"
              }`}
            >
              {checkReferenceStability("stableConfig", stableConfig)
                ? "✅ Stable"
                : "🔄 Updated"}
            </div>
            <pre>{JSON.stringify(stableConfig, null, 2)}</pre>
          </div>

          <div className="stability-item">
            <h5>Unstable Array</h5>
            <div
              className={`stability-indicator ${
                checkReferenceStability("unstableArray", unstableArray)
                  ? "stable"
                  : "unstable"
              }`}
            >
              {checkReferenceStability("unstableArray", unstableArray)
                ? "✅ Stable"
                : "❌ New Reference"}
            </div>
            <div>Array: [{unstableArray.join(", ")}]</div>
          </div>

          <div className="stability-item">
            <h5>Stable Array (useMemo)</h5>
            <div
              className={`stability-indicator ${
                checkReferenceStability("stableArray", stableArray)
                  ? "stable"
                  : "unstable"
              }`}
            >
              {checkReferenceStability("stableArray", stableArray)
                ? "✅ Stable"
                : "🔄 Updated"}
            </div>
            <div>Array: [{stableArray.join(", ")}]</div>
          </div>
        </div>

        <div className="complex-object">
          <h5>Complex Configuration Object</h5>
          <div
            className={`stability-indicator ${
              checkReferenceStability("complexConfig", complexConfig)
                ? "stable"
                : "unstable"
```

```
          }`}
        >
          {checkReferenceStability("complexConfig", complexConfig)
            ? "✅ Stable"
            : "🔄 Updated"}
        </div>
        <details>
          <summary>View Complex Config</summary>
          <pre>{JSON.stringify(complexConfig, null, 2)}</pre>
        </details>
      </div>

      <div className="processed-items">
        <h5>Processed Items</h5>
        <div
          className={`stability-indicator ${
            checkReferenceStability("processedItems", processedItems)
              ? "stable"
              : "unstable"
          }`}
        >
          {checkReferenceStability("processedItems", processedItems)
            ? "✅ Stable"
            : "🔄 Updated"}
        </div>
        <div className="items-display">
          {processedItems.map((item) => (
            <div key={item.id} className="processed-item">
              <strong>{item.display}</strong>
              <br />
              <small>
                Length: {item.metadata.length}, Vowels: {item.metadata.vowels}
              </small>
            </div>
          ))}
        </div>
      </div>
    </div>

    <div className="explanation">
      <h4>Reference Stability Impact:</h4>
      <ul>
        <li>
          <strong>❌ Unstable references:</strong> Create new objects/arrays
          every render
        </li>
        <li>
          <strong>✅ Stable references:</strong> Only create new when
          dependencies change
        </li>
        <li>
          <strong>🔄 Updated references:</strong> New reference due to
          dependency change
        </li>
```

```
            <li>
              <strong>Performance:</strong> Stable references prevent unnecessary
              child re-renders
            </li>
            <li>
              <strong>useEffect:</strong> Stable references prevent infinite loops
            </li>
          </ul>
        </div>
      </div>
    );
}
```

Child Component Re-render Prevention

```
// 🎯 Child component that shows re-render behavior
const ExpensiveChild = React.memo(({ config, onAction }) => {
  const renderCount = useRef(0);
  renderCount.current += 1;

  console.log(`ExpensiveChild rendered ${renderCount.current} times`);

  // Simulate expensive rendering
  const expensiveValue = useMemo(() => {
    console.log("ExpensiveChild: Performing expensive calculation");
    let result = 0;
    for (let i = 0; i < 100000; i++) {
      result += Math.random();
    }
    return result;
  }, [config.seed]);

  return (
    <div className="expensive-child">
      <h4>Expensive Child Component</h4>
      <p>Render count: {renderCount.current}</p>
      <p>Config theme: {config.theme}</p>
      <p>Config seed: {config.seed}</p>
      <p>Expensive value: {expensiveValue.toFixed(2)}</p>
      <button onClick={() => onAction("child-action")}>Child Action</button>
    </div>
  );
});

function ChildReRenderDemo() {
  const [parentCount, setParentCount] = useState(0);
  const [childSeed, setChildSeed] = useState(1);
  const [theme, setTheme] = useState("light");

  // 🎯 Unstable config - causes unnecessary re-renders
  const unstableConfig = {
```

```
    theme,
    seed: childSeed,
    timestamp: Date.now(), // ❌ Always different!
  };

  // 🎯 Stable config - only changes when dependencies change
  const stableConfig = useMemo(
    () => ({
      theme,
      seed: childSeed,
      // timestamp: Date.now() // ❌ Don't include changing values
    }),
    [theme, childSeed]
  );

  // 🎯 Unstable callback - new function every render
  const unstableCallback = (action) => {
    console.log("Action:", action, "Parent count:", parentCount);
  };

  // 🎯 Stable callback - memoized function
  const stableCallback = useCallback(
    (action) => {
      console.log("Action:", action, "Parent count:", parentCount);
    },
    [parentCount]
  );

  const parentRenderCount = useRef(0);
  parentRenderCount.current += 1;

  return (
    <div className="child-rerender-demo">
      <h3>Child Re-render Prevention Demo</h3>

      <div className="parent-info">
        <p>
          <strong>Parent render count:</strong> {parentRenderCount.current}
        </p>
        <p>
          <em>Check console for child render logs</em>
        </p>
      </div>

      <div className="controls">
        <div className="control-group">
          <button onClick={() => setParentCount((c) => c + 1)}>
            Parent Count: {parentCount}
          </button>
          <p>
            <em>This should NOT cause child re-renders with stable props</em>
          </p>
        </div>
```

```
          <div className="control-group">
            <button onClick={() => setChildSeed((s) => s + 1)}>
              Child Seed: {childSeed}
            </button>
            <p>
              <em>This SHOULD cause child re-renders</em>
            </p>
          </div>

          <div className="control-group">
            <button
              onClick={() => setTheme((t) => (t === "light" ? "dark" : "light"))}
            >
              Theme: {theme}
            </button>
            <p>
              <em>This SHOULD cause child re-renders</em>
            </p>
          </div>
        </div>

        <div className="children-comparison">
          <div className="child-section">
            <h4>❌ With Unstable Props</h4>
            <p>
              <em>Re-renders on every parent render</em>
            </p>
            <ExpensiveChild config={unstableConfig} onAction={unstableCallback} />
          </div>

          <div className="child-section">
            <h4>✅ With Stable Props (useMemo + useCallback)</h4>
            <p>
              <em>Only re-renders when props actually change</em>
            </p>
            <ExpensiveChild config={stableConfig} onAction={stableCallback} />
          </div>
        </div>

        <div className="explanation">
          <h4>Optimization Techniques:</h4>
          <ul>
            <li>
              <strong>React.memo:</strong> Prevents re-renders when props haven't
              changed
            </li>
            <li>
              <strong>useMemo:</strong> Stabilizes object/array references
            </li>
            <li>
              <strong>useCallback:</strong> Stabilizes function references
            </li>
            <li>
              <strong>Dependency arrays:</strong> Control when memoization updates
```

```
          </li>
        </ul>
      </div>
    </div>
  );
}
```

🎯 Dependency Array Best Practices

Dependency Array Patterns

```
function DependencyArrayDemo() {
  const [user, setUser] = useState({
    name: "John",
    age: 30,
    email: "john@example.com",
  });
  const [settings, setSettings] = useState({
    theme: "light",
    notifications: true,
  });
  const [items, setItems] = useState([1, 2, 3, 4, 5]);
  const [filter, setFilter] = useState("");

  // 🎯 Primitive dependencies
  const userDisplayName = useMemo(() => {
    console.log("Computing user display name");
    return `${user.name} (${user.age} years old)`;
  }, [user.name, user.age]); // ✅ Only specific properties

  // ❌ WRONG: Entire object as dependency
  const wrongUserDisplay = useMemo(() => {
    console.log("Computing wrong user display (will run too often)");
    return `${user.name} (${user.age} years old)`;
  }, [user]); // ❌ Runs when ANY user property changes

  // 🎯 Array length as dependency
  const itemsInfo = useMemo(() => {
    console.log("Computing items info");
    return {
      count: items.length,
      isEmpty: items.length === 0,
      hasMany: items.length > 10,
    };
  }, [items.length]); // ✅ Only when length changes

  // 🎯 Computed dependencies
  const hasNotifications = settings.notifications;
  const isDarkTheme = settings.theme === "dark";

  const uiConfig = useMemo(() => {
```

```
    console.log("Computing UI config");
    return {
      showNotificationBadge: hasNotifications,
      darkMode: isDarkTheme,
      className: `theme-${settings.theme} ${
        hasNotifications ? "with-notifications" : ""
      }`,
      styles: {
        backgroundColor: isDarkTheme ? "#333" : "#fff",
        color: isDarkTheme ? "#fff" : "#333",
      },
    };
  }, [hasNotifications, isDarkTheme, settings.theme]); // ✅ Specific computed
values

  // 🎯 Complex filtering with multiple dependencies
  const filteredItems = useMemo(() => {
    console.log("Filtering items");

    if (!filter) return items;

    return items.filter((item) => {
      const itemStr = item.toString();
      return itemStr.includes(filter);
    });
  }, [items, filter]); // ✅ Both items and filter

  // 🎯 Expensive calculation with conditional dependencies
  const expensiveCalculation = useMemo(() => {
    console.log("Performing expensive calculation");

    // Only calculate if we have items and user is adult
    if (items.length === 0 || user.age < 18) {
      return { result: 0, message: "No calculation needed" };
    }

    let result = 0;
    for (let i = 0; i < items.length * 1000; i++) {
      result += Math.random() * user.age;
    }

    return {
      result: result.toFixed(2),
      message: `Calculated for ${items.length} items and age ${user.age}`,
    };
  }, [items.length, user.age]); // ✅ Only the values we actually use

  // 🎯 Object with stable keys
  const stableObjectKeys = useMemo(() => Object.keys(user).sort(), [user]);

  const userMetadata = useMemo(() => {
    console.log("Computing user metadata");
    return {
      fieldCount: stableObjectKeys.length,
```

```
      hasEmail: stableObjectKeys.includes("email"),
      hasAge: stableObjectKeys.includes("age"),
      summary: `User has ${
        stableObjectKeys.length
      } fields: ${stableObjectKeys.join(", ")}`,
    };
  }, [stableObjectKeys]); // ✅ Depends on stable key array

  // 🎯 Functions in dependencies (should be memoized)
  const processItem = useCallback(
    (item) => {
      return {
        original: item,
        doubled: item * 2,
        userAgeMultiplied: item * user.age,
      };
    },
    [user.age]
  );

  const processedItems = useMemo(() => {
    console.log("Processing items with function");
    return filteredItems.map(processItem);
  }, [filteredItems, processItem]); // ✅ processItem is memoized

  const updateUser = (field, value) => {
    setUser((prev) => ({ ...prev, [field]: value }));
  };

  const updateSettings = (field, value) => {
    setSettings((prev) => ({ ...prev, [field]: value }));
  };

  const addItem = () => {
    setItems((prev) => [...prev, Math.max(...prev) + 1]);
  };

  const removeItem = () => {
    setItems((prev) => prev.slice(0, -1));
  };

  return (
    <div className="dependency-array-demo">
      <h3>Dependency Array Best Practices</h3>

      <div className="controls">
        <div className="control-section">
          <h4>User Controls</h4>
          <div className="control-group">
            <label>Name:</label>
            <input
              value={user.name}
              onChange={(e) => updateUser("name", e.target.value)}
            />
```

```
            </div>
            <div className="control-group">
              <label>Age:</label>
              <input
                type="number"
                value={user.age}
                onChange={(e) => updateUser("age", Number(e.target.value))}
              />
            </div>
            <div className="control-group">
              <label>Email:</label>
              <input
                value={user.email}
                onChange={(e) => updateUser("email", e.target.value)}
              />
            </div>
          </div>

          <div className="control-section">
            <h4>Settings Controls</h4>
            <div className="control-group">
              <label>Theme:</label>
              <select
                value={settings.theme}
                onChange={(e) => updateSettings("theme", e.target.value)}
              >
                <option value="light">Light</option>
                <option value="dark">Dark</option>
              </select>
            </div>
            <div className="control-group">
              <label>
                <input
                  type="checkbox"
                  checked={settings.notifications}
                  onChange={(e) =>
                    updateSettings("notifications", e.target.checked)
                  }
                />
                Notifications
              </label>
            </div>
          </div>

          <div className="control-section">
            <h4>Items Controls</h4>
            <div className="control-group">
              <button onClick={addItem}>Add Item</button>
              <button onClick={removeItem}>Remove Item</button>
              <span>Items: [{items.join(", ")}]</span>
            </div>
            <div className="control-group">
              <label>Filter:</label>
              <input
```

```
            value={filter}
            onChange={(e) => setFilter(e.target.value)}
            placeholder="Filter items"
          />
        </div>
      </div>
    </div>

    <div className="results">
      <div className="result-section">
        <h4>User Display Names</h4>
        <p>
          <strong>Optimized (name + age only):</strong> {userDisplayName}
        </p>
        <p>
          <strong>Unoptimized (entire user object):</strong>{" "}
          {wrongUserDisplay}
        </p>
        <p>
          <em>Check console - unoptimized version runs more often</em>
        </p>
      </div>

      <div className="result-section">
        <h4>Items Information</h4>
        <pre>{JSON.stringify(itemsInfo, null, 2)}</pre>
      </div>

      <div className="result-section">
        <h4>UI Configuration</h4>
        <div style={uiConfig.styles} className={uiConfig.className}>
          <p>Theme: {settings.theme}</p>
          <p>
            Notifications: {settings.notifications ? "Enabled" : "Disabled"}
          </p>
          <p>Dark mode: {uiConfig.darkMode ? "Yes" : "No"}</p>
        </div>
      </div>

      <div className="result-section">
        <h4>Filtered Items</h4>
        <p>Filter: "{filter}"</p>
        <p>Results: [{filteredItems.join(", ")}]</p>
      </div>

      <div className="result-section">
        <h4>Expensive Calculation</h4>
        <p>Result: {expensiveCalculation.result}</p>
        <p>Message: {expensiveCalculation.message}</p>
      </div>

      <div className="result-section">
        <h4>User Metadata</h4>
        <pre>{JSON.stringify(userMetadata, null, 2)}</pre>
```

```
          </div>

          <div className="result-section">
            <h4>Processed Items</h4>
            <div className="processed-items">
              {processedItems.map((item, index) => (
                <div key={index} className="processed-item">
                  Original: {item.original}, Doubled: {item.doubled}, Age
                  Multiplied: {item.userAgeMultiplied}
                </div>
              ))}
            </div>
          </div>
        </div>

        <div className="best-practices">
          <h4>Dependency Array Best Practices:</h4>
          <ul>
            <li>
              <strong>✅ Use specific properties:</strong> [user.name, user.age]
              instead of [user]
            </li>
            <li>
              <strong>✅ Use computed values:</strong> [items.length] instead of
              [items]
            </li>
            <li>
              <strong>✅ Memoize functions:</strong> Use useCallback for function
              dependencies
            </li>
            <li>
              <strong>✅ Stable references:</strong> Extract stable values outside
              useMemo
            </li>
            <li>
              <strong>❌ Avoid objects/arrays:</strong> Unless you need the entire
              reference
            </li>
            <li>
              <strong>❌ Don't omit dependencies:</strong> Include all values used
              inside
            </li>
          </ul>
        </div>
      </div>
    );
  }
```

⚠ Common Mistakes & Anti-Patterns

1. Premature Optimization

```jsx
// ❌ WRONG: Memoizing everything unnecessarily
function OverMemoizedComponent() {
  const [count, setCount] = useState(0);

  // ❌ Unnecessary - simple calculation
  const doubledCount = useMemo(() => count * 2, [count]);

  // ❌ Unnecessary - primitive value
  const isEven = useMemo(() => count % 2 === 0, [count]);

  // ❌ Unnecessary - simple string
  const message = useMemo(() => `Count is ${count}`, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Doubled: {doubledCount}</p>
      <p>Is Even: {isEven ? "Yes" : "No"}</p>
      <p>{message}</p>
    </div>
  );
}

// ✅ CORRECT: Only memoize when necessary
function OptimallyMemoizedComponent() {
  const [count, setCount] = useState(0);

  // ✅ Simple calculations - no memo needed
  const doubledCount = count * 2;
  const isEven = count % 2 === 0;
  const message = `Count is ${count}`;

  // ✅ Only memoize expensive operations
  const expensiveCalculation = useMemo(() => {
    let result = 0;
    for (let i = 0; i < count * 100000; i++) {
      result += Math.random();
    }
    return result;
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Doubled: {doubledCount}</p>
      <p>Is Even: {isEven ? "Yes" : "No"}</p>
      <p>{message}</p>
      <p>Expensive: {expensiveCalculation.toFixed(2)}</p>
    </div>
  );
}
```

2. Missing Dependencies

```
// ❌ WRONG: Missing dependencies
function MissingDependenciesExample() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  // ❌ Missing 'multiplier' in dependencies
  const calculation = useMemo(() => {
    return count * multiplier; // Uses multiplier but not in deps
  }, [count]); // ❌ Stale closure!

  return (
    <div>
      <p>Count: {count}</p>
      <p>Multiplier: {multiplier}</p>
      <p>Result: {calculation}</p>
    </div>
  );
}

// ✅ CORRECT: All dependencies included
function CorrectDependenciesExample() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  // ✅ All dependencies included
  const calculation = useMemo(() => {
    return count * multiplier;
  }, [count, multiplier]); // ✅ Complete dependency array

  return (
    <div>
      <p>Count: {count}</p>
      <p>Multiplier: {multiplier}</p>
      <p>Result: {calculation}</p>
    </div>
  );
}
```

3. Object/Array Dependencies

```
// ❌ WRONG: Using entire objects as dependencies
function WrongObjectDependencies() {
  const [user, setUser] = useState({
    name: "John",
    age: 30,
    email: "john@example.com",
  });
```

```
  // ❌ Will run whenever ANY user property changes
  const userDisplayName = useMemo(() => {
    return `${user.name} (${user.age})`; // Only uses name and age
  }, [user]); // ❌ Depends on entire user object

  return <p>{userDisplayName}</p>;
}

// ✅ CORRECT: Use specific properties
function CorrectObjectDependencies() {
  const [user, setUser] = useState({
    name: "John",
    age: 30,
    email: "john@example.com",
  });

  // ✅ Only runs when name or age changes
  const userDisplayName = useMemo(() => {
    return `${user.name} (${user.age})`;
  }, [user.name, user.age]); // ✅ Only specific properties

  return <p>{userDisplayName}</p>;
}
```

4. Expensive Dependency Calculations

```
// ❌ WRONG: Expensive calculations in dependency array
function ExpensiveDependencyCalculation() {
  const [items, setItems] = useState([1, 2, 3, 4, 5]);

  // ❌ Expensive operation in dependency array
  const processedItems = useMemo(() => {
    return items.map((item) => item * 2);
  }, [items.map((item) => item.toString())]); // ❌ Creates new array every
render!

  return (
    <div>
      {processedItems.map((item) => (
        <div key={item}>{item}</div>
      ))}
    </div>
  );
}

// ✅ CORRECT: Simple dependencies
function SimpleDependencyCalculation() {
  const [items, setItems] = useState([1, 2, 3, 4, 5]);

  // ✅ Simple dependency
  const processedItems = useMemo(() => {
```

```
    return items.map((item) => ({
      original: item,
      doubled: item * 2,
      stringified: item.toString(),
    }));
  }, [items]); // ✅ Simple array reference

  return (
    <div>
      {processedItems.map((item) => (
        <div key={item.original}>
          {item.original} → {item.doubled} ({item.stringified})
        </div>
      ))}
    </div>
  );
}
```

🧪 Mini Challenges

Challenge 1: Data Dashboard Optimization

Build a data dashboard that:

- Processes large datasets efficiently
- Calculates multiple statistics
- Filters and sorts data
- Only recalculates when necessary

▶ 💡 Solution

```
function DataDashboard() {
  const [rawData] = useState(
    Array.from({ length: 50000 }, (_, i) => ({
      id: i,
      value: Math.random() * 1000,
      category: ["A", "B", "C", "D"][Math.floor(Math.random() * 4)],
      date: new Date(
        2023,
        Math.floor(Math.random() * 12),
        Math.floor(Math.random() * 28) + 1
      ),
      status: ["active", "inactive", "pending"][Math.floor(Math.random() * 3)],
    }))
  );

  const [filters, setFilters] = useState({
    category: "all",
    status: "all",
    minValue: 0,
    maxValue: 1000,
```

```
    dateRange: "all",
  });

  const [sortConfig, setSortConfig] = useState({
    field: "id",
    direction: "asc",
  });

  // 🎯 Filtered data
  const filteredData = useMemo(() => {
    console.time("Data filtering");

    let result = rawData;

    if (filters.category !== "all") {
      result = result.filter((item) => item.category === filters.category);
    }

    if (filters.status !== "all") {
      result = result.filter((item) => item.status === filters.status);
    }

    result = result.filter(
      (item) => item.value >= filters.minValue && item.value <= filters.maxValue
    );

    if (filters.dateRange !== "all") {
      const now = new Date();
      const cutoff = new Date();

      switch (filters.dateRange) {
        case "last30":
          cutoff.setDate(now.getDate() - 30);
          break;
        case "last90":
          cutoff.setDate(now.getDate() - 90);
          break;
        case "thisYear":
          cutoff.setFullYear(now.getFullYear(), 0, 1);
          break;
      }

      result = result.filter((item) => item.date >= cutoff);
    }

    console.timeEnd("Data filtering");
    return result;
  }, [rawData, filters]);

  // 🎯 Sorted data
  const sortedData = useMemo(() => {
    console.time("Data sorting");

    const result = [...filteredData].sort((a, b) => {
```

```
      const aVal = a[sortConfig.field];
      const bVal = b[sortConfig.field];

      let comparison = 0;
      if (aVal > bVal) comparison = 1;
      if (aVal < bVal) comparison = -1;

      return sortConfig.direction === "desc" ? -comparison : comparison;
    });

    console.timeEnd("Data sorting");
    return result;
  }, [filteredData, sortConfig]);

  // 🎯 Statistics
  const statistics = useMemo(() => {
    console.time("Statistics calculation");

    if (filteredData.length === 0) {
      return {
        count: 0,
        sum: 0,
        average: 0,
        min: 0,
        max: 0,
        median: 0,
        categoryBreakdown: {},
        statusBreakdown: {},
      };
    }

    const values = filteredData.map((item) => item.value);
    const sum = values.reduce((acc, val) => acc + val, 0);
    const sortedValues = [...values].sort((a, b) => a - b);
    const median =
      sortedValues.length % 2 === 0
        ? (sortedValues[sortedValues.length / 2 - 1] +
            sortedValues[sortedValues.length / 2]) /
          2
        : sortedValues[Math.floor(sortedValues.length / 2)];

    const categoryBreakdown = filteredData.reduce((acc, item) => {
      acc[item.category] = (acc[item.category] || 0) + 1;
      return acc;
    }, {});

    const statusBreakdown = filteredData.reduce((acc, item) => {
      acc[item.status] = (acc[item.status] || 0) + 1;
      return acc;
    }, {});

    console.timeEnd("Statistics calculation");

    return {
```

```
      count: filteredData.length,
      sum: sum.toFixed(2),
      average: (sum / filteredData.length).toFixed(2),
      min: Math.min(...values).toFixed(2),
      max: Math.max(...values).toFixed(2),
      median: median.toFixed(2),
      categoryBreakdown,
      statusBreakdown,
    };
  }, [filteredData]);

  // 🎯 Chart data
  const chartData = useMemo(() => {
    console.time("Chart data preparation");

    const categoryData = Object.entries(statistics.categoryBreakdown).map(
      ([category, count]) => ({
        category,
        count,
        percentage: ((count / statistics.count) * 100).toFixed(1),
      })
    );

    const statusData = Object.entries(statistics.statusBreakdown).map(
      ([status, count]) => ({
        status,
        count,
        percentage: ((count / statistics.count) * 100).toFixed(1),
      })
    );

    console.timeEnd("Chart data preparation");

    return { categoryData, statusData };
  }, [statistics]);

  const updateFilter = (key, value) => {
    setFilters((prev) => ({ ...prev, [key]: value }));
  };

  const updateSort = (field) => {
    setSortConfig((prev) => ({
      field,
      direction:
        prev.field === field && prev.direction === "asc" ? "desc" : "asc",
    }));
  };

  return (
    <div className="data-dashboard">
      <h3>Optimized Data Dashboard</h3>

      {/* Filters */}
      <div className="filters">
```

```
          <div className="filter-group">
            <label>Category:</label>
            <select
              value={filters.category}
              onChange={(e) => updateFilter("category", e.target.value)}
            >
              <option value="all">All</option>
              <option value="A">A</option>
              <option value="B">B</option>
              <option value="C">C</option>
              <option value="D">D</option>
            </select>
          </div>

          <div className="filter-group">
            <label>Status:</label>
            <select
              value={filters.status}
              onChange={(e) => updateFilter("status", e.target.value)}
            >
              <option value="all">All</option>
              <option value="active">Active</option>
              <option value="inactive">Inactive</option>
              <option value="pending">Pending</option>
            </select>
          </div>

          <div className="filter-group">
            <label>Value Range:</label>
            <input
              type="range"
              min="0"
              max="1000"
              value={filters.minValue}
              onChange={(e) => updateFilter("minValue", Number(e.target.value))}
            />
            <span>
              {filters.minValue} - {filters.maxValue}
            </span>
            <input
              type="range"
              min="0"
              max="1000"
              value={filters.maxValue}
              onChange={(e) => updateFilter("maxValue", Number(e.target.value))}
            />
          </div>
        </div>

        {/* Statistics */}
        <div className="statistics">
          <h4>Statistics</h4>
          <div className="stats-grid">
            <div className="stat">Count: {statistics.count}</div>
```

```
            <div className="stat">Sum: {statistics.sum}</div>
            <div className="stat">Average: {statistics.average}</div>
            <div className="stat">Min: {statistics.min}</div>
            <div className="stat">Max: {statistics.max}</div>
            <div className="stat">Median: {statistics.median}</div>
          </div>
        </div>

        {/* Charts */}
        <div className="charts">
          <div className="chart">
            <h5>Category Breakdown</h5>
            {chartData.categoryData.map((item) => (
              <div key={item.category} className="chart-bar">
                {item.category}: {item.count} ({item.percentage}%)
              </div>
            ))}
          </div>

          <div className="chart">
            <h5>Status Breakdown</h5>
            {chartData.statusData.map((item) => (
              <div key={item.status} className="chart-bar">
                {item.status}: {item.count} ({item.percentage}%)
              </div>
            ))}
          </div>
        </div>

        {/* Data Table */}
        <div className="data-table">
          <h4>Data (First 100 rows)</h4>
          <table>
            <thead>
              <tr>
                <th onClick={() => updateSort("id")}>ID</th>
                <th onClick={() => updateSort("value")}>Value</th>
                <th onClick={() => updateSort("category")}>Category</th>
                <th onClick={() => updateSort("status")}>Status</th>
                <th onClick={() => updateSort("date")}>Date</th>
              </tr>
            </thead>
            <tbody>
              {sortedData.slice(0, 100).map((item) => (
                <tr key={item.id}>
                  <td>{item.id}</td>
                  <td>{item.value.toFixed(2)}</td>
                  <td>{item.category}</td>
                  <td>{item.status}</td>
                  <td>{item.date.toLocaleDateString()}</td>
                </tr>
              ))}
            </tbody>
          </table>
```

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

Challenge 2: Smart Shopping Cart

Create a shopping cart that:

- Calculates totals, taxes, and discounts efficiently
- Handles complex pricing rules
- Updates only when necessary
- Provides real-time validation

▶ 💡 Solution

```
function SmartShoppingCart() {
  const [items, setItems] = useState([
    {
      id: 1,
      name: "Laptop",
      price: 999.99,
      quantity: 1,
      category: "electronics",
      taxable: true,
    },
    {
      id: 2,
      name: "Book",
      price: 29.99,
      quantity: 2,
      category: "books",
      taxable: false,
    },
    {
      id: 3,
      name: "Headphones",
      price: 199.99,
      quantity: 1,
      category: "electronics",
      taxable: true,
    },
  ]);

  const [discountCode, setDiscountCode] = useState("");
  const [shippingMethod, setShippingMethod] = useState("standard");
  const [customerType, setCustomerType] = useState("regular");

  // 🎯 Discount rules
  const discountRules = useMemo(
    () => ({
```

```
      SAVE10: { type: "percentage", value: 0.1, minAmount: 50 },
      ELECTRONICS20: {
        type: "percentage",
        value: 0.2,
        category: "electronics",
      },
      FREESHIP: { type: "shipping", value: 0 },
      STUDENT15: { type: "percentage", value: 0.15, customerType: "student" },
    }),
    []
  );

  // 🎯 Shipping rates
  const shippingRates = useMemo(
    () => ({
      standard: 9.99,
      express: 19.99,
      overnight: 39.99,
      free: 0,
    }),
    []
  );

  // 🎯 Tax rates by category
  const taxRates = useMemo(
    () => ({
      electronics: 0.08,
      books: 0,
      clothing: 0.06,
      food: 0.03,
    }),
    []
  );

  // 🎯 Calculate subtotal
  const subtotal = useMemo(() => {
    console.log("Calculating subtotal");
    return items.reduce((total, item) => total + item.price * item.quantity, 0);
  }, [items]);

  // 🎯 Calculate item-level discounts
  const itemDiscounts = useMemo(() => {
    console.log("Calculating item discounts");

    const discount = discountRules[discountCode];
    if (!discount) return {};

    const discounts = {};

    items.forEach((item) => {
      let itemDiscount = 0;

      if (discount.type === "percentage") {
        // Category-specific discount
```

```
        if (discount.category && item.category === discount.category) {
          itemDiscount = item.price * item.quantity * discount.value;
        }
        // Customer type discount
        else if (
          discount.customerType &&
          customerType === discount.customerType
        ) {
          itemDiscount = item.price * item.quantity * discount.value;
        }
        // General percentage discount
        else if (!discount.category && !discount.customerType) {
          itemDiscount = item.price * item.quantity * discount.value;
        }
      }

      if (itemDiscount > 0) {
        discounts[item.id] = itemDiscount;
      }
    });

    return discounts;
  }, [items, discountCode, customerType, discountRules]);

  // 🎯 Calculate total discount
  const totalDiscount = useMemo(() => {
    console.log("Calculating total discount");

    const itemDiscountTotal = Object.values(itemDiscounts).reduce(
      (sum, discount) => sum + discount,
      0
    );
    const discount = discountRules[discountCode];

    // Check minimum amount requirement
    if (discount && discount.minAmount && subtotal < discount.minAmount) {
      return 0;
    }

    return itemDiscountTotal;
  }, [itemDiscounts, discountRules, discountCode, subtotal]);

  // 🎯 Calculate taxes
  const taxes = useMemo(() => {
    console.log("Calculating taxes");

    const discountedSubtotal = subtotal - totalDiscount;

    return items.reduce((totalTax, item) => {
      if (!item.taxable) return totalTax;

      const taxRate = taxRates[item.category] || 0;
      const itemTotal = item.price * item.quantity;
      const itemDiscountRatio = (itemDiscounts[item.id] || 0) / itemTotal;
```

```
      const discountedItemTotal = itemTotal * (1 - itemDiscountRatio);

      return totalTax + discountedItemTotal * taxRate;
    }, 0);
  }, [items, subtotal, totalDiscount, itemDiscounts, taxRates]);

  // 🎯 Calculate shipping
  const shipping = useMemo(() => {
    console.log("Calculating shipping");

    const discount = discountRules[discountCode];
    if (discount && discount.type === "shipping") {
      return discount.value;
    }

    // Free shipping for orders over $100
    if (subtotal - totalDiscount > 100) {
      return 0;
    }

    return shippingRates[shippingMethod] || 0;
  }, [
    discountCode,
    discountRules,
    subtotal,
    totalDiscount,
    shippingMethod,
    shippingRates,
  ]);

  // 🎯 Calculate final total
  const total = useMemo(() => {
    console.log("Calculating final total");
    return subtotal - totalDiscount + taxes + shipping;
  }, [subtotal, totalDiscount, taxes, shipping]);

  // 🎯 Validation messages
  const validationMessages = useMemo(() => {
    console.log("Calculating validation messages");

    const messages = [];
    const discount = discountRules[discountCode];

    if (discountCode && !discount) {
      messages.push({ type: "error", text: "Invalid discount code" });
    }

    if (discount && discount.minAmount && subtotal < discount.minAmount) {
      messages.push({
        type: "warning",
        text: `Add $${(discount.minAmount - subtotal).toFixed(
          2
        )} more to qualify for discount`,
      });
```

```
    }

    if (
      discount &&
      discount.customerType &&
      customerType !== discount.customerType
    ) {
      messages.push({
        type: "error",
        text: `This discount is only for ${discount.customerType} customers`,
      });
    }

    if (subtotal - totalDiscount > 100 && shipping > 0) {
      messages.push({
        type: "info",
        text: "You qualify for free shipping!",
      });
    }

    return messages;
  }, [
    discountCode,
    discountRules,
    subtotal,
    totalDiscount,
    customerType,
    shipping,
  ]);

  const updateQuantity = (id, quantity) => {
    setItems((prev) =>
      prev.map((item) =>
        item.id === id ? { ...item, quantity: Math.max(0, quantity) } : item
      )
    );
  };

  const removeItem = (id) => {
    setItems((prev) => prev.filter((item) => item.id !== id));
  };

  return (
    <div className="smart-shopping-cart">
      <h3>Smart Shopping Cart</h3>

      {/* Controls */}
      <div className="controls">
        <div className="control-group">
          <label>Customer Type:</label>
          <select
            value={customerType}
            onChange={(e) => setCustomerType(e.target.value)}
          >
```

```
            <option value="regular">Regular</option>
            <option value="student">Student</option>
            <option value="premium">Premium</option>
          </select>
        </div>

        <div className="control-group">
          <label>Discount Code:</label>
          <input
            value={discountCode}
            onChange={(e) => setDiscountCode(e.target.value.toUpperCase())}
            placeholder="Enter discount code"
          />
        </div>

        <div className="control-group">
          <label>Shipping:</label>
          <select
            value={shippingMethod}
            onChange={(e) => setShippingMethod(e.target.value)}
          >
            <option value="standard">Standard ($9.99)</option>
            <option value="express">Express ($19.99)</option>
            <option value="overnight">Overnight ($39.99)</option>
          </select>
        </div>
      </div>

      {/* Validation Messages */}
      {validationMessages.length > 0 && (
        <div className="validation-messages">
          {validationMessages.map((message, index) => (
            <div key={index} className={`message ${message.type}`}>
              {message.text}
            </div>
          ))}
        </div>
      )}

      {/* Cart Items */}
      <div className="cart-items">
        <h4>Cart Items</h4>
        {items.map((item) => (
          <div key={item.id} className="cart-item">
            <div className="item-info">
              <h5>{item.name}</h5>
              <p>Category: {item.category}</p>
              <p>Price: ${item.price}</p>
              <p>Taxable: {item.taxable ? "Yes" : "No"}</p>
            </div>

            <div className="item-controls">
              <button
                onClick={() => updateQuantity(item.id, item.quantity - 1)}
```

```
                >
                  -
                </button>
                <span>Qty: {item.quantity}</span>
                <button
                  onClick={() => updateQuantity(item.id, item.quantity + 1)}
                >
                  +
                </button>
                <button onClick={() => removeItem(item.id)}>Remove</button>
              </div>

              <div className="item-totals">
                <p>Subtotal: ${(item.price * item.quantity).toFixed(2)}</p>
                {itemDiscounts[item.id] && (
                  <p>Discount: -${itemDiscounts[item.id].toFixed(2)}</p>
                )}
              </div>
            </div>
          ))}
        </div>

        {/* Cart Summary */}
        <div className="cart-summary">
          <h4>Order Summary</h4>
          <div className="summary-line">
            <span>Subtotal:</span>
            <span>${subtotal.toFixed(2)}</span>
          </div>
          {totalDiscount > 0 && (
            <div className="summary-line discount">
              <span>Discount ({discountCode}):</span>
              <span>-${totalDiscount.toFixed(2)}</span>
            </div>
          )}
          <div className="summary-line">
            <span>Taxes:</span>
            <span>${taxes.toFixed(2)}</span>
          </div>
          <div className="summary-line">
            <span>Shipping:</span>
            <span>${shipping.toFixed(2)}</span>
          </div>
          <div className="summary-line total">
            <span>
              <strong>Total:</strong>
            </span>
            <span>
              <strong>${total.toFixed(2)}</strong>
            </span>
          </div>
        </div>
      </div>
```

```
    );
}
```

📊 Performance Measurement

Measuring useMemo Impact

```
function PerformanceMeasurement() {
  const [dataSize, setDataSize] = useState(10000);
  const [useOptimization, setUseOptimization] = useState(true);
  const [triggerUpdate, setTriggerUpdate] = useState(0);

  // Generate test data
  const testData = useMemo(() => {
    console.log("Generating test data");
    return Array.from({ length: dataSize }, (_, i) => ({
      id: i,
      value: Math.random() * 100,
      category: ["A", "B", "C"][Math.floor(Math.random() * 3)],
    }));
  }, [dataSize]);

  // 🎯 Performance timing hook
  const usePerformanceTimer = (label) => {
    const startTimeRef = useRef(null);
    const [duration, setDuration] = useState(0);

    const start = useCallback(() => {
      startTimeRef.current = performance.now();
    }, []);

    const end = useCallback(() => {
      if (startTimeRef.current) {
        const duration = performance.now() - startTimeRef.current;
        setDuration(duration);
        console.log(`${label}: ${duration.toFixed(2)}ms`);
      }
    }, [label]);

    return { start, end, duration };
  };

  const optimizedTimer = usePerformanceTimer("Optimized calculation");
  const unoptimizedTimer = usePerformanceTimer("Unoptimized calculation");

  // 🎯 Expensive calculation with optimization
  const optimizedResult = useMemo(() => {
    if (!useOptimization) return null;

    optimizedTimer.start();
```

```
    const result = testData.reduce(
      (acc, item) => {
        acc.total += item.value;
        acc.count += 1;
        acc.categories[item.category] =
          (acc.categories[item.category] || 0) + 1;

        if (item.value > acc.max) acc.max = item.value;
        if (item.value < acc.min) acc.min = item.value;

        return acc;
      },
      {
        total: 0,
        count: 0,
        categories: {},
        max: -Infinity,
        min: Infinity,
      }
    );

    result.average = result.total / result.count;

    optimizedTimer.end();
    return result;
  }, [testData, useOptimization, optimizedTimer]);

  // 🎯 Expensive calculation without optimization
  const calculateUnoptimized = () => {
    if (useOptimization) return null;

    unoptimizedTimer.start();

    const result = testData.reduce(
      (acc, item) => {
        acc.total += item.value;
        acc.count += 1;
        acc.categories[item.category] =
          (acc.categories[item.category] || 0) + 1;

        if (item.value > acc.max) acc.max = item.value;
        if (item.value < acc.min) acc.min = item.value;

        return acc;
      },
      {
        total: 0,
        count: 0,
        categories: {},
        max: -Infinity,
        min: Infinity,
      }
    );
```

```
      result.average = result.total / result.count;

      unoptimizedTimer.end();
      return result;
    };

    const unoptimizedResult = !useOptimization ? calculateUnoptimized() : null;
    const currentResult = useOptimization ? optimizedResult : unoptimizedResult;

    const renderCount = useRef(0);
    renderCount.current += 1;

    return (
      <div className="performance-measurement">
        <h3>Performance Measurement Demo</h3>

        <div className="controls">
          <div className="control-group">
            <label>Data Size:</label>
            <select
              value={dataSize}
              onChange={(e) => setDataSize(Number(e.target.value))}
            >
              <option value={1000}>1,000 items</option>
              <option value={10000}>10,000 items</option>
              <option value={50000}>50,000 items</option>
              <option value={100000}>100,000 items</option>
            </select>
          </div>

          <div className="control-group">
            <label>
              <input
                type="checkbox"
                checked={useOptimization}
                onChange={(e) => setUseOptimization(e.target.checked)}
              />
              Use useMemo Optimization
            </label>
          </div>

          <div className="control-group">
            <button onClick={() => setTriggerUpdate((prev) => prev + 1)}>
              Force Re-render (Trigger: {triggerUpdate})
            </button>
          </div>
        </div>

        <div className="performance-stats">
          <h4>Performance Statistics</h4>
          <div className="stats-grid">
            <div className="stat">
              <strong>Render Count:</strong> {renderCount.current}
            </div>
```

```
            <div className="stat">
              <strong>Data Size:</strong> {dataSize.toLocaleString()} items
            </div>
            <div className="stat">
              <strong>Optimization:</strong>{" "}
              {useOptimization ? "Enabled" : "Disabled"}
            </div>
            <div className="stat">
              <strong>Last Calculation Time:</strong>
              {useOptimization
                ? `${optimizedTimer.duration.toFixed(2)}ms`
                : `${unoptimizedTimer.duration.toFixed(2)}ms`}
            </div>
          </div>
        </div>

        {currentResult && (
          <div className="calculation-results">
            <h4>Calculation Results</h4>
            <div className="results-grid">
              <div className="result">
                <strong>Total:</strong> {currentResult.total.toFixed(2)}
              </div>
              <div className="result">
                <strong>Count:</strong> {currentResult.count}
              </div>
              <div className="result">
                <strong>Average:</strong> {currentResult.average.toFixed(2)}
              </div>
              <div className="result">
                <strong>Min:</strong> {currentResult.min.toFixed(2)}
              </div>
              <div className="result">
                <strong>Max:</strong> {currentResult.max.toFixed(2)}
              </div>
            </div>

            <div className="category-breakdown">
              <h5>Category Breakdown:</h5>
              {Object.entries(currentResult.categories).map(
                ([category, count]) => (
                  <div key={category} className="category-stat">
                    {category}: {count} items
                  </div>
                )
              )}
            </div>
          </div>
        )}

        <div className="performance-notes">
          <h4>Performance Notes:</h4>
          <ul>
            <li>
```

```
            <strong>With useMemo:</strong> Calculation only runs when data
            changes
          </li>
          <li>
            <strong>Without useMemo:</strong> Calculation runs on every render
          </li>
          <li>
            <strong>Force re-render:</strong> Test how optimization affects
            performance
          </li>
          <li>
            <strong>Data size:</strong> Larger datasets show bigger performance
            differences
          </li>
        </ul>
      </div>
    </div>
  );
}
```

🎯 When and Why: useMemo Decision Framework

Quick Decision Tree

```
🤔 Should I use useMemo?

├── Is this an expensive calculation?
│   ├── Yes → Continue evaluation...
│   └── No → Don't use useMemo ❌
│
├── Does the calculation run on every render?
│   ├── Yes → Continue evaluation...
│   └── No → Don't use useMemo ❌
│
├── Do the inputs change frequently?
│   ├── Yes → useMemo might not help ⚠️
│   └── No → Continue evaluation...
│
├── Is this for reference stability?
│   ├── Yes → Use useMemo ✅
│   └── No → Continue evaluation...
│
├── Is this preventing child re-renders?
│   ├── Yes → Use useMemo ✅
│   └── No → Continue evaluation...
│
└── Is the calculation actually expensive?
    ├── Yes → Use useMemo ✅
    └── No → Don't use useMemo ❌
```

Performance Guidelines

```
// 🎯 When TO use useMemo:

// ✅ Expensive calculations
const expensiveValue = useMemo(() => {
  return heavyCalculation(data);
}, [data]);

// ✅ Object/array reference stability
const stableConfig = useMemo(
  () => ({
    theme: "dark",
    features: ["a", "b", "c"],
  }),
  []
);

// ✅ Preventing child re-renders
const memoizedProps = useMemo(
  () => ({
    data: processedData,
    config: settings,
  }),
  [processedData, settings]
);

// ✅ Complex filtering/sorting
const filteredData = useMemo(() => {
  return data
    .filter((item) => item.active)
    .sort((a, b) => a.name.localeCompare(b.name));
}, [data]);

// 🎯 When NOT to use useMemo:

// ❌ Simple calculations
const doubled = count * 2; // Don't memo

// ❌ Primitive values
const isEven = count % 2 === 0; // Don't memo

// ❌ Always changing dependencies
const timestamp = useMemo(() => Date.now(), [Date.now()]); // Useless

// ❌ More expensive than the calculation
const simple = useMemo(() => a + b, [a, b]); // Overhead > benefit
```

🔬 Interview Insights

Common Interview Questions

1. **"When would you use useMemo?"**
    - Expensive calculations that don't need to run on every render
    - Object/array reference stability for child components
    - Complex data transformations (filtering, sorting, grouping)
    - Preventing unnecessary re-renders in React.memo components
2. **"What's the difference between useMemo and useCallback?"**
    - useMemo: Memoizes the result of a calculation
    - useCallback: Memoizes the function itself
    - Both prevent unnecessary re-computations
    - Show examples of when to use each
3. **"How do you measure if useMemo is actually helping?"**
    - Use React DevTools Profiler
    - Console.time() for manual timing
    - Performance.now() for precise measurements
    - Compare render times with/without memoization
4. **"What are the downsides of useMemo?"**
    - Memory overhead (storing cached values)
    - Comparison overhead (checking dependencies)
    - Can make code more complex
    - Premature optimization can hurt performance

Code Review Red Flags

```
// 🚨 Red Flags in Interviews:

// ❌ Memoizing everything
const simple = useMemo(() => a + b, [a, b]);

// ❌ Missing dependencies
const calc = useMemo(() => a * b * c, [a, b]); // Missing 'c'

// ❌ Expensive dependencies
const result = useMemo(() => process(data), [data.map((x) => x.id)]);

// ❌ Object dependencies
const result = useMemo(() => process(user.name), [user]); // Use user.name

// ❌ Always changing dependencies
const result = useMemo(() => calculate(), [Math.random()]);
```

🎯 Key Takeaways

Mental Model

```
// 🤔 Think of useMemo as:

// "A cache that stores the result of expensive calculations
//  and only recalculates when dependencies change"

const memoizedValue = useMemo(() => {
  // This only runs when dependencies change
  return expensiveCalculation(dependencies);
}, [dependencies]);
```

Best Practices Summary

1. **Profile first** - Measure before optimizing
2. **Use for expensive operations** - Not simple calculations
3. **Stabilize references** - For objects/arrays passed to children
4. **Specific dependencies** - Use primitive values when possible
5. **Don't over-memoize** - It has overhead too
6. **Consider alternatives** - Sometimes restructuring is better
7. **Test performance impact** - Verify it actually helps

Production Tips

- **Start without useMemo** - Add only when needed
- **Use React DevTools Profiler** - Identify actual bottlenecks
- **Consider component splitting** - Sometimes better than memoization
- **Watch bundle size** - Don't memoize everything
- **Document expensive operations** - Help future developers understand

---

Next up: useCallback: Function Memoization - Master function memoization and prevent unnecessary re-renders.

Previous: useRef: Refs vs Variables

💡 Pro tip: In interviews, always explain the trade-offs. Show that you understand useMemo has overhead and should only be used when the benefits outweigh the costs. Demonstrate performance measurement techniques.

useCallback: Function Memoization

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What useCallback is and when to use it
- How function references affect React re-renders
- The relationship between useCallback and React.memo
- Performance implications and trade-offs

- Common patterns and anti-patterns
- Real-world optimization scenarios

🤔 What is useCallback?

`useCallback` is a React Hook that **memoizes a function** and returns the same function reference between renders unless its dependencies change.

The Problem: Function Recreation

```
// ❌ Problem: New function on every render
function ParentComponent() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("");

  // 😱 This creates a NEW function on every render
  const handleClick = () => {
    console.log("Button clicked!");
  };

  return (
    <div>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <ExpensiveChild onClick={handleClick} />
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
    </div>
  );
}

// This component re-renders even when only 'name' changes!
const ExpensiveChild = React.memo(({ onClick }) => {
  console.log("ExpensiveChild rendered");
  return <button onClick={onClick}>Click me</button>;
});
```

The Solution: useCallback

```
// ✅ Solution: Stable function reference
function ParentComponent() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("");

  // 🎯 Same function reference unless dependencies change
  const handleClick = useCallback(() => {
    console.log("Button clicked!");
  }, []); // Empty dependencies = never changes

  return (
    <div>
```

```
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <ExpensiveChild onClick={handleClick} />
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
    </div>
  );
}

// Now this only re-renders when onClick actually changes!
const ExpensiveChild = React.memo(({ onClick }) => {
  console.log("ExpensiveChild rendered");
  return <button onClick={onClick}>Click me</button>;
});
```

🔍 Deep Dive: How useCallback Works

Basic Syntax

```
const memoizedCallback = useCallback(
  () => {
    // Function body
    doSomething(a, b);
  },
  [a, b] // Dependencies
);
```

Dependency Array Behavior

```
function CallbackDemo() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(1);
  const [name, setName] = useState("");

  // 🎯 No dependencies - function never changes
  const handleReset = useCallback(() => {
    setCount(0);
    setMultiplier(1);
  }, []);

  // 🎯 Depends on count - changes when count changes
  const handleIncrement = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []); // ✅ No dependencies needed (using functional update)

  // 🎯 Depends on count and multiplier
  const handleCalculate = useCallback(() => {
    return count * multiplier;
  }, [count, multiplier]);

  // 🎯 Depends on external value
```

```
  const handleLog = useCallback(() => {
    console.log(`Current name: ${name}`);
  }, [name]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Multiplier: {multiplier}</p>
      <p>Name: {name}</p>

      <button onClick={handleIncrement}>Increment</button>
      <button onClick={handleReset}>Reset</button>
      <button onClick={handleLog}>Log Name</button>

      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter name"
      />

      <input
        type="number"
        value={multiplier}
        onChange={(e) => setMultiplier(Number(e.target.value))}
      />
    </div>
  );
}
```

🎯 useCallback vs useMemo: Key Differences

```
function ComparisonDemo() {
  const [items, setItems] = useState([1, 2, 3, 4, 5]);

  // 🎯 useMemo: Memoizes the RESULT of a calculation
  const expensiveValue = useMemo(() => {
    console.log("Calculating expensive value");
    return items.reduce((sum, item) => sum + item * item, 0);
  }, [items]);

  // 🎯 useCallback: Memoizes the FUNCTION itself
  const expensiveFunction = useCallback(() => {
    console.log("Creating expensive function");
    return items.reduce((sum, item) => sum + item * item, 0);
  }, [items]);

  // 🎯 Alternative: useMemo to create a function
  const expensiveFunctionAlt = useMemo(() => {
    console.log("Creating function with useMemo");
    return () => items.reduce((sum, item) => sum + item * item, 0);
  }, [items]);
```

```
  return (
    <div>
      <p>Expensive Value: {expensiveValue}</p>
      <button onClick={() => console.log(expensiveFunction())}>
        Call Function
      </button>
      <button onClick={() => console.log(expensiveFunctionAlt())}>
        Call Alt Function
      </button>
    </div>
  );
}
```

Mental Model

```
// 🧠 Think of it this way:

// useMemo: "Remember this VALUE"
const value = useMemo(() => expensiveCalculation(), [deps]);

// useCallback: "Remember this FUNCTION"
const func = useCallback(() => doSomething(), [deps]);

// They're equivalent:
useCallback(fn, deps) === useMemo(() => fn, deps);
```

🚀 Real-World Patterns

Pattern 1: Event Handlers with Parameters

```
function TodoList() {
  const [todos, setTodos] = useState([
    { id: 1, text: "Learn React", completed: false },
    { id: 2, text: "Build an app", completed: false },
    { id: 3, text: "Deploy to production", completed: true },
  ]);

  // ❌ Bad: Creates new function for each todo
  const badRender = () => {
    return todos.map((todo) => (
      <TodoItem
        key={todo.id}
        todo={todo}
        onToggle={() => toggleTodo(todo.id)} // New function every render!
        onDelete={() => deleteTodo(todo.id)} // New function every render!
      />
    ));
  };
```

```
  // ✅ Good: Stable function references
  const handleToggle = useCallback((id) => {
    setTodos((prev) =>
      prev.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  }, []);

  const handleDelete = useCallback((id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
  }, []);

  const handleAdd = useCallback((text) => {
    const newTodo = {
      id: Date.now(),
      text,
      completed: false,
    };
    setTodos((prev) => [...prev, newTodo]);
  }, []);

  return (
    <div className="todo-list">
      <AddTodoForm onAdd={handleAdd} />
      {todos.map((todo) => (
        <TodoItem
          key={todo.id}
          todo={todo}
          onToggle={handleToggle}
          onDelete={handleDelete}
        />
      ))}
    </div>
  );
}

// Optimized child component
const TodoItem = React.memo(({ todo, onToggle, onDelete }) => {
  console.log(`Rendering todo: ${todo.id}`);

  return (
    <div className={`todo-item ${todo.completed ? "completed" : ""}`}>
      <span>{todo.text}</span>
      <button onClick={() => onToggle(todo.id)}>Toggle</button>
      <button onClick={() => onDelete(todo.id)}>Delete</button>
    </div>
  );
});

const AddTodoForm = React.memo(({ onAdd }) => {
  const [text, setText] = useState("");
```

```
  const handleSubmit = useCallback(
    (e) => {
      e.preventDefault();
      if (text.trim()) {
        onAdd(text.trim());
        setText("");
      }
    },
    [text, onAdd]
  );

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add new todo"
      />
      <button type="submit">Add</button>
    </form>
  );
});
```

Pattern 2: API Calls and Async Operations

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  // 🎯 Stable API call function
  const fetchUser = useCallback(async (id) => {
    setLoading(true);
    setError(null);

    try {
      const response = await fetch(`/api/users/${id}`);
      if (!response.ok) throw new Error("Failed to fetch user");

      const userData = await response.json();
      setUser(userData);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }, []); // No dependencies - function is stable

  // 🎯 Refresh function
  const refreshUser = useCallback(() => {
    if (userId) {
```

```
      fetchUser(userId);
    }
  }, [userId, fetchUser]);

  // 🎯 Update user function
  const updateUser = useCallback(
    async (updates) => {
      setLoading(true);

      try {
        const response = await fetch(`/api/users/${userId}`, {
          method: "PATCH",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(updates),
        });

        if (!response.ok) throw new Error("Failed to update user");

        const updatedUser = await response.json();
        setUser(updatedUser);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    },
    [userId]
  );

  // Fetch user on mount and when userId changes
  useEffect(() => {
    if (userId) {
      fetchUser(userId);
    }
  }, [userId, fetchUser]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;
  if (!user) return <div>No user found</div>;

  return (
    <div className="user-profile">
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
      <p>Role: {user.role}</p>

      <UserActions user={user} onRefresh={refreshUser} onUpdate={updateUser} />
    </div>
  );
}

const UserActions = React.memo(({ user, onRefresh, onUpdate }) => {
  const [isEditing, setIsEditing] = useState(false);
  const [formData, setFormData] = useState({
```

```
    name: user.name,
    email: user.email,
  });

  const handleSave = useCallback(() => {
    onUpdate(formData);
    setIsEditing(false);
  }, [formData, onUpdate]);

  const handleCancel = useCallback(() => {
    setFormData({ name: user.name, email: user.email });
    setIsEditing(false);
  }, [user.name, user.email]);

  if (isEditing) {
    return (
      <div className="edit-form">
        <input
          value={formData.name}
          onChange={(e) =>
            setFormData((prev) => ({ ...prev, name: e.target.value }))
          }
          placeholder="Name"
        />
        <input
          value={formData.email}
          onChange={(e) =>
            setFormData((prev) => ({ ...prev, email: e.target.value }))
          }
          placeholder="Email"
        />
        <button onClick={handleSave}>Save</button>
        <button onClick={handleCancel}>Cancel</button>
      </div>
    );
  }

  return (
    <div className="user-actions">
      <button onClick={() => setIsEditing(true)}>Edit</button>
      <button onClick={onRefresh}>Refresh</button>
    </div>
  );
});
```

Pattern 3: Custom Hooks with Callbacks

```
// 🎯 Custom hook for debounced callbacks
function useDebounceCallback(callback, delay) {
  const timeoutRef = useRef(null);
```

```
  const debouncedCallback = useCallback(
    (...args) => {
      if (timeoutRef.current) {
        clearTimeout(timeoutRef.current);
      }

      timeoutRef.current = setTimeout(() => {
        callback(...args);
      }, delay);
    },
    [callback, delay]
  );

  // Cleanup on unmount
  useEffect(() => {
    return () => {
      if (timeoutRef.current) {
        clearTimeout(timeoutRef.current);
      }
    };
  }, []);

  return debouncedCallback;
}

// 🎯 Custom hook for throttled callbacks
function useThrottleCallback(callback, delay) {
  const lastRun = useRef(Date.now());

  const throttledCallback = useCallback(
    (...args) => {
      if (Date.now() - lastRun.current >= delay) {
        callback(...args);
        lastRun.current = Date.now();
      }
    },
    [callback, delay]
  );

  return throttledCallback;
}

// 🎯 Usage example
function SearchComponent() {
  const [query, setQuery] = useState("");
  const [results, setResults] = useState([]);
  const [loading, setLoading] = useState(false);

  // 🎯 Actual search function
  const performSearch = useCallback(async (searchQuery) => {
    if (!searchQuery.trim()) {
      setResults([]);
      return;
    }
```

```
    setLoading(true);
    try {
      const response = await fetch(
        `/api/search?q=${encodeURIComponent(searchQuery)}`
      );
      const data = await response.json();
      setResults(data.results || []);
    } catch (error) {
      console.error("Search failed:", error);
      setResults([]);
    } finally {
      setLoading(false);
    }
  }, []);

  // 🎯 Debounced search (waits 300ms after user stops typing)
  const debouncedSearch = useDebounceCallback(performSearch, 300);

  // 🎯 Throttled search (max once per 1000ms)
  const throttledSearch = useThrottleCallback(performSearch, 1000);

  // 🎯 Handle input change
  const handleInputChange = useCallback(
    (e) => {
      const value = e.target.value;
      setQuery(value);
      debouncedSearch(value);
    },
    [debouncedSearch]
  );

  // 🎯 Handle manual search
  const handleManualSearch = useCallback(() => {
    throttledSearch(query);
  }, [query, throttledSearch]);

  return (
    <div className="search-component">
      <div className="search-input">
        <input
          type="text"
          value={query}
          onChange={handleInputChange}
          placeholder="Search..."
        />
        <button onClick={handleManualSearch}>Search</button>
      </div>

      {loading && <div className="loading">Searching...</div>}

      <div className="search-results">
        {results.map((result, index) => (
          <div key={result.id || index} className="search-result">
```

```
            <h3>{result.title}</h3>
            <p>{result.description}</p>
          </div>
        ))}
      </div>
    </div>
  );
}
```

⚠ Common Mistakes and Anti-Patterns

Mistake 1: Unnecessary useCallback

```
// ❌ Bad: useCallback for simple functions that don't need optimization
function SimpleComponent() {
  const [count, setCount] = useState(0);

  // ❌ Unnecessary - this function is simple and not passed to children
  const handleClick = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  // ✅ Better - just use a regular function
  const handleClickBetter = () => {
    setCount((prev) => prev + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

Mistake 2: Missing Dependencies

```
// ❌ Bad: Missing dependencies
function BuggyComponent() {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  // ❌ Missing 'multiplier' in dependencies
  const handleCalculate = useCallback(() => {
    console.log(count * multiplier); // This will use stale 'multiplier'
  }, [count]); // Missing multiplier!

  // ✅ Correct dependencies
  const handleCalculateFixed = useCallback(() => {
```

```
    console.log(count * multiplier);
  }, [count, multiplier]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Multiplier: {multiplier}</p>
      <button onClick={handleCalculate}>Calculate (Buggy)</button>
      <button onClick={handleCalculateFixed}>Calculate (Fixed)</button>
    </div>
  );
}
```

Mistake 3: Object/Array Dependencies

```
// ❌ Bad: Object as dependency
function ProblematicComponent({ config }) {
  // ❌ 'config' object changes on every render
  const handleProcess = useCallback(() => {
    processData(config);
  }, [config]); // This will recreate the function every time!

  // ✅ Better: Extract specific properties
  const handleProcessFixed = useCallback(() => {
    processData(config);
  }, [config.apiUrl, config.timeout]); // Use specific properties

  // ✅ Even better: Memoize the config object in parent
  // const config = useMemo(() => ({ apiUrl, timeout }), [apiUrl, timeout]);

  return <ProcessButton onClick={handleProcess} />;
}
```

Mistake 4: Inline Object Creation

```
// ❌ Bad: Creating objects inline
function BadParent() {
  const [data, setData] = useState([]);

  const handleUpdate = useCallback((item) => {
    setData((prev) => prev.map((d) => (d.id === item.id ? item : d)));
  }, []);

  return (
    <div>
      {data.map((item) => (
        <ChildComponent
          key={item.id}
          item={item}
```

```
            onUpdate={handleUpdate}
            // ❌ This creates a new object every render!
            config={{ theme: "dark", size: "large" }}
          />
        ))}
      </div>
    );
}

// ✅ Good: Memoize objects
function GoodParent() {
  const [data, setData] = useState([]);

  const handleUpdate = useCallback((item) => {
    setData((prev) => prev.map((d) => (d.id === item.id ? item : d)));
  }, []);

  // ✅ Stable object reference
  const config = useMemo(() => ({ theme: "dark", size: "large" }), []);

  return (
    <div>
      {data.map((item) => (
        <ChildComponent
          key={item.id}
          item={item}
          onUpdate={handleUpdate}
          config={config}
        />
      ))}
    </div>
  );
}
```

🎯 Mini-Challenges

Challenge 1: Optimized Data Table

Create a data table with sorting, filtering, and pagination that doesn't re-render unnecessarily.

▶ 💡 Solution

```
function OptimizedDataTable() {
  const [data, setData] = useState([
    { id: 1, name: "Alice", age: 30, department: "Engineering", salary: 75000 },
    { id: 2, name: "Bob", age: 25, department: "Design", salary: 65000 },
    {
      id: 3,
      name: "Charlie",
      age: 35,
      department: "Engineering",
```

```
    salary: 85000,
  },
  { id: 4, name: "Diana", age: 28, department: "Marketing", salary: 60000 },
  { id: 5, name: "Eve", age: 32, department: "Engineering", salary: 80000 },
  { id: 6, name: "Frank", age: 29, department: "Design", salary: 70000 },
  { id: 7, name: "Grace", age: 31, department: "Marketing", salary: 62000 },
  { id: 8, name: "Henry", age: 27, department: "Engineering", salary: 78000 },
]);

const [sortConfig, setSortConfig] = useState({ key: null, direction: "asc" });
const [filterText, setFilterText] = useState("");
const [currentPage, setCurrentPage] = useState(1);
const [pageSize, setPageSize] = useState(5);

// 🎯 Memoized filtered data
const filteredData = useMemo(() => {
  if (!filterText) return data;

  return data.filter((item) =>
    Object.values(item).some((value) =>
      value.toString().toLowerCase().includes(filterText.toLowerCase())
    )
  );
}, [data, filterText]);

// 🎯 Memoized sorted data
const sortedData = useMemo(() => {
  if (!sortConfig.key) return filteredData;

  return [...filteredData].sort((a, b) => {
    const aValue = a[sortConfig.key];
    const bValue = b[sortConfig.key];

    if (aValue < bValue) return sortConfig.direction === "asc" ? -1 : 1;
    if (aValue > bValue) return sortConfig.direction === "asc" ? 1 : -1;
    return 0;
  });
}, [filteredData, sortConfig]);

// 🎯 Memoized paginated data
const paginatedData = useMemo(() => {
  const startIndex = (currentPage - 1) * pageSize;
  return sortedData.slice(startIndex, startIndex + pageSize);
}, [sortedData, currentPage, pageSize]);

// 🎯 Memoized pagination info
const paginationInfo = useMemo(() => {
  const totalItems = sortedData.length;
  const totalPages = Math.ceil(totalItems / pageSize);
  const startItem = (currentPage - 1) * pageSize + 1;
  const endItem = Math.min(currentPage * pageSize, totalItems);

  return { totalItems, totalPages, startItem, endItem };
}, [sortedData.length, currentPage, pageSize]);
```

```
  // 🎯 Stable callback functions
  const handleSort = useCallback((key) => {
    setSortConfig((prev) => ({
      key,
      direction: prev.key === key && prev.direction === "asc" ? "desc" : "asc",
    }));
  }, []);

  const handleFilter = useCallback((text) => {
    setFilterText(text);
    setCurrentPage(1); // Reset to first page when filtering
  }, []);

  const handlePageChange = useCallback((page) => {
    setCurrentPage(page);
  }, []);

  const handlePageSizeChange = useCallback((size) => {
    setPageSize(size);
    setCurrentPage(1); // Reset to first page when changing page size
  }, []);

  const handleEdit = useCallback((id, field, value) => {
    setData((prev) =>
      prev.map((item) => (item.id === id ? { ...item, [field]: value } : item))
    );
  }, []);

  const handleDelete = useCallback((id) => {
    setData((prev) => prev.filter((item) => item.id !== id));
  }, []);

  return (
    <div className="optimized-data-table">
      <h3>Optimized Data Table</h3>

      {/* Controls */}
      <TableControls
        filterText={filterText}
        onFilter={handleFilter}
        pageSize={pageSize}
        onPageSizeChange={handlePageSizeChange}
        paginationInfo={paginationInfo}
      />

      {/* Table */}
      <DataTable
        data={paginatedData}
        sortConfig={sortConfig}
        onSort={handleSort}
        onEdit={handleEdit}
        onDelete={handleDelete}
      />
```

```
      {/* Pagination */}
      <Pagination
        currentPage={currentPage}
        totalPages={paginationInfo.totalPages}
        onPageChange={handlePageChange}
      />
    </div>
  );
}

// Memoized table controls
const TableControls = React.memo(
  ({ filterText, onFilter, pageSize, onPageSizeChange, paginationInfo }) => {
    console.log("TableControls rendered");

    return (
      <div className="table-controls">
        <div className="filter-section">
          <input
            type="text"
            value={filterText}
            onChange={(e) => onFilter(e.target.value)}
            placeholder="Filter data..."
          />
        </div>

        <div className="page-size-section">
          <label>Items per page:</label>
          <select
            value={pageSize}
            onChange={(e) => onPageSizeChange(Number(e.target.value))}
          >
            <option value={5}>5</option>
            <option value={10}>10</option>
            <option value={20}>20</option>
          </select>
        </div>

        <div className="info-section">
          Showing {paginationInfo.startItem}-{paginationInfo.endItem} of{" "}
          {paginationInfo.totalItems} items
        </div>
      </div>
    );
  }
);

// Memoized data table
const DataTable = React.memo(
  ({ data, sortConfig, onSort, onEdit, onDelete }) => {
    console.log("DataTable rendered");

    const columns = [
```

```
      { key: "name", label: "Name", sortable: true },
      { key: "age", label: "Age", sortable: true },
      { key: "department", label: "Department", sortable: true },
      { key: "salary", label: "Salary", sortable: true },
    ];

    return (
      <table className="data-table">
        <thead>
          <tr>
            {columns.map((column) => (
              <TableHeader
                key={column.key}
                column={column}
                sortConfig={sortConfig}
                onSort={onSort}
              />
            ))}
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          {data.map((item) => (
            <TableRow
              key={item.id}
              item={item}
              onEdit={onEdit}
              onDelete={onDelete}
            />
          ))}
        </tbody>
      </table>
    );
  }
);

// Memoized table header
const TableHeader = React.memo(({ column, sortConfig, onSort }) => {
  const handleClick = useCallback(() => {
    if (column.sortable) {
      onSort(column.key);
    }
  }, [column.key, column.sortable, onSort]);

  const getSortIcon = () => {
    if (sortConfig.key !== column.key) return "↕";
    return sortConfig.direction === "asc" ? "↑" : "↓";
  };

  return (
    <th className={column.sortable ? "sortable" : ""} onClick={handleClick}>
      {column.label}
      {column.sortable && <span className="sort-icon">{getSortIcon()}</span>}
    </th>
```

```
  );
});

// Memoized table row
const TableRow = React.memo(({ item, onEdit, onDelete }) => {
  console.log(`Rendering row for ${item.name}`);

  const [isEditing, setIsEditing] = useState(false);
  const [editData, setEditData] = useState(item);

  const handleEdit = useCallback(() => {
    setIsEditing(true);
    setEditData(item);
  }, [item]);

  const handleSave = useCallback(() => {
    Object.keys(editData).forEach((key) => {
      if (editData[key] !== item[key]) {
        onEdit(item.id, key, editData[key]);
      }
    });
    setIsEditing(false);
  }, [editData, item, onEdit]);

  const handleCancel = useCallback(() => {
    setEditData(item);
    setIsEditing(false);
  }, [item]);

  const handleDelete = useCallback(() => {
    if (window.confirm(`Delete ${item.name}?`)) {
      onDelete(item.id);
    }
  }, [item.id, item.name, onDelete]);

  if (isEditing) {
    return (
      <tr className="editing">
        <td>
          <input
            value={editData.name}
            onChange={(e) =>
              setEditData((prev) => ({ ...prev, name: e.target.value }))
            }
          />
        </td>
        <td>
          <input
            type="number"
            value={editData.age}
            onChange={(e) =>
              setEditData((prev) => ({ ...prev, age: Number(e.target.value) }))
            }
          />
```

```
          </td>
          <td>
            <select
              value={editData.department}
              onChange={(e) =>
                setEditData((prev) => ({ ...prev, department: e.target.value }))
              }
            >
              <option value="Engineering">Engineering</option>
              <option value="Design">Design</option>
              <option value="Marketing">Marketing</option>
            </select>
          </td>
          <td>
            <input
              type="number"
              value={editData.salary}
              onChange={(e) =>
                setEditData((prev) => ({
                  ...prev,
                  salary: Number(e.target.value),
                }))
              }
            />
          </td>
          <td>
            <button onClick={handleSave}>Save</button>
            <button onClick={handleCancel}>Cancel</button>
          </td>
        </tr>
      );
    }

    return (
      <tr>
        <td>{item.name}</td>
        <td>{item.age}</td>
        <td>{item.department}</td>
        <td>${item.salary.toLocaleString()}</td>
        <td>
          <button onClick={handleEdit}>Edit</button>
          <button onClick={handleDelete}>Delete</button>
        </td>
      </tr>
    );
  });

  // Memoized pagination
  const Pagination = React.memo(({ currentPage, totalPages, onPageChange }) => {
    console.log("Pagination rendered");

    const handlePrevious = useCallback(() => {
      if (currentPage > 1) {
        onPageChange(currentPage - 1);
```

```
    }
  }, [currentPage, onPageChange]);

  const handleNext = useCallback(() => {
    if (currentPage < totalPages) {
      onPageChange(currentPage + 1);
    }
  }, [currentPage, totalPages, onPageChange]);

  const handlePageClick = useCallback(
    (page) => {
      onPageChange(page);
    },
    [onPageChange]
  );

  const getPageNumbers = () => {
    const pages = [];
    const maxVisible = 5;

    let start = Math.max(1, currentPage - Math.floor(maxVisible / 2));
    let end = Math.min(totalPages, start + maxVisible - 1);

    if (end - start + 1 < maxVisible) {
      start = Math.max(1, end - maxVisible + 1);
    }

    for (let i = start; i <= end; i++) {
      pages.push(i);
    }

    return pages;
  };

  if (totalPages <= 1) return null;

  return (
    <div className="pagination">
      <button onClick={handlePrevious} disabled={currentPage === 1}>
        Previous
      </button>

      {getPageNumbers().map((page) => (
        <button
          key={page}
          onClick={() => handlePageClick(page)}
          className={page === currentPage ? "active" : ""}
        >
          {page}
        </button>
      ))}

      <button onClick={handleNext} disabled={currentPage === totalPages}>
        Next
```

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

Challenge 2: Real-time Chat Interface

Build a chat interface with message sending, typing indicators, and user presence that optimizes re-renders.

▶ 💡 Solution

```
function ChatInterface() {
  const [messages, setMessages] = useState([
    {
      id: 1,
      user: "Alice",
      text: "Hello everyone!",
      timestamp: Date.now() - 60000,
    },
    {
      id: 2,
      user: "Bob",
      text: "Hey Alice! How are you?",
      timestamp: Date.now() - 45000,
    },
    {
      id: 3,
      user: "Alice",
      text: "I'm doing great, thanks!",
      timestamp: Date.now() - 30000,
    },
  ]);

  const [currentUser, setCurrentUser] = useState("Charlie");
  const [newMessage, setNewMessage] = useState("");
  const [typingUsers, setTypingUsers] = useState([]);
  const [onlineUsers, setOnlineUsers] = useState([
    "Alice",
    "Bob",
    "Charlie",
    "Diana",
  ]);
  const [isConnected, setIsConnected] = useState(true);

  // 🎯 Stable message sending function
  const sendMessage = useCallback(
    (text) => {
      if (!text.trim()) return;

      const newMsg = {
        id: Date.now(),
```

```
        user: currentUser,
        text: text.trim(),
        timestamp: Date.now(),
      };

      setMessages((prev) => [...prev, newMsg]);
      setNewMessage("");

      // Simulate removing typing indicator
      setTypingUsers((prev) => prev.filter((user) => user !== currentUser));
    },
    [currentUser]
  );

  // 🎯 Stable typing handler
  const handleTyping = useCallback((user, isTyping) => {
    setTypingUsers((prev) => {
      if (isTyping) {
        return prev.includes(user) ? prev : [...prev, user];
      } else {
        return prev.filter((u) => u !== user);
      }
    });
  }, []);

  // 🎯 Stable user presence handler
  const handleUserPresence = useCallback((user, isOnline) => {
    setOnlineUsers((prev) => {
      if (isOnline) {
        return prev.includes(user) ? prev : [...prev, user];
      } else {
        return prev.filter((u) => u !== user);
      }
    });
  }, []);

  // 🎯 Stable message deletion
  const deleteMessage = useCallback((messageId) => {
    setMessages((prev) => prev.filter((msg) => msg.id !== messageId));
  }, []);

  // 🎯 Stable message editing
  const editMessage = useCallback((messageId, newText) => {
    setMessages((prev) =>
      prev.map((msg) =>
        msg.id === messageId ? { ...msg, text: newText, edited: true } : msg
      )
    );
  }, []);

  // 🎯 Simulate typing detection
  const typingTimeoutRef = useRef(null);

  const handleInputChange = useCallback(
```

```
      (value) => {
        setNewMessage(value);

        // Start typing indicator
        handleTyping(currentUser, true);

        // Clear existing timeout
        if (typingTimeoutRef.current) {
          clearTimeout(typingTimeoutRef.current);
        }

        // Stop typing after 1 second of inactivity
        typingTimeoutRef.current = setTimeout(() => {
          handleTyping(currentUser, false);
        }, 1000);
      },
      [currentUser, handleTyping]
    );

    // 🎯 Cleanup typing timeout
    useEffect(() => {
      return () => {
        if (typingTimeoutRef.current) {
          clearTimeout(typingTimeoutRef.current);
        }
      };
    }, []);

    return (
      <div className="chat-interface">
        <ChatHeader
          isConnected={isConnected}
          onlineUsers={onlineUsers}
          currentUser={currentUser}
          onUserChange={setCurrentUser}
        />

        <MessageList
          messages={messages}
          currentUser={currentUser}
          onDelete={deleteMessage}
          onEdit={editMessage}
        />

        <TypingIndicator
          typingUsers={typingUsers.filter((user) => user !== currentUser)}
        />

        <MessageInput
          value={newMessage}
          onChange={handleInputChange}
          onSend={sendMessage}
          disabled={!isConnected}
        />
```

```
    </div>
  );
}

// Memoized chat header
const ChatHeader = React.memo(
  ({ isConnected, onlineUsers, currentUser, onUserChange }) => {
    console.log("ChatHeader rendered");

    return (
      <div className="chat-header">
        <div className="connection-status">
          <span
            className={`status-indicator ${
              isConnected ? "connected" : "disconnected"
            }`}
          >
            {isConnected ? "🟢" : "🔴"}
          </span>
          {isConnected ? "Connected" : "Disconnected"}
        </div>

        <div className="user-selector">
          <label>You are:</label>
          <select
            value={currentUser}
            onChange={(e) => onUserChange(e.target.value)}
          >
            {onlineUsers.map((user) => (
              <option key={user} value={user}>
                {user}
              </option>
            ))}
          </select>
        </div>

        <div className="online-users">
          <span>Online: {onlineUsers.length}</span>
          <div className="user-list">
            {onlineUsers.map((user) => (
              <span key={user} className="online-user">
                🟢 {user}
              </span>
            ))}
          </div>
        </div>
      </div>
    );
  }
);

// Memoized message list
const MessageList = React.memo(
  ({ messages, currentUser, onDelete, onEdit }) => {
```

```
    console.log("MessageList rendered");

    const messagesEndRef = useRef(null);

    // Auto-scroll to bottom
    useEffect(() => {
      messagesEndRef.current?.scrollIntoView({ behavior: "smooth" });
    }, [messages]);

    return (
      <div className="message-list">
        {messages.map((message) => (
          <Message
            key={message.id}
            message={message}
            isOwn={message.user === currentUser}
            onDelete={onDelete}
            onEdit={onEdit}
          />
        ))}
        <div ref={messagesEndRef} />
      </div>
    );
  }
);

// Memoized individual message
const Message = React.memo(({ message, isOwn, onDelete, onEdit }) => {
  console.log(`Rendering message ${message.id}`);

  const [isEditing, setIsEditing] = useState(false);
  const [editText, setEditText] = useState(message.text);

  const handleEdit = useCallback(() => {
    setIsEditing(true);
    setEditText(message.text);
  }, [message.text]);

  const handleSave = useCallback(() => {
    if (editText.trim() && editText !== message.text) {
      onEdit(message.id, editText.trim());
    }
    setIsEditing(false);
  }, [editText, message.id, message.text, onEdit]);

  const handleCancel = useCallback(() => {
    setEditText(message.text);
    setIsEditing(false);
  }, [message.text]);

  const handleDelete = useCallback(() => {
    if (window.confirm("Delete this message?")) {
      onDelete(message.id);
    }
```

```
  }, [message.id, onDelete]);

  const formatTime = (timestamp) => {
    return new Date(timestamp).toLocaleTimeString([], {
      hour: "2-digit",
      minute: "2-digit",
    });
  };

  return (
    <div className={`message ${isOwn ? "own" : "other"}`}>
      <div className="message-header">
        <span className="username">{message.user}</span>
        <span className="timestamp">{formatTime(message.timestamp)}</span>
        {message.edited && <span className="edited-indicator">(edited)</span>}
      </div>

      <div className="message-content">
        {isEditing ? (
          <div className="edit-form">
            <textarea
              value={editText}
              onChange={(e) => setEditText(e.target.value)}
              onKeyDown={(e) => {
                if (e.key === "Enter" && !e.shiftKey) {
                  e.preventDefault();
                  handleSave();
                } else if (e.key === "Escape") {
                  handleCancel();
                }
              }}
            />
            <div className="edit-actions">
              <button onClick={handleSave}>Save</button>
              <button onClick={handleCancel}>Cancel</button>
            </div>
          </div>
        ) : (
          <div className="message-text">{message.text}</div>
        )}
      </div>

      {isOwn && !isEditing && (
        <div className="message-actions">
          <button onClick={handleEdit}>Edit</button>
          <button onClick={handleDelete}>Delete</button>
        </div>
      )}
    </div>
  );
});

// Memoized typing indicator
const TypingIndicator = React.memo(({ typingUsers }) => {
```

```
  console.log("TypingIndicator rendered");

  if (typingUsers.length === 0) return null;

  const getTypingText = () => {
    if (typingUsers.length === 1) {
      return `${typingUsers[0]} is typing...`;
    } else if (typingUsers.length === 2) {
      return `${typingUsers[0]} and ${typingUsers[1]} are typing...`;
    } else {
      return `${typingUsers.length} people are typing...`;
    }
  };

  return (
    <div className="typing-indicator">
      <span className="typing-text">{getTypingText()}</span>
      <span className="typing-dots">
        <span>.</span>
        <span>.</span>
        <span>.</span>
      </span>
    </div>
  );
});

// Memoized message input
const MessageInput = React.memo(({ value, onChange, onSend, disabled }) => {
  console.log("MessageInput rendered");

  const handleSubmit = useCallback(
    (e) => {
      e.preventDefault();
      onSend(value);
    },
    [value, onSend]
  );

  const handleKeyDown = useCallback(
    (e) => {
      if (e.key === "Enter" && !e.shiftKey) {
        e.preventDefault();
        onSend(value);
      }
    },
    [value, onSend]
  );

  return (
    <form className="message-input" onSubmit={handleSubmit}>
      <textarea
        value={value}
        onChange={(e) => onChange(e.target.value)}
        onKeyDown={handleKeyDown}
```

```
        placeholder={disabled ? "Disconnected..." : "Type a message..."}
        disabled={disabled}
        rows={3}
      />
      <button type="submit" disabled={disabled || !value.trim()}>
        Send
      </button>
    </form>
  );
});
```

🎯 When and Why: useCallback Decision Framework

Quick Decision Tree

```
🤔 Should I use useCallback?

├── Is this function passed to a child component?
│   ├── Yes → Continue evaluation...
│   └── No → Probably don't need useCallback ❌
│
├── Is the child component wrapped in React.memo?
│   ├── Yes → Use useCallback ✅
│   └── No → Continue evaluation...
│
├── Is this function used in a dependency array?
│   ├── Yes → Use useCallback ✅
│   └── No → Continue evaluation...
│
├── Is this an expensive function to recreate?
│   ├── Yes → Use useCallback ✅
│   └── No → Continue evaluation...
│
└── Does the function have complex dependencies?
    ├── Yes → Consider useCallback ⚠️
    └── No → Probably don't need useCallback ❌
```

Performance Guidelines

```
// 🎯 When TO use useCallback:

// ✅ Passed to memoized children
const ExpensiveChild = React.memo(({ onClick }) => {
  /* ... */
});
const handleClick = useCallback(() => {
  /* ... */
}, []);
<ExpensiveChild onClick={handleClick} />;
```

```
// ✅ Used in dependency arrays
const fetchData = useCallback(async () => {
  /* ... */
}, [userId]);
useEffect(() => {
  fetchData();
}, [fetchData]);

// ✅ Event handlers with parameters
const handleItemClick = useCallback((id) => {
  /* ... */
}, []);

// ✅ Custom hook callbacks
const debouncedCallback = useDebounceCallback(callback, 300);

// 🎯 When NOT to use useCallback:

// ❌ Simple event handlers not passed to children
const handleClick = () => setCount((c) => c + 1); // Don't memo

// ❌ Functions that change on every render anyway
const handleClick = useCallback(() => {
  console.log(Math.random()); // Dependencies change constantly
}, [Math.random()]);

// ❌ More overhead than benefit
const simple = useCallback(() => a + b, [a, b]); // Too simple
```

Interview Insights

Common Interview Questions

1. **"What's the difference between useCallback and useMemo?"**
   - useCallback memoizes functions, useMemo memoizes values
   - useCallback(fn, deps) === useMemo(() => fn, deps)
   - Show practical examples of when to use each
2. **"When would you use useCallback?"**
   - Preventing unnecessary re-renders of memoized children
   - Stabilizing function references for dependency arrays
   - Optimizing expensive function creation
   - Custom hooks that return callbacks
3. **"How does useCallback help with React.memo?"**
   - React.memo does shallow comparison of props
   - Functions are recreated on every render (new reference)
   - useCallback provides stable function reference

- Prevents unnecessary child re-renders

4. **"What are the trade-offs of useCallback?"**
   - Memory overhead (storing function reference)
   - Comparison overhead (checking dependencies)
   - Code complexity
   - Can prevent garbage collection if overused

Code Review Red Flags

```
// 🚩 Red Flags in Interviews:

// ❌ Using useCallback everywhere
const simple = useCallback(() => setCount((c) => c + 1), []);

// ❌ Missing dependencies
const handler = useCallback(() => {
  console.log(name); // Uses 'name' but not in deps
}, []); // Missing 'name'

// ❌ Expensive dependencies
const handler = useCallback(() => {
  process(data);
}, [data.map((x) => x.id)]); // Creates new array every time

// ❌ Not using with React.memo
const Child = ({ onClick }) => <button onClick={onClick}>Click</button>;
const handler = useCallback(() => {}, []); // Useless without memo

// ❌ Inline object creation
<Child
  onClick={useCallback(() => {}, [])}
  config={{ theme: "dark" }} // This breaks memoization anyway!
/>;
```

🎯 Key Takeaways

Mental Model

```
// 🧠 Think of useCallback as:

// "Give me the same function reference unless dependencies change"

const stableFunction = useCallback(() => {
  // This function object stays the same
  // unless dependencies in the array change
}, [dependencies]);
```

Best Practices Summary

1. **Use with React.memo** - Primary use case for optimization
2. **Stabilize dependencies** - For useEffect and other hooks
3. **Don't overuse** - Has overhead, use judiciously
4. **Include all dependencies** - Avoid stale closures
5. **Consider alternatives** - Sometimes restructuring is better
6. **Profile performance** - Measure actual impact
7. **Use ESLint rules** - Catch missing dependencies

Production Tips

- **Start without useCallback** - Add only when needed
- **Use React DevTools Profiler** - Identify actual bottlenecks
- **Combine with useMemo** - For complete optimization
- **Watch for stale closures** - Common bug with missing deps
- **Document optimization intent** - Help future developers

Next up: useContext: Global State Management - Master React's built-in state management solution.

Previous: useMemo: What, When, Why

💡 Pro tip: In interviews, demonstrate that you understand useCallback is an optimization tool, not a requirement. Show that you can identify when it's beneficial vs. when it adds unnecessary complexity.

useContext: Global State Management

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What React Context is and when to use it
- How to create and consume context effectively
- Context vs prop drilling solutions
- Performance implications and optimization strategies
- Common patterns and anti-patterns
- Real-world state management scenarios

🧠 What is useContext?

`useContext` is a React Hook that allows you to **consume context values** without wrapping your component in a Context Consumer. It provides a way to share data across the component tree without passing props down manually at every level.

The Problem: Prop Drilling

```jsx
// ❌ Problem: Prop drilling through multiple levels
function App() {
  const [user, setUser] = useState({ name: "Alice", theme: "dark" });

  return (
    <div>
      <Header user={user} />
      <Main user={user} setUser={setUser} />
      <Footer user={user} />
    </div>
  );
}

function Header({ user }) {
  return (
    <header>
      <Navigation user={user} />
    </header>
  );
}

function Navigation({ user }) {
  return (
    <nav>
      <UserProfile user={user} />
    </nav>
  );
}

function UserProfile({ user }) {
  return <span>Welcome, {user.name}!</span>;
}

function Main({ user, setUser }) {
  return (
    <main>
      <Content user={user} setUser={setUser} />
    </main>
  );
}

function Content({ user, setUser }) {
  return (
    <div>
      <Settings user={user} setUser={setUser} />
    </div>
  );
}

function Settings({ user, setUser }) {
  return (
    <div>
      <ThemeToggle user={user} setUser={setUser} />
```

```
    </div>
  );
}

function ThemeToggle({ user, setUser }) {
  const toggleTheme = () => {
    setUser((prev) => ({
      ...prev,
      theme: prev.theme === "dark" ? "light" : "dark",
    }));
  };

  return <button onClick={toggleTheme}>Current theme: {user.theme}</button>;
}
```

The Solution: Context

```
// ✅ Solution: Context eliminates prop drilling
import { createContext, useContext, useState } from "react";

// 🎯 Create context
const UserContext = createContext();

// 🎯 Context provider component
function UserProvider({ children }) {
  const [user, setUser] = useState({ name: "Alice", theme: "dark" });

  const value = {
    user,
    setUser,
    toggleTheme: () => {
      setUser((prev) => ({
        ...prev,
        theme: prev.theme === "dark" ? "light" : "dark",
      }));
    },
  };

  return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
}

// 🎯 Custom hook for consuming context
function useUser() {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error("useUser must be used within a UserProvider");
  }
  return context;
}

// 🎯 Clean component tree
```

```
function App() {
  return (
    <UserProvider>
      <div>
        <Header />
        <Main />
        <Footer />
      </div>
    </UserProvider>
  );
}

function Header() {
  return (
    <header>
      <Navigation />
    </header>
  );
}

function Navigation() {
  return (
    <nav>
      <UserProfile />
    </nav>
  );
}

function UserProfile() {
  const { user } = useUser(); // Direct access!
  return <span>Welcome, {user.name}!</span>;
}

function Main() {
  return (
    <main>
      <Content />
    </main>
  );
}

function Content() {
  return (
    <div>
      <Settings />
    </div>
  );
}

function Settings() {
  return (
    <div>
      <ThemeToggle />
    </div>
```

```
  );
}

function ThemeToggle() {
  const { user, toggleTheme } = useUser(); // Direct access!

  return <button onClick={toggleTheme}>Current theme: {user.theme}</button>;
}
```

🔍 Deep Dive: Context Fundamentals

Creating Context

```
import { createContext } from "react";

// 🎯 Basic context creation
const MyContext = createContext();

// 🎯 Context with default value
const ThemeContext = createContext("light");

// 🎯 Context with complex default value
const AppContext = createContext({
  user: null,
  theme: "light",
  language: "en",
  updateUser: () => {},
  toggleTheme: () => {},
  setLanguage: () => {},
});

// 🎯 Context with TypeScript (if using TypeScript)
// interface UserContextType {
//   user: User | null;
//   login: (user: User) => void;
//   logout: () => void;
// }
// const UserContext = createContext<UserContextType | undefined>(undefined);
```

Provider Pattern

```
function ContextDemo() {
  // 🎯 Multiple context providers
  return (
    <ThemeProvider>
      <UserProvider>
        <LanguageProvider>
          <App />
        </LanguageProvider>
```

```
      </UserProvider>
    </ThemeProvider>
  );
}

// 🎯 Theme context
const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const [primaryColor, setPrimaryColor] = useState("#007bff");

  const value = {
    theme,
    primaryColor,
    setTheme,
    setPrimaryColor,
    toggleTheme: () =>
      setTheme((prev) => (prev === "light" ? "dark" : "light")),
    isDark: theme === "dark",
  };

  return (
    <ThemeContext.Provider value={value}>{children}</ThemeContext.Provider>
  );
}

// 🎯 User context
const UserContext = createContext();

function UserProvider({ children }) {
  const [user, setUser] = useState(null);
  const [isLoading, setIsLoading] = useState(false);

  const login = async (credentials) => {
    setIsLoading(true);
    try {
      const response = await fetch("/api/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(credentials),
      });
      const userData = await response.json();
      setUser(userData);
    } catch (error) {
      console.error("Login failed:", error);
    } finally {
      setIsLoading(false);
    }
  };

  const logout = () => {
    setUser(null);
    localStorage.removeItem("token");
```

```
  };

  const value = {
    user,
    isLoading,
    login,
    logout,
    isAuthenticated: !!user,
  };

  return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
}

// 🎯 Language context
const LanguageContext = createContext();

function LanguageProvider({ children }) {
  const [language, setLanguage] = useState("en");
  const [translations, setTranslations] = useState({});

  const loadTranslations = async (lang) => {
    try {
      const response = await fetch(`/api/translations/${lang}`);
      const data = await response.json();
      setTranslations(data);
    } catch (error) {
      console.error("Failed to load translations:", error);
    }
  };

  const translate = (key, fallback = key) => {
    return translations[key] || fallback;
  };

  const value = {
    language,
    setLanguage,
    translate,
    loadTranslations,
  };

  return (
    <LanguageContext.Provider value={value}>
      {children}
    </LanguageContext.Provider>
  );
}
```

Custom Hooks for Context

```
// 🎯 Custom hooks with error handling
function useTheme() {
  const context = useContext(ThemeContext);
  if (context === undefined) {
    throw new Error("useTheme must be used within a ThemeProvider");
  }
  return context;
}

function useUser() {
  const context = useContext(UserContext);
  if (context === undefined) {
    throw new Error("useUser must be used within a UserProvider");
  }
  return context;
}

function useLanguage() {
  const context = useContext(LanguageContext);
  if (context === undefined) {
    throw new Error("useLanguage must be used within a LanguageProvider");
  }
  return context;
}

// 🎯 Selective context consumption
function useThemeActions() {
  const { setTheme, setPrimaryColor, toggleTheme } = useTheme();
  return { setTheme, setPrimaryColor, toggleTheme };
}

function useThemeValues() {
  const { theme, primaryColor, isDark } = useTheme();
  return { theme, primaryColor, isDark };
}

function useAuth() {
  const { user, isAuthenticated, login, logout } = useUser();
  return { user, isAuthenticated, login, logout };
}
```

🚀 Real-World Patterns

Pattern 1: Shopping Cart Context

```
const CartContext = createContext();

function CartProvider({ children }) {
  const [items, setItems] = useState([]);
  const [isOpen, setIsOpen] = useState(false);
```

```
// 🎯 Cart calculations
const totalItems = useMemo(() => {
  return items.reduce((sum, item) => sum + item.quantity, 0);
}, [items]);

const totalPrice = useMemo(() => {
  return items.reduce((sum, item) => sum + item.price * item.quantity, 0);
}, [items]);

// 🎯 Cart actions
const addItem = useCallback((product) => {
  setItems((prev) => {
    const existingItem = prev.find((item) => item.id === product.id);

    if (existingItem) {
      return prev.map((item) =>
        item.id === product.id
          ? { ...item, quantity: item.quantity + 1 }
          : item
      );
    }

    return [...prev, { ...product, quantity: 1 }];
  });
}, []);

const removeItem = useCallback((productId) => {
  setItems((prev) => prev.filter((item) => item.id !== productId));
}, []);

const updateQuantity = useCallback(
  (productId, quantity) => {
    if (quantity <= 0) {
      removeItem(productId);
      return;
    }

    setItems((prev) =>
      prev.map((item) =>
        item.id === productId ? { ...item, quantity } : item
      )
    );
  },
  [removeItem]
);

const clearCart = useCallback(() => {
  setItems([]);
}, []);

const toggleCart = useCallback(() => {
  setIsOpen((prev) => !prev);
}, []);
```

```
  // 🎯 Persist cart to localStorage
  useEffect(() => {
    const savedCart = localStorage.getItem("cart");
    if (savedCart) {
      try {
        setItems(JSON.parse(savedCart));
      } catch (error) {
        console.error("Failed to load cart from localStorage:", error);
      }
    }
  }, []);

  useEffect(() => {
    localStorage.setItem("cart", JSON.stringify(items));
  }, [items]);

  const value = {
    items,
    totalItems,
    totalPrice,
    isOpen,
    addItem,
    removeItem,
    updateQuantity,
    clearCart,
    toggleCart,
  };

  return <CartContext.Provider value={value}>{children}</CartContext.Provider>;
}

function useCart() {
  const context = useContext(CartContext);
  if (!context) {
    throw new Error("useCart must be used within a CartProvider");
  }
  return context;
}

// 🎯 Usage examples
function ProductCard({ product }) {
  const { addItem } = useCart();

  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <p>${product.price}</p>
      <button onClick={() => addItem(product)}>Add to Cart</button>
    </div>
  );
}

function CartIcon() {
```

```
  const { totalItems, toggleCart } = useCart();

  return (
    <button className="cart-icon" onClick={toggleCart}>
      🛒 {totalItems > 0 && <span className="badge">{totalItems}</span>}
    </button>
  );
}

function CartSidebar() {
  const { items, totalPrice, isOpen, updateQuantity, removeItem, clearCart } =
    useCart();

  if (!isOpen) return null;

  return (
    <div className="cart-sidebar">
      <h2>Shopping Cart</h2>

      {items.length === 0 ? (
        <p>Your cart is empty</p>
      ) : (
        <>
          {items.map((item) => (
            <div key={item.id} className="cart-item">
              <h4>{item.name}</h4>
              <p>${item.price} each</p>
              <div className="quantity-controls">
                <button
                  onClick={() => updateQuantity(item.id, item.quantity - 1)}
                >
                  -
                </button>
                <span>{item.quantity}</span>
                <button
                  onClick={() => updateQuantity(item.id, item.quantity + 1)}
                >
                  +
                </button>
              </div>
              <button onClick={() => removeItem(item.id)}>Remove</button>
            </div>
          ))}

          <div className="cart-total">
            <strong>Total: ${totalPrice.toFixed(2)}</strong>
          </div>

          <div className="cart-actions">
            <button onClick={clearCart}>Clear Cart</button>
            <button className="checkout-btn">Checkout</button>
          </div>
        </>
      )}
```

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

Pattern 2: Notification System

```
const NotificationContext = createContext();

function NotificationProvider({ children }) {
  const [notifications, setNotifications] = useState([]);

  const addNotification = useCallback((notification) => {
    const id = Date.now() + Math.random();
    const newNotification = {
      id,
      type: "info",
      duration: 5000,
      ...notification,
    };

    setNotifications((prev) => [...prev, newNotification]);

    // Auto-remove after duration
    if (newNotification.duration > 0) {
      setTimeout(() => {
        removeNotification(id);
      }, newNotification.duration);
    }
  }, []);

  const removeNotification = useCallback((id) => {
    setNotifications((prev) =>
      prev.filter((notification) => notification.id !== id)
    );
  }, []);

  const clearAll = useCallback(() => {
    setNotifications([]);
  }, []);

  // 🎯 Convenience methods
  const showSuccess = useCallback(
    (message, options = {}) => {
      addNotification({ type: "success", message, ...options });
    },
    [addNotification]
  );

  const showError = useCallback(
    (message, options = {}) => {
      addNotification({ type: "error", message, duration: 0, ...options });
```

```
    },
    [addNotification]
  );

  const showWarning = useCallback(
    (message, options = {}) => {
      addNotification({ type: "warning", message, ...options });
    },
    [addNotification]
  );

  const showInfo = useCallback(
    (message, options = {}) => {
      addNotification({ type: "info", message, ...options });
    },
    [addNotification]
  );

  const value = {
    notifications,
    addNotification,
    removeNotification,
    clearAll,
    showSuccess,
    showError,
    showWarning,
    showInfo,
  };

  return (
    <NotificationContext.Provider value={value}>
      {children}
      <NotificationContainer />
    </NotificationContext.Provider>
  );
}

function useNotifications() {
  const context = useContext(NotificationContext);
  if (!context) {
    throw new Error(
      "useNotifications must be used within a NotificationProvider"
    );
  }
  return context;
}

function NotificationContainer() {
  const { notifications } = useNotifications();

  return (
    <div className="notification-container">
      {notifications.map((notification) => (
        <Notification key={notification.id} notification={notification} />
```

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

  function Notification({ notification }) {
    const { removeNotification } = useNotifications();
    const [isVisible, setIsVisible] = useState(false);

    useEffect(() => {
      // Trigger animation
      const timer = setTimeout(() => setIsVisible(true), 10);
      return () => clearTimeout(timer);
    }, []);

    const handleClose = () => {
      setIsVisible(false);
      setTimeout(() => removeNotification(notification.id), 300);
    };

    const getIcon = () => {
      switch (notification.type) {
        case "success":
          return "✅";
        case "error":
          return "❌";
        case "warning":
          return "⚠️";
        default:
          return "ℹ️";
      }
    };

    return (
      <div
        className={`notification notification-${notification.type} ${
          isVisible ? "visible" : ""
        }`}
      >
        <span className="notification-icon">{getIcon()}</span>
        <span className="notification-message">{notification.message}</span>
        <button className="notification-close" onClick={handleClose}>
          ×
        </button>
      </div>
    );
  }

  // 🎯 Usage examples
  function LoginForm() {
    const { showSuccess, showError } = useNotifications();
    const [credentials, setCredentials] = useState({ email: "", password: "" });

    const handleSubmit = async (e) => {
```

```
    e.preventDefault();

    try {
      const response = await fetch("/api/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(credentials),
      });

      if (response.ok) {
        showSuccess("Login successful! Welcome back.");
      } else {
        showError("Login failed. Please check your credentials.");
      }
    } catch (error) {
      showError("Network error. Please try again.");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={credentials.email}
        onChange={(e) =>
          setCredentials((prev) => ({ ...prev, email: e.target.value }))
        }
        placeholder="Email"
      />
      <input
        type="password"
        value={credentials.password}
        onChange={(e) =>
          setCredentials((prev) => ({ ...prev, password: e.target.value }))
        }
        placeholder="Password"
      />
      <button type="submit">Login</button>
    </form>
  );
}
```

Pattern 3: Modal Management

```
const ModalContext = createContext();

function ModalProvider({ children }) {
  const [modals, setModals] = useState([]);

  const openModal = useCallback((modalConfig) => {
    const id = Date.now() + Math.random();
```

```
    const modal = {
      id,
      ...modalConfig,
    };

    setModals((prev) => [...prev, modal]);
    return id;
  }, []);

  const closeModal = useCallback((id) => {
    setModals((prev) => prev.filter((modal) => modal.id !== id));
  }, []);

  const closeAllModals = useCallback(() => {
    setModals([]);
  }, []);

  const updateModal = useCallback((id, updates) => {
    setModals((prev) =>
      prev.map((modal) => (modal.id === id ? { ...modal, ...updates } : modal))
    );
  }, []);

  // 🎯 Convenience methods
  const confirm = useCallback(
    (message, options = {}) => {
      return new Promise((resolve) => {
        const id = openModal({
          type: "confirm",
          message,
          onConfirm: () => {
            closeModal(id);
            resolve(true);
          },
          onCancel: () => {
            closeModal(id);
            resolve(false);
          },
          ...options,
        });
      });
    },
    [openModal, closeModal]
  );

  const alert = useCallback(
    (message, options = {}) => {
      return new Promise((resolve) => {
        const id = openModal({
          type: "alert",
          message,
          onClose: () => {
            closeModal(id);
            resolve();
```

```
          },
          ...options,
        });
      });
    },
    [openModal, closeModal]
  );

  const prompt = useCallback(
    (message, defaultValue = "", options = {}) => {
      return new Promise((resolve) => {
        const id = openModal({
          type: "prompt",
          message,
          defaultValue,
          onSubmit: (value) => {
            closeModal(id);
            resolve(value);
          },
          onCancel: () => {
            closeModal(id);
            resolve(null);
          },
          ...options,
        });
      });
    },
    [openModal, closeModal]
  );

  const value = {
    modals,
    openModal,
    closeModal,
    closeAllModals,
    updateModal,
    confirm,
    alert,
    prompt,
  };

  return (
    <ModalContext.Provider value={value}>
      {children}
      <ModalContainer />
    </ModalContext.Provider>
  );
}

function useModal() {
  const context = useContext(ModalContext);
  if (!context) {
    throw new Error("useModal must be used within a ModalProvider");
  }
```

```
  return context;
}

function ModalContainer() {
  const { modals } = useModal();

  if (modals.length === 0) return null;

  return (
    <div className="modal-overlay">
      {modals.map((modal) => (
        <Modal key={modal.id} modal={modal} />
      ))}
    </div>
  );
}

function Modal({ modal }) {
  const { closeModal } = useModal();
  const [inputValue, setInputValue] = useState(modal.defaultValue || "");

  const handleBackdropClick = (e) => {
    if (e.target === e.currentTarget && modal.closeOnBackdrop !== false) {
      closeModal(modal.id);
    }
  };

  const handleKeyDown = (e) => {
    if (e.key === "Escape" && modal.closeOnEscape !== false) {
      closeModal(modal.id);
    }
  };

  useEffect(() => {
    document.addEventListener("keydown", handleKeyDown);
    return () => document.removeEventListener("keydown", handleKeyDown);
  }, []);

  const renderContent = () => {
    switch (modal.type) {
      case "confirm":
        return (
          <div className="modal-content">
            <h3>{modal.title || "Confirm"}</h3>
            <p>{modal.message}</p>
            <div className="modal-actions">
              <button onClick={modal.onCancel}>Cancel</button>
              <button onClick={modal.onConfirm} className="primary">
                Confirm
              </button>
            </div>
          </div>
        );
```

```
        case "alert":
          return (
            <div className="modal-content">
              <h3>{modal.title || "Alert"}</h3>
              <p>{modal.message}</p>
              <div className="modal-actions">
                <button onClick={modal.onClose} className="primary">
                  OK
                </button>
              </div>
            </div>
          );

        case "prompt":
          return (
            <div className="modal-content">
              <h3>{modal.title || "Input Required"}</h3>
              <p>{modal.message}</p>
              <input
                type="text"
                value={inputValue}
                onChange={(e) => setInputValue(e.target.value)}
                placeholder={modal.placeholder}
                autoFocus
              />
              <div className="modal-actions">
                <button onClick={modal.onCancel}>Cancel</button>
                <button
                  onClick={() => modal.onSubmit(inputValue)}
                  className="primary"
                >
                  Submit
                </button>
              </div>
            </div>
          );

        default:
          return modal.content;
      }
    };

    return (
      <div className="modal-backdrop" onClick={handleBackdropClick}>
        <div className="modal">{renderContent()}</div>
      </div>
    );
  }

  // 🎯 Usage examples
  function DeleteButton({ itemId, itemName }) {
    const { confirm } = useModal();

    const handleDelete = async () => {
```

```
    const confirmed = await confirm(
      `Are you sure you want to delete "${itemName}"? This action cannot be
undone.`,
      { title: "Delete Item" }
    );

    if (confirmed) {
      // Proceed with deletion
      console.log("Deleting item:", itemId);
    }
  };

  return (
    <button onClick={handleDelete} className="delete-btn">
      Delete
    </button>
  );
}

function RenameButton({ itemId, currentName }) {
  const { prompt } = useModal();

  const handleRename = async () => {
    const newName = await prompt("Enter a new name:", currentName, {
      title: "Rename Item",
      placeholder: "Item name",
    });

    if (newName && newName !== currentName) {
      // Proceed with rename
      console.log("Renaming item:", itemId, "to:", newName);
    }
  };

  return <button onClick={handleRename}>Rename</button>;
}
```

⚠ Common Mistakes and Anti-Patterns

Mistake 1: Context for Everything

```
// ❌ Bad: Using context for local state
function BadExample() {
  // Don't create context for state that's only used in one component
  const CountContext = createContext();

  function CountProvider({ children }) {
    const [count, setCount] = useState(0);
    return (
      <CountContext.Provider value={{ count, setCount }}>
        {children}
```

```
      </CountContext.Provider>
    );
  }

  // This is overkill for local state
  return (
    <CountProvider>
      <Counter />
    </CountProvider>
  );
}

// ✅ Good: Use local state for local concerns
function GoodExample() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>Increment</button>
    </div>
  );
}
```

Mistake 2: Not Memoizing Context Values

```
// ❌ Bad: Creating new objects on every render
function BadProvider({ children }) {
  const [user, setUser] = useState(null);

  // This creates a new object on every render!
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}

// ✅ Good: Memoize the context value
function GoodProvider({ children }) {
  const [user, setUser] = useState(null);

  const value = useMemo(
    () => ({
      user,
      setUser,
    }),
    [user]
  );

  return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
```

```
}

// ✅ Even better: Separate stable and changing values
function BetterProvider({ children }) {
  const [user, setUser] = useState(null);

  const actions = useMemo(
    () => ({
      setUser,
      login: async (credentials) => {
        /* ... */
      },
      logout: () => setUser(null),
    }),
    []
  ); // Stable functions

  const state = useMemo(
    () => ({
      user,
      isAuthenticated: !!user,
    }),
    [user]
  ); // Changing state

  const value = useMemo(
    () => ({
      ...state,
      ...actions,
    }),
    [state, actions]
  );

  return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
}
```

Mistake 3: Too Many Contexts

```
// ❌ Bad: Separate context for every piece of state
function BadApp() {
  return (
    <UserProvider>
      <ThemeProvider>
        <LanguageProvider>
          <NotificationProvider>
            <CartProvider>
              <ModalProvider>
                <LoadingProvider>
                  <ErrorProvider>
                    <App />
                  </ErrorProvider>
```

```
                </LoadingProvider>
              </ModalProvider>
            </CartProvider>
          </NotificationProvider>
        </LanguageProvider>
      </ThemeProvider>
    </UserProvider>
  );
}

// ✅ Good: Combine related contexts
function GoodApp() {
  return (
    <AppProvider>
      {" "}
      {/* Combines user, theme, language */}
      <UIProvider>
        {" "}
        {/* Combines notifications, modals, loading */}
        <ShoppingProvider>
          {" "}
          {/* Combines cart, wishlist, orders */}
          <App />
        </ShoppingProvider>
      </UIProvider>
    </AppProvider>
  );
}

// ✅ Combined context example
function AppProvider({ children }) {
  const [user, setUser] = useState(null);
  const [theme, setTheme] = useState("light");
  const [language, setLanguage] = useState("en");

  const value = useMemo(
    () => ({
      // User state
      user,
      setUser,
      isAuthenticated: !!user,

      // Theme state
      theme,
      setTheme,
      toggleTheme: () =>
        setTheme((prev) => (prev === "light" ? "dark" : "light")),

      // Language state
      language,
      setLanguage,
    }),
    [user, theme, language]
  );
```

```
  return <AppContext.Provider value={value}>{children}</AppContext.Provider>;
}
```

Mistake 4: Context Performance Issues

```
// ❌ Bad: Single context with frequently changing values
function BadPerformanceProvider({ children }) {
  const [user, setUser] = useState(null);
  const [mousePosition, setMousePosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMouseMove = (e) => {
      setMousePosition({ x: e.clientX, y: e.clientY }); // Causes all consumers to
re-render!
    };

    window.addEventListener("mousemove", handleMouseMove);
    return () => window.removeEventListener("mousemove", handleMouseMove);
  }, []);

  const value = useMemo(
    () => ({
      user,
      setUser,
      mousePosition, // This changes constantly!
    }),
    [user, mousePosition]
  );

  return <AppContext.Provider value={value}>{children}</AppContext.Provider>;
}

// ✅ Good: Separate contexts for different update frequencies
function GoodPerformanceProvider({ children }) {
  return (
    <UserProvider>
      {" "}
      {/* Stable user data */}
      <MouseProvider>
        {" "}
        {/* Frequently changing mouse data */}
        {children}
      </MouseProvider>
    </UserProvider>
  );
}

function UserProvider({ children }) {
  const [user, setUser] = useState(null);
```

```
  const value = useMemo(
    () => ({
      user,
      setUser,
    }),
    [user]
  );

  return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
}

function MouseProvider({ children }) {
  const [mousePosition, setMousePosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMouseMove = (e) => {
      setMousePosition({ x: e.clientX, y: e.clientY });
    };

    window.addEventListener("mousemove", handleMouseMove);
    return () => window.removeEventListener("mousemove", handleMouseMove);
  }, []);

  return (
    <MouseContext.Provider value={mousePosition}>
      {children}
    </MouseContext.Provider>
  );
}
```

Mini-Challenges

Challenge 1: Multi-Step Form Context

Create a context for managing a multi-step form with validation, progress tracking, and data persistence.

▶ Solution

```
const FormContext = createContext();

function FormProvider({ children }) {
  const [currentStep, setCurrentStep] = useState(0);
  const [formData, setFormData] = useState({
    personal: { firstName: "", lastName: "", email: "", phone: "" },
    address: { street: "", city: "", state: "", zipCode: "", country: "" },
    preferences: { newsletter: false, notifications: true, theme: "light" },
  });
  const [errors, setErrors] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  const steps = [
```

```
    {
      id: "personal",
      title: "Personal Information",
      fields: ["firstName", "lastName", "email", "phone"],
    },
    {
      id: "address",
      title: "Address Information",
      fields: ["street", "city", "state", "zipCode", "country"],
    },
    {
      id: "preferences",
      title: "Preferences",
      fields: ["newsletter", "notifications", "theme"],
    },
    { id: "review", title: "Review & Submit", fields: [] },
  ];

  // 🎯 Validation rules
  const validationRules = {
    firstName: { required: true, minLength: 2 },
    lastName: { required: true, minLength: 2 },
    email: { required: true, pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/ },
    phone: { required: true, pattern: /^[\d\s\-\(\)\+]+$/ },
    street: { required: true, minLength: 5 },
    city: { required: true, minLength: 2 },
    state: { required: true, minLength: 2 },
    zipCode: { required: true, pattern: /^\d{5}(-\d{4})?$/ },
    country: { required: true },
  };

  // 🎯 Validate field
  const validateField = useCallback((field, value) => {
    const rules = validationRules[field];
    if (!rules) return null;

    if (rules.required && (!value || value.toString().trim() === "")) {
      return `${field} is required`;
    }

    if (rules.minLength && value.length < rules.minLength) {
      return `${field} must be at least ${rules.minLength} characters`;
    }

    if (rules.pattern && !rules.pattern.test(value)) {
      return `${field} format is invalid`;
    }

    return null;
  }, []);

  // 🎯 Validate step
  const validateStep = useCallback(
    (stepIndex) => {
```

```
      const step = steps[stepIndex];
      const stepErrors = {};

      step.fields.forEach((field) => {
        const section = Object.keys(formData).find((key) =>
          formData[key].hasOwnProperty(field)
        );

        if (section) {
          const error = validateField(field, formData[section][field]);
          if (error) {
            stepErrors[field] = error;
          }
        }
      });

      return stepErrors;
    },
    [formData, validateField, steps]
  );

  // 🎯 Update form data
  const updateFormData = useCallback((section, field, value) => {
    setFormData((prev) => ({
      ...prev,
      [section]: {
        ...prev[section],
        [field]: value,
      },
    }));

    // Clear error for this field
    setErrors((prev) => {
      const newErrors = { ...prev };
      delete newErrors[field];
      return newErrors;
    });
  }, []);

  // 🎯 Navigation
  const nextStep = useCallback(() => {
    const stepErrors = validateStep(currentStep);

    if (Object.keys(stepErrors).length > 0) {
      setErrors(stepErrors);
      return false;
    }

    setErrors({});
    setCurrentStep((prev) => Math.min(prev + 1, steps.length - 1));
    return true;
  }, [currentStep, validateStep, steps.length]);

  const prevStep = useCallback(() => {
```

```
    setCurrentStep((prev) => Math.max(prev - 1, 0));
    setErrors({});
  }, []);

  const goToStep = useCallback(
    (stepIndex) => {
      if (stepIndex >= 0 && stepIndex < steps.length) {
        setCurrentStep(stepIndex);
        setErrors({});
      }
    },
    [steps.length]
  );

  // 🎯 Submit form
  const submitForm = useCallback(async () => {
    setIsSubmitting(true);

    try {
      const response = await fetch("/api/submit-form", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(formData),
      });

      if (response.ok) {
        // Clear form data
        setFormData({
          personal: { firstName: "", lastName: "", email: "", phone: "" },
          address: {
            street: "",
            city: "",
            state: "",
            zipCode: "",
            country: "",
          },
          preferences: {
            newsletter: false,
            notifications: true,
            theme: "light",
          },
        });
        setCurrentStep(0);
        return { success: true };
      } else {
        return { success: false, error: "Submission failed" };
      }
    } catch (error) {
      return { success: false, error: error.message };
    } finally {
      setIsSubmitting(false);
    }
  }, [formData]);
```

```
  // 🎯 Progress calculation
  const progress = useMemo(() => {
    return ((currentStep + 1) / steps.length) * 100;
  }, [currentStep, steps.length]);

  // 🎯 Completion status
  const isStepComplete = useCallback(
    (stepIndex) => {
      const stepErrors = validateStep(stepIndex);
      return Object.keys(stepErrors).length === 0;
    },
    [validateStep]
  );

  // 🎯 Persist to localStorage
  useEffect(() => {
    const savedData = localStorage.getItem("multiStepForm");
    if (savedData) {
      try {
        const parsed = JSON.parse(savedData);
        setFormData(parsed.formData || formData);
        setCurrentStep(parsed.currentStep || 0);
      } catch (error) {
        console.error("Failed to load form data:", error);
      }
    }
  }, []);

  useEffect(() => {
    localStorage.setItem(
      "multiStepForm",
      JSON.stringify({
        formData,
        currentStep,
      })
    );
  }, [formData, currentStep]);

  const value = {
    // State
    currentStep,
    formData,
    errors,
    isSubmitting,
    steps,
    progress,

    // Actions
    updateFormData,
    nextStep,
    prevStep,
    goToStep,
    submitForm,
    validateField,
```

```
    validateStep,
    isStepComplete,
  };

  return <FormContext.Provider value={value}>{children}</FormContext.Provider>;
}

function useForm() {
  const context = useContext(FormContext);
  if (!context) {
    throw new Error("useForm must be used within a FormProvider");
  }
  return context;
}

// 🎯 Form components
function MultiStepForm() {
  const { currentStep, steps } = useForm();

  return (
    <div className="multi-step-form">
      <FormProgress />
      <FormSteps />
      <FormNavigation />
    </div>
  );
}

function FormProgress() {
  const { progress, currentStep, steps } = useForm();

  return (
    <div className="form-progress">
      <div className="progress-bar">
        <div className="progress-fill" style={{ width: `${progress}%` }} />
      </div>
      <div className="step-indicators">
        {steps.map((step, index) => (
          <StepIndicator key={step.id} step={step} index={index} />
        ))}
      </div>
    </div>
  );
}

function StepIndicator({ step, index }) {
  const { currentStep, goToStep, isStepComplete } = useForm();

  const isActive = index === currentStep;
  const isComplete = index < currentStep || isStepComplete(index);
  const isClickable = index <= currentStep;

  return (
    <div
```

```
        className={`step-indicator ${isActive ? "active" : ""} ${
          isComplete ? "complete" : ""
        } ${isClickable ? "clickable" : ""}`}
        onClick={() => isClickable && goToStep(index)}
      >
        <div className="step-number">{isComplete ? "✓" : index + 1}</div>
        <div className="step-title">{step.title}</div>
      </div>
    );
  }

  function FormSteps() {
    const { currentStep } = useForm();

    const renderStep = () => {
      switch (currentStep) {
        case 0:
          return <PersonalInfoStep />;
        case 1:
          return <AddressInfoStep />;
        case 2:
          return <PreferencesStep />;
        case 3:
          return <ReviewStep />;
        default:
          return null;
      }
    };

    return <div className="form-steps">{renderStep()}</div>;
  }

  function PersonalInfoStep() {
    const { formData, updateFormData, errors } = useForm();

    return (
      <div className="form-step">
        <h2>Personal Information</h2>

        <FormField
          label="First Name"
          value={formData.personal.firstName}
          onChange={(value) => updateFormData("personal", "firstName", value)}
          error={errors.firstName}
          required
        />

        <FormField
          label="Last Name"
          value={formData.personal.lastName}
          onChange={(value) => updateFormData("personal", "lastName", value)}
          error={errors.lastName}
          required
        />
```

```
        <FormField
          label="Email"
          type="email"
          value={formData.personal.email}
          onChange={(value) => updateFormData("personal", "email", value)}
          error={errors.email}
          required
        />

        <FormField
          label="Phone"
          type="tel"
          value={formData.personal.phone}
          onChange={(value) => updateFormData("personal", "phone", value)}
          error={errors.phone}
          required
        />
      </div>
    );
}

function AddressInfoStep() {
  const { formData, updateFormData, errors } = useForm();

  return (
    <div className="form-step">
      <h2>Address Information</h2>

      <FormField
        label="Street Address"
        value={formData.address.street}
        onChange={(value) => updateFormData("address", "street", value)}
        error={errors.street}
        required
      />

      <div className="form-row">
        <FormField
          label="City"
          value={formData.address.city}
          onChange={(value) => updateFormData("address", "city", value)}
          error={errors.city}
          required
        />

        <FormField
          label="State"
          value={formData.address.state}
          onChange={(value) => updateFormData("address", "state", value)}
          error={errors.state}
          required
        />
      </div>
```

```
        <div className="form-row">
          <FormField
            label="ZIP Code"
            value={formData.address.zipCode}
            onChange={(value) => updateFormData("address", "zipCode", value)}
            error={errors.zipCode}
            required
          />

          <FormField
            label="Country"
            type="select"
            value={formData.address.country}
            onChange={(value) => updateFormData("address", "country", value)}
            error={errors.country}
            options={[
              { value: "", label: "Select Country" },
              { value: "US", label: "United States" },
              { value: "CA", label: "Canada" },
              { value: "UK", label: "United Kingdom" },
            ]}
            required
          />
        </div>
      </div>
    );
  }

  function PreferencesStep() {
    const { formData, updateFormData } = useForm();

    return (
      <div className="form-step">
        <h2>Preferences</h2>

        <FormField
          label="Newsletter Subscription"
          type="checkbox"
          checked={formData.preferences.newsletter}
          onChange={(checked) =>
            updateFormData("preferences", "newsletter", checked)
          }
        />

        <FormField
          label="Email Notifications"
          type="checkbox"
          checked={formData.preferences.notifications}
          onChange={(checked) =>
            updateFormData("preferences", "notifications", checked)
          }
        />
```

```
        <FormField
          label="Theme Preference"
          type="select"
          value={formData.preferences.theme}
          onChange={(value) => updateFormData("preferences", "theme", value)}
          options={[
            { value: "light", label: "Light" },
            { value: "dark", label: "Dark" },
            { value: "auto", label: "Auto" },
          ]}
        />
      </div>
    );
  }

  function ReviewStep() {
    const { formData, submitForm, isSubmitting } = useForm();
    const [submitResult, setSubmitResult] = useState(null);

    const handleSubmit = async () => {
      const result = await submitForm();
      setSubmitResult(result);
    };

    if (submitResult?.success) {
      return (
        <div className="form-step">
          <div className="success-message">
            <h2>✅ Form Submitted Successfully!</h2>
            <p>Thank you for your submission. We'll be in touch soon.</p>
          </div>
        </div>
      );
    }

    return (
      <div className="form-step">
        <h2>Review & Submit</h2>

        <div className="review-section">
          <h3>Personal Information</h3>
          <p>
            <strong>Name:</strong> {formData.personal.firstName}{" "}
            {formData.personal.lastName}
          </p>
          <p>
            <strong>Email:</strong> {formData.personal.email}
          </p>
          <p>
            <strong>Phone:</strong> {formData.personal.phone}
          </p>
        </div>

        <div className="review-section">
```

```
          <h3>Address</h3>
          <p>
            <strong>Street:</strong> {formData.address.street}
          </p>
          <p>
            <strong>City:</strong> {formData.address.city},{" "}
            {formData.address.state} {formData.address.zipCode}
          </p>
          <p>
            <strong>Country:</strong> {formData.address.country}
          </p>
        </div>

        <div className="review-section">
          <h3>Preferences</h3>
          <p>
            <strong>Newsletter:</strong>{" "}
            {formData.preferences.newsletter ? "Yes" : "No"}
          </p>
          <p>
            <strong>Notifications:</strong>{" "}
            {formData.preferences.notifications ? "Yes" : "No"}
          </p>
          <p>
            <strong>Theme:</strong> {formData.preferences.theme}
          </p>
        </div>

        {submitResult?.error && (
          <div className="error-message">Error: {submitResult.error}</div>
        )}

        <button
          onClick={handleSubmit}
          disabled={isSubmitting}
          className="submit-btn"
        >
          {isSubmitting ? "Submitting..." : "Submit Form"}
        </button>
      </div>
    );
  }

  function FormField({
    label,
    type = "text",
    value,
    checked,
    onChange,
    error,
    required,
    options,
    ...props
  }) {
```

```
  const renderInput = () => {
    switch (type) {
      case "checkbox":
        return (
          <input
            type="checkbox"
            checked={checked}
            onChange={(e) => onChange(e.target.checked)}
            {...props}
          />
        );

      case "select":
        return (
          <select
            value={value}
            onChange={(e) => onChange(e.target.value)}
            {...props}
          >
            {options?.map((option) => (
              <option key={option.value} value={option.value}>
                {option.label}
              </option>
            ))}
          </select>
        );

      default:
        return (
          <input
            type={type}
            value={value}
            onChange={(e) => onChange(e.target.value)}
            {...props}
          />
        );
    }
  };

  return (
    <div className={`form-field ${error ? "error" : ""}`}>
      <label>
        {label} {required && <span className="required">*</span>}
      </label>
      {renderInput()}
      {error && <span className="error-text">{error}</span>}
    </div>
  );
}

function FormNavigation() {
  const { currentStep, steps, nextStep, prevStep, isStepComplete } = useForm();

  const canGoNext = currentStep < steps.length - 1;
```

```
  const canGoPrev = currentStep > 0;
  const isLastStep = currentStep === steps.length - 1;

  return (
    <div className="form-navigation">
      <button
        onClick={prevStep}
        disabled={!canGoPrev}
        className="nav-btn prev-btn"
      >
        Previous
      </button>

      {!isLastStep && (
        <button
          onClick={nextStep}
          disabled={!canGoNext}
          className="nav-btn next-btn"
        >
          Next
        </button>
      )}
    </div>
  );
}
```

Challenge 2: Real-time Collaboration Context

Build a context for managing real-time collaboration features like user presence, cursors, and live updates.

▶ 💡 Solution

```
const CollaborationContext = createContext();

function CollaborationProvider({ children, documentId }) {
  const [currentUser, setCurrentUser] = useState(null);
  const [connectedUsers, setConnectedUsers] = useState([]);
  const [userCursors, setUserCursors] = useState({});
  const [documentContent, setDocumentContent] = useState("");
  const [isConnected, setIsConnected] = useState(false);
  const [connectionStatus, setConnectionStatus] = useState("disconnected");
  const [lastSaved, setLastSaved] = useState(null);

  const wsRef = useRef(null);
  const saveTimeoutRef = useRef(null);

  // 🎯 WebSocket connection
  useEffect(() => {
    if (!documentId || !currentUser) return;

    const ws = new WebSocket(`ws://localhost:8080/collaborate/${documentId}`);
```

```
      wsRef.current = ws;

      ws.onopen = () => {
        setIsConnected(true);
        setConnectionStatus("connected");

        // Send user join message
        ws.send(
          JSON.stringify({
            type: "user_join",
            user: currentUser,
          })
        );
      };

      ws.onmessage = (event) => {
        const message = JSON.parse(event.data);
        handleWebSocketMessage(message);
      };

      ws.onclose = () => {
        setIsConnected(false);
        setConnectionStatus("disconnected");
      };

      ws.onerror = () => {
        setConnectionStatus("error");
      };

      return () => {
        if (ws.readyState === WebSocket.OPEN) {
          ws.send(
            JSON.stringify({
              type: "user_leave",
              user: currentUser,
            })
          );
        }
        ws.close();
      };
    }, [documentId, currentUser]);

    // 🎯 Handle WebSocket messages
    const handleWebSocketMessage = useCallback(
      (message) => {
        switch (message.type) {
          case "user_joined":
            setConnectedUsers((prev) => {
              if (prev.find((user) => user.id === message.user.id)) {
                return prev;
              }
              return [...prev, message.user];
            });
            break;
```

```
        case "user_left":
          setConnectedUsers((prev) =>
            prev.filter((user) => user.id !== message.user.id)
          );
          setUserCursors((prev) => {
            const newCursors = { ...prev };
            delete newCursors[message.user.id];
            return newCursors;
          });
          break;

        case "users_list":
          setConnectedUsers(
            message.users.filter((user) => user.id !== currentUser?.id)
          );
          break;

        case "cursor_update":
          setUserCursors((prev) => ({
            ...prev,
            [message.userId]: {
              position: message.position,
              selection: message.selection,
              timestamp: Date.now(),
            },
          }));
          break;

        case "content_update":
          setDocumentContent(message.content);
          break;

        case "document_saved":
          setLastSaved(new Date(message.timestamp));
          break;

        default:
          console.log("Unknown message type:", message.type);
      }
    },
    [currentUser]
  );

  // 🎯 Send cursor position
  const updateCursor = useCallback(
    (position, selection = null) => {
      if (!wsRef.current || wsRef.current.readyState !== WebSocket.OPEN) return;

      wsRef.current.send(
        JSON.stringify({
          type: "cursor_update",
          position,
          selection,
```

```
            userId: currentUser?.id,
          })
        );
      },
      [currentUser]
    );

    // 🎯 Update document content
    const updateContent = useCallback(
      (content) => {
        setDocumentContent(content);

        // Debounced save
        if (saveTimeoutRef.current) {
          clearTimeout(saveTimeoutRef.current);
        }

        saveTimeoutRef.current = setTimeout(() => {
          if (wsRef.current && wsRef.current.readyState === WebSocket.OPEN) {
            wsRef.current.send(
              JSON.stringify({
                type: "content_update",
                content,
                userId: currentUser?.id,
              })
            );
          }
        }, 500);
      },
      [currentUser]
    );

    // 🎯 Join collaboration session
    const joinSession = useCallback((user) => {
      setCurrentUser(user);
    }, []);

    // 🎯 Leave collaboration session
    const leaveSession = useCallback(() => {
      if (wsRef.current && wsRef.current.readyState === WebSocket.OPEN) {
        wsRef.current.send(
          JSON.stringify({
            type: "user_leave",
            user: currentUser,
          })
        );
      }
      setCurrentUser(null);
      setConnectedUsers([]);
      setUserCursors({});
    }, [currentUser]);

    // 🎯 Get user color
    const getUserColor = useCallback((userId) => {
```

```
    const colors = [
      "#FF6B6B",
      "#4ECDC4",
      "#45B7D1",
      "#96CEB4",
      "#FFEAA7",
      "#DDA0DD",
      "#98D8C8",
    ];
    const index = userId.charCodeAt(0) % colors.length;
    return colors[index];
  }, []);

  // 🎯 Clean up old cursors
  useEffect(() => {
    const interval = setInterval(() => {
      const now = Date.now();
      setUserCursors((prev) => {
        const filtered = {};
        Object.entries(prev).forEach(([userId, cursor]) => {
          if (now - cursor.timestamp < 10000) {
            // Keep cursors for 10 seconds
            filtered[userId] = cursor;
          }
        });
        return filtered;
      });
    }, 5000);

    return () => clearInterval(interval);
  }, []);

  const value = {
    // State
    currentUser,
    connectedUsers,
    userCursors,
    documentContent,
    isConnected,
    connectionStatus,
    lastSaved,

    // Actions
    joinSession,
    leaveSession,
    updateCursor,
    updateContent,
    getUserColor,
  };

  return (
    <CollaborationContext.Provider value={value}>
      {children}
    </CollaborationContext.Provider>
```

```
    );
  }

  function useCollaboration() {
    const context = useContext(CollaborationContext);
    if (!context) {
      throw new Error(
        "useCollaboration must be used within a CollaborationProvider"
      );
    }
    return context;
  }

  // 🎯 Collaboration components
  function CollaborativeEditor() {
    const {
      documentContent,
      updateContent,
      updateCursor,
      connectedUsers,
      userCursors,
      getUserColor,
    } = useCollaboration();

    const editorRef = useRef(null);

    const handleContentChange = (e) => {
      updateContent(e.target.value);
    };

    const handleSelectionChange = () => {
      if (!editorRef.current) return;

      const { selectionStart, selectionEnd } = editorRef.current;
      updateCursor(selectionStart, { start: selectionStart, end: selectionEnd });
    };

    return (
      <div className="collaborative-editor">
        <div className="editor-header">
          <UserPresence />
          <ConnectionStatus />
        </div>

        <div className="editor-container">
          <textarea
            ref={editorRef}
            value={documentContent}
            onChange={handleContentChange}
            onSelect={handleSelectionChange}
            onKeyUp={handleSelectionChange}
            onClick={handleSelectionChange}
            className="editor-textarea"
            placeholder="Start typing..."
```

```
          />

          {/* Render user cursors */}
          {Object.entries(userCursors).map(([userId, cursor]) => {
            const user = connectedUsers.find((u) => u.id === userId);
            if (!user) return null;

            return (
              <UserCursor
                key={userId}
                user={user}
                cursor={cursor}
                color={getUserColor(userId)}
              />
            );
          })}
        </div>
      </div>
    );
  }

  function UserPresence() {
    const { connectedUsers, getUserColor } = useCollaboration();

    return (
      <div className="user-presence">
        <span className="presence-label">Online:</span>
        {connectedUsers.map((user) => (
          <div
            key={user.id}
            className="user-avatar"
            style={{ backgroundColor: getUserColor(user.id) }}
            title={user.name}
          >
            {user.name.charAt(0).toUpperCase()}
          </div>
        ))}
      </div>
    );
  }

  function ConnectionStatus() {
    const { connectionStatus, lastSaved } = useCollaboration();

    const getStatusIcon = () => {
      switch (connectionStatus) {
        case "connected":
          return "🟢";
        case "disconnected":
          return "🔴";
        case "error":
          return "🟡";
        default:
          return "⚪";
```

```
    }
  };

  return (
    <div className="connection-status">
      <span className="status-indicator">
        {getStatusIcon()} {connectionStatus}
      </span>
      {lastSaved && (
        <span className="last-saved">
          Last saved: {lastSaved.toLocaleTimeString()}
        </span>
      )}
    </div>
  );
}

function UserCursor({ user, cursor, color }) {
  const [position, setPosition] = useState({ top: 0, left: 0 });

  // Calculate cursor position based on text position
  useEffect(() => {
    // This would need more complex calculation in a real implementation
    // For demo purposes, we'll use a simple approximation
    const lineHeight = 20;
    const charWidth = 8;
    const line = Math.floor(cursor.position / 80); // Assuming 80 chars per line
    const char = cursor.position % 80;

    setPosition({
      top: line * lineHeight,
      left: char * charWidth,
    });
  }, [cursor.position]);

  return (
    <div
      className="user-cursor"
      style={{
        position: "absolute",
        top: position.top,
        left: position.left,
        borderColor: color,
      }}
    >
      <div className="cursor-line" style={{ backgroundColor: color }} />
      <div className="cursor-label" style={{ backgroundColor: color }}>
        {user.name}
      </div>
    </div>
  );
}
```

🎯 When and Why to Use Context

✅ Good Use Cases

1. **Theme/UI State**: Colors, fonts, layout preferences
2. **User Authentication**: Current user, permissions, auth status
3. **Language/Localization**: Current language, translations
4. **Global App State**: Shopping cart, notifications, modals
5. **Feature Flags**: A/B testing, feature toggles
6. **API Configuration**: Base URLs, auth tokens, request settings

❌ Avoid Context For

1. **Local Component State**: State used by only one component
2. **Frequently Changing Values**: Mouse position, scroll position
3. **Performance-Critical Data**: Large datasets, real-time updates
4. **Server State**: Use React Query, SWR, or similar instead
5. **Form State**: Use local state or form libraries

🤔 Decision Framework

```
Do multiple components need this data?
├─ No → Use local state
└─ Yes
   ├─ Does it change frequently?
   │  ├─ Yes → Consider alternatives (state management library)
   │  └─ No → Context is good
   └─ Is it server data?
      ├─ Yes → Use data fetching library
      └─ No → Context is perfect
```

🚀 Performance Optimization

Split Contexts by Update Frequency

```
// ✅ Separate stable and changing values
function OptimizedProvider({ children }) {
  return (
    <StableDataProvider>
      {" "}
      {/* User, theme, config */}
      <ChangingDataProvider>
        {" "}
        {/* Notifications, real-time data */}
        {children}
      </ChangingDataProvider>
    </StableDataProvider>
```

```
  );
}
```

Use React.memo with Context

```
// ✅ Prevent unnecessary re-renders
const ExpensiveComponent = React.memo(function ExpensiveComponent() {
  const { stableValue } = useStableContext();

  return (
    <div>
      {/* Expensive rendering logic */}
      {stableValue}
    </div>
  );
});
```

Selective Context Consumption

```
// ✅ Only consume what you need
function useThemeActions() {
  const { setTheme, toggleTheme } = useTheme();
  return useMemo(() => ({ setTheme, toggleTheme }), [setTheme, toggleTheme]);
}

function useThemeValues() {
  const { theme, colors } = useTheme();
  return useMemo(() => ({ theme, colors }), [theme, colors]);
}
```

Interview Insights

Common Questions

Q: When would you use Context vs Redux?

A: Context is great for:

- Simple global state
- Theme/user preferences
- Authentication state
- Small to medium apps

Redux is better for:

- Complex state logic
- Time-travel debugging

- Predictable state updates
- Large applications
- Team collaboration

Q: How do you prevent Context performance issues?

A:

1. Split contexts by update frequency
2. Memoize context values
3. Use React.memo for expensive components
4. Avoid putting frequently changing data in context
5. Consider state colocation

Q: What's the difference between Context and prop drilling?

A: Prop drilling passes data through intermediate components that don't use it. Context allows direct access to data from any component in the tree, eliminating unnecessary prop passing.

Code Review Red Flags

✕ Creating context for every piece of state ✕ Not memoizing context values ✕ Putting frequently changing data in context ✕ Missing error boundaries around context providers ✕ Not providing default values ✕ Using context for server state

🎯 Key Takeaways

1. **Context solves prop drilling** - Share data across component trees
2. **Not a replacement for all state management** - Use appropriately
3. **Performance matters** - Memoize values and split contexts
4. **Custom hooks improve DX** - Encapsulate context logic
5. **Error handling is crucial** - Always check if context exists
6. **Default values help** - Provide sensible defaults
7. **Think about update frequency** - Separate stable from changing data

🏆 Best Practices

1. **Always provide custom hooks** for consuming context
2. **Memoize context values** to prevent unnecessary re-renders
3. **Split contexts** by concern and update frequency
4. **Use TypeScript** for better developer experience
5. **Provide meaningful error messages** when context is missing
6. **Keep context values focused** - single responsibility
7. **Document your contexts** - explain when and how to use them

🔧 Production Tips

- **Monitor context re-renders** with React DevTools Profiler
- **Use React.StrictMode** to catch context-related issues
- **Consider context composition** for complex applications

- **Implement proper error boundaries** around providers
- **Test context providers** in isolation
- **Use context for configuration** that rarely changes
- **Avoid context for derived state** - compute in components instead

---

Next up: We'll explore `useMemo` and learn how to optimize expensive calculations and prevent unnecessary re-renders! 🚀

useReducer: Complex State Management

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What `useReducer` is and when to use it over `useState`
- How to design reducer functions and action patterns
- State management patterns for complex applications
- Integration with Context for global state
- Performance optimization techniques
- Real-world state machine implementations

🧠 What is useReducer?

`useReducer` is a React Hook that provides an alternative to `useState` for managing **complex state logic**. It's particularly useful when:

- State has multiple sub-values
- Next state depends on the previous state
- State transitions are complex
- You need predictable state updates

The Problem: Complex State with useState

```
// ❌ Complex state management with useState becomes unwieldy
function ComplexForm() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [confirmPassword, setConfirmPassword] = useState("");
  const [errors, setErrors] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [isValid, setIsValid] = useState(false);
  const [touched, setTouched] = useState({});
  const [submitCount, setSubmitCount] = useState(0);

  // Multiple state updates scattered throughout
```

```
  const handleSubmit = async () => {
    setIsSubmitting(true);
    setSubmitCount((prev) => prev + 1);
    setErrors({});

    try {
      // Validation logic
      const newErrors = {};
      if (!firstName) newErrors.firstName = "Required";
      if (!lastName) newErrors.lastName = "Required";
      if (!email) newErrors.email = "Required";
      if (password !== confirmPassword)
        newErrors.confirmPassword = "Passwords must match";

      if (Object.keys(newErrors).length > 0) {
        setErrors(newErrors);
        setIsValid(false);
        setIsSubmitting(false);
        return;
      }

      // Submit logic
      await submitForm({ firstName, lastName, email, password });

      // Reset form
      setFirstName("");
      setLastName("");
      setEmail("");
      setPassword("");
      setConfirmPassword("");
      setErrors({});
      setIsValid(false);
      setTouched({});
    } catch (error) {
      setErrors({ submit: error.message });
    } finally {
      setIsSubmitting(false);
    }
  };

  // More scattered state updates...
}
```

The Solution: useReducer

```
// ✅ Clean, predictable state management with useReducer
const initialState = {
  fields: {
    firstName: "",
    lastName: "",
    email: "",
```

```
    password: "",
    confirmPassword: "",
  },
  errors: {},
  touched: {},
  isSubmitting: false,
  isValid: false,
  submitCount: 0,
};

function formReducer(state, action) {
  switch (action.type) {
    case "FIELD_CHANGE":
      return {
        ...state,
        fields: {
          ...state.fields,
          [action.field]: action.value,
        },
        errors: {
          ...state.errors,
          [action.field]: undefined, // Clear error for this field
        },
      };

    case "FIELD_BLUR":
      return {
        ...state,
        touched: {
          ...state.touched,
          [action.field]: true,
        },
      };

    case "SUBMIT_START":
      return {
        ...state,
        isSubmitting: true,
        submitCount: state.submitCount + 1,
        errors: {},
      };

    case "SUBMIT_SUCCESS":
      return {
        ...initialState, // Reset to initial state
      };

    case "SUBMIT_ERROR":
      return {
        ...state,
        isSubmitting: false,
        errors: action.errors,
      };
```

```
      case "VALIDATION_RESULT":
        return {
          ...state,
          isValid: action.isValid,
          errors: {
            ...state.errors,
            ...action.errors,
          },
        };

      default:
        return state;
    }
  }

  function CleanForm() {
    const [state, dispatch] = useReducer(formReducer, initialState);

    const handleFieldChange = (field, value) => {
      dispatch({ type: "FIELD_CHANGE", field, value });
    };

    const handleFieldBlur = (field) => {
      dispatch({ type: "FIELD_BLUR", field });
    };

    const handleSubmit = async () => {
      dispatch({ type: "SUBMIT_START" });

      try {
        // Validation
        const errors = validateForm(state.fields);
        if (Object.keys(errors).length > 0) {
          dispatch({
            type: "VALIDATION_RESULT",
            isValid: false,
            errors,
          });
          return;
        }

        // Submit
        await submitForm(state.fields);
        dispatch({ type: "SUBMIT_SUCCESS" });
      } catch (error) {
        dispatch({
          type: "SUBMIT_ERROR",
          errors: { submit: error.message },
        });
      }
    };

    return (
      <form onSubmit={handleSubmit}>
```

```
      {/* Form fields using state and dispatch */}
    </form>
  );
}
```

🔍 Deep Dive: useReducer Fundamentals

Basic Syntax

```
const [state, dispatch] = useReducer(reducer, initialState, init?);
```

- **reducer**: `(state, action) => newState`
- **initialState**: The initial state value
- **init**: Optional lazy initialization function
- **state**: Current state value
- **dispatch**: Function to trigger state updates

Reducer Function Patterns

```
// 🎯 Basic reducer pattern
function counterReducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    case "RESET":
      return { count: 0 };
    default:
      throw new Error(`Unknown action type: ${action.type}`);
  }
}

// 🎯 Reducer with payload
function todoReducer(state, action) {
  switch (action.type) {
    case "ADD_TODO":
      return {
        ...state,
        todos: [
          ...state.todos,
          {
            id: Date.now(),
            text: action.payload.text,
            completed: false,
            createdAt: new Date().toISOString(),
          },
        ],
```

```
      };

    case "TOGGLE_TODO":
      return {
        ...state,
        todos: state.todos.map((todo) =>
          todo.id === action.payload.id
            ? { ...todo, completed: !todo.completed }
            : todo
        ),
      };

    case "DELETE_TODO":
      return {
        ...state,
        todos: state.todos.filter((todo) => todo.id !== action.payload.id),
      };

    case "EDIT_TODO":
      return {
        ...state,
        todos: state.todos.map((todo) =>
          todo.id === action.payload.id
            ? { ...todo, text: action.payload.text }
            : todo
        ),
      };

    case "CLEAR_COMPLETED":
      return {
        ...state,
        todos: state.todos.filter((todo) => !todo.completed),
      };

    case "SET_FILTER":
      return {
        ...state,
        filter: action.payload.filter,
      };

    default:
      return state;
  }
}

// 🎯 Reducer with complex state updates
function shoppingCartReducer(state, action) {
  switch (action.type) {
    case "ADD_ITEM": {
      const { product } = action.payload;
      const existingItem = state.items.find((item) => item.id === product.id);

      if (existingItem) {
        return {
```

```
            ...state,
            items: state.items.map((item) =>
              item.id === product.id
                ? { ...item, quantity: item.quantity + 1 }
                : item
            ),
          };
        }

        return {
          ...state,
          items: [...state.items, { ...product, quantity: 1 }],
        };
      }

      case "REMOVE_ITEM":
        return {
          ...state,
          items: state.items.filter((item) => item.id !== action.payload.id),
        };

      case "UPDATE_QUANTITY": {
        const { id, quantity } = action.payload;

        if (quantity <= 0) {
          return {
            ...state,
            items: state.items.filter((item) => item.id !== id),
          };
        }

        return {
          ...state,
          items: state.items.map((item) =>
            item.id === id ? { ...item, quantity } : item
          ),
        };
      }

      case "APPLY_DISCOUNT": {
        const { code, percentage } = action.payload;
        return {
          ...state,
          discount: {
            code,
            percentage,
            applied: true,
          },
        };
      }

      case "CLEAR_CART":
        return {
          ...state,
```

```
        items: [],
        discount: null,
      };

    default:
      return state;
  }
}
```

Action Creators Pattern

```
// 🎯 Action creators for better maintainability
const todoActions = {
  addTodo: (text) => ({
    type: "ADD_TODO",
    payload: { text },
  }),

  toggleTodo: (id) => ({
    type: "TOGGLE_TODO",
    payload: { id },
  }),

  deleteTodo: (id) => ({
    type: "DELETE_TODO",
    payload: { id },
  }),

  editTodo: (id, text) => ({
    type: "EDIT_TODO",
    payload: { id, text },
  }),

  clearCompleted: () => ({
    type: "CLEAR_COMPLETED",
  }),

  setFilter: (filter) => ({
    type: "SET_FILTER",
    payload: { filter },
  }),
};

// 🎯 Usage with action creators
function TodoApp() {
  const [state, dispatch] = useReducer(todoReducer, initialState);

  const addTodo = (text) => {
    dispatch(todoActions.addTodo(text));
  };
```

```
  const toggleTodo = (id) => {
    dispatch(todoActions.toggleTodo(id));
  };

  // ... rest of component
}
```

🚀 Real-World Patterns

Pattern 1: State Machine with useReducer

```
// 🎯 Async data fetching state machine
const initialState = {
  status: "idle", // idle | loading | success | error
  data: null,
  error: null,
  retryCount: 0,
};

function dataFetchReducer(state, action) {
  switch (action.type) {
    case "FETCH_START":
      return {
        ...state,
        status: "loading",
        error: null,
      };

    case "FETCH_SUCCESS":
      return {
        ...state,
        status: "success",
        data: action.payload.data,
        error: null,
        retryCount: 0,
      };

    case "FETCH_ERROR":
      return {
        ...state,
        status: "error",
        error: action.payload.error,
        data: null,
      };

    case "RETRY":
      return {
        ...state,
        status: "loading",
        error: null,
        retryCount: state.retryCount + 1,
```

```
      };

    case "RESET":
      return initialState;

    default:
      return state;
  }
}

function useAsyncData(url) {
  const [state, dispatch] = useReducer(dataFetchReducer, initialState);

  const fetchData = useCallback(async () => {
    dispatch({ type: "FETCH_START" });

    try {
      const response = await fetch(url);
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }
      const data = await response.json();
      dispatch({ type: "FETCH_SUCCESS", payload: { data } });
    } catch (error) {
      dispatch({ type: "FETCH_ERROR", payload: { error: error.message } });
    }
  }, [url]);

  const retry = useCallback(() => {
    if (state.retryCount < 3) {
      dispatch({ type: "RETRY" });
      fetchData();
    }
  }, [fetchData, state.retryCount]);

  const reset = useCallback(() => {
    dispatch({ type: "RESET" });
  }, []);

  useEffect(() => {
    fetchData();
  }, [fetchData]);

  return {
    ...state,
    refetch: fetchData,
    retry,
    reset,
    canRetry: state.retryCount < 3,
  };
}

// 🎯 Usage
function DataComponent() {
```

```
  const { status, data, error, retry, canRetry } = useAsyncData("/api/users");

  if (status === "loading") {
    return <div>Loading...</div>;
  }

  if (status === "error") {
    return (
      <div>
        <p>Error: {error}</p>
        {canRetry && <button onClick={retry}>Retry</button>}
      </div>
    );
  }

  if (status === "success") {
    return (
      <div>
        {data.map((user) => (
          <div key={user.id}>{user.name}</div>
        ))}
      </div>
    );
  }

  return null;
}
```

Pattern 2: Multi-Step Wizard

```
// 🎯 Multi-step wizard with validation
const wizardSteps = [
  { id: "personal", title: "Personal Info", component: PersonalStep },
  { id: "address", title: "Address", component: AddressStep },
  { id: "payment", title: "Payment", component: PaymentStep },
  { id: "review", title: "Review", component: ReviewStep },
];

const initialWizardState = {
  currentStep: 0,
  data: {
    personal: {},
    address: {},
    payment: {},
  },
  errors: {},
  touched: {},
  isValid: false,
  isSubmitting: false,
  completedSteps: [],
};
```

```
function wizardReducer(state, action) {
  switch (action.type) {
    case "NEXT_STEP": {
      const nextStep = Math.min(state.currentStep + 1, wizardSteps.length - 1);
      return {
        ...state,
        currentStep: nextStep,
        completedSteps: [
          ...new Set([...state.completedSteps, state.currentStep]),
        ],
      };
    }

    case "PREV_STEP": {
      const prevStep = Math.max(state.currentStep - 1, 0);
      return {
        ...state,
        currentStep: prevStep,
      };
    }

    case "GO_TO_STEP": {
      const { step } = action.payload;
      // Only allow going to completed steps or next step
      if (
        state.completedSteps.includes(step) ||
        step === state.currentStep + 1
      ) {
        return {
          ...state,
          currentStep: step,
        };
      }
      return state;
    }

    case "UPDATE_STEP_DATA": {
      const { step, data } = action.payload;
      return {
        ...state,
        data: {
          ...state.data,
          [step]: {
            ...state.data[step],
            ...data,
          },
        },
      };
    }

    case "SET_STEP_ERRORS": {
      const { step, errors } = action.payload;
      return {
```

```
        ...state,
        errors: {
          ...state.errors,
          [step]: errors,
        },
      };
    }

    case "SET_STEP_TOUCHED": {
      const { step, touched } = action.payload;
      return {
        ...state,
        touched: {
          ...state.touched,
          [step]: {
            ...state.touched[step],
            ...touched,
          },
        },
      };
    }

    case "VALIDATE_STEP": {
      const { step, isValid, errors } = action.payload;
      return {
        ...state,
        isValid,
        errors: {
          ...state.errors,
          [step]: errors,
        },
      };
    }

    case "SUBMIT_START":
      return {
        ...state,
        isSubmitting: true,
      };

    case "SUBMIT_SUCCESS":
      return {
        ...initialWizardState,
        // Maybe keep some success state
      };

    case "SUBMIT_ERROR":
      return {
        ...state,
        isSubmitting: false,
        errors: {
          ...state.errors,
          submit: action.payload.error,
        },
```

```
      };

    case "RESET_WIZARD":
      return initialWizardState;

    default:
      return state;
  }
}

function useWizard() {
  const [state, dispatch] = useReducer(wizardReducer, initialWizardState);

  const nextStep = useCallback(() => {
    const currentStepId = wizardSteps[state.currentStep].id;
    const stepData = state.data[currentStepId];
    const errors = validateStep(currentStepId, stepData);

    if (Object.keys(errors).length === 0) {
      dispatch({ type: "NEXT_STEP" });
    } else {
      dispatch({
        type: "VALIDATE_STEP",
        payload: { step: currentStepId, isValid: false, errors },
      });
    }
  }, [state.currentStep, state.data]);

  const prevStep = useCallback(() => {
    dispatch({ type: "PREV_STEP" });
  }, []);

  const goToStep = useCallback((step) => {
    dispatch({ type: "GO_TO_STEP", payload: { step } });
  }, []);

  const updateStepData = useCallback((step, data) => {
    dispatch({ type: "UPDATE_STEP_DATA", payload: { step, data } });
  }, []);

  const submitWizard = useCallback(async () => {
    dispatch({ type: "SUBMIT_START" });

    try {
      // Validate all steps
      const allErrors = {};
      let hasErrors = false;

      Object.keys(state.data).forEach((step) => {
        const errors = validateStep(step, state.data[step]);
        if (Object.keys(errors).length > 0) {
          allErrors[step] = errors;
          hasErrors = true;
        }
```

```
      });

      if (hasErrors) {
        dispatch({
          type: "SUBMIT_ERROR",
          payload: { error: "Please fix validation errors" },
        });
        return;
      }

      // Submit data
      const response = await fetch("/api/wizard-submit", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(state.data),
      });

      if (!response.ok) {
        throw new Error("Submission failed");
      }

      dispatch({ type: "SUBMIT_SUCCESS" });
    } catch (error) {
      dispatch({ type: "SUBMIT_ERROR", payload: { error: error.message } });
    }
  }, [state.data]);

  return {
    ...state,
    nextStep,
    prevStep,
    goToStep,
    updateStepData,
    submitWizard,
    currentStepData: state.data[wizardSteps[state.currentStep]?.id] || {},
    canGoNext: state.currentStep < wizardSteps.length - 1,
    canGoPrev: state.currentStep > 0,
    isLastStep: state.currentStep === wizardSteps.length - 1,
  };
}

// 🎯 Wizard component
function MultiStepWizard() {
  const wizard = useWizard();
  const currentStep = wizardSteps[wizard.currentStep];
  const StepComponent = currentStep.component;

  return (
    <div className="wizard">
      <WizardProgress
        steps={wizardSteps}
        currentStep={wizard.currentStep}
        completedSteps={wizard.completedSteps}
        onStepClick={wizard.goToStep}
```

```
        />

        <div className="wizard-content">
          <StepComponent
            data={wizard.currentStepData}
            errors={wizard.errors[currentStep.id] || {}}
            onChange={(data) => wizard.updateStepData(currentStep.id, data)}
          />
        </div>

        <WizardNavigation
          canGoNext={wizard.canGoNext}
          canGoPrev={wizard.canGoPrev}
          isLastStep={wizard.isLastStep}
          isSubmitting={wizard.isSubmitting}
          onNext={wizard.nextStep}
          onPrev={wizard.prevStep}
          onSubmit={wizard.submitWizard}
        />
      </div>
    );
}
```

Pattern 3: Game State Management

```
// 🎯 Tic-tac-toe game with useReducer
const initialGameState = {
  board: Array(9).fill(null),
  currentPlayer: "X",
  winner: null,
  gameStatus: "playing", // playing | won | draw
  moveHistory: [],
  scores: { X: 0, O: 0, draws: 0 },
};

function gameReducer(state, action) {
  switch (action.type) {
    case "MAKE_MOVE": {
      const { index } = action.payload;

      // Invalid move
      if (state.board[index] || state.winner) {
        return state;
      }

      const newBoard = [...state.board];
      newBoard[index] = state.currentPlayer;

      const winner = calculateWinner(newBoard);
      const isDraw = !winner && newBoard.every((cell) => cell !== null);
```

```
        let newScores = state.scores;
        let gameStatus = "playing";

        if (winner) {
          gameStatus = "won";
          newScores = {
            ...state.scores,
            [winner]: state.scores[winner] + 1,
          };
        } else if (isDraw) {
          gameStatus = "draw";
          newScores = {
            ...state.scores,
            draws: state.scores.draws + 1,
          };
        }

        return {
          ...state,
          board: newBoard,
          currentPlayer: state.currentPlayer === "X" ? "O" : "X",
          winner,
          gameStatus,
          scores: newScores,
          moveHistory: [
            ...state.moveHistory,
            {
              player: state.currentPlayer,
              index,
              board: newBoard,
              timestamp: Date.now(),
            },
          ],
        };
      }

      case "RESET_GAME":
        return {
          ...initialGameState,
          scores: state.scores, // Keep scores
        };

      case "NEW_GAME":
        return initialGameState;

      case "UNDO_MOVE": {
        if (state.moveHistory.length === 0) {
          return state;
        }

        const newHistory = state.moveHistory.slice(0, -1);
        const lastMove = newHistory[newHistory.length - 1];

        if (lastMove) {
```

```
        return {
          ...state,
          board: lastMove.board,
          currentPlayer: lastMove.player === "X" ? "O" : "X",
          winner: calculateWinner(lastMove.board),
          gameStatus: "playing",
          moveHistory: newHistory,
        };
      } else {
        return {
          ...state,
          board: Array(9).fill(null),
          currentPlayer: "X",
          winner: null,
          gameStatus: "playing",
          moveHistory: [],
        };
      }
    }

    default:
      return state;
  }
}

function calculateWinner(board) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8], // rows
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8], // columns
    [0, 4, 8],
    [2, 4, 6], // diagonals
  ];

  for (let line of lines) {
    const [a, b, c] = line;
    if (board[a] && board[a] === board[b] && board[a] === board[c]) {
      return board[a];
    }
  }

  return null;
}

function TicTacToe() {
  const [state, dispatch] = useReducer(gameReducer, initialGameState);

  const makeMove = (index) => {
    dispatch({ type: "MAKE_MOVE", payload: { index } });
  };
```

```
  const resetGame = () => {
    dispatch({ type: "RESET_GAME" });
  };

  const newGame = () => {
    dispatch({ type: "NEW_GAME" });
  };

  const undoMove = () => {
    dispatch({ type: "UNDO_MOVE" });
  };

  return (
    <div className="tic-tac-toe">
      <div className="game-header">
        <div className="scores">
          <span>X: {state.scores.X}</span>
          <span>O: {state.scores.O}</span>
          <span>Draws: {state.scores.draws}</span>
        </div>

        <div className="game-controls">
          <button onClick={undoMove} disabled={state.moveHistory.length === 0}>
            Undo
          </button>
          <button onClick={resetGame}>Reset Game</button>
          <button onClick={newGame}>New Game</button>
        </div>
      </div>

      <div className="game-status">
        {state.gameStatus === "won" && <h2>🎉 Player {state.winner} wins!</h2>}
        {state.gameStatus === "draw" && <h2>🤝 It's a draw!</h2>}
        {state.gameStatus === "playing" && (
          <h2>Current player: {state.currentPlayer}</h2>
        )}
      </div>

      <div className="board">
        {state.board.map((cell, index) => (
          <button
            key={index}
            className="cell"
            onClick={() => makeMove(index)}
            disabled={cell !== null || state.gameStatus !== "playing"}
          >
            {cell}
          </button>
        ))}
      </div>

      <div className="move-history">
        <h3>Move History</h3>
        {state.moveHistory.map((move, index) => (
```

```
            <div key={index} className="move">
              Move {index + 1}: Player {move.player} → Position {move.index}
            </div>
          ))}
        </div>
      </div>
    );
}
```

🔄 useReducer + Context Pattern

```
// 🎯 Global state management with useReducer + Context
const AppStateContext = createContext();
const AppDispatchContext = createContext();

const initialAppState = {
  user: null,
  theme: "light",
  notifications: [],
  cart: {
    items: [],
    total: 0,
  },
  ui: {
    sidebarOpen: false,
    modalStack: [],
  },
};

function appReducer(state, action) {
  switch (action.type) {
    // User actions
    case "USER_LOGIN":
      return {
        ...state,
        user: action.payload.user,
      };

    case "USER_LOGOUT":
      return {
        ...state,
        user: null,
        cart: { items: [], total: 0 }, // Clear cart on logout
      };

    // Theme actions
    case "TOGGLE_THEME":
      return {
        ...state,
        theme: state.theme === "light" ? "dark" : "light",
      };
```

```
    // Notification actions
    case "ADD_NOTIFICATION":
      return {
        ...state,
        notifications: [
          ...state.notifications,
          {
            id: Date.now(),
            ...action.payload,
          },
        ],
      };

    case "REMOVE_NOTIFICATION":
      return {
        ...state,
        notifications: state.notifications.filter(
          (notification) => notification.id !== action.payload.id
        ),
      };

    // Cart actions
    case "ADD_TO_CART": {
      const { product } = action.payload;
      const existingItem = state.cart.items.find(
        (item) => item.id === product.id
      );

      let newItems;
      if (existingItem) {
        newItems = state.cart.items.map((item) =>
          item.id === product.id
            ? { ...item, quantity: item.quantity + 1 }
            : item
        );
      } else {
        newItems = [...state.cart.items, { ...product, quantity: 1 }];
      }

      const newTotal = newItems.reduce(
        (sum, item) => sum + item.price * item.quantity,
        0
      );

      return {
        ...state,
        cart: {
          items: newItems,
          total: newTotal,
        },
      };
    }
```

```
    // UI actions
    case "TOGGLE_SIDEBAR":
      return {
        ...state,
        ui: {
          ...state.ui,
          sidebarOpen: !state.ui.sidebarOpen,
        },
      };

    case "OPEN_MODAL":
      return {
        ...state,
        ui: {
          ...state.ui,
          modalStack: [...state.ui.modalStack, action.payload.modal],
        },
      };

    case "CLOSE_MODAL":
      return {
        ...state,
        ui: {
          ...state.ui,
          modalStack: state.ui.modalStack.slice(0, -1),
        },
      };

    default:
      return state;
  }
}

function AppProvider({ children }) {
  const [state, dispatch] = useReducer(appReducer, initialAppState);

  return (
    <AppStateContext.Provider value={state}>
      <AppDispatchContext.Provider value={dispatch}>
        {children}
      </AppDispatchContext.Provider>
    </AppStateContext.Provider>
  );
}

// 🎯 Custom hooks for consuming state and dispatch
function useAppState() {
  const context = useContext(AppStateContext);
  if (!context) {
    throw new Error("useAppState must be used within an AppProvider");
  }
  return context;
}
```

```
function useAppDispatch() {
  const context = useContext(AppDispatchContext);
  if (!context) {
    throw new Error("useAppDispatch must be used within an AppProvider");
  }
  return context;
}

// 🎯 Specialized hooks for specific state slices
function useUser() {
  const { user } = useAppState();
  const dispatch = useAppDispatch();

  const login = useCallback(
    (userData) => {
      dispatch({ type: "USER_LOGIN", payload: { user: userData } });
    },
    [dispatch]
  );

  const logout = useCallback(() => {
    dispatch({ type: "USER_LOGOUT" });
  }, [dispatch]);

  return {
    user,
    isAuthenticated: !!user,
    login,
    logout,
  };
}

function useTheme() {
  const { theme } = useAppState();
  const dispatch = useAppDispatch();

  const toggleTheme = useCallback(() => {
    dispatch({ type: "TOGGLE_THEME" });
  }, [dispatch]);

  return {
    theme,
    isDark: theme === "dark",
    toggleTheme,
  };
}

function useNotifications() {
  const { notifications } = useAppState();
  const dispatch = useAppDispatch();

  const addNotification = useCallback(
    (notification) => {
      dispatch({ type: "ADD_NOTIFICATION", payload: notification });
```

```
    },
    [dispatch]
  );

  const removeNotification = useCallback(
    (id) => {
      dispatch({ type: "REMOVE_NOTIFICATION", payload: { id } });
    },
    [dispatch]
  );

  return {
    notifications,
    addNotification,
    removeNotification,
  };
}

function useCart() {
  const { cart } = useAppState();
  const dispatch = useAppDispatch();

  const addToCart = useCallback(
    (product) => {
      dispatch({ type: "ADD_TO_CART", payload: { product } });
    },
    [dispatch]
  );

  return {
    ...cart,
    addToCart,
    itemCount: cart.items.reduce((sum, item) => sum + item.quantity, 0),
  };
}
```

⚠ Common Mistakes and Anti-Patterns

Mistake 1: Mutating State

```
// ❌ Bad: Mutating state directly
function badReducer(state, action) {
  switch (action.type) {
    case "ADD_ITEM":
      state.items.push(action.payload); // Mutation!
      return state;

    case "UPDATE_ITEM":
      const item = state.items.find((item) => item.id === action.payload.id);
      item.name = action.payload.name; // Mutation!
      return state;
```

```
    default:
      return state;
  }
}

// ✅ Good: Returning new state objects
function goodReducer(state, action) {
  switch (action.type) {
    case "ADD_ITEM":
      return {
        ...state,
        items: [...state.items, action.payload],
      };

    case "UPDATE_ITEM":
      return {
        ...state,
        items: state.items.map((item) =>
          item.id === action.payload.id
            ? { ...item, name: action.payload.name }
            : item
        ),
      };

    default:
      return state;
  }
}
```

Mistake 2: Complex Logic in Components

```
// ❌ Bad: Complex state logic in component
function BadTodoList() {
  const [state, dispatch] = useReducer(todoReducer, initialState);

  const addTodo = (text) => {
    // Complex logic in component
    if (text.trim() === "") return;

    const newTodo = {
      id: Date.now(),
      text: text.trim(),
      completed: false,
      createdAt: new Date().toISOString(),
      priority: text.includes("!") ? "high" : "normal",
    };

    dispatch({ type: "ADD_TODO", payload: newTodo });
  };
```

```
  // More complex logic...
}

// ✅ Good: Logic in reducer or custom hooks
function goodTodoReducer(state, action) {
  switch (action.type) {
    case "ADD_TODO": {
      const { text } = action.payload;

      // Validation and processing in reducer
      if (!text || text.trim() === "") {
        return state;
      }

      const newTodo = {
        id: Date.now(),
        text: text.trim(),
        completed: false,
        createdAt: new Date().toISOString(),
        priority: text.includes("!") ? "high" : "normal",
      };

      return {
        ...state,
        todos: [...state.todos, newTodo],
      };
    }

    default:
      return state;
  }
}

function GoodTodoList() {
  const [state, dispatch] = useReducer(goodTodoReducer, initialState);

  const addTodo = (text) => {
    dispatch({ type: "ADD_TODO", payload: { text } });
  };

  // Clean component logic
}
```

Mistake 3: Not Using Action Types Constants

```
// ❌ Bad: Magic strings
function badComponent() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleClick = () => {
    dispatch({ type: "INCREMENT_COUNTER" }); // Typo-prone
```

```
  };
}

// ✅ Good: Action type constants
const ACTION_TYPES = {
  INCREMENT_COUNTER: "INCREMENT_COUNTER",
  DECREMENT_COUNTER: "DECREMENT_COUNTER",
  RESET_COUNTER: "RESET_COUNTER",
};

function goodReducer(state, action) {
  switch (action.type) {
    case ACTION_TYPES.INCREMENT_COUNTER:
      return { count: state.count + 1 };

    case ACTION_TYPES.DECREMENT_COUNTER:
      return { count: state.count - 1 };

    case ACTION_TYPES.RESET_COUNTER:
      return { count: 0 };

    default:
      return state;
  }
}

function goodComponent() {
  const [state, dispatch] = useReducer(goodReducer, initialState);

  const handleClick = () => {
    dispatch({ type: ACTION_TYPES.INCREMENT_COUNTER }); // Type-safe
  };
}
```

Mini-Challenges

Challenge 1: Advanced Todo App

Build a todo app with categories, due dates, priorities, and search functionality using `useReducer`.

▶ Solution

```
const initialTodoState = {
  todos: [],
  categories: ["Personal", "Work", "Shopping"],
  filter: {
    category: "all",
    priority: "all",
    status: "all",
    search: "",
  },
```

```
  sort: {
    field: "createdAt",
    direction: "desc",
  },
};

function todoReducer(state, action) {
  switch (action.type) {
    case "ADD_TODO": {
      const { text, category, priority, dueDate } = action.payload;

      const newTodo = {
        id: Date.now(),
        text: text.trim(),
        category: category || "Personal",
        priority: priority || "medium",
        dueDate: dueDate || null,
        completed: false,
        createdAt: new Date().toISOString(),
        completedAt: null,
      };

      return {
        ...state,
        todos: [...state.todos, newTodo],
      };
    }

    case "TOGGLE_TODO": {
      const { id } = action.payload;
      return {
        ...state,
        todos: state.todos.map((todo) =>
          todo.id === id
            ? {
                ...todo,
                completed: !todo.completed,
                completedAt: !todo.completed ? new Date().toISOString() : null,
              }
            : todo
        ),
      };
    }

    case "UPDATE_TODO": {
      const { id, updates } = action.payload;
      return {
        ...state,
        todos: state.todos.map((todo) =>
          todo.id === id ? { ...todo, ...updates } : todo
        ),
      };
    }
```

```
      case "DELETE_TODO": {
        const { id } = action.payload;
        return {
          ...state,
          todos: state.todos.filter((todo) => todo.id !== id),
        };
      }

      case "SET_FILTER": {
        const { filterType, value } = action.payload;
        return {
          ...state,
          filter: {
            ...state.filter,
            [filterType]: value,
          },
        };
      }

      case "SET_SORT": {
        const { field, direction } = action.payload;
        return {
          ...state,
          sort: { field, direction },
        };
      }

      case "ADD_CATEGORY": {
        const { category } = action.payload;
        if (!state.categories.includes(category)) {
          return {
            ...state,
            categories: [...state.categories, category],
          };
        }
        return state;
      }

      default:
        return state;
    }
  }

  function useTodos() {
    const [state, dispatch] = useReducer(todoReducer, initialTodoState);

    // Filtered and sorted todos
    const filteredTodos = useMemo(() => {
      let filtered = state.todos;

      // Filter by category
      if (state.filter.category !== "all") {
        filtered = filtered.filter(
          (todo) => todo.category === state.filter.category
```

```
      );
    }

    // Filter by priority
    if (state.filter.priority !== "all") {
      filtered = filtered.filter(
        (todo) => todo.priority === state.filter.priority
      );
    }

    // Filter by status
    if (state.filter.status !== "all") {
      filtered = filtered.filter((todo) =>
        state.filter.status === "completed" ? todo.completed : !todo.completed
      );
    }

    // Filter by search
    if (state.filter.search) {
      filtered = filtered.filter((todo) =>
        todo.text.toLowerCase().includes(state.filter.search.toLowerCase())
      );
    }

    // Sort
    filtered.sort((a, b) => {
      const { field, direction } = state.sort;
      let aValue = a[field];
      let bValue = b[field];

      if (field === "priority") {
        const priorityOrder = { low: 1, medium: 2, high: 3 };
        aValue = priorityOrder[aValue];
        bValue = priorityOrder[bValue];
      }

      if (direction === "asc") {
        return aValue > bValue ? 1 : -1;
      } else {
        return aValue < bValue ? 1 : -1;
      }
    });

    return filtered;
  }, [state.todos, state.filter, state.sort]);

  // Actions
  const addTodo = useCallback((todoData) => {
    dispatch({ type: "ADD_TODO", payload: todoData });
  }, []);

  const toggleTodo = useCallback((id) => {
    dispatch({ type: "TOGGLE_TODO", payload: { id } });
  }, []);
```

```
  const updateTodo = useCallback((id, updates) => {
    dispatch({ type: "UPDATE_TODO", payload: { id, updates } });
  }, []);

  const deleteTodo = useCallback((id) => {
    dispatch({ type: "DELETE_TODO", payload: { id } });
  }, []);

  const setFilter = useCallback((filterType, value) => {
    dispatch({ type: "SET_FILTER", payload: { filterType, value } });
  }, []);

  const setSort = useCallback((field, direction) => {
    dispatch({ type: "SET_SORT", payload: { field, direction } });
  }, []);

  const addCategory = useCallback((category) => {
    dispatch({ type: "ADD_CATEGORY", payload: { category } });
  }, []);

  // Statistics
  const stats = useMemo(() => {
    const total = state.todos.length;
    const completed = state.todos.filter((todo) => todo.completed).length;
    const overdue = state.todos.filter((todo) => {
      if (!todo.dueDate || todo.completed) return false;
      return new Date(todo.dueDate) < new Date();
    }).length;

    return {
      total,
      completed,
      pending: total - completed,
      overdue,
      completionRate: total > 0 ? Math.round((completed / total) * 100) : 0,
    };
  }, [state.todos]);

  return {
    todos: filteredTodos,
    allTodos: state.todos,
    categories: state.categories,
    filter: state.filter,
    sort: state.sort,
    stats,
    addTodo,
    toggleTodo,
    updateTodo,
    deleteTodo,
    setFilter,
    setSort,
    addCategory,
  };
```

```
}

function AdvancedTodoApp() {
  const {
    todos,
    categories,
    filter,
    sort,
    stats,
    addTodo,
    toggleTodo,
    updateTodo,
    deleteTodo,
    setFilter,
    setSort,
    addCategory,
  } = useTodos();

  return (
    <div className="advanced-todo-app">
      <TodoStats stats={stats} />
      <TodoFilters
        filter={filter}
        categories={categories}
        onFilterChange={setFilter}
        onSortChange={setSort}
      />
      <TodoForm
        categories={categories}
        onAddTodo={addTodo}
        onAddCategory={addCategory}
      />
      <TodoList
        todos={todos}
        onToggle={toggleTodo}
        onUpdate={updateTodo}
        onDelete={deleteTodo}
      />
    </div>
  );
}
```

Challenge 2: Shopping Cart with Discounts

Create a shopping cart system with multiple discount types, tax calculation, and shipping options.

▶ 💡 Solution

```
const initialCartState = {
  items: [],
  discounts: {
```

```
    coupon: null,
    bulk: null,
    loyalty: null,
  },
  shipping: {
    method: "standard",
    cost: 0,
    estimatedDays: 5,
  },
  tax: {
    rate: 0.08,
    amount: 0,
  },
  totals: {
    subtotal: 0,
    discountAmount: 0,
    taxAmount: 0,
    shippingCost: 0,
    total: 0,
  },
};

function cartReducer(state, action) {
  switch (action.type) {
    case "ADD_ITEM": {
      const { product, quantity = 1 } = action.payload;
      const existingItem = state.items.find((item) => item.id === product.id);

      let newItems;
      if (existingItem) {
        newItems = state.items.map((item) =>
          item.id === product.id
            ? { ...item, quantity: item.quantity + quantity }
            : item
        );
      } else {
        newItems = [...state.items, { ...product, quantity }];
      }

      return calculateTotals({ ...state, items: newItems });
    }

    case "UPDATE_QUANTITY": {
      const { id, quantity } = action.payload;

      if (quantity <= 0) {
        return cartReducer(state, { type: "REMOVE_ITEM", payload: { id } });
      }

      const newItems = state.items.map((item) =>
        item.id === id ? { ...item, quantity } : item
      );

      return calculateTotals({ ...state, items: newItems });
```

```
    }

    case "REMOVE_ITEM": {
      const { id } = action.payload;
      const newItems = state.items.filter((item) => item.id !== id);
      return calculateTotals({ ...state, items: newItems });
    }

    case "APPLY_COUPON": {
      const { coupon } = action.payload;
      const newState = {
        ...state,
        discounts: {
          ...state.discounts,
          coupon,
        },
      };
      return calculateTotals(newState);
    }

    case "REMOVE_COUPON": {
      const newState = {
        ...state,
        discounts: {
          ...state.discounts,
          coupon: null,
        },
      };
      return calculateTotals(newState);
    }

    case "SET_SHIPPING": {
      const { method, cost, estimatedDays } = action.payload;
      const newState = {
        ...state,
        shipping: { method, cost, estimatedDays },
      };
      return calculateTotals(newState);
    }

    case "CLEAR_CART":
      return initialCartState;

    default:
      return state;
  }
}

function calculateTotals(state) {
  const subtotal = state.items.reduce(
    (sum, item) => sum + item.price * item.quantity,
    0
  );
```

```
    // Calculate discounts
    let discountAmount = 0;

    // Coupon discount
    if (state.discounts.coupon) {
      const coupon = state.discounts.coupon;
      if (coupon.type === "percentage") {
        discountAmount += subtotal * (coupon.value / 100);
      } else if (coupon.type === "fixed") {
        discountAmount += Math.min(coupon.value, subtotal);
      }
    }

    // Bulk discount (10% off if subtotal > $100)
    if (subtotal > 100) {
      const bulkDiscount = subtotal * 0.1;
      discountAmount += bulkDiscount;
      state.discounts.bulk = {
        type: "bulk",
        description: "10% off orders over $100",
        amount: bulkDiscount,
      };
    } else {
      state.discounts.bulk = null;
    }

    // Loyalty discount (5% off for returning customers)
    // This would typically come from user data
    const isLoyalCustomer = true; // Placeholder
    if (isLoyalCustomer && subtotal > 50) {
      const loyaltyDiscount = subtotal * 0.05;
      discountAmount += loyaltyDiscount;
      state.discounts.loyalty = {
        type: "loyalty",
        description: "5% loyalty discount",
        amount: loyaltyDiscount,
      };
    } else {
      state.discounts.loyalty = null;
    }

    const discountedSubtotal = Math.max(0, subtotal - discountAmount);
    const taxAmount = discountedSubtotal * state.tax.rate;
    const total = discountedSubtotal + taxAmount + state.shipping.cost;

    return {
      ...state,
      totals: {
        subtotal,
        discountAmount,
        taxAmount,
        shippingCost: state.shipping.cost,
        total,
      },
```

```
  };
}

function useShoppingCart() {
  const [state, dispatch] = useReducer(cartReducer, initialCartState);

  const addItem = useCallback((product, quantity = 1) => {
    dispatch({ type: "ADD_ITEM", payload: { product, quantity } });
  }, []);

  const updateQuantity = useCallback((id, quantity) => {
    dispatch({ type: "UPDATE_QUANTITY", payload: { id, quantity } });
  }, []);

  const removeItem = useCallback((id) => {
    dispatch({ type: "REMOVE_ITEM", payload: { id } });
  }, []);

  const applyCoupon = useCallback((coupon) => {
    dispatch({ type: "APPLY_COUPON", payload: { coupon } });
  }, []);

  const removeCoupon = useCallback(() => {
    dispatch({ type: "REMOVE_COUPON" });
  }, []);

  const setShipping = useCallback((method, cost, estimatedDays) => {
    dispatch({
      type: "SET_SHIPPING",
      payload: { method, cost, estimatedDays },
    });
  }, []);

  const clearCart = useCallback(() => {
    dispatch({ type: "CLEAR_CART" });
  }, []);

  const itemCount = useMemo(() => {
    return state.items.reduce((sum, item) => sum + item.quantity, 0);
  }, [state.items]);

  return {
    ...state,
    itemCount,
    addItem,
    updateQuantity,
    removeItem,
    applyCoupon,
    removeCoupon,
    setShipping,
    clearCart,
  };
}
```

```
function ShoppingCart() {
  const {
    items,
    discounts,
    shipping,
    totals,
    itemCount,
    addItem,
    updateQuantity,
    removeItem,
    applyCoupon,
    removeCoupon,
    setShipping,
  } = useShoppingCart();

  return (
    <div className="shopping-cart">
      <div className="cart-header">
        <h2>Shopping Cart ({itemCount} items)</h2>
      </div>

      <div className="cart-items">
        {items.map((item) => (
          <CartItem
            key={item.id}
            item={item}
            onUpdateQuantity={updateQuantity}
            onRemove={removeItem}
          />
        ))}
      </div>

      <CouponSection
        coupon={discounts.coupon}
        onApplyCoupon={applyCoupon}
        onRemoveCoupon={removeCoupon}
      />

      <ShippingOptions shipping={shipping} onSetShipping={setShipping} />

      <CartSummary totals={totals} discounts={discounts} shipping={shipping} />
    </div>
  );
}
```

🎯 When and Why to Use useReducer

✅ Use useReducer When:

1. **Complex State Logic**: Multiple related state values that change together
2. **State Transitions**: Next state depends on previous state
3. **Predictable Updates**: Need consistent, testable state changes

4. **Multiple Actions**: Different ways to update the same state
5. **State Validation**: Need to validate state changes
6. **Undo/Redo**: Need to track state history
7. **Global State**: Combined with Context for app-wide state

❌ Avoid useReducer When:

1. **Simple State**: Single primitive values (string, number, boolean)
2. **Independent State**: State values that don't relate to each other
3. **Frequent Updates**: High-frequency state changes (animations, mouse tracking)
4. **Server State**: Use data fetching libraries instead
5. **Form State**: Consider form libraries for complex forms

🤔 Decision Framework

```
Do you have multiple related state values?
├─ No → Use useState
└─ Yes
   ├─ Do state updates depend on previous state?
   │  ├─ No → Consider multiple useState
   │  └─ Yes → useReducer is good
   └─ Do you need predictable state transitions?
      ├─ No → useState might be simpler
      └─ Yes → useReducer is perfect
```

🚀 Performance Optimization

Memoizing Dispatch

```jsx
// ✅ dispatch is automatically stable
function MyComponent() {
  const [state, dispatch] = useReducer(reducer, initialState);

  // dispatch never changes, so this is safe
  const memoizedCallback = useCallback(() => {
    dispatch({ type: "SOME_ACTION" });
  }, [dispatch]); // dispatch can be omitted from deps

  return <ChildComponent onAction={memoizedCallback} />;
}
```

Optimizing Reducer Performance

```jsx
// ✅ Use early returns for performance
function optimizedReducer(state, action) {
  switch (action.type) {
```

```
    case "UPDATE_ITEM": {
      const { id, updates } = action.payload;

      // Early return if item doesn't exist
      const itemIndex = state.items.findIndex((item) => item.id === id);
      if (itemIndex === -1) {
        return state;
      }

      // Early return if no actual changes
      const currentItem = state.items[itemIndex];
      const hasChanges = Object.keys(updates).some(
        (key) => currentItem[key] !== updates[key]
      );
      if (!hasChanges) {
        return state;
      }

      // Only update if necessary
      const newItems = [...state.items];
      newItems[itemIndex] = { ...currentItem, ...updates };

      return {
        ...state,
        items: newItems,
      };
    }

    default:
      return state;
  }
}
```

Lazy Initialization

```
// ✅ Expensive initial state calculation
function init(initialCount) {
  // Expensive computation
  const expensiveValue = computeExpensiveValue(initialCount);
  return {
    count: initialCount,
    expensiveValue,
    history: [],
  };
}

function Counter({ initialCount }) {
  // init function only runs once
  const [state, dispatch] = useReducer(reducer, initialCount, init);

  return (
```

```
      <div>
        Count: {state.count}
        <button onClick={() => dispatch({ type: "increment" })}>+</button>
      </div>
    );
  }
```

Interview Insights

Common Questions

Q: When would you use useReducer instead of useState?

A: useReducer is better when:

- Managing complex state with multiple sub-values
- State updates depend on previous state
- Need predictable state transitions
- Multiple components need to trigger the same state changes
- Building state machines or complex workflows

Q: How does useReducer compare to Redux?

A: Similarities:

- Both use reducer pattern
- Predictable state updates
- Action-based state changes

Differences:

- useReducer is local to component tree
- Redux has global state and middleware
- Redux has time-travel debugging
- useReducer is simpler for component-level state

Q: Can you combine useReducer with Context?

A: Yes! This is a powerful pattern for global state management:

```
const StateContext = createContext();
const DispatchContext = createContext();

function AppProvider({ children }) {
  const [state, dispatch] = useReducer(appReducer, initialState);

  return (
    <StateContext.Provider value={state}>
      <DispatchContext.Provider value={dispatch}>
        {children}
      </DispatchContext.Provider>
```

```
      </StateContext.Provider>
    );
  }
```

Q: How do you test components that use useReducer?

A: Test the reducer function separately, then test component behavior:

```
// Test reducer
describe("todoReducer", () => {
  it("should add a todo", () => {
    const initialState = { todos: [] };
    const action = { type: "ADD_TODO", payload: { text: "Test" } };
    const newState = todoReducer(initialState, action);

    expect(newState.todos).toHaveLength(1);
    expect(newState.todos[0].text).toBe("Test");
  });
});

// Test component
test("adds todo when form is submitted", () => {
  render(<TodoApp />);

  fireEvent.change(screen.getByPlaceholderText("Add todo"), {
    target: { value: "New todo" },
  });
  fireEvent.click(screen.getByText("Add"));

  expect(screen.getByText("New todo")).toBeInTheDocument();
});
```

Code Review Red Flags

✕ Mutating state in reducer ✕ Complex logic in components instead of reducer ✕ Not handling all action types ✕ Missing default case in reducer ✕ Using useReducer for simple state ✕ Not memoizing expensive calculations

🎯 Key Takeaways

1. **useReducer is for complex state** - Use when useState becomes unwieldy
2. **Reducers must be pure** - No side effects, always return new state
3. **Actions describe what happened** - Use descriptive action types
4. **Combine with Context for global state** - Powerful alternative to Redux
5. **Test reducers separately** - Pure functions are easy to test
6. **Use action creators** - Improve maintainability and type safety
7. **Consider performance** - Early returns and memoization help

🏆 Best Practices

1. **Keep reducers pure** - No side effects or mutations
2. **Use action type constants** - Prevent typos and improve maintainability
3. **Structure actions consistently** - Use `{ type, payload }` pattern
4. **Handle all cases** - Always include default case
5. **Split large reducers** - Use reducer composition for complex state
6. **Document state shape** - Clear interfaces help team collaboration
7. **Use TypeScript** - Better developer experience and fewer bugs

🔧 Production Tips

- **Monitor reducer performance** with React DevTools Profiler
- **Use immer** for complex immutable updates
- **Implement middleware pattern** for logging and debugging
- **Consider reducer composition** for large applications
- **Use action creators** for complex action logic
- **Implement optimistic updates** for better UX
- **Add error boundaries** around components using reducers

Next up: We'll explore custom hooks and learn how to create reusable stateful logic! 🚀

13. Custom Hooks: Reusable Logic 🎣

> "Custom hooks are where React's composition model truly shines. They let you extract component logic into reusable functions." - React Team

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What custom hooks are and why they're powerful
- How to extract stateful logic into reusable hooks
- Common patterns for building custom hooks
- Best practices for hook composition
- Testing strategies for custom hooks
- Performance considerations

🔍 What Are Custom Hooks?

Custom hooks are JavaScript functions that:

1. **Start with "use"** (naming convention)
2. **Can call other hooks** (built-in or custom)
3. **Extract stateful logic** from components
4. **Enable logic reuse** across components
5. **Follow the rules of hooks**

Basic Example

```
// ✅ Simple custom hook
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  const decrement = useCallback(() => {
    setCount((prev) => prev - 1);
  }, []);

  const reset = useCallback(() => {
    setCount(initialValue);
  }, [initialValue]);

  return {
    count,
    increment,
    decrement,
    reset,
  };
}

// Usage in component
function Counter() {
  const { count, increment, decrement, reset } = useCounter(10);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

🏗️ Common Custom Hook Patterns

1. State Management Hooks

```
// ✅ Boolean state hook
function useToggle(initialValue = false) {
  const [value, setValue] = useState(initialValue);

  const toggle = useCallback(() => {
    setValue((prev) => !prev);
  }, []);
```

```
  const setTrue = useCallback(() => {
    setValue(true);
  }, []);

  const setFalse = useCallback(() => {
    setValue(false);
  }, []);

  return [value, { toggle, setTrue, setFalse }];
}

// ✅ Array state hook
function useArray(initialArray = []) {
  const [array, setArray] = useState(initialArray);

  const push = useCallback((element) => {
    setArray((prev) => [...prev, element]);
  }, []);

  const remove = useCallback((index) => {
    setArray((prev) => prev.filter((_, i) => i !== index));
  }, []);

  const update = useCallback((index, newElement) => {
    setArray((prev) =>
      prev.map((item, i) => (i === index ? newElement : item))
    );
  }, []);

  const clear = useCallback(() => {
    setArray([]);
  }, []);

  return {
    array,
    set: setArray,
    push,
    remove,
    update,
    clear,
  };
}

// Usage
function TodoList() {
  const { array: todos, push, remove, update } = useArray([]);
  const [newTodo, setNewTodo] = useState("");

  const addTodo = () => {
    if (newTodo.trim()) {
      push({ id: Date.now(), text: newTodo, completed: false });
      setNewTodo("");
    }
  };
```

```
  const toggleTodo = (index) => {
    const todo = todos[index];
    update(index, { ...todo, completed: !todo.completed });
  };

  return (
    <div>
      <input
        value={newTodo}
        onChange={(e) => setNewTodo(e.target.value)}
        onKeyPress={(e) => e.key === "Enter" && addTodo()}
      />
      <button onClick={addTodo}>Add</button>

      {todos.map((todo, index) => (
        <div key={todo.id}>
          <span
            style={{
              textDecoration: todo.completed ? "line-through" : "none",
            }}
            onClick={() => toggleTodo(index)}
          >
            {todo.text}
          </span>
          <button onClick={() => remove(index)}>Delete</button>
        </div>
      ))}
    </div>
  );
}
```

2. Data Fetching Hooks

```
// ✅ Generic fetch hook
function useFetch(url, options = {}) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let isCancelled = false;

    const fetchData = async () => {
      try {
        setLoading(true);
        setError(null);

        const response = await fetch(url, options);

        if (!response.ok) {
```

```
          throw new Error(`HTTP error! status: ${response.status}`);
        }

        const result = await response.json();

        if (!isCancelled) {
          setData(result);
        }
      } catch (err) {
        if (!isCancelled) {
          setError(err.message);
        }
      } finally {
        if (!isCancelled) {
          setLoading(false);
        }
      }
    };

    fetchData();

    return () => {
      isCancelled = true;
    };
  }, [url, JSON.stringify(options)]);

  const refetch = useCallback(() => {
    setLoading(true);
    setError(null);
    // Trigger useEffect by updating a dependency
  }, []);

  return { data, loading, error, refetch };
}

// ✅ Specific API hook
function useUsers() {
  const { data, loading, error, refetch } = useFetch("/api/users");

  const users = useMemo(() => data || [], [data]);

  return {
    users,
    loading,
    error,
    refetch,
  };
}

// Usage
function UserList() {
  const { users, loading, error, refetch } = useUsers();

  if (loading) return <div>Loading users...</div>;
```

```
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <button onClick={refetch}>Refresh</button>
      {users.map((user) => (
        <div key={user.id}>{user.name}</div>
      ))}
    </div>
  );
}
```

3. Local Storage Hooks

```
// ✅ Local storage hook with sync
function useLocalStorage(key, initialValue) {
  // Get initial value from localStorage or use provided initial value
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(`Error reading localStorage key "${key}":`, error);
      return initialValue;
    }
  });

  // Return a wrapped version of useState's setter function that persists the new
value to localStorage
  const setValue = useCallback(
    (value) => {
      try {
        // Allow value to be a function so we have the same API as useState
        const valueToStore =
          value instanceof Function ? value(storedValue) : value;

        setStoredValue(valueToStore);
        window.localStorage.setItem(key, JSON.stringify(valueToStore));
      } catch (error) {
        console.error(`Error setting localStorage key "${key}":`, error);
      }
    },
    [key, storedValue]
  );

  // Listen for changes to this key in other tabs/windows
  useEffect(() => {
    const handleStorageChange = (e) => {
      if (e.key === key && e.newValue !== null) {
        try {
          setStoredValue(JSON.parse(e.newValue));
```

```
        } catch (error) {
          console.error(
            `Error parsing localStorage value for key "${key}":`,
            error
          );
        }
      }
    };

    window.addEventListener("storage", handleStorageChange);
    return () => window.removeEventListener("storage", handleStorageChange);
  }, [key]);

  return [storedValue, setValue];
}

// Usage
function Settings() {
  const [theme, setTheme] = useLocalStorage("theme", "light");
  const [language, setLanguage] = useLocalStorage("language", "en");

  return (
    <div>
      <select value={theme} onChange={(e) => setTheme(e.target.value)}>
        <option value="light">Light</option>
        <option value="dark">Dark</option>
      </select>

      <select value={language} onChange={(e) => setLanguage(e.target.value)}>
        <option value="en">English</option>
        <option value="es">Spanish</option>
        <option value="fr">French</option>
      </select>
    </div>
  );
}
```

4. Event Listener Hooks

```
// ✅ Generic event listener hook
function useEventListener(eventName, handler, element = window) {
  // Create a ref that stores handler
  const savedHandler = useRef();

  // Update ref.current value if handler changes
  useEffect(() => {
    savedHandler.current = handler;
  }, [handler]);

  useEffect(() => {
    // Make sure element supports addEventListener
```

```
    const isSupported = element && element.addEventListener;
    if (!isSupported) return;

    // Create event listener that calls handler function stored in ref
    const eventListener = (event) => savedHandler.current(event);

    element.addEventListener(eventName, eventListener);

    // Remove event listener on cleanup
    return () => {
      element.removeEventListener(eventName, eventListener);
    };
  }, [eventName, element]);
}

// ✅ Specific keyboard hook
function useKeyPress(targetKey) {
  const [keyPressed, setKeyPressed] = useState(false);

  const downHandler = useCallback(
    (event) => {
      if (event.key === targetKey) {
        setKeyPressed(true);
      }
    },
    [targetKey]
  );

  const upHandler = useCallback(
    (event) => {
      if (event.key === targetKey) {
        setKeyPressed(false);
      }
    },
    [targetKey]
  );

  useEventListener("keydown", downHandler);
  useEventListener("keyup", upHandler);

  return keyPressed;
}

// ✅ Window size hook
function useWindowSize() {
  const [windowSize, setWindowSize] = useState({
    width: undefined,
    height: undefined,
  });

  useEffect(() => {
    function handleResize() {
      setWindowSize({
        width: window.innerWidth,
```

```
        height: window.innerHeight,
      });
    }

    // Set initial size
    handleResize();

    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return windowSize;
}

// Usage
function ResponsiveComponent() {
  const { width } = useWindowSize();
  const escapePressed = useKeyPress("Escape");

  useEffect(() => {
    if (escapePressed) {
      console.log("Escape was pressed!");
    }
  }, [escapePressed]);

  return (
    <div>
      <p>Window width: {width}px</p>
      <p>Screen size: {width < 768 ? "Mobile" : "Desktop"}</p>
      {escapePressed && <p>Escape is being pressed!</p>}
    </div>
  );
}
```

🔄 Hook Composition

Combining Multiple Hooks

```
// ✅ Complex hook that combines multiple concerns
function useShoppingCart() {
  // Use multiple custom hooks
  const { array: items, push, remove, update, clear } = useArray([]);
  const [cartTotal, setCartTotal] = useLocalStorage("cartTotal", 0);
  const [isOpen, { toggle: toggleCart, setFalse: closeCart }] =
    useToggle(false);

  // Calculate total whenever items change
  useEffect(() => {
    const total = items.reduce(
      (sum, item) => sum + item.price * item.quantity,
      0
```

```
    );
    setCartTotal(total);
  }, [items, setCartTotal]);

  // Add item to cart
  const addItem = useCallback(
    (product) => {
      const existingIndex = items.findIndex((item) => item.id === product.id);

      if (existingIndex >= 0) {
        // Update quantity if item exists
        const existingItem = items[existingIndex];
        update(existingIndex, {
          ...existingItem,
          quantity: existingItem.quantity + 1,
        });
      } else {
        // Add new item
        push({ ...product, quantity: 1 });
      }
    },
    [items, push, update]
  );

  // Remove item from cart
  const removeItem = useCallback(
    (productId) => {
      const index = items.findIndex((item) => item.id === productId);
      if (index >= 0) {
        remove(index);
      }
    },
    [items, remove]
  );

  // Update item quantity
  const updateQuantity = useCallback(
    (productId, newQuantity) => {
      if (newQuantity <= 0) {
        removeItem(productId);
        return;
      }

      const index = items.findIndex((item) => item.id === productId);
      if (index >= 0) {
        update(index, { ...items[index], quantity: newQuantity });
      }
    },
    [items, update, removeItem]
  );

  return {
    items,
    total: cartTotal,
```

```
    itemCount: items.length,
    isOpen,
    addItem,
    removeItem,
    updateQuantity,
    clearCart: clear,
    toggleCart,
    closeCart,
  };
}

// Usage
function App() {
  const cart = useShoppingCart();

  const sampleProduct = {
    id: 1,
    name: "React Handbook",
    price: 29.99,
  };

  return (
    <div>
      <button onClick={() => cart.addItem(sampleProduct)}>Add to Cart</button>

      <button onClick={cart.toggleCart}>
        Cart ({cart.itemCount}) - ${cart.total.toFixed(2)}
      </button>

      {cart.isOpen && (
        <div className="cart-dropdown">
          {cart.items.map((item) => (
            <div key={item.id}>
              <span>{item.name}</span>
              <input
                type="number"
                value={item.quantity}
                onChange={(e) =>
                  cart.updateQuantity(item.id, parseInt(e.target.value))
                }
              />
              <button onClick={() => cart.removeItem(item.id)}>Remove</button>
            </div>
          ))}
          <button onClick={cart.clearCart}>Clear Cart</button>
        </div>
      )}
    </div>
  );
}
```

Mini-Challenge: Build a Form Hook

Challenge 1: useForm Hook

Create a custom hook that manages form state, validation, and submission:

```
// Your task: Implement this hook
function useForm(initialValues, validationRules) {
  // Should return:
  // - values: current form values
  // - errors: validation errors
  // - touched: which fields have been touched
  // - isValid: whether form is valid
  // - isSubmitting: submission state
  // - handleChange: function to update field values
  // - handleBlur: function to mark fields as touched
  // - handleSubmit: function to handle form submission
  // - reset: function to reset form
}

// Usage should work like this:
function ContactForm() {
  const {
    values,
    errors,
    touched,
    isValid,
    isSubmitting,
    handleChange,
    handleBlur,
    handleSubmit,
    reset,
  } = useForm(
    { name: "", email: "", message: "" },
    {
      name: (value) =>
        value.length < 2 ? "Name must be at least 2 characters" : "",
      email: (value) => (!/\S+@\S+\.\S+/.test(value) ? "Invalid email" : ""),
      message: (value) =>
        value.length < 10 ? "Message must be at least 10 characters" : "",
    }
  );

  const onSubmit = async (formData) => {
    // Simulate API call
    await new Promise((resolve) => setTimeout(resolve, 1000));
    console.log("Form submitted:", formData);
    reset();
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        name="name"
```

```
          value={values.name}
          onChange={handleChange}
          onBlur={handleBlur}
          placeholder="Name"
        />
        {touched.name && errors.name && <span>{errors.name}</span>}

        <input
          name="email"
          value={values.email}
          onChange={handleChange}
          onBlur={handleBlur}
          placeholder="Email"
        />
        {touched.email && errors.email && <span>{errors.email}</span>}

        <textarea
          name="message"
          value={values.message}
          onChange={handleChange}
          onBlur={handleBlur}
          placeholder="Message"
        />
        {touched.message && errors.message && <span>{errors.message}</span>}

        <button type="submit" disabled={!isValid || isSubmitting}>
          {isSubmitting ? "Submitting..." : "Submit"}
        </button>

        <button type="button" onClick={reset}>
          Reset
        </button>
      </form>
    );
}
```

Solution:

```
function useForm(initialValues, validationRules = {}) {
  const [values, setValues] = useState(initialValues);
  const [errors, setErrors] = useState({});
  const [touched, setTouched] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  // Validate a single field
  const validateField = useCallback(
    (name, value) => {
      const rule = validationRules[name];
      return rule ? rule(value) : "";
    },
    [validationRules]
```

```
  );

  // Validate all fields
  const validateForm = useCallback(() => {
    const newErrors = {};

    Object.keys(values).forEach((name) => {
      const error = validateField(name, values[name]);
      if (error) {
        newErrors[name] = error;
      }
    });

    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  }, [values, validateField]);

  // Check if form is valid
  const isValid = useMemo(() => {
    return Object.keys(values).every((name) => {
      const error = validateField(name, values[name]);
      return !error;
    });
  }, [values, validateField]);

  // Handle input change
  const handleChange = useCallback(
    (event) => {
      const { name, value } = event.target;

      setValues((prev) => ({ ...prev, [name]: value }));

      // Validate field if it's been touched
      if (touched[name]) {
        const error = validateField(name, value);
        setErrors((prev) => ({ ...prev, [name]: error }));
      }
    },
    [touched, validateField]
  );

  // Handle input blur
  const handleBlur = useCallback(
    (event) => {
      const { name } = event.target;

      setTouched((prev) => ({ ...prev, [name]: true }));

      // Validate field on blur
      const error = validateField(name, values[name]);
      setErrors((prev) => ({ ...prev, [name]: error }));
    },
    [values, validateField]
  );
```

```
  // Handle form submission
  const handleSubmit = useCallback(
    (onSubmit) => {
      return async (event) => {
        event.preventDefault();

        // Mark all fields as touched
        const allTouched = Object.keys(values).reduce((acc, key) => {
          acc[key] = true;
          return acc;
        }, {});
        setTouched(allTouched);

        // Validate form
        if (!validateForm()) {
          return;
        }

        setIsSubmitting(true);

        try {
          await onSubmit(values);
        } catch (error) {
          console.error("Form submission error:", error);
        } finally {
          setIsSubmitting(false);
        }
      };
    },
    [values, validateForm]
  );

  // Reset form
  const reset = useCallback(() => {
    setValues(initialValues);
    setErrors({});
    setTouched({});
    setIsSubmitting(false);
  }, [initialValues]);

  return {
    values,
    errors,
    touched,
    isValid,
    isSubmitting,
    handleChange,
    handleBlur,
    handleSubmit,
    reset,
  };
}
```

❌ Common Mistakes

1. Breaking Rules of Hooks

```
// ❌ Calling hooks conditionally
function BadHook(shouldUseState) {
  if (shouldUseState) {
    const [state, setState] = useState(0); // This breaks rules!
  }
  return null;
}

// ✅ Always call hooks at top level
function GoodHook(shouldUseState) {
  const [state, setState] = useState(shouldUseState ? 0 : null);

  if (!shouldUseState) {
    return { state: null, setState: () => {} };
  }

  return { state, setState };
}
```

2. Not Memoizing Callbacks

```
// ❌ Creating new functions on every render
function useCounter() {
  const [count, setCount] = useState(0);

  // These functions are recreated on every render!
  const increment = () => setCount((prev) => prev + 1);
  const decrement = () => setCount((prev) => prev - 1);

  return { count, increment, decrement };
}

// ✅ Memoize callbacks
function useCounter() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  const decrement = useCallback(() => {
    setCount((prev) => prev - 1);
  }, []);
```

```
  return { count, increment, decrement };
}
```

3. Missing Dependencies

```
// ❌ Missing dependencies in useEffect
function useFetch(url) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then(setData);
  }, []); // Missing 'url' dependency!

  return data;
}

// ✅ Include all dependencies
function useFetch(url) {
  const [data, setData] = useState(null);

  useEffect(() => {
    let isCancelled = false;

    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        if (!isCancelled) {
          setData(data);
        }
      });

    return () => {
      isCancelled = true;
    };
  }, [url]); // Include 'url' dependency

  return data;
}
```

4. Not Handling Cleanup

```
// ❌ Memory leak - no cleanup
function useInterval(callback, delay) {
  useEffect(() => {
    const id = setInterval(callback, delay);
    // Missing cleanup!
  }, [callback, delay]);
```

```
}

// ✅ Proper cleanup
function useInterval(callback, delay) {
  const savedCallback = useRef();

  useEffect(() => {
    savedCallback.current = callback;
  }, [callback]);

  useEffect(() => {
    function tick() {
      savedCallback.current();
    }

    if (delay !== null) {
      const id = setInterval(tick, delay);
      return () => clearInterval(id); // Cleanup!
    }
  }, [delay]);
}
```

🧪 Testing Custom Hooks

Using React Testing Library

```
import { renderHook, act } from "@testing-library/react";
import { useCounter } from "./useCounter";

describe("useCounter", () => {
  it("should initialize with default value", () => {
    const { result } = renderHook(() => useCounter());

    expect(result.current.count).toBe(0);
  });

  it("should initialize with custom value", () => {
    const { result } = renderHook(() => useCounter(10));

    expect(result.current.count).toBe(10);
  });

  it("should increment count", () => {
    const { result } = renderHook(() => useCounter());

    act(() => {
      result.current.increment();
    });

    expect(result.current.count).toBe(1);
  });
```

```
  it("should decrement count", () => {
    const { result } = renderHook(() => useCounter(5));

    act(() => {
      result.current.decrement();
    });

    expect(result.current.count).toBe(4);
  });

  it("should reset to initial value", () => {
    const { result } = renderHook(() => useCounter(10));

    act(() => {
      result.current.increment();
      result.current.increment();
    });

    expect(result.current.count).toBe(12);

    act(() => {
      result.current.reset();
    });

    expect(result.current.count).toBe(10);
  });
});
```

Testing Hooks with Dependencies

```
import { renderHook } from "@testing-library/react";
import { useFetch } from "./useFetch";

// Mock fetch
global.fetch = jest.fn();

describe("useFetch", () => {
  beforeEach(() => {
    fetch.mockClear();
  });

  it("should fetch data successfully", async () => {
    const mockData = { id: 1, name: "Test" };
    fetch.mockResolvedValueOnce({
      ok: true,
      json: async () => mockData,
    });

    const { result, waitForNextUpdate } = renderHook(() =>
      useFetch("/api/test")
```

```
    );

    expect(result.current.loading).toBe(true);
    expect(result.current.data).toBe(null);

    await waitForNextUpdate();

    expect(result.current.loading).toBe(false);
    expect(result.current.data).toEqual(mockData);
    expect(result.current.error).toBe(null);
  });

  it("should handle fetch errors", async () => {
    fetch.mockRejectedValueOnce(new Error("Network error"));

    const { result, waitForNextUpdate } = renderHook(() =>
      useFetch("/api/test")
    );

    await waitForNextUpdate();

    expect(result.current.loading).toBe(false);
    expect(result.current.data).toBe(null);
    expect(result.current.error).toBe("Network error");
  });
});
```

🎯 When and Why to Use Custom Hooks

✅ Use Custom Hooks When:

1. **Reusing Stateful Logic**: Same logic needed in multiple components
2. **Complex State Management**: Multiple related state values
3. **Side Effect Patterns**: Common useEffect patterns
4. **API Integration**: Consistent data fetching patterns
5. **Event Handling**: Reusable event listener logic
6. **Form Management**: Common form state and validation
7. **Local Storage**: Persistent state management

❌ Avoid Custom Hooks When:

1. **Simple State**: Single useState is sufficient
2. **Component-Specific Logic**: Logic only used in one place
3. **No State or Effects**: Pure utility functions don't need hooks
4. **Over-Abstraction**: Making simple things complex

🤔 Decision Framework

```
Do you have stateful logic?
├─ No → Use regular functions
```

```
└─ Yes
   ├─ Is it used in multiple components?
   │  ├─ No → Keep in component (for now)
   │  └─ Yes → Custom hook is good
   └─ Is the logic complex?
      ├─ No → Consider if abstraction adds value
      └─ Yes → Custom hook helps organization
```

🚀 Performance Considerations

Memoization in Custom Hooks

```
// ✅ Properly memoized custom hook
function useExpensiveCalculation(data, options) {
  // Memoize expensive calculation
  const result = useMemo(() => {
    return performExpensiveCalculation(data, options);
  }, [data, options]);

  // Memoize callbacks
  const recalculate = useCallback(() => {
    // Force recalculation
  }, []);

  // Memoize returned object to prevent unnecessary re-renders
  return useMemo(
    () => ({
      result,
      recalculate,
    }),
    [result, recalculate]
  );
}
```

Avoiding Unnecessary Re-renders

```
// ❌ This will cause re-renders even when values don't change
function useBadHook() {
  const [count, setCount] = useState(0);

  // New object on every render!
  return {
    count,
    increment: () => setCount((prev) => prev + 1),
  };
}

// ✅ Stable references
function useGoodHook() {
```

```
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  // Memoize the returned object
  return useMemo(
    () => ({
      count,
      increment,
    }),
    [count, increment]
  );
}
```

Next up: We'll explore React.memo and optimization techniques to make your apps lightning fast! ⚡

14. React.memo & Optimization 🚀

> "Premature optimization is the root of all evil, but knowing when and how to optimize is the mark of a skilled developer." - Adapted from Donald Knuth

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What React.memo is and how it works
- When and why to use React.memo
- How to optimize component re-renders
- Performance measurement techniques
- Common optimization patterns and anti-patterns
- The relationship between memo, useMemo, and useCallback

🔍 What is React.memo?

React.memo is a higher-order component that memoizes the result of a component. It only re-renders when its props change, similar to `PureComponent` for class components.

Basic Example

```
// ✅ Basic React.memo usage
const ExpensiveComponent = React.memo(function ExpensiveComponent({
  name,
  age,
}) {
  console.log("ExpensiveComponent rendered");
```

```
  // Simulate expensive calculation
  const expensiveValue = useMemo(() => {
    let result = 0;
    for (let i = 0; i < 1000000; i++) {
      result += i;
    }
    return result;
  }, []);

  return (
    <div>
      <h3>
        {name} (Age: {age})
      </h3>
      <p>Expensive calculation result: {expensiveValue}</p>
    </div>
  );
});

// Parent component
function App() {
  const [count, setCount] = useState(0);
  const [user] = useState({ name: "John", age: 30 });

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>

      {/* This won't re-render when count changes */}
      <ExpensiveComponent name={user.name} age={user.age} />
    </div>
  );
}
```

🔧 How React.memo Works

Shallow Comparison

React.memo performs a shallow comparison of props by default:

```
// ✅ These will NOT trigger re-render (shallow equal)
const props1 = { name: "John", age: 30 };
const props2 = { name: "John", age: 30 };
// React.memo will prevent re-render

// ❌ These WILL trigger re-render (not shallow equal)
const props3 = { name: "John", user: { id: 1 } };
const props4 = { name: "John", user: { id: 1 } };
// Different object references, so re-render happens
```

Custom Comparison Function

```
// ✅ Custom comparison for complex props
const UserCard = React.memo(
  function UserCard({ user, settings }) {
    return (
      <div>
        <h3>{user.name}</h3>
        <p>Theme: {settings.theme}</p>
      </div>
    );
  },
  (prevProps, nextProps) => {
    // Return true if props are equal (skip re-render)
    // Return false if props are different (re-render)

    return (
      prevProps.user.id === nextProps.user.id &&
      prevProps.user.name === nextProps.user.name &&
      prevProps.settings.theme === nextProps.settings.theme
    );
  }
);

// Usage
function App() {
  const [count, setCount] = useState(0);
  const [user] = useState({ id: 1, name: "John", email: "john@example.com" });
  const [settings] = useState({ theme: "dark", notifications: true });

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>

      {/* Won't re-render when count changes, even though user/settings are
objects */}
      <UserCard user={user} settings={settings} />
    </div>
  );
}
```

🎯 When to Use React.memo

✅ Good Use Cases

1. **Expensive Rendering**: Components with heavy calculations or complex UI
2. **Stable Props**: Props that don't change often
3. **Large Lists**: List items that render frequently
4. **Pure Components**: Components that only depend on props
5. **Leaf Components**: Components at the end of the component tree

```
// ✅ Good candidate for React.memo
const ProductCard = React.memo(function ProductCard({ product }) {
  // Expensive image processing
  const processedImage = useMemo(() => {
    return processImage(product.image);
  }, [product.image]);

  return (
    <div className="product-card">
      <img src={processedImage} alt={product.name} />
      <h3>{product.name}</h3>
      <p>${product.price}</p>
    </div>
  );
});

// Usage in a list
function ProductList({ products, searchTerm }) {
  const [sortOrder, setSortOrder] = useState("name");

  const filteredProducts = useMemo(() => {
    return products
      .filter((p) => p.name.toLowerCase().includes(searchTerm.toLowerCase()))
      .sort((a, b) => a[sortOrder].localeCompare(b[sortOrder]));
  }, [products, searchTerm, sortOrder]);

  return (
    <div>
      <select value={sortOrder} onChange={(e) => setSortOrder(e.target.value)}>
        <option value="name">Name</option>
        <option value="price">Price</option>
      </select>

      {/* Each ProductCard will only re-render if its product prop changes */}
      {filteredProducts.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}
```

❌ When NOT to Use React.memo

1. **Props Change Frequently**: If props change on every render
2. **Simple Components**: Very lightweight components
3. **Always Re-rendering Parent**: If parent always re-renders anyway
4. **Complex Comparison**: When custom comparison is more expensive than re-rendering

```
// ❌ Bad candidate for React.memo
const SimpleButton = React.memo(function SimpleButton({ onClick, children }) {
```

```
  return <button onClick={onClick}>{children}</button>;
});
// This is too simple to benefit from memoization

// ❌ Props change every render
function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
      {/* onClick is a new function every render, so memo won't help */}
      <SimpleButton onClick={() => setCount(count + 1)}>
        Count: {count}
      </SimpleButton>
    </div>
  );
}
```

🔄 React.memo + useCallback + useMemo

The Holy Trinity of Optimization

```
// ✅ Properly optimized component hierarchy
const TodoItem = React.memo(function TodoItem({ todo, onToggle, onDelete }) {
  console.log(`TodoItem ${todo.id} rendered`);

  return (
    <div className={`todo-item ${todo.completed ? "completed" : ""}`}>
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => onToggle(todo.id)}
      />
      <span>{todo.text}</span>
      <button onClick={() => onDelete(todo.id)}>Delete</button>
    </div>
  );
});

const TodoList = React.memo(function TodoList({ todos, onToggle, onDelete }) {
  console.log("TodoList rendered");

  return (
    <div className="todo-list">
      {todos.map((todo) => (
        <TodoItem
          key={todo.id}
          todo={todo}
          onToggle={onToggle}
          onDelete={onDelete}
        />
```

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

  function App() {
    const [todos, setTodos] = useState([
      { id: 1, text: "Learn React", completed: false },
      { id: 2, text: "Build an app", completed: false },
    ]);
    const [newTodo, setNewTodo] = useState("");

    // ✅ Memoized callbacks prevent unnecessary re-renders
    const handleToggle = useCallback((id) => {
      setTodos((prev) =>
        prev.map((todo) =>
          todo.id === id ? { ...todo, completed: !todo.completed } : todo
        )
      );
    }, []);

    const handleDelete = useCallback((id) => {
      setTodos((prev) => prev.filter((todo) => todo.id !== id));
    }, []);

    const handleAdd = useCallback(() => {
      if (newTodo.trim()) {
        setTodos((prev) => [
          ...prev,
          {
            id: Date.now(),
            text: newTodo,
            completed: false,
          },
        ]);
        setNewTodo("");
      }
    }, [newTodo]);

    // ✅ Memoized filtered todos
    const incompleteTodos = useMemo(() => {
      return todos.filter((todo) => !todo.completed);
    }, [todos]);

    return (
      <div>
        <div>
          <input
            value={newTodo}
            onChange={(e) => setNewTodo(e.target.value)}
            placeholder="Add new todo"
          />
          <button onClick={handleAdd}>Add</button>
        </div>
```

```
      <h3>Incomplete Todos ({incompleteTodos.length})</h3>
      <TodoList
        todos={incompleteTodos}
        onToggle={handleToggle}
        onDelete={handleDelete}
      />

      <h3>All Todos</h3>
      <TodoList todos={todos} onToggle={handleToggle} onDelete={handleDelete} />
    </div>
  );
}
```

📊 Performance Measurement

Using React DevTools Profiler

```
// ✅ Component with performance monitoring
const PerformanceDemo = React.memo(function PerformanceDemo({ data }) {
  const startTime = performance.now();

  // Expensive calculation
  const processedData = useMemo(() => {
    console.log("Processing data...");
    return data.map((item) => ({
      ...item,
      processed: true,
      timestamp: Date.now(),
    }));
  }, [data]);

  useEffect(() => {
    const endTime = performance.now();
    console.log(
      `PerformanceDemo render took ${endTime - startTime} milliseconds`
    );
  });

  return (
    <div>
      <h3>Processed Data ({processedData.length} items)</h3>
      {processedData.map((item) => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  );
});
```

Custom Performance Hook

```
// ✅ Hook to measure render performance
function useRenderTime(componentName) {
  const renderStartTime = useRef();

  // Mark start of render
  renderStartTime.current = performance.now();

  useEffect(() => {
    const renderEndTime = performance.now();
    const renderTime = renderEndTime - renderStartTime.current;

    console.log(`${componentName} render time: ${renderTime.toFixed(2)}ms`);
  });
}

// Usage
const MyComponent = React.memo(function MyComponent({ data }) {
  useRenderTime("MyComponent");

  return <div>{/* Component content */}</div>;
});
```

🎯 Mini-Challenge: Optimize a Data Dashboard

Challenge: Build an Optimized Dashboard

Create a dashboard that efficiently renders large amounts of data:

```
// Your task: Optimize this dashboard for performance
function DataDashboard() {
  const [data, setData] = useState([]);
  const [filter, setFilter] = useState("");
  const [sortBy, setSortBy] = useState("name");
  const [refreshCount, setRefreshCount] = useState(0);

  // Simulate data fetching
  useEffect(() => {
    const fetchData = async () => {
      // Simulate API call
      const newData = Array.from({ length: 1000 }, (_, i) => ({
        id: i,
        name: `Item ${i}`,
        value: Math.random() * 100,
        category: ["A", "B", "C"][i % 3],
        timestamp: new Date(Date.now() - Math.random() * 86400000),
      }));
      setData(newData);
    };

    fetchData();
```

```
  }, [refreshCount]);

  // Filter and sort data
  const processedData = data
    .filter((item) => item.name.toLowerCase().includes(filter.toLowerCase()))
    .sort((a, b) => {
      if (sortBy === "name") return a.name.localeCompare(b.name);
      if (sortBy === "value") return b.value - a.value;
      return 0;
    });

  return (
    <div>
      <div className="controls">
        <input
          placeholder="Filter items..."
          value={filter}
          onChange={(e) => setFilter(e.target.value)}
        />

        <select value={sortBy} onChange={(e) => setSortBy(e.target.value)}>
          <option value="name">Sort by Name</option>
          <option value="value">Sort by Value</option>
        </select>

        <button onClick={() => setRefreshCount((c) => c + 1)}>
          Refresh Data
        </button>
      </div>

      <div className="stats">
        <div>Total Items: {data.length}</div>
        <div>Filtered Items: {processedData.length}</div>
        <div>
          Average Value:{" "}
          {(
            processedData.reduce((sum, item) => sum + item.value, 0) /
            processedData.length
          ).toFixed(2)}
        </div>
      </div>

      <div className="data-grid">
        {processedData.map((item) => (
          <div key={item.id} className="data-item">
            <h4>{item.name}</h4>
            <p>Value: {item.value.toFixed(2)}</p>
            <p>Category: {item.category}</p>
            <small>{item.timestamp.toLocaleDateString()}</small>
          </div>
        ))}
      </div>
    </div>
```

```
  );
}
```

Solution: Optimized Dashboard

```
// ✅ Optimized data item component
const DataItem = React.memo(function DataItem({ item }) {
  return (
    <div className="data-item">
      <h4>{item.name}</h4>
      <p>Value: {item.value.toFixed(2)}</p>
      <p>Category: {item.category}</p>
      <small>{item.timestamp.toLocaleDateString()}</small>
    </div>
  );
});

// ✅ Optimized stats component
const DataStats = React.memo(function DataStats({
  totalItems,
  filteredItems,
  averageValue,
}) {
  return (
    <div className="stats">
      <div>Total Items: {totalItems}</div>
      <div>Filtered Items: {filteredItems}</div>
      <div>Average Value: {averageValue}</div>
    </div>
  );
});

// ✅ Optimized controls component
const DataControls = React.memo(function DataControls({
  filter,
  onFilterChange,
  sortBy,
  onSortChange,
  onRefresh,
}) {
  return (
    <div className="controls">
      <input
        placeholder="Filter items..."
        value={filter}
        onChange={onFilterChange}
      />

      <select value={sortBy} onChange={onSortChange}>
        <option value="name">Sort by Name</option>
        <option value="value">Sort by Value</option>
```

```
        </select>

        <button onClick={onRefresh}>Refresh Data</button>
      </div>
    );
  });

  // ✅ Optimized main dashboard
  function DataDashboard() {
    const [data, setData] = useState([]);
    const [filter, setFilter] = useState("");
    const [sortBy, setSortBy] = useState("name");
    const [refreshCount, setRefreshCount] = useState(0);

    // Simulate data fetching
    useEffect(() => {
      const fetchData = async () => {
        const newData = Array.from({ length: 1000 }, (_, i) => ({
          id: i,
          name: `Item ${i}`,
          value: Math.random() * 100,
          category: ["A", "B", "C"][i % 3],
          timestamp: new Date(Date.now() - Math.random() * 86400000),
        }));
        setData(newData);
      };

      fetchData();
    }, [refreshCount]);

    // ✅ Memoized data processing
    const processedData = useMemo(() => {
      return data
        .filter((item) => item.name.toLowerCase().includes(filter.toLowerCase()))
        .sort((a, b) => {
          if (sortBy === "name") return a.name.localeCompare(b.name);
          if (sortBy === "value") return b.value - a.value;
          return 0;
        });
    }, [data, filter, sortBy]);

    // ✅ Memoized statistics
    const stats = useMemo(() => {
      const averageValue =
        processedData.length > 0
          ? (
              processedData.reduce((sum, item) => sum + item.value, 0) /
              processedData.length
            ).toFixed(2)
          : "0.00";

      return {
        totalItems: data.length,
        filteredItems: processedData.length,
```

```
      averageValue,
    };
  }, [data.length, processedData]);

  // ✅ Memoized event handlers
  const handleFilterChange = useCallback((e) => {
    setFilter(e.target.value);
  }, []);

  const handleSortChange = useCallback((e) => {
    setSortBy(e.target.value);
  }, []);

  const handleRefresh = useCallback(() => {
    setRefreshCount((c) => c + 1);
  }, []);

  return (
    <div>
      <DataControls
        filter={filter}
        onFilterChange={handleFilterChange}
        sortBy={sortBy}
        onSortChange={handleSortChange}
        onRefresh={handleRefresh}
      />

      <DataStats
        totalItems={stats.totalItems}
        filteredItems={stats.filteredItems}
        averageValue={stats.averageValue}
      />

      <div className="data-grid">
        {processedData.map((item) => (
          <DataItem key={item.id} item={item} />
        ))}
      </div>
    </div>
  );
}
```

❌ Common Optimization Mistakes

1. Over-Memoization

```
// ❌ Unnecessary memoization
const SimpleText = React.memo(function SimpleText({ text }) {
  return <span>{text}</span>; // Too simple to benefit from memo
});
```

```
// ❌ Memoizing everything
function App() {
  const [count, setCount] = useState(0);

  // These change every render anyway!
  const memoizedCount = useMemo(() => count, [count]);
  const memoizedIncrement = useCallback(() => setCount(count + 1), [count]);

  return (
    <div>
      <span>{memoizedCount}</span>
      <button onClick={memoizedIncrement}>+</button>
    </div>
  );
}
```

2. Incorrect Dependencies

```
// ❌ Missing dependencies
const BadComponent = React.memo(function BadComponent({ items, multiplier }) {
  const expensiveValue = useMemo(() => {
    return items.reduce((sum, item) => sum + item.value * multiplier, 0);
  }, [items]); // Missing 'multiplier' dependency!

  return <div>{expensiveValue}</div>;
});

// ✅ Correct dependencies
const GoodComponent = React.memo(function GoodComponent({ items, multiplier }) {
  const expensiveValue = useMemo(() => {
    return items.reduce((sum, item) => sum + item.value * multiplier, 0);
  }, [items, multiplier]); // All dependencies included

  return <div>{expensiveValue}</div>;
});
```

3. Comparing Functions in memo

```
// ❌ Functions will always be different
const BadComponent = React.memo(
  function BadComponent({ onClick }) {
    return <button onClick={onClick}>Click me</button>;
  },
  (prevProps, nextProps) => {
    // This will always return false because functions are recreated
    return prevProps.onClick === nextProps.onClick;
  }
);
```

```
// ✅ Use useCallback in parent instead
function Parent() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    setCount((c) => c + 1);
  }, []);

  return <BadComponent onClick={handleClick} />;
}
```

🎤 Interview Insights

Common Questions

Q: When should you use React.memo?

A: Use React.memo when:

- Component renders frequently with the same props
- Component has expensive rendering logic
- Component is a leaf node in the component tree
- Props are stable and don't change often

Q: What's the difference between React.memo, useMemo, and useCallback?

A:

- **React.memo**: Memoizes entire component based on props
- **useMemo**: Memoizes computed values within a component
- **useCallback**: Memoizes function references

Q: How do you measure if React.memo is helping performance?

A: Use:

- React DevTools Profiler
- Console.log in components to track renders
- Performance.now() to measure render times
- React's built-in performance monitoring

Q: Can React.memo cause memory leaks?

A: React.memo itself doesn't cause memory leaks, but:

- Memoized components hold references to props
- Custom comparison functions might hold references
- Always clean up subscriptions and timers

Code Review Red Flags

✕ Memoizing every component without measurement ✕ Complex custom comparison functions ✕ Missing dependencies in useMemo/useCallback ✕ Memoizing props that change every render ✕ Not using React DevTools to verify optimizations ✕ Optimizing before identifying performance bottlenecks

🎯 Key Takeaways

1. **Measure first, optimize second** - Use React DevTools Profiler
2. **React.memo prevents re-renders** - Only when props haven't changed
3. **Shallow comparison by default** - Deep objects need custom comparison
4. **Combine with useCallback/useMemo** - For stable prop references
5. **Don't over-optimize** - Simple components don't need memoization
6. **Profile your optimizations** - Verify they actually help
7. **Consider the whole tree** - Optimize parent components first

🏆 Best Practices

1. **Start with React DevTools** - Identify actual performance bottlenecks
2. **Optimize from top down** - Parent optimizations often help more
3. **Use memo for expensive components** - Complex calculations or large lists
4. **Memoize stable callbacks** - Use useCallback for event handlers
5. **Avoid premature optimization** - Don't memo everything by default
6. **Test your optimizations** - Measure before and after
7. **Consider code splitting** - Sometimes better than memoization

🔧 Production Tips

- **Use React.memo for list items** - Especially in large lists
- **Memoize context values** - Prevent unnecessary provider re-renders
- **Consider virtualization** - For very large lists (react-window)
- **Profile in production mode** - Development mode has extra overhead
- **Monitor bundle size** - Memoization adds code
- **Use React Concurrent features** - Suspense and transitions
- **Implement error boundaries** - Around memoized components

---

Next up: We'll explore advanced patterns like render props, compound components, and more! 🎨

15. Advanced Patterns 🎨

> "Good architecture is not about the framework you choose, but about the patterns you apply." - Clean Architecture Principles

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- Render Props pattern and its use cases

- Compound Components pattern
- Higher-Order Components (HOCs)
- Provider pattern with Context
- State Reducer pattern
- Controlled vs Uncontrolled component patterns
- When and why to use each pattern

🎭 Render Props Pattern

Render Props is a technique for sharing code between components using a prop whose value is a function.

Basic Render Props

```
// ✅ Mouse tracker with render props
class MouseTracker extends React.Component {
  state = { x: 0, y: 0 };

  handleMouseMove = (event) => {
    this.setState({
      x: event.clientX,
      y: event.clientY,
    });
  };

  render() {
    return (
      <div style={{ height: "100vh" }} onMouseMove={this.handleMouseMove}>
        {/* Call the render prop function with current state */}
        {this.props.render(this.state)}
      </div>
    );
  }
}

// Usage
function App() {
  return (
    <div>
      <h1>Mouse Tracker Demo</h1>

      <MouseTracker
        render={({ x, y }) => (
          <div>
            <h2>
              Mouse position: ({x}, {y})
            </h2>
            <div
              style={{
                position: "absolute",
                left: x - 10,
                top: y - 10,
```

```
                width: 20,
                height: 20,
                backgroundColor: "red",
                borderRadius: "50%",
                pointerEvents: "none",
              }}
            />
          </div>
        )}
      />
    </div>
  );
}
```

Modern Render Props with Hooks

```
// ✅ Hook-based render props
function useMouse() {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMouseMove = (event) => {
      setPosition({ x: event.clientX, y: event.clientY });
    };

    window.addEventListener("mousemove", handleMouseMove);
    return () => window.removeEventListener("mousemove", handleMouseMove);
  }, []);

  return position;
}

// Render props component using the hook
function MouseProvider({ children }) {
  const mousePosition = useMouse();

  return (
    <div>
      {typeof children === "function" ? children(mousePosition) : children}
    </div>
  );
}

// Usage
function App() {
  return (
    <MouseProvider>
      {({ x, y }) => (
        <div>
          <h2>
            Mouse at: ({x}, {y})
```

```
            </h2>
            <MouseCursor x={x} y={y} />
          </div>
        )}
      </MouseProvider>
    );
}

function MouseCursor({ x, y }) {
  return (
    <div
      style={{
        position: "fixed",
        left: x - 10,
        top: y - 10,
        width: 20,
        height: 20,
        backgroundColor: "blue",
        borderRadius: "50%",
        pointerEvents: "none",
        zIndex: 9999,
      }}
    />
  );
}
```

Data Fetching with Render Props

```
// ✅ Generic data fetcher
function DataFetcher({ url, children }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let isCancelled = false;

    const fetchData = async () => {
      try {
        setLoading(true);
        setError(null);

        const response = await fetch(url);
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }

        const result = await response.json();

        if (!isCancelled) {
          setData(result);
```

```
        }
      } catch (err) {
        if (!isCancelled) {
          setError(err.message);
        }
      } finally {
        if (!isCancelled) {
          setLoading(false);
        }
      }
    };

    fetchData();

    return () => {
      isCancelled = true;
    };
  }, [url]);

  return children({ data, loading, error });
}

// Usage
function UserProfile({ userId }) {
  return (
    <DataFetcher url={`/api/users/${userId}`}>
      {({ data: user, loading, error }) => {
        if (loading) return <div>Loading user...</div>;
        if (error) return <div>Error: {error}</div>;
        if (!user) return <div>User not found</div>;

        return (
          <div>
            <h2>{user.name}</h2>
            <p>Email: {user.email}</p>
            <p>Role: {user.role}</p>
          </div>
        );
      }}
    </DataFetcher>
  );
}

function PostsList() {
  return (
    <DataFetcher url="/api/posts">
      {({ data: posts, loading, error }) => {
        if (loading) return <div>Loading posts...</div>;
        if (error) return <div>Error: {error}</div>;

        return (
          <div>
            <h2>Posts</h2>
            {posts?.map((post) => (
```

```
              <article key={post.id}>
                <h3>{post.title}</h3>
                <p>{post.excerpt}</p>
              </article>
            ))}
          </div>
        );
      }}
    </DataFetcher>
  );
}
```

🧩 Compound Components Pattern

Compound components work together to form a complete UI. Think of HTML elements like `<select>` and `<option>`.

Basic Compound Components

```
// ✅ Accordion compound component
const AccordionContext = createContext();

function Accordion({ children, allowMultiple = false }) {
  const [openItems, setOpenItems] = useState(new Set());

  const toggleItem = useCallback(
    (id) => {
      setOpenItems((prev) => {
        const newSet = new Set(prev);

        if (newSet.has(id)) {
          newSet.delete(id);
        } else {
          if (!allowMultiple) {
            newSet.clear();
          }
          newSet.add(id);
        }

        return newSet;
      });
    },
    [allowMultiple]
  );

  const value = useMemo(
    () => ({
      openItems,
      toggleItem,
    }),
    [openItems, toggleItem]
```

```
  );

  return (
    <AccordionContext.Provider value={value}>
      <div className="accordion">{children}</div>
    </AccordionContext.Provider>
  );
}

function AccordionItem({ id, children }) {
  const { openItems, toggleItem } = useContext(AccordionContext);
  const isOpen = openItems.has(id);

  return (
    <div className={`accordion-item ${isOpen ? "open" : ""}`}>
      {React.Children.map(children, (child) => {
        if (React.isValidElement(child)) {
          return React.cloneElement(child, { id, isOpen });
        }
        return child;
      })}
    </div>
  );
}

function AccordionHeader({ id, isOpen, children }) {
  const { toggleItem } = useContext(AccordionContext);

  return (
    <button
      className="accordion-header"
      onClick={() => toggleItem(id)}
      aria-expanded={isOpen}
    >
      {children}
      <span className={`accordion-icon ${isOpen ? "rotated" : ""}`}>▼</span>
    </button>
  );
}

function AccordionContent({ isOpen, children }) {
  return (
    <div className={`accordion-content ${isOpen ? "open" : ""}`}>
      <div className="accordion-content-inner">{children}</div>
    </div>
  );
}

// Attach sub-components to main component
Accordion.Item = AccordionItem;
Accordion.Header = AccordionHeader;
Accordion.Content = AccordionContent;

// Usage
```

```
function App() {
  return (
    <div>
      <h1>FAQ</h1>

      <Accordion allowMultiple={false}>
        <Accordion.Item id="item1">
          <Accordion.Header>What is React?</Accordion.Header>
          <Accordion.Content>
            React is a JavaScript library for building user interfaces. It was
            created by Facebook and is now maintained by Meta.
          </Accordion.Content>
        </Accordion.Item>

        <Accordion.Item id="item2">
          <Accordion.Header>What are React Hooks?</Accordion.Header>
          <Accordion.Content>
            Hooks are functions that let you use state and other React features
            in functional components. They were introduced in React 16.8.
          </Accordion.Content>
        </Accordion.Item>

        <Accordion.Item id="item3">
          <Accordion.Header>What is JSX?</Accordion.Header>
          <Accordion.Content>
            JSX is a syntax extension for JavaScript that looks similar to XML
            or HTML. It allows you to write HTML-like code in your JavaScript
            files.
          </Accordion.Content>
        </Accordion.Item>
      </Accordion>
    </div>
  );
}
```

Advanced Compound Components with Flexible API

```
// ✅ Flexible Modal compound component
const ModalContext = createContext();

function Modal({ children, isOpen, onClose }) {
  const value = useMemo(
    () => ({
      isOpen,
      onClose,
    }),
    [isOpen, onClose]
  );

  useEffect(() => {
    const handleEscape = (e) => {
```

```
      if (e.key === "Escape" && isOpen) {
        onClose();
      }
    };

    document.addEventListener("keydown", handleEscape);
    return () => document.removeEventListener("keydown", handleEscape);
  }, [isOpen, onClose]);

  if (!isOpen) return null;

  return (
    <ModalContext.Provider value={value}>
      <div className="modal-overlay" onClick={onClose}>
        <div className="modal-content" onClick={(e) => e.stopPropagation()}>
          {children}
        </div>
      </div>
    </ModalContext.Provider>
  );
}

function ModalHeader({ children }) {
  const { onClose } = useContext(ModalContext);

  return (
    <div className="modal-header">
      <h2>{children}</h2>
      <button className="modal-close" onClick={onClose}>
        ×
      </button>
    </div>
  );
}

function ModalBody({ children }) {
  return <div className="modal-body">{children}</div>;
}

function ModalFooter({ children }) {
  return <div className="modal-footer">{children}</div>;
}

// Attach sub-components
Modal.Header = ModalHeader;
Modal.Body = ModalBody;
Modal.Footer = ModalFooter;

// Usage
function App() {
  const [isModalOpen, setIsModalOpen] = useState(false);

  return (
    <div>
```

```
      <button onClick={() => setIsModalOpen(true)}>Open Modal</button>

      <Modal isOpen={isModalOpen} onClose={() => setIsModalOpen(false)}>
        <Modal.Header>Confirm Action</Modal.Header>

        <Modal.Body>
          <p>Are you sure you want to delete this item?</p>
          <p>This action cannot be undone.</p>
        </Modal.Body>

        <Modal.Footer>
          <button onClick={() => setIsModalOpen(false)}>Cancel</button>
          <button
            className="danger"
            onClick={() => {
              // Perform delete action
              console.log("Item deleted");
              setIsModalOpen(false);
            }}
          >
            Delete
          </button>
        </Modal.Footer>
      </Modal>
    </div>
  );
}
```

🎯 Higher-Order Components (HOCs)

HOCs are functions that take a component and return a new component with additional functionality.

Basic HOC Example

```
// ✅ withLoading HOC
function withLoading(WrappedComponent) {
  return function WithLoadingComponent(props) {
    if (props.isLoading) {
      return (
        <div className="loading-spinner">
          <div className="spinner"></div>
          <p>Loading...</p>
        </div>
      );
    }

    return <WrappedComponent {...props} />;
  };
}

// Usage
```

```
function UserProfile({ user }) {
  return (
    <div>
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
    </div>
  );
}

const UserProfileWithLoading = withLoading(UserProfile);

function App() {
  const [user, setUser] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    // Simulate API call
    setTimeout(() => {
      setUser({ name: "John Doe", email: "john@example.com" });
      setIsLoading(false);
    }, 2000);
  }, []);

  return <UserProfileWithLoading user={user} isLoading={isLoading} />;
}
```

Advanced HOC with Configuration

```
// ✅ withAuth HOC with configuration
function withAuth(WrappedComponent, options = {}) {
  const {
    redirectTo = "/login",
    requiredRoles = [],
    fallbackComponent: FallbackComponent,
  } = options;

  return function WithAuthComponent(props) {
    const { user, isAuthenticated } = useAuth(); // Custom hook

    // Not authenticated
    if (!isAuthenticated) {
      if (FallbackComponent) {
        return <FallbackComponent />;
      }

      // In a real app, you'd use React Router
      window.location.href = redirectTo;
      return null;
    }

    // Check roles if required
```

```
    if (requiredRoles.length > 0) {
      const hasRequiredRole = requiredRoles.some((role) =>
        user.roles?.includes(role)
      );

      if (!hasRequiredRole) {
        return (
          <div className="access-denied">
            <h2>Access Denied</h2>
            <p>You don't have permission to view this page.</p>
          </div>
        );
      }
    }

    return <WrappedComponent {...props} user={user} />;
  };
}

// Usage
function AdminPanel({ user }) {
  return (
    <div>
      <h1>Admin Panel</h1>
      <p>Welcome, {user.name}!</p>
      <button>Manage Users</button>
      <button>View Analytics</button>
    </div>
  );
}

function LoginPrompt() {
  return (
    <div>
      <h2>Please log in</h2>
      <button>Login</button>
    </div>
  );
}

const ProtectedAdminPanel = withAuth(AdminPanel, {
  requiredRoles: ["admin"],
  fallbackComponent: LoginPrompt,
});
```

Provider Pattern

The Provider pattern uses React Context to share data across the component tree.

Theme Provider Example

```
// ✅ Theme provider with multiple themes
const ThemeContext = createContext();

const themes = {
  light: {
    background: "#ffffff",
    text: "#000000",
    primary: "#007bff",
    secondary: "#6c757d",
  },
  dark: {
    background: "#121212",
    text: "#ffffff",
    primary: "#bb86fc",
    secondary: "#03dac6",
  },
  blue: {
    background: "#e3f2fd",
    text: "#0d47a1",
    primary: "#1976d2",
    secondary: "#42a5f5",
  },
};

function ThemeProvider({ children }) {
  const [currentTheme, setCurrentTheme] = useState("light");

  const theme = themes[currentTheme];

  const toggleTheme = useCallback(() => {
    setCurrentTheme((prev) => {
      const themeNames = Object.keys(themes);
      const currentIndex = themeNames.indexOf(prev);
      const nextIndex = (currentIndex + 1) % themeNames.length;
      return themeNames[nextIndex];
    });
  }, []);

  const setTheme = useCallback((themeName) => {
    if (themes[themeName]) {
      setCurrentTheme(themeName);
    }
  }, []);

  const value = useMemo(
    () => ({
      theme,
      currentTheme,
      toggleTheme,
      setTheme,
      availableThemes: Object.keys(themes),
    }),
    [theme, currentTheme, toggleTheme, setTheme]
```

```
  );

  return (
    <ThemeContext.Provider value={value}>
      <div
        style={{
          backgroundColor: theme.background,
          color: theme.text,
          minHeight: "100vh",
          transition: "all 0.3s ease",
        }}
      >
        {children}
      </div>
    </ThemeContext.Provider>
  );
}

// Custom hook for using theme
function useTheme() {
  const context = useContext(ThemeContext);

  if (!context) {
    throw new Error("useTheme must be used within a ThemeProvider");
  }

  return context;
}

// Components using the theme
function Header() {
  const { theme, toggleTheme, currentTheme } = useTheme();

  return (
    <header
      style={{
        padding: "1rem",
        borderBottom: `2px solid ${theme.primary}`,
        display: "flex",
        justifyContent: "space-between",
        alignItems: "center",
      }}
    >
      <h1 style={{ color: theme.primary }}>My App</h1>

      <button
        onClick={toggleTheme}
        style={{
          backgroundColor: theme.primary,
          color: theme.background,
          border: "none",
          padding: "0.5rem 1rem",
          borderRadius: "4px",
          cursor: "pointer",
```

```
          }}
        >
          Theme: {currentTheme}
        </button>
      </header>
    );
  }

  function Card({ title, children }) {
    const { theme } = useTheme();

    return (
      <div
        style={{
          border: `1px solid ${theme.secondary}`,
          borderRadius: "8px",
          padding: "1rem",
          margin: "1rem",
          backgroundColor: theme.background,
          boxShadow: `0 2px 4px ${theme.secondary}30`,
        }}
      >
        <h3 style={{ color: theme.primary, marginTop: 0 }}>{title}</h3>
        {children}
      </div>
    );
  }

  // App using the provider
  function App() {
    return (
      <ThemeProvider>
        <Header />

        <main style={{ padding: "1rem" }}>
          <Card title="Welcome">
            <p>This is a themed application!</p>
            <p>Click the theme button to cycle through different themes.</p>
          </Card>

          <Card title="Features">
            <ul>
              <li>Multiple theme support</li>
              <li>Smooth transitions</li>
              <li>Context-based state management</li>
            </ul>
          </Card>
        </main>
      </ThemeProvider>
    );
  }
```

🔄 State Reducer Pattern

The State Reducer pattern gives users control over how state updates work.

Flexible Counter with State Reducer

```
// ✅ Counter with customizable state reducer
function useCounter({
  initialValue = 0,
  min = -Infinity,
  max = Infinity,
  step = 1,
  reducer = (state, action) => state, // Default: no-op reducer
} = {}) {
  // Internal reducer that handles the core logic
  const internalReducer = useCallback(
    (state, action) => {
      switch (action.type) {
        case "INCREMENT":
          return Math.min(max, state + step);
        case "DECREMENT":
          return Math.max(min, state - step);
        case "SET":
          return Math.max(min, Math.min(max, action.payload));
        case "RESET":
          return initialValue;
        default:
          return state;
      }
    },
    [min, max, step, initialValue]
  );

  // Combined reducer: user reducer gets first chance to handle actions
  const combinedReducer = useCallback(
    (state, action) => {
      // Let user reducer handle the action first
      const userResult = reducer(state, action);

      // If user reducer returned the same state, use internal reducer
      if (userResult === state) {
        return internalReducer(state, action);
      }

      // User reducer modified state, respect their decision
      return userResult;
    },
    [reducer, internalReducer]
  );

  const [count, dispatch] = useReducer(combinedReducer, initialValue);
```

```
  const increment = useCallback(() => {
    dispatch({ type: "INCREMENT" });
  }, []);

  const decrement = useCallback(() => {
    dispatch({ type: "DECREMENT" });
  }, []);

  const set = useCallback((value) => {
    dispatch({ type: "SET", payload: value });
  }, []);

  const reset = useCallback(() => {
    dispatch({ type: "RESET" });
  }, []);

  return {
    count,
    increment,
    decrement,
    set,
    reset,
    dispatch, // Allow direct dispatch for custom actions
  };
}

// Usage examples
function BasicCounter() {
  const counter = useCounter({ initialValue: 10, min: 0, max: 100 });

  return (
    <div>
      <h3>Basic Counter: {counter.count}</h3>
      <button onClick={counter.decrement}>-</button>
      <button onClick={counter.increment}>+</button>
      <button onClick={counter.reset}>Reset</button>
    </div>
  );
}

function CustomCounter() {
  // Custom reducer that adds logging and prevents odd numbers
  const customReducer = useCallback((state, action) => {
    console.log("Counter action:", action.type, "Current state:", state);

    // Prevent setting odd numbers
    if (action.type === "SET" && action.payload % 2 !== 0) {
      console.log("Odd numbers not allowed!");
      return state; // Don't change state
    }

    // Let the default reducer handle other cases
    return state;
  }, []);
```

```
  const counter = useCounter({
    initialValue: 0,
    step: 2,
    reducer: customReducer,
  });

  return (
    <div>
      <h3>Custom Counter (Even Numbers Only): {counter.count}</h3>
      <button onClick={counter.decrement}>-2</button>
      <button onClick={counter.increment}>+2</button>
      <button onClick={() => counter.set(7)}>Set to 7 (will fail)</button>
      <button onClick={() => counter.set(8)}>Set to 8 (will work)</button>
      <button onClick={counter.reset}>Reset</button>
    </div>
  );
}

function App() {
  return (
    <div>
      <BasicCounter />
      <hr />
      <CustomCounter />
    </div>
  );
}
```

🎯 Mini-Challenge: Build a Flexible Form Component

Challenge: Create a Compound Form Component

Build a form system using compound components that supports:

- Field validation
- Custom field types
- Form-level state management
- Flexible layout

```
// Your task: Implement this form system
// Usage should work like this:
function ContactForm() {
  const handleSubmit = (data) => {
    console.log("Form submitted:", data);
  };

  return (
    <Form onSubmit={handleSubmit}>
      <Form.Field name="name" required>
        <Form.Label>Full Name</Form.Label>
```

```
        <Form.Input placeholder="Enter your name" />
        <Form.Error />
      </Form.Field>

      <Form.Field name="email" required validate={validateEmail}>
        <Form.Label>Email Address</Form.Label>
        <Form.Input type="email" placeholder="Enter your email" />
        <Form.Error />
      </Form.Field>

      <Form.Field name="message" required>
        <Form.Label>Message</Form.Label>
        <Form.Textarea rows={4} placeholder="Enter your message" />
        <Form.Error />
      </Form.Field>

      <Form.Actions>
        <Form.Reset>Clear</Form.Reset>
        <Form.Submit>Send Message</Form.Submit>
      </Form.Actions>
    </Form>
  );
}

function validateEmail(value) {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return emailRegex.test(value) ? "" : "Please enter a valid email address";
}
```

Solution: Compound Form Component

```
// ✅ Form context and provider
const FormContext = createContext();

function Form({ children, onSubmit, initialValues = {} }) {
  const [values, setValues] = useState(initialValues);
  const [errors, setErrors] = useState({});
  const [touched, setTouched] = useState({});

  const setValue = useCallback(
    (name, value) => {
      setValues((prev) => ({ ...prev, [name]: value }));

      // Clear error when user starts typing
      if (errors[name]) {
        setErrors((prev) => ({ ...prev, [name]: "" }));
      }
    },
    [errors]
  );
```

```
  const setError = useCallback((name, error) => {
    setErrors((prev) => ({ ...prev, [name]: error }));
  }, []);

  const setTouched = useCallback((name) => {
    setTouched((prev) => ({ ...prev, [name]: true }));
  }, []);

  const validateField = useCallback(
    (name, value, validator) => {
      if (validator) {
        const error = validator(value);
        setError(name, error);
        return !error;
      }
      return true;
    },
    [setError]
  );

  const reset = useCallback(() => {
    setValues(initialValues);
    setErrors({});
    setTouched({});
  }, [initialValues]);

  const handleSubmit = useCallback(
    (e) => {
      e.preventDefault();

      // Validate all fields
      const formData = new FormData(e.target);
      const fieldValidations = [];

      // Get all field validators from form elements
      const fields = e.target.querySelectorAll("[data-field-name]");
      fields.forEach((field) => {
        const name = field.dataset.fieldName;
        const validator = field.dataset.validator;
        const required = field.hasAttribute("required");
        const value = values[name] || "";

        if (required && !value.trim()) {
          setError(name, "This field is required");
          fieldValidations.push(false);
        } else if (validator && window[validator]) {
          const isValid = validateField(name, value, window[validator]);
          fieldValidations.push(isValid);
        } else {
          fieldValidations.push(true);
        }

        setTouched(name);
      });
```

```
      // Submit if all validations pass
      if (fieldValidations.every(Boolean)) {
        onSubmit(values);
      }
    },
    [values, validateField, setError, setTouched, onSubmit]
  );

  const value = useMemo(
    () => ({
      values,
      errors,
      touched,
      setValue,
      setError,
      setTouched,
      validateField,
      reset,
    }),
    [
      values,
      errors,
      touched,
      setValue,
      setError,
      setTouched,
      validateField,
      reset,
    ]
  );

  return (
    <FormContext.Provider value={value}>
      <form onSubmit={handleSubmit}>{children}</form>
    </FormContext.Provider>
  );
}

// Field wrapper component
function FormField({ name, required, validate, children }) {
  const { values, errors, touched } = useContext(FormContext);

  const fieldValue = values[name] || "";
  const fieldError = errors[name];
  const fieldTouched = touched[name];
  const showError = fieldTouched && fieldError;

  const fieldContext = useMemo(
    () => ({
      name,
      value: fieldValue,
      error: fieldError,
      touched: fieldTouched,
```

```
      showError,
      required,
      validate,
    }),
    [name, fieldValue, fieldError, fieldTouched, showError, required, validate]
  );

  return (
    <FieldContext.Provider value={fieldContext}>
      <div className={`form-field ${showError ? "has-error" : ""}`}>
        {children}
      </div>
    </FieldContext.Provider>
  );
}

const FieldContext = createContext();

// Form input components
function FormLabel({ children }) {
  const { name, required } = useContext(FieldContext);

  return (
    <label htmlFor={name} className="form-label">
      {children}
      {required && <span className="required">*</span>}
    </label>
  );
}

function FormInput({ type = "text", ...props }) {
  const { name, value, required, validate } = useContext(FieldContext);
  const { setValue, setTouched, validateField } = useContext(FormContext);

  const handleChange = (e) => {
    setValue(name, e.target.value);
  };

  const handleBlur = () => {
    setTouched(name);
    if (validate) {
      validateField(name, value, validate);
    }
  };

  return (
    <input
      id={name}
      name={name}
      type={type}
      value={value}
      onChange={handleChange}
      onBlur={handleBlur}
      required={required}
```

```
        data-field-name={name}
        data-validator={validate?.name}
        className="form-input"
        {...props}
      />
    );
  }

  function FormTextarea({ ...props }) {
    const { name, value, required, validate } = useContext(FieldContext);
    const { setValue, setTouched, validateField } = useContext(FormContext);

    const handleChange = (e) => {
      setValue(name, e.target.value);
    };

    const handleBlur = () => {
      setTouched(name);
      if (validate) {
        validateField(name, value, validate);
      }
    };

    return (
      <textarea
        id={name}
        name={name}
        value={value}
        onChange={handleChange}
        onBlur={handleBlur}
        required={required}
        data-field-name={name}
        data-validator={validate?.name}
        className="form-textarea"
        {...props}
      />
    );
  }

  function FormError() {
    const { showError, error } = useContext(FieldContext);

    if (!showError) return null;

    return <div className="form-error">{error}</div>;
  }

  function FormActions({ children }) {
    return <div className="form-actions">{children}</div>;
  }

  function FormSubmit({ children, ...props }) {
    return (
      <button type="submit" className="form-submit" {...props}>
```

```
      {children}
    </button>
  );
}

function FormReset({ children, ...props }) {
  const { reset } = useContext(FormContext);

  return (
    <button type="button" onClick={reset} className="form-reset" {...props}>
      {children}
    </button>
  );
}

// Attach sub-components
Form.Field = FormField;
Form.Label = FormLabel;
Form.Input = FormInput;
Form.Textarea = FormTextarea;
Form.Error = FormError;
Form.Actions = FormActions;
Form.Submit = FormSubmit;
Form.Reset = FormReset;
```

🎯 When to Use Each Pattern

Pattern Selection Guide

Pattern	Use When	Avoid When
Render Props	Sharing stateful logic, flexible rendering	Simple prop passing
Compound Components	Related components that work together	Unrelated components
HOCs	Cross-cutting concerns, legacy code	Modern React (prefer hooks)
Provider	Global state, theme, auth	Local component state
State Reducer	Complex state logic, user customization	Simple state updates

Decision Framework

```
What are you trying to achieve?

├─ Share stateful logic between components
│  ├─ Need flexible rendering → Render Props
│  └─ Logic only → Custom Hook
│
├─ Create reusable component API
│  ├─ Components work together → Compound Components
│  └─ Single component → Regular props
```

```
│
├─ Add functionality to existing components
│  ├─ Modern React → Custom Hook + Component
│  └─ Legacy/Class components → HOC
│
├─ Share data across component tree
│  ├─ Global state → Provider Pattern
│  └─ Local state → Props/State lifting
│
└─ Complex state management
   ├─ User needs control → State Reducer
   └─ Internal only → useReducer
```

Next up: We'll explore error boundaries and learn how to handle errors gracefully in React! 🛡️

16. Error Boundaries & Error Handling 🛡️

> "Errors are not failures, they're learning opportunities. In React, Error Boundaries help us learn gracefully." - React Philosophy

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- What Error Boundaries are and how they work
- How to create and implement Error Boundaries
- Different error handling strategies in React
- Best practices for error reporting and recovery
- How to handle async errors and promise rejections
- Testing error scenarios
- Production error monitoring

🔍 What are Error Boundaries?

Error Boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

Key Characteristics

- **Catch errors during rendering**
- **Catch errors in lifecycle methods**
- **Catch errors in constructors**
- **Display fallback UI**
- **Log error information**
- **Only work in class components** (for now)

What Error Boundaries DON'T Catch

- Event handlers
- Asynchronous code (setTimeout, promises)
- Errors during server-side rendering
- Errors thrown in the error boundary itself

🏗️ Creating Error Boundaries

Basic Error Boundary

```
// ✅ Basic Error Boundary class component
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null, errorInfo: null };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // Log the error to an error reporting service
    console.error("Error Boundary caught an error:", error, errorInfo);

    this.setState({
      error: error,
      errorInfo: errorInfo,
    });

    // You can also log the error to an error reporting service here
    // logErrorToService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // Fallback UI
      return (
        <div className="error-boundary">
          <h2>🚨 Something went wrong!</h2>
          <p>We're sorry, but something unexpected happened.</p>

          {/* Show error details in development */}
          {process.env.NODE_ENV === "development" && (
            <details style={{ whiteSpace: "pre-wrap" }}>
              <summary>Error Details (Development Only)</summary>
              <p>
                <strong>Error:</strong>{" "}
                {this.state.error && this.state.error.toString()}
              </p>
              <p>
                <strong>Stack Trace:</strong>
```

```
              </p>
              <pre>{this.state.errorInfo.componentStack}</pre>
            </details>
          )}

          <button
            onClick={() =>
              this.setState({ hasError: false, error: null, errorInfo: null })
            }
          >
            Try Again
          </button>
        </div>
      );
    }

    return this.props.children;
  }
}

// Usage
function App() {
  return (
    <ErrorBoundary>
      <Header />
      <MainContent />
      <Footer />
    </ErrorBoundary>
  );
}
```

Advanced Error Boundary with Hooks Support

```
// ✅ Advanced Error Boundary with more features
class AdvancedErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      hasError: false,
      error: null,
      errorInfo: null,
      eventId: null,
    };
  }

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

  componentDidCatch(error, errorInfo) {
    const { onError, enableLogging = true } = this.props;
```

```
    // Generate unique error ID
    const eventId = `error_${Date.now()}_${Math.random()
      .toString(36)
      .substr(2, 9)}`;

    this.setState({
      error,
      errorInfo,
      eventId,
    });

    // Custom error handler
    if (onError) {
      onError(error, errorInfo, eventId);
    }

    // Log to console in development
    if (enableLogging && process.env.NODE_ENV === "development") {
      console.group("🔒 Error Boundary Caught Error");
      console.error("Error:", error);
      console.error("Error Info:", errorInfo);
      console.error("Event ID:", eventId);
      console.groupEnd();
    }

    // Log to external service in production
    if (process.env.NODE_ENV === "production") {
      this.logErrorToService(error, errorInfo, eventId);
    }
  }

  logErrorToService = (error, errorInfo, eventId) => {
    // Example: Send to error monitoring service
    const errorData = {
      message: error.message,
      stack: error.stack,
      componentStack: errorInfo.componentStack,
      eventId,
      timestamp: new Date().toISOString(),
      userAgent: navigator.userAgent,
      url: window.location.href,
    };

    // Send to your error monitoring service
    // fetch('/api/errors', {
    //   method: 'POST',
    //   headers: { 'Content-Type': 'application/json' },
    //   body: JSON.stringify(errorData)
    // });

    console.log("Would send to error service:", errorData);
  };
```

```
  handleRetry = () => {
    this.setState({
      hasError: false,
      error: null,
      errorInfo: null,
      eventId: null,
    });
  };

  handleReload = () => {
    window.location.reload();
  };

  render() {
    const { fallback: Fallback, children } = this.props;
    const { hasError, error, errorInfo, eventId } = this.state;

    if (hasError) {
      // Use custom fallback component if provided
      if (Fallback) {
        return (
          <Fallback
            error={error}
            errorInfo={errorInfo}
            eventId={eventId}
            onRetry={this.handleRetry}
            onReload={this.handleReload}
          />
        );
      }

      // Default fallback UI
      return (
        <div className="error-boundary-container">
          <div className="error-boundary-content">
            <div className="error-icon">🛸</div>
            <h1>Oops! Something went wrong</h1>
            <p>We're sorry, but an unexpected error occurred.</p>

            {eventId && (
              <p className="error-id">
                Error ID: <code>{eventId}</code>
              </p>
            )}

            <div className="error-actions">
              <button className="btn btn-primary" onClick={this.handleRetry}>
                Try Again
              </button>

              <button className="btn btn-secondary" onClick={this.handleReload}>
                Reload Page
              </button>
            </div>
```

```
            {/* Development error details */}
            {process.env.NODE_ENV === "development" && (
              <details className="error-details">
                <summary>🔍 Error Details (Development)</summary>
                <div className="error-content">
                  <h3>Error Message:</h3>
                  <pre>{error?.toString()}</pre>

                  <h3>Component Stack:</h3>
                  <pre>{errorInfo?.componentStack}</pre>

                  <h3>Error Stack:</h3>
                  <pre>{error?.stack}</pre>
                </div>
              </details>
            )}
          </div>
        </div>
      );
    }

    return children;
  }
}

// Custom fallback component
function CustomErrorFallback({ error, errorInfo, eventId, onRetry, onReload }) {
  return (
    <div className="custom-error-fallback">
      <h2>🎭 Custom Error Handler</h2>
      <p>Something unexpected happened in our application.</p>

      {eventId && (
        <div className="error-reference">
          <strong>Reference ID:</strong> {eventId}
        </div>
      )}

      <div className="error-actions">
        <button onClick={onRetry}>🔄 Try Again</button>
        <button onClick={onReload}>🔃 Reload Page</button>
        <button onClick={() => window.history.back()}>⬅ Go Back</button>
      </div>
    </div>
  );
}

// Usage with custom fallback
function App() {
  const handleError = (error, errorInfo, eventId) => {
    // Custom error handling logic
    console.log("Custom error handler called:", { error, errorInfo, eventId });
```

```
    // Send to analytics
    // analytics.track('Error Occurred', { eventId, errorMessage: error.message
});
  };

  return (
    <AdvancedErrorBoundary fallback={CustomErrorFallback} onError={handleError}>
      <Router>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/profile" element={<Profile />} />
          <Route path="/settings" element={<Settings />} />
        </Routes>
      </Router>
    </AdvancedErrorBoundary>
  );
}
```

🎧 Error Boundary Hook (Custom Implementation)

Since Error Boundaries only work with class components, here's a custom hook approach:

```
// ✅ Custom hook for error handling
function useErrorHandler() {
  const [error, setError] = useState(null);

  const resetError = useCallback(() => {
    setError(null);
  }, []);

  const captureError = useCallback((error, errorInfo = {}) => {
    console.error("Error captured by useErrorHandler:", error);

    setError({
      message: error.message,
      stack: error.stack,
      timestamp: new Date().toISOString(),
      ...errorInfo,
    });

    // Log to external service
    if (process.env.NODE_ENV === "production") {
      // logErrorToService(error, errorInfo);
    }
  }, []);

  // Automatically capture unhandled promise rejections
  useEffect(() => {
    const handleUnhandledRejection = (event) => {
      captureError(new Error(event.reason), {
        type: "unhandledRejection",
```

```
        reason: event.reason,
      });
    };

    window.addEventListener("unhandledrejection", handleUnhandledRejection);

    return () => {
      window.removeEventListener(
        "unhandledrejection",
        handleUnhandledRejection
      );
    };
  }, [captureError]);

  return {
    error,
    captureError,
    resetError,
    hasError: !!error,
  };
}

// Error boundary component using the hook
function ErrorBoundaryWithHook({ children, fallback: Fallback }) {
  const { error, resetError, hasError } = useErrorHandler();

  if (hasError) {
    if (Fallback) {
      return <Fallback error={error} onRetry={resetError} />;
    }

    return (
      <div className="error-boundary">
        <h2>Something went wrong</h2>
        <p>{error.message}</p>
        <button onClick={resetError}>Try Again</button>
      </div>
    );
  }

  return children;
}

// Usage in functional components
function MyComponent() {
  const { captureError } = useErrorHandler();

  const handleAsyncOperation = async () => {
    try {
      const result = await riskyAsyncOperation();
      // Handle success
    } catch (error) {
      captureError(error, { context: "async operation" });
    }
```

```
  };

  return (
    <div>
      <button onClick={handleAsyncOperation}>Perform Risky Operation</button>
    </div>
  );
}
```

🔄 Handling Different Types of Errors

1. Async Errors and Promise Rejections

```
// ✅ Async error handling component
function AsyncErrorHandler({ children }) {
  const [asyncError, setAsyncError] = useState(null);

  useEffect(() => {
    // Handle unhandled promise rejections
    const handleUnhandledRejection = (event) => {
      console.error("Unhandled promise rejection:", event.reason);
      setAsyncError({
        type: "unhandledRejection",
        message: event.reason?.message || "Unhandled promise rejection",
        timestamp: new Date().toISOString(),
      });

      // Prevent the default browser behavior
      event.preventDefault();
    };

    // Handle general JavaScript errors
    const handleError = (event) => {
      console.error("JavaScript error:", event.error);
      setAsyncError({
        type: "javascriptError",
        message: event.error?.message || "JavaScript error",
        filename: event.filename,
        lineno: event.lineno,
        colno: event.colno,
        timestamp: new Date().toISOString(),
      });
    };

    window.addEventListener("unhandledrejection", handleUnhandledRejection);
    window.addEventListener("error", handleError);

    return () => {
      window.removeEventListener(
        "unhandledrejection",
        handleUnhandledRejection
```

```
      );
      window.removeEventListener("error", handleError);
    };
  }, []);

  if (asyncError) {
    return (
      <div className="async-error">
        <h3>🔒 Async Error Detected</h3>
        <p>
          <strong>Type:</strong> {asyncError.type}
        </p>
        <p>
          <strong>Message:</strong> {asyncError.message}
        </p>
        <p>
          <strong>Time:</strong> {asyncError.timestamp}
        </p>

        <button onClick={() => setAsyncError(null)}>Dismiss</button>

        <button onClick={() => window.location.reload()}>Reload Page</button>
      </div>
    );
  }

  return children;
}
```

2. Network Error Handling

```
// ✅ Network error handling hook
function useNetworkErrorHandler() {
  const [networkError, setNetworkError] = useState(null);
  const [isOnline, setIsOnline] = useState(navigator.onLine);

  useEffect(() => {
    const handleOnline = () => {
      setIsOnline(true);
      setNetworkError(null);
    };

    const handleOffline = () => {
      setIsOnline(false);
      setNetworkError({
        type: "offline",
        message:
          "You are currently offline. Please check your internet connection.",
      });
    };
```

```
    window.addEventListener("online", handleOnline);
    window.addEventListener("offline", handleOffline);

    return () => {
      window.removeEventListener("online", handleOnline);
      window.removeEventListener("offline", handleOffline);
    };
  }, []);

  const handleNetworkError = useCallback((error, context = {}) => {
    let errorMessage = "Network error occurred";
    let errorType = "network";

    if (error.name === "TypeError" && error.message.includes("fetch")) {
      errorMessage = "Failed to connect to server";
      errorType = "connection";
    } else if (error.status) {
      errorMessage = `Server error: ${error.status} ${error.statusText}`;
      errorType = "server";
    }

    setNetworkError({
      type: errorType,
      message: errorMessage,
      status: error.status,
      context,
      timestamp: new Date().toISOString(),
    });
  }, []);

  const clearNetworkError = useCallback(() => {
    setNetworkError(null);
  }, []);

  return {
    networkError,
    isOnline,
    handleNetworkError,
    clearNetworkError,
  };
}

// Network-aware fetch wrapper
function useNetworkFetch() {
  const { handleNetworkError, isOnline } = useNetworkErrorHandler();

  const networkFetch = useCallback(
    async (url, options = {}) => {
      if (!isOnline) {
        throw new Error("No internet connection");
      }

      try {
        const response = await fetch(url, {
```

```
          ...options,
          signal: AbortSignal.timeout(10000), // 10 second timeout
        });

        if (!response.ok) {
          const error = new Error(
            `HTTP ${response.status}: ${response.statusText}`
          );
          error.status = response.status;
          error.statusText = response.statusText;
          throw error;
        }

        return response;
      } catch (error) {
        handleNetworkError(error, { url, options });
        throw error;
      }
    },
    [isOnline, handleNetworkError]
  );

  return networkFetch;
}

// Usage
function DataComponent() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const networkFetch = useNetworkFetch();
  const { networkError, clearNetworkError } = useNetworkErrorHandler();

  const fetchData = async () => {
    setLoading(true);
    try {
      const response = await networkFetch("/api/data");
      const result = await response.json();
      setData(result);
    } catch (error) {
      console.error("Failed to fetch data:", error);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData();
  }, []);

  if (networkError) {
    return (
      <div className="network-error">
        <h3>🌐 Network Error</h3>
        <p>{networkError.message}</p>
```

```
        <div className="error-actions">
          <button onClick={fetchData}>Retry</button>
          <button onClick={clearNetworkError}>Dismiss</button>
        </div>
      </div>
    );
  }

  if (loading) return <div>Loading...</div>;

  return (
    <div>
      <h2>Data</h2>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}
```

🎯 Mini-Challenge: Build a Comprehensive Error System

Challenge: Create an Error Management System

Build a complete error handling system that includes:

- Error boundaries for different app sections
- Async error handling
- Error reporting and logging
- User-friendly error recovery

```
// Your task: Implement this error management system
function App() {
  return (
    <ErrorProvider>
      <div className="app">
        <ErrorBoundary name="header">
          <Header />
        </ErrorBoundary>

        <ErrorBoundary name="main" fallback={MainErrorFallback}>
          <main>
            <ErrorBoundary name="sidebar">
              <Sidebar />
            </ErrorBoundary>

            <ErrorBoundary name="content">
              <Routes>
                <Route path="/" element={<Home />} />
                <Route path="/profile" element={<Profile />} />
                <Route path="/settings" element={<Settings />} />
              </Routes>
```

```
            </ErrorBoundary>
          </main>
        </ErrorBoundary>

        <ErrorBoundary name="footer">
          <Footer />
        </ErrorBoundary>

        <ErrorNotifications />
      </div>
    </ErrorProvider>
  );
}

// Components that might throw errors
function BuggyComponent() {
  const [shouldThrow, setShouldThrow] = useState(false);

  if (shouldThrow) {
    throw new Error("Intentional error for testing!");
  }

  return (
    <div>
      <h3>Buggy Component</h3>
      <button onClick={() => setShouldThrow(true)}>Throw Error</button>
    </div>
  );
}

function AsyncBuggyComponent() {
  const handleAsyncError = async () => {
    // This should be caught by async error handlers
    await new Promise((_, reject) => {
      setTimeout(() => reject(new Error("Async error!")), 1000);
    });
  };

  return (
    <div>
      <h3>Async Buggy Component</h3>
      <button onClick={handleAsyncError}>Trigger Async Error</button>
    </div>
  );
}
```

Solution: Comprehensive Error Management System

```
// ✅ Error context for global error management
const ErrorContext = createContext();
```

```
function ErrorProvider({ children }) {
  const [errors, setErrors] = useState([]);
  const [errorStats, setErrorStats] = useState({
    total: 0,
    byType: {},
    byComponent: {},
  });

  const addError = useCallback((error, context = {}) => {
    const errorId = `error_${Date.now()}_${Math.random()
      .toString(36)
      .substr(2, 9)}`;

    const errorEntry = {
      id: errorId,
      message: error.message,
      stack: error.stack,
      timestamp: new Date().toISOString(),
      context,
      dismissed: false,
    };

    setErrors((prev) => [errorEntry, ...prev].slice(0, 50)); // Keep last 50
errors

    // Update statistics
    setErrorStats((prev) => ({
      total: prev.total + 1,
      byType: {
        ...prev.byType,
        [context.type || "unknown"]:
          (prev.byType[context.type || "unknown"] || 0) + 1,
      },
      byComponent: {
        ...prev.byComponent,
        [context.component || "unknown"]:
          (prev.byComponent[context.component || "unknown"] || 0) + 1,
      },
    }));

    // Log to external service in production
    if (process.env.NODE_ENV === "production") {
      logErrorToService(errorEntry);
    }

    return errorId;
  }, []);

  const dismissError = useCallback((errorId) => {
    setErrors((prev) =>
      prev.map((error) =>
        error.id === errorId ? { ...error, dismissed: true } : error
      )
    );
```

```
  }, []);

  const clearErrors = useCallback(() => {
    setErrors([]);
  }, []);

  const logErrorToService = useCallback((errorEntry) => {
    // Mock error logging service
    console.log("📊 Logging error to service:", errorEntry);

    // In real app, send to your error monitoring service
    // fetch('/api/errors', {
    //   method: 'POST',
    //   headers: { 'Content-Type': 'application/json' },
    //   body: JSON.stringify(errorEntry)
    // });
  }, []);

  const value = useMemo(
    () => ({
      errors: errors.filter((error) => !error.dismissed),
      allErrors: errors,
      errorStats,
      addError,
      dismissError,
      clearErrors,
    }),
    [errors, errorStats, addError, dismissError, clearErrors]
  );

  return (
    <ErrorContext.Provider value={value}>{children}</ErrorContext.Provider>
  );
}

// Custom hook to use error context
function useErrorContext() {
  const context = useContext(ErrorContext);
  if (!context) {
    throw new Error("useErrorContext must be used within ErrorProvider");
  }
  return context;
}

// Enhanced Error Boundary with context integration
class ContextualErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null };
  }

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

```
  componentDidCatch(error, errorInfo) {
    const { name, onError } = this.props;

    // Add error to global context
    if (this.context?.addError) {
      this.context.addError(error, {
        type: "boundary",
        component: name,
        componentStack: errorInfo.componentStack,
      });
    }

    // Call custom error handler
    if (onError) {
      onError(error, errorInfo);
    }
  }

  render() {
    const { hasError, error } = this.state;
    const { children, fallback: Fallback, name } = this.props;

    if (hasError) {
      if (Fallback) {
        return (
          <Fallback
            error={error}
            componentName={name}
            onRetry={() => this.setState({ hasError: false, error: null })}
          />
        );
      }

      return (
        <div className="error-boundary-fallback">
          <h3>⚠ Error in {name}</h3>
          <p>Something went wrong in this section.</p>
          <button
            onClick={() => this.setState({ hasError: false, error: null })}
          >
            Try Again
          </button>
        </div>
      );
    }

    return children;
  }
}

ContextualErrorBoundary.contextType = ErrorContext;

// Error notification component
```

```
function ErrorNotifications() {
  const { errors, dismissError } = useErrorContext();

  if (errors.length === 0) return null;

  return (
    <div className="error-notifications">
      {errors.slice(0, 3).map((error) => (
        <div key={error.id} className="error-notification">
          <div className="error-content">
            <strong>Error:</strong> {error.message}
            {error.context.component && (
              <div className="error-context">
                Component: {error.context.component}
              </div>
            )}
          </div>

          <button
            className="error-dismiss"
            onClick={() => dismissError(error.id)}
          >
            ×
          </button>
        </div>
      ))}
    </div>
  );
}

// Error statistics component
function ErrorStats() {
  const { errorStats, allErrors } = useErrorContext();

  return (
    <div className="error-stats">
      <h3>📊 Error Statistics</h3>

      <div className="stat-item">
        <strong>Total Errors:</strong> {errorStats.total}
      </div>

      <div className="stat-section">
        <h4>By Type:</h4>
        {Object.entries(errorStats.byType).map(([type, count]) => (
          <div key={type} className="stat-item">
            {type}: {count}
          </div>
        ))}
      </div>

      <div className="stat-section">
        <h4>By Component:</h4>
        {Object.entries(errorStats.byComponent).map(([component, count]) => (
```

```
            <div key={component} className="stat-item">
              {component}: {count}
            </div>
          ))}
        </div>

        <div className="stat-section">
          <h4>Recent Errors:</h4>
          {allErrors.slice(0, 5).map((error) => (
            <div key={error.id} className="recent-error">
              <div>{error.message}</div>
              <small>{new Date(error.timestamp).toLocaleString()}</small>
            </div>
          ))}
        </div>
      </div>
    );
}

// Custom fallback components
function MainErrorFallback({ error, componentName, onRetry }) {
  return (
    <div className="main-error-fallback">
      <h2>🚨 Main Content Error</h2>
      <p>The main content area encountered an error.</p>

      <div className="error-details">
        <strong>Component:</strong> {componentName}
        <br />
        <strong>Error:</strong> {error?.message}
      </div>

      <div className="error-actions">
        <button onClick={onRetry}>🔄 Retry</button>
        <button onClick={() => (window.location.href = "/")}>🏠 Go Home</button>
      </div>
    </div>
  );
}

// Async error handler component
function AsyncErrorHandler({ children }) {
  const { addError } = useErrorContext();

  useEffect(() => {
    const handleUnhandledRejection = (event) => {
      addError(new Error(event.reason), {
        type: "unhandledRejection",
        component: "global",
      });
      event.preventDefault();
    };

    const handleError = (event) => {
```

```
        addError(event.error, {
          type: "javascript",
          component: "global",
          filename: event.filename,
          lineno: event.lineno,
        });
      };

      window.addEventListener("unhandledrejection", handleUnhandledRejection);
      window.addEventListener("error", handleError);

      return () => {
        window.removeEventListener(
          "unhandledrejection",
          handleUnhandledRejection
        );
        window.removeEventListener("error", handleError);
      };
    }, [addError]);

    return children;
  }

  // Main app with comprehensive error handling
  function App() {
    return (
      <ErrorProvider>
        <AsyncErrorHandler>
          <div className="app">
            <ContextualErrorBoundary name="header">
              <Header />
            </ContextualErrorBoundary>

            <ContextualErrorBoundary name="main" fallback={MainErrorFallback}>
              <main className="main-content">
                <ContextualErrorBoundary name="sidebar">
                  <Sidebar />
                </ContextualErrorBoundary>

                <ContextualErrorBoundary name="content">
                  <div className="content">
                    <BuggyComponent />
                    <AsyncBuggyComponent />
                    <ErrorStats />
                  </div>
                </ContextualErrorBoundary>
              </main>
            </ContextualErrorBoundary>

            <ContextualErrorBoundary name="footer">
              <Footer />
            </ContextualErrorBoundary>

            <ErrorNotifications />
```

```
        </div>
      </AsyncErrorHandler>
    </ErrorProvider>
  );
}

// Test components
function Header() {
  return <header>📱 My App Header</header>;
}

function Sidebar() {
  return (
    <aside>
      <h3>Sidebar</h3>
      <nav>
        <a href="/">Home</a>
        <a href="/profile">Profile</a>
      </nav>
    </aside>
  );
}

function Footer() {
  return <footer>© 2024 My App</footer>;
}

function BuggyComponent() {
  const [shouldThrow, setShouldThrow] = useState(false);

  if (shouldThrow) {
    throw new Error("Intentional boundary error for testing!");
  }

  return (
    <div className="buggy-component">
      <h3>🐛 Boundary Error Test</h3>
      <button onClick={() => setShouldThrow(true)}>Throw Boundary Error</button>
    </div>
  );
}

function AsyncBuggyComponent() {
  const { addError } = useErrorContext();

  const handleAsyncError = async () => {
    try {
      await new Promise((_, reject) => {
        setTimeout(() => reject(new Error("Intentional async error!")), 1000);
      });
    } catch (error) {
      addError(error, {
        type: "async",
        component: "AsyncBuggyComponent",
```

```
      });
    }
  };

  const handleUnhandledAsyncError = () => {
    // This will be caught by the global handler
    setTimeout(() => {
      throw new Error("Unhandled async error!");
    }, 1000);
  };

  return (
    <div className="async-buggy-component">
      <h3>⚡ Async Error Test</h3>
      <button onClick={handleAsyncError}>Handled Async Error</button>
      <button onClick={handleUnhandledAsyncError}>Unhandled Async Error</button>
    </div>
  );
}
```

🧪 Testing Error Scenarios

Testing Error Boundaries

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";

// Mock console.error to avoid noise in tests
const originalError = console.error;
beforeAll(() => {
  console.error = jest.fn();
});

afterAll(() => {
  console.error = originalError;
});

// Test component that throws on command
function ThrowError({ shouldThrow }) {
  if (shouldThrow) {
    throw new Error("Test error");
  }
  return <div>No error</div>;
}

describe("ErrorBoundary", () => {
  it("should catch and display error", () => {
    render(
      <ErrorBoundary>
        <ThrowError shouldThrow={true} />
      </ErrorBoundary>
```

```
    );

    expect(screen.getByText(/something went wrong/i)).toBeInTheDocument();
  });

  it("should render children when no error", () => {
    render(
      <ErrorBoundary>
        <ThrowError shouldThrow={false} />
      </ErrorBoundary>
    );

    expect(screen.getByText("No error")).toBeInTheDocument();
  });

  it("should allow retry after error", async () => {
    const user = userEvent.setup();

    function TestComponent() {
      const [shouldThrow, setShouldThrow] = useState(true);

      return (
        <ErrorBoundary>
          <button onClick={() => setShouldThrow(false)}>Fix Error</button>
          <ThrowError shouldThrow={shouldThrow} />
        </ErrorBoundary>
      );
    }

    render(<TestComponent />);

    // Error should be displayed
    expect(screen.getByText(/something went wrong/i)).toBeInTheDocument();

    // Click retry
    await user.click(screen.getByText("Try Again"));

    // Should show the component again
    expect(screen.getByText("No error")).toBeInTheDocument();
  });
});
```

🎯 Best Practices

1. Error Boundary Placement

```
// ✅ Strategic error boundary placement
function App() {
  return (
    <ErrorBoundary name="app">
      {" "}
```

```
      {/* Top-level boundary */}
      <Header />
      <ErrorBoundary name="main">
        {" "}
        {/* Section-level boundary */}
        <Router>
          <Routes>
            <Route
              path="/"
              element={
                <ErrorBoundary name="home">
                  {" "}
                  {/* Page-level boundary */}
                  <Home />
                </ErrorBoundary>
              }
            />

            <Route
              path="/profile"
              element={
                <ErrorBoundary name="profile">
                  <Profile />
                </ErrorBoundary>
              }
            />
          </Routes>
        </Router>
      </ErrorBoundary>
      <Footer />
    </ErrorBoundary>
  );
}
```

2. Error Logging and Monitoring

```
// ✅ Production error logging
function logErrorToService(error, errorInfo, context = {}) {
  const errorData = {
    message: error.message,
    stack: error.stack,
    componentStack: errorInfo?.componentStack,
    timestamp: new Date().toISOString(),
    url: window.location.href,
    userAgent: navigator.userAgent,
    userId: getCurrentUserId(), // If available
    sessionId: getSessionId(), // If available
    buildVersion: process.env.REACT_APP_VERSION,
    ...context,
  };
```

```
  // Send to error monitoring service (Sentry, LogRocket, etc.)
  if (window.Sentry) {
    window.Sentry.captureException(error, {
      extra: errorData,
    });
  }

  // Send to custom analytics
  if (window.analytics) {
    window.analytics.track("Error Occurred", errorData);
  }

  // Send to custom error endpoint
  fetch("/api/errors", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(errorData),
  }).catch((err) => {
    console.error("Failed to log error:", err);
  });
}
```

3. User-Friendly Error Messages

```
// ✅ Context-aware error messages
function getErrorMessage(error, context) {
  const errorMessages = {
    ChunkLoadError:
      "The application has been updated. Please refresh the page.",
    NetworkError: "Please check your internet connection and try again.",
    AuthenticationError: "Your session has expired. Please log in again.",
    ValidationError: "Please check your input and try again.",
    PermissionError: "You don't have permission to perform this action.",
    default: "Something unexpected happened. Please try again.",
  };

  // Determine error type
  if (error.name === "ChunkLoadError") return errorMessages.ChunkLoadError;
  if (error.message.includes("fetch")) return errorMessages.NetworkError;
  if (error.status === 401) return errorMessages.AuthenticationError;
  if (error.status === 403) return errorMessages.PermissionError;
  if (context.type === "validation") return errorMessages.ValidationError;

  return errorMessages.default;
}
```

Next up: We'll explore testing strategies and learn how to write comprehensive tests for React applications! 🔦

17. Testing React Applications 🧪

> "Testing is not about finding bugs, it's about building confidence in your code." - Kent C. Dodds

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- Different types of testing in React (Unit, Integration, E2E)
- Setting up and using React Testing Library
- Testing components, hooks, and user interactions
- Mocking dependencies and external services
- Testing async operations and error scenarios
- Best practices for maintainable tests
- Performance testing and accessibility testing

🔍 Types of Testing

Testing Pyramid

```
        /\     E2E Tests
       /  \    (Few, Slow, Expensive)
      /____\
     /      \  Integration Tests
    /        \ (Some, Medium Speed)
   /__________\
  /            \ Unit Tests
 /              \ (Many, Fast, Cheap)
/________________\
```

1. Unit Tests

- Test individual components in isolation
- Fast and focused
- Easy to debug
- High code coverage

2. Integration Tests

- Test component interactions
- Test with real dependencies
- More realistic scenarios
- Catch integration issues

3. End-to-End (E2E) Tests

- Test complete user workflows
- Test in real browser environment

- Catch system-level issues
- Slower but comprehensive

🛠️ Setting Up Testing Environment

React Testing Library Setup

```
# Install testing dependencies
npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event

# For custom render utilities
npm install --save-dev @testing-library/react-hooks
```

Test Setup File

```
// src/setupTests.js
import "@testing-library/jest-dom";

// Mock IntersectionObserver
global.IntersectionObserver = class IntersectionObserver {
  constructor() {}
  observe() {
    return null;
  }
  disconnect() {
    return null;
  }
  unobserve() {
    return null;
  }
};

// Mock ResizeObserver
global.ResizeObserver = class ResizeObserver {
  constructor() {}
  observe() {
    return null;
  }
  disconnect() {
    return null;
  }
  unobserve() {
    return null;
  }
};

// Mock matchMedia
Object.defineProperty(window, "matchMedia", {
  writable: true,
```

```
    value: jest.fn().mockImplementation((query) => ({
      matches: false,
      media: query,
      onchange: null,
      addListener: jest.fn(),
      removeListener: jest.fn(),
      addEventListener: jest.fn(),
      removeEventListener: jest.fn(),
      dispatchEvent: jest.fn(),
    })),
  });

  // Mock localStorage
  const localStorageMock = {
    getItem: jest.fn(),
    setItem: jest.fn(),
    removeItem: jest.fn(),
    clear: jest.fn(),
  };
  global.localStorage = localStorageMock;
```

Custom Render Utility

```
// src/test-utils.js
import React from "react";
import { render } from "@testing-library/react";
import { BrowserRouter } from "react-router-dom";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import { ThemeProvider } from "./contexts/ThemeContext";
import { AuthProvider } from "./contexts/AuthContext";

// Create a custom render function that includes providers
function customRender(ui, options = {}) {
  const {
    initialEntries = ["/"],
    queryClient = new QueryClient({
      defaultOptions: {
        queries: { retry: false },
        mutations: { retry: false },
      },
    }),
    ...renderOptions
  } = options;

  function Wrapper({ children }) {
    return (
      <QueryClientProvider client={queryClient}>
        <BrowserRouter>
          <ThemeProvider>
            <AuthProvider>{children}</AuthProvider>
          </ThemeProvider>
```

```
        </BrowserRouter>
      </QueryClientProvider>
    );
  }

  return {
    ...render(ui, { wrapper: Wrapper, ...renderOptions }),
    queryClient,
  };
}

// Re-export everything
export * from "@testing-library/react";
export { customRender as render };
export { default as userEvent } from "@testing-library/user-event";
```

Testing Components

Basic Component Testing

```
// Button.jsx
function Button({ children, onClick, disabled = false, variant = "primary" }) {
  return (
    <button
      onClick={onClick}
      disabled={disabled}
      className={`btn btn-${variant}`}
      data-testid="button"
    >
      {children}
    </button>
  );
}

export default Button;
```

```
// Button.test.jsx
import { render, screen } from "../test-utils";
import userEvent from "@testing-library/user-event";
import Button from "./Button";

describe("Button Component", () => {
  it("renders with correct text", () => {
    render(<Button>Click me</Button>);

    expect(
      screen.getByRole("button", { name: "Click me" })
    ).toBeInTheDocument();
  });
```

```
  it("calls onClick when clicked", async () => {
    const user = userEvent.setup();
    const handleClick = jest.fn();

    render(<Button onClick={handleClick}>Click me</Button>);

    await user.click(screen.getByRole("button"));

    expect(handleClick).toHaveBeenCalledTimes(1);
  });

  it("is disabled when disabled prop is true", () => {
    render(<Button disabled>Click me</Button>);

    expect(screen.getByRole("button")).toBeDisabled();
  });

  it("applies correct CSS class for variant", () => {
    render(<Button variant="secondary">Click me</Button>);

    expect(screen.getByRole("button")).toHaveClass("btn-secondary");
  });

  it("does not call onClick when disabled", async () => {
    const user = userEvent.setup();
    const handleClick = jest.fn();

    render(
      <Button onClick={handleClick} disabled>
        Click me
      </Button>
    );

    await user.click(screen.getByRole("button"));

    expect(handleClick).not.toHaveBeenCalled();
  });
});
```

Testing Forms and User Input

```
// LoginForm.jsx
import { useState } from "react";

function LoginForm({ onSubmit, isLoading = false }) {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [errors, setErrors] = useState({});

  const validateForm = () => {
```

```
    const newErrors = {};

    if (!email) {
      newErrors.email = "Email is required";
    } else if (!/\S+@\S+\.\S+/.test(email)) {
      newErrors.email = "Email is invalid";
    }

    if (!password) {
      newErrors.password = "Password is required";
    } else if (password.length < 6) {
      newErrors.password = "Password must be at least 6 characters";
    }

    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  const handleSubmit = (e) => {
    e.preventDefault();

    if (validateForm()) {
      onSubmit({ email, password });
    }
  };

  return (
    <form onSubmit={handleSubmit} noValidate>
      <div className="form-group">
        <label htmlFor="email">Email</label>
        <input
          id="email"
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          aria-invalid={errors.email ? "true" : "false"}
          aria-describedby={errors.email ? "email-error" : undefined}
        />
        {errors.email && (
          <div id="email-error" role="alert" className="error">
            {errors.email}
          </div>
        )}
      </div>

      <div className="form-group">
        <label htmlFor="password">Password</label>
        <input
          id="password"
          type="password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
          aria-invalid={errors.password ? "true" : "false"}
          aria-describedby={errors.password ? "password-error" : undefined}
```

```
        />
        {errors.password && (
          <div id="password-error" role="alert" className="error">
            {errors.password}
          </div>
        )}
      </div>

      <button type="submit" disabled={isLoading}>
        {isLoading ? "Logging in..." : "Login"}
      </button>
    </form>
  );
}

export default LoginForm;
```

```
// LoginForm.test.jsx
import { render, screen, waitFor } from "../test-utils";
import userEvent from "@testing-library/user-event";
import LoginForm from "./LoginForm";

describe("LoginForm", () => {
  const mockOnSubmit = jest.fn();

  beforeEach(() => {
    mockOnSubmit.mockClear();
  });

  it("renders form fields", () => {
    render(<LoginForm onSubmit={mockOnSubmit} />);

    expect(screen.getByLabelText(/email/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/password/i)).toBeInTheDocument();
    expect(screen.getByRole("button", { name: /login/i })).toBeInTheDocument();
  });

  it("submits form with valid data", async () => {
    const user = userEvent.setup();

    render(<LoginForm onSubmit={mockOnSubmit} />);

    await user.type(screen.getByLabelText(/email/i), "test@example.com");
    await user.type(screen.getByLabelText(/password/i), "password123");
    await user.click(screen.getByRole("button", { name: /login/i }));

    expect(mockOnSubmit).toHaveBeenCalledWith({
      email: "test@example.com",
      password: "password123",
    });
  });
```

```
  it("shows validation errors for empty fields", async () => {
    const user = userEvent.setup();

    render(<LoginForm onSubmit={mockOnSubmit} />);

    await user.click(screen.getByRole("button", { name: /login/i }));

    expect(
      screen.getByRole("alert", { name: /email is required/i })
    ).toBeInTheDocument();
    expect(
      screen.getByRole("alert", { name: /password is required/i })
    ).toBeInTheDocument();
    expect(mockOnSubmit).not.toHaveBeenCalled();
  });

  it("shows validation error for invalid email", async () => {
    const user = userEvent.setup();

    render(<LoginForm onSubmit={mockOnSubmit} />);

    await user.type(screen.getByLabelText(/email/i), "invalid-email");
    await user.type(screen.getByLabelText(/password/i), "password123");
    await user.click(screen.getByRole("button", { name: /login/i }));

    expect(
      screen.getByRole("alert", { name: /email is invalid/i })
    ).toBeInTheDocument();
    expect(mockOnSubmit).not.toHaveBeenCalled();
  });

  it("shows validation error for short password", async () => {
    const user = userEvent.setup();

    render(<LoginForm onSubmit={mockOnSubmit} />);

    await user.type(screen.getByLabelText(/email/i), "test@example.com");
    await user.type(screen.getByLabelText(/password/i), "123");
    await user.click(screen.getByRole("button", { name: /login/i }));

    expect(
      screen.getByRole("alert", {
        name: /password must be at least 6 characters/i,
      })
    ).toBeInTheDocument();
    expect(mockOnSubmit).not.toHaveBeenCalled();
  });

  it("disables submit button when loading", () => {
    render(<LoginForm onSubmit={mockOnSubmit} isLoading={true} />);

    expect(screen.getByRole("button", { name: /logging in/i })).toBeDisabled();
  });
```

```
  it("clears errors when user starts typing", async () => {
    const user = userEvent.setup();

    render(<LoginForm onSubmit={mockOnSubmit} />);

    // Trigger validation errors
    await user.click(screen.getByRole("button", { name: /login/i }));
    expect(
      screen.getByRole("alert", { name: /email is required/i })
    ).toBeInTheDocument();

    // Start typing to clear error
    await user.type(screen.getByLabelText(/email/i), "test@example.com");

    await waitFor(() => {
      expect(
        screen.queryByRole("alert", { name: /email is required/i })
      ).not.toBeInTheDocument();
    });
  });
});
```

□ Testing Custom Hooks

Testing Hooks with renderHook

```
// useCounter.js
import { useState, useCallback } from "react";

function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  const decrement = useCallback(() => {
    setCount((prev) => prev - 1);
  }, []);

  const reset = useCallback(() => {
    setCount(initialValue);
  }, [initialValue]);

  const setValue = useCallback((value) => {
    setCount(value);
  }, []);

  return {
    count,
```

```
    increment,
    decrement,
    reset,
    setValue,
  };
}

export default useCounter;
```

```
// useCounter.test.js
import { renderHook, act } from "@testing-library/react";
import useCounter from "./useCounter";

describe("useCounter", () => {
  it("initializes with default value", () => {
    const { result } = renderHook(() => useCounter());

    expect(result.current.count).toBe(0);
  });

  it("initializes with custom value", () => {
    const { result } = renderHook(() => useCounter(10));

    expect(result.current.count).toBe(10);
  });

  it("increments count", () => {
    const { result } = renderHook(() => useCounter());

    act(() => {
      result.current.increment();
    });

    expect(result.current.count).toBe(1);
  });

  it("decrements count", () => {
    const { result } = renderHook(() => useCounter(5));

    act(() => {
      result.current.decrement();
    });

    expect(result.current.count).toBe(4);
  });

  it("resets to initial value", () => {
    const { result } = renderHook(() => useCounter(10));

    act(() => {
      result.current.increment();
```

```
      result.current.increment();
    });

    expect(result.current.count).toBe(12);

    act(() => {
      result.current.reset();
    });

    expect(result.current.count).toBe(10);
  });

  it("sets specific value", () => {
    const { result } = renderHook(() => useCounter());

    act(() => {
      result.current.setValue(42);
    });

    expect(result.current.count).toBe(42);
  });

  it("updates reset when initial value changes", () => {
    let initialValue = 0;
    const { result, rerender } = renderHook(() => useCounter(initialValue));

    act(() => {
      result.current.increment();
    });

    expect(result.current.count).toBe(1);

    // Change initial value and rerender
    initialValue = 10;
    rerender();

    act(() => {
      result.current.reset();
    });

    expect(result.current.count).toBe(10);
  });
});
```

Testing Async Hooks

```
// useApi.js
import { useState, useEffect, useCallback } from "react";

function useApi(url, options = {}) {
  const [data, setData] = useState(null);
```

```
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const fetchData = useCallback(async () => {
    setLoading(true);
    setError(null);

    try {
      const response = await fetch(url, options);

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
      }

      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }, [url, options]);

  useEffect(() => {
    if (url) {
      fetchData();
    }
  }, [fetchData, url]);

  const refetch = useCallback(() => {
    fetchData();
  }, [fetchData]);

  return {
    data,
    loading,
    error,
    refetch,
  };
}

export default useApi;
```

```
// useApi.test.js
import { renderHook, waitFor } from "@testing-library/react";
import useApi from "./useApi";

// Mock fetch
global.fetch = jest.fn();

describe("useApi", () => {
```

```
  beforeEach(() => {
    fetch.mockClear();
  });

  it("fetches data successfully", async () => {
    const mockData = { id: 1, name: "Test" };

    fetch.mockResolvedValueOnce({
      ok: true,
      json: async () => mockData,
    });

    const { result } = renderHook(() => useApi("/api/test"));

    expect(result.current.loading).toBe(true);
    expect(result.current.data).toBe(null);
    expect(result.current.error).toBe(null);

    await waitFor(() => {
      expect(result.current.loading).toBe(false);
    });

    expect(result.current.data).toEqual(mockData);
    expect(result.current.error).toBe(null);
    expect(fetch).toHaveBeenCalledWith("/api/test", {});
  });

  it("handles fetch error", async () => {
    fetch.mockRejectedValueOnce(new Error("Network error"));

    const { result } = renderHook(() => useApi("/api/test"));

    await waitFor(() => {
      expect(result.current.loading).toBe(false);
    });

    expect(result.current.data).toBe(null);
    expect(result.current.error).toBe("Network error");
  });

  it("handles HTTP error status", async () => {
    fetch.mockResolvedValueOnce({
      ok: false,
      status: 404,
      statusText: "Not Found",
    });

    const { result } = renderHook(() => useApi("/api/test"));

    await waitFor(() => {
      expect(result.current.loading).toBe(false);
    });

    expect(result.current.data).toBe(null);
```

```
    expect(result.current.error).toBe("HTTP 404: Not Found");
  });

  it("refetches data when refetch is called", async () => {
    const mockData = { id: 1, name: "Test" };

    fetch.mockResolvedValue({
      ok: true,
      json: async () => mockData,
    });

    const { result } = renderHook(() => useApi("/api/test"));

    await waitFor(() => {
      expect(result.current.loading).toBe(false);
    });

    expect(fetch).toHaveBeenCalledTimes(1);

    // Call refetch
    result.current.refetch();

    await waitFor(() => {
      expect(fetch).toHaveBeenCalledTimes(2);
    });
  });

  it("does not fetch when url is null", () => {
    renderHook(() => useApi(null));

    expect(fetch).not.toHaveBeenCalled();
  });
});
```

🎭 Mocking Dependencies

Mocking External Libraries

```
// Mock axios
jest.mock('axios');
const mockedAxios = axios as jest.Mocked<typeof axios>;

// Mock React Router
jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useNavigate: () => jest.fn(),
  useParams: () => ({ id: '123' }),
}));

// Mock date-fns
jest.mock('date-fns', () => ({
```

```
  format: jest.fn(() => '2024-01-01'),
  isValid: jest.fn(() => true),
}));
```

Mocking Custom Hooks

```
// Mock custom hook
jest.mock('../hooks/useAuth');
const mockUseAuth = useAuth as jest.MockedFunction<typeof useAuth>;

describe('ProtectedComponent', () => {
  it('renders when user is authenticated', () => {
    mockUseAuth.mockReturnValue({
      user: { id: 1, name: 'John' },
      isAuthenticated: true,
      login: jest.fn(),
      logout: jest.fn(),
    });

    render(<ProtectedComponent />);

    expect(screen.getByText('Welcome, John!')).toBeInTheDocument();
  });

  it('shows login prompt when user is not authenticated', () => {
    mockUseAuth.mockReturnValue({
      user: null,
      isAuthenticated: false,
      login: jest.fn(),
      logout: jest.fn(),
    });

    render(<ProtectedComponent />);

    expect(screen.getByText('Please log in')).toBeInTheDocument();
  });
});
```

Mocking API Calls with MSW

```
// src/mocks/handlers.js
import { rest } from "msw";

export const handlers = [
  // Mock GET request
  rest.get("/api/users", (req, res, ctx) => {
    return res(
      ctx.status(200),
      ctx.json([
```

```
        { id: 1, name: "John Doe", email: "john@example.com" },
        { id: 2, name: "Jane Smith", email: "jane@example.com" },
      ])
    );
  }),

  // Mock POST request
  rest.post("/api/users", (req, res, ctx) => {
    return res(ctx.status(201), ctx.json({ id: 3, ...req.body }));
  }),

  // Mock error response
  rest.get("/api/error", (req, res, ctx) => {
    return res(ctx.status(500), ctx.json({ message: "Internal server error" }));
  }),
];
```

```
// src/mocks/server.js
import { setupServer } from "msw/node";
import { handlers } from "./handlers";

export const server = setupServer(...handlers);
```

```
// src/setupTests.js
import { server } from "./mocks/server";

// Enable API mocking before tests
beforeAll(() => server.listen());

// Reset any request handlers that are declared as a part of our tests
afterEach(() => server.resetHandlers());

// Clean up after the tests are finished
afterAll(() => server.close());
```

Testing Async Operations

Testing Components with Async Data

```
// UserList.jsx
import { useState, useEffect } from "react";

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

```
  useEffect(() => {
    const fetchUsers = async () => {
      try {
        const response = await fetch("/api/users");
        if (!response.ok) {
          throw new Error("Failed to fetch users");
        }
        const data = await response.json();
        setUsers(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchUsers();
  }, []);

  if (loading) return <div>Loading users...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <h2>Users</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>
            {user.name} - {user.email}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default UserList;
```

```
// UserList.test.jsx
import { render, screen, waitFor } from "../test-utils";
import { server } from "../mocks/server";
import { rest } from "msw";
import UserList from "./UserList";

describe("UserList", () => {
  it("displays loading state initially", () => {
    render(<UserList />);

    expect(screen.getByText("Loading users...")).toBeInTheDocument();
  });
```

```
  it("displays users after successful fetch", async () => {
    render(<UserList />);

    await waitFor(() => {
      expect(screen.getByText("Users")).toBeInTheDocument();
    });

    expect(screen.getByText("John Doe - john@example.com")).toBeInTheDocument();
    expect(
      screen.getByText("Jane Smith - jane@example.com")
    ).toBeInTheDocument();
  });

  it("displays error message when fetch fails", async () => {
    // Override the default handler for this test
    server.use(
      rest.get("/api/users", (req, res, ctx) => {
        return res(ctx.status(500), ctx.json({ message: "Server error" }));
      })
    );

    render(<UserList />);

    await waitFor(() => {
      expect(
        screen.getByText("Error: Failed to fetch users")
      ).toBeInTheDocument();
    });
  });

  it("does not show loading state after data is loaded", async () => {
    render(<UserList />);

    await waitFor(() => {
      expect(screen.queryByText("Loading users...")).not.toBeInTheDocument();
    });
  });
});
```

Testing with React Query

```
// UserListWithQuery.jsx
import { useQuery } from "@tanstack/react-query";

function UserListWithQuery() {
  const {
    data: users,
    isLoading,
    error,
    refetch,
  } = useQuery({
```

```
    queryKey: ["users"],
    queryFn: async () => {
      const response = await fetch("/api/users");
      if (!response.ok) {
        throw new Error("Failed to fetch users");
      }
      return response.json();
    },
  });

  if (isLoading) return <div>Loading users...</div>;
  if (error) {
    return (
      <div>
        <div>Error: {error.message}</div>
        <button onClick={() => refetch()}>Retry</button>
      </div>
    );
  }

  return (
    <div>
      <h2>Users</h2>
      <button onClick={() => refetch()}>Refresh</button>
      <ul>
        {users?.map((user) => (
          <li key={user.id}>
            {user.name} - {user.email}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default UserListWithQuery;
```

```
// UserListWithQuery.test.jsx
import { render, screen, waitFor } from "../test-utils";
import { server } from "../mocks/server";
import { rest } from "msw";
import userEvent from "@testing-library/user-event";
import UserListWithQuery from "./UserListWithQuery";

describe("UserListWithQuery", () => {
  it("displays users and allows refresh", async () => {
    const user = userEvent.setup();

    render(<UserListWithQuery />);

    // Wait for initial load
```

```
    await waitFor(() => {
      expect(screen.getByText("Users")).toBeInTheDocument();
    });

    expect(screen.getByText("John Doe - john@example.com")).toBeInTheDocument();

    // Test refresh functionality
    await user.click(screen.getByText("Refresh"));

    // Should still show users after refresh
    await waitFor(() => {
      expect(
        screen.getByText("John Doe - john@example.com")
      ).toBeInTheDocument();
    });
  });

  it("shows retry button on error", async () => {
    const user = userEvent.setup();

    // Mock error response
    server.use(
      rest.get("/api/users", (req, res, ctx) => {
        return res(ctx.status(500), ctx.json({ message: "Server error" }));
      })
    );

    render(<UserListWithQuery />);

    await waitFor(() => {
      expect(screen.getByText(/Error:/)).toBeInTheDocument();
    });

    expect(screen.getByText("Retry")).toBeInTheDocument();

    // Reset to successful response
    server.use(
      rest.get("/api/users", (req, res, ctx) => {
        return res(
          ctx.status(200),
          ctx.json([{ id: 1, name: "John Doe", email: "john@example.com" }])
        );
      })
    );

    // Click retry
    await user.click(screen.getByText("Retry"));

    await waitFor(() => {
      expect(
        screen.getByText("John Doe - john@example.com")
      ).toBeInTheDocument();
    });
```

```
  });
});
```

🎯 Mini-Challenge: Build a Comprehensive Test Suite

Challenge: Test a Todo Application

Create comprehensive tests for a todo application with the following features:

- Add new todos
- Mark todos as complete
- Delete todos
- Filter todos (all, active, completed)
- Persist todos to localStorage

```
// TodoApp.jsx - Your task: Write comprehensive tests for this component
import { useState, useEffect } from "react";

function TodoApp() {
  const [todos, setTodos] = useState([]);
  const [filter, setFilter] = useState("all");
  const [newTodo, setNewTodo] = useState("");

  // Load todos from localStorage on mount
  useEffect(() => {
    const savedTodos = localStorage.getItem("todos");
    if (savedTodos) {
      setTodos(JSON.parse(savedTodos));
    }
  }, []);

  // Save todos to localStorage whenever todos change
  useEffect(() => {
    localStorage.setItem("todos", JSON.stringify(todos));
  }, [todos]);

  const addTodo = () => {
    if (newTodo.trim()) {
      const todo = {
        id: Date.now(),
        text: newTodo.trim(),
        completed: false,
        createdAt: new Date().toISOString(),
      };
      setTodos((prev) => [...prev, todo]);
      setNewTodo("");
    }
  };

  const toggleTodo = (id) => {
    setTodos((prev) =>
```

```
      prev.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  const deleteTodo = (id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
  };

  const clearCompleted = () => {
    setTodos((prev) => prev.filter((todo) => !todo.completed));
  };

  const filteredTodos = todos.filter((todo) => {
    if (filter === "active") return !todo.completed;
    if (filter === "completed") return todo.completed;
    return true;
  });

  const activeCount = todos.filter((todo) => !todo.completed).length;
  const completedCount = todos.filter((todo) => todo.completed).length;

  return (
    <div className="todo-app">
      <h1>Todo App</h1>

      <div className="add-todo">
        <input
          type="text"
          value={newTodo}
          onChange={(e) => setNewTodo(e.target.value)}
          onKeyPress={(e) => e.key === "Enter" && addTodo()}
          placeholder="What needs to be done?"
          aria-label="New todo"
        />
        <button onClick={addTodo}>Add</button>
      </div>

      <div className="filters">
        <button
          className={filter === "all" ? "active" : ""}
          onClick={() => setFilter("all")}
        >
          All ({todos.length})
        </button>
        <button
          className={filter === "active" ? "active" : ""}
          onClick={() => setFilter("active")}
        >
          Active ({activeCount})
        </button>
        <button
          className={filter === "completed" ? "active" : ""}
```

```
            onClick={() => setFilter("completed")}
          >
            Completed ({completedCount})
          </button>
        </div>

        <ul className="todo-list">
          {filteredTodos.map((todo) => (
            <li key={todo.id} className={todo.completed ? "completed" : ""}>
              <input
                type="checkbox"
                checked={todo.completed}
                onChange={() => toggleTodo(todo.id)}
                aria-label={`Mark "${todo.text}" as ${
                  todo.completed ? "incomplete" : "complete"
                }`}
              />
              <span className="todo-text">{todo.text}</span>
              <button
                onClick={() => deleteTodo(todo.id)}
                aria-label={`Delete "${todo.text}"`}
              >
                Delete
              </button>
            </li>
          ))}
        </ul>

        {completedCount > 0 && (
          <button onClick={clearCompleted} className="clear-completed">
            Clear Completed ({completedCount})
          </button>
        )}

        {todos.length === 0 && (
          <p className="empty-state">No todos yet. Add one above!</p>
        )}
      </div>
    );
}

export default TodoApp;
```

Solution: Comprehensive Todo App Tests

```
// TodoApp.test.jsx
import { render, screen, waitFor } from "../test-utils";
import userEvent from "@testing-library/user-event";
import TodoApp from "./TodoApp";

// Mock localStorage
```

```
const localStorageMock = {
  getItem: jest.fn(),
  setItem: jest.fn(),
  removeItem: jest.fn(),
  clear: jest.fn(),
};
global.localStorage = localStorageMock;

describe("TodoApp", () => {
  beforeEach(() => {
    localStorageMock.getItem.mockClear();
    localStorageMock.setItem.mockClear();
    localStorageMock.removeItem.mockClear();
    localStorageMock.clear.mockClear();
  });

  describe("Initial Render", () => {
    it("renders the todo app with empty state", () => {
      localStorageMock.getItem.mockReturnValue(null);

      render(<TodoApp />);

      expect(screen.getByText("Todo App")).toBeInTheDocument();
      expect(
        screen.getByPlaceholderText("What needs to be done?")
      ).toBeInTheDocument();
      expect(
        screen.getByText("No todos yet. Add one above!")
      ).toBeInTheDocument();
      expect(screen.getByText("All (0)")).toBeInTheDocument();
      expect(screen.getByText("Active (0)")).toBeInTheDocument();
      expect(screen.getByText("Completed (0)")).toBeInTheDocument();
    });

    it("loads todos from localStorage on mount", () => {
      const savedTodos = [
        { id: 1, text: "Test todo", completed: false, createdAt: "2024-01-01" },
        {
          id: 2,
          text: "Completed todo",
          completed: true,
          createdAt: "2024-01-02",
        },
      ];

      localStorageMock.getItem.mockReturnValue(JSON.stringify(savedTodos));

      render(<TodoApp />);

      expect(screen.getByText("Test todo")).toBeInTheDocument();
      expect(screen.getByText("Completed todo")).toBeInTheDocument();
      expect(screen.getByText("All (2)")).toBeInTheDocument();
      expect(screen.getByText("Active (1)")).toBeInTheDocument();
      expect(screen.getByText("Completed (1)")).toBeInTheDocument();
```

```
  });
});

describe("Adding Todos", () => {
  it("adds a new todo when clicking Add button", async () => {
    const user = userEvent.setup();
    localStorageMock.getItem.mockReturnValue(null);

    render(<TodoApp />);

    const input = screen.getByPlaceholderText("What needs to be done?");
    const addButton = screen.getByText("Add");

    await user.type(input, "New todo item");
    await user.click(addButton);

    expect(screen.getByText("New todo item")).toBeInTheDocument();
    expect(screen.getByText("All (1)")).toBeInTheDocument();
    expect(screen.getByText("Active (1)")).toBeInTheDocument();
    expect(input).toHaveValue("");
  });

  it("adds a new todo when pressing Enter", async () => {
    const user = userEvent.setup();
    localStorageMock.getItem.mockReturnValue(null);

    render(<TodoApp />);

    const input = screen.getByPlaceholderText("What needs to be done?");

    await user.type(input, "New todo item{enter}");

    expect(screen.getByText("New todo item")).toBeInTheDocument();
    expect(input).toHaveValue("");
  });

  it("does not add empty todos", async () => {
    const user = userEvent.setup();
    localStorageMock.getItem.mockReturnValue(null);

    render(<TodoApp />);

    const addButton = screen.getByText("Add");

    await user.click(addButton);

    expect(
      screen.getByText("No todos yet. Add one above!")
    ).toBeInTheDocument();
    expect(screen.getByText("All (0)")).toBeInTheDocument();
  });

  it("trims whitespace from todo text", async () => {
    const user = userEvent.setup();
```

```
      localStorageMock.getItem.mockReturnValue(null);

      render(<TodoApp />);

      const input = screen.getByPlaceholderText("What needs to be done?");

      await user.type(input, "  Trimmed todo  ");
      await user.click(screen.getByText("Add"));

      expect(screen.getByText("Trimmed todo")).toBeInTheDocument();
    });
  });

  describe("Todo Interactions", () => {
    beforeEach(() => {
      const savedTodos = [
        {
          id: 1,
          text: "Active todo",
          completed: false,
          createdAt: "2024-01-01",
        },
        {
          id: 2,
          text: "Completed todo",
          completed: true,
          createdAt: "2024-01-02",
        },
      ];
      localStorageMock.getItem.mockReturnValue(JSON.stringify(savedTodos));
    });

    it("toggles todo completion status", async () => {
      const user = userEvent.setup();

      render(<TodoApp />);

      const activeCheckbox = screen.getByLabelText(
        'Mark "Active todo" as complete'
      );
      const completedCheckbox = screen.getByLabelText(
        'Mark "Completed todo" as incomplete'
      );

      expect(activeCheckbox).not.toBeChecked();
      expect(completedCheckbox).toBeChecked();

      await user.click(activeCheckbox);

      expect(activeCheckbox).toBeChecked();
      expect(screen.getByText("Active (0)")).toBeInTheDocument();
      expect(screen.getByText("Completed (2)")).toBeInTheDocument();

      await user.click(completedCheckbox);
```

```
    expect(completedCheckbox).not.toBeChecked();
    expect(screen.getByText("Active (1)")).toBeInTheDocument();
    expect(screen.getByText("Completed (1)")).toBeInTheDocument();
  });

  it("deletes todos", async () => {
    const user = userEvent.setup();

    render(<TodoApp />);

    const deleteButton = screen.getByLabelText('Delete "Active todo"');

    await user.click(deleteButton);

    expect(screen.queryByText("Active todo")).not.toBeInTheDocument();
    expect(screen.getByText("Completed todo")).toBeInTheDocument();
    expect(screen.getByText("All (1)")).toBeInTheDocument();
    expect(screen.getByText("Active (0)")).toBeInTheDocument();
    expect(screen.getByText("Completed (1)")).toBeInTheDocument();
  });

  it("clears completed todos", async () => {
    const user = userEvent.setup();

    render(<TodoApp />);

    const clearButton = screen.getByText("Clear Completed (1)");

    await user.click(clearButton);

    expect(screen.queryByText("Completed todo")).not.toBeInTheDocument();
    expect(screen.getByText("Active todo")).toBeInTheDocument();
    expect(screen.getByText("All (1)")).toBeInTheDocument();
    expect(screen.getByText("Active (1)")).toBeInTheDocument();
    expect(screen.getByText("Completed (0)")).toBeInTheDocument();
    expect(screen.queryByText("Clear Completed")).not.toBeInTheDocument();
  });
});

describe("Filtering", () => {
  beforeEach(() => {
    const savedTodos = [
      {
        id: 1,
        text: "Active todo 1",
        completed: false,
        createdAt: "2024-01-01",
      },
      {
        id: 2,
        text: "Active todo 2",
        completed: false,
        createdAt: "2024-01-02",
```

```
      },
      {
        id: 3,
        text: "Completed todo 1",
        completed: true,
        createdAt: "2024-01-03",
      },
      {
        id: 4,
        text: "Completed todo 2",
        completed: true,
        createdAt: "2024-01-04",
      },
    ];
    localStorageMock.getItem.mockReturnValue(JSON.stringify(savedTodos));
  });

  it("shows all todos by default", () => {
    render(<TodoApp />);

    expect(screen.getByText("Active todo 1")).toBeInTheDocument();
    expect(screen.getByText("Active todo 2")).toBeInTheDocument();
    expect(screen.getByText("Completed todo 1")).toBeInTheDocument();
    expect(screen.getByText("Completed todo 2")).toBeInTheDocument();

    const allButton = screen.getByText("All (4)");
    expect(allButton).toHaveClass("active");
  });

  it("filters to show only active todos", async () => {
    const user = userEvent.setup();

    render(<TodoApp />);

    await user.click(screen.getByText("Active (2)"));

    expect(screen.getByText("Active todo 1")).toBeInTheDocument();
    expect(screen.getByText("Active todo 2")).toBeInTheDocument();
    expect(screen.queryByText("Completed todo 1")).not.toBeInTheDocument();
    expect(screen.queryByText("Completed todo 2")).not.toBeInTheDocument();

    const activeButton = screen.getByText("Active (2)");
    expect(activeButton).toHaveClass("active");
  });

  it("filters to show only completed todos", async () => {
    const user = userEvent.setup();

    render(<TodoApp />);

    await user.click(screen.getByText("Completed (2)"));

    expect(screen.queryByText("Active todo 1")).not.toBeInTheDocument();
    expect(screen.queryByText("Active todo 2")).not.toBeInTheDocument();
```

```
      expect(screen.getByText("Completed todo 1")).toBeInTheDocument();
      expect(screen.getByText("Completed todo 2")).toBeInTheDocument();

      const completedButton = screen.getByText("Completed (2)");
      expect(completedButton).toHaveClass("active");
    });

    it("updates filter counts when todos change", async () => {
      const user = userEvent.setup();

      render(<TodoApp />);

      // Toggle an active todo to completed
      await user.click(
        screen.getByLabelText('Mark "Active todo 1" as complete')
      );

      expect(screen.getByText("All (4)")).toBeInTheDocument();
      expect(screen.getByText("Active (1)")).toBeInTheDocument();
      expect(screen.getByText("Completed (3)")).toBeInTheDocument();
    });
  });

  describe("LocalStorage Integration", () => {
    it("saves todos to localStorage when todos change", async () => {
      const user = userEvent.setup();
      localStorageMock.getItem.mockReturnValue(null);

      render(<TodoApp />);

      const input = screen.getByPlaceholderText("What needs to be done?");

      await user.type(input, "New todo");
      await user.click(screen.getByText("Add"));

      await waitFor(() => {
        expect(localStorageMock.setItem).toHaveBeenCalledWith(
          "todos",
          expect.stringContaining("New todo")
        );
      });
    });

    it("handles invalid localStorage data gracefully", () => {
      localStorageMock.getItem.mockReturnValue("invalid json");

      // Should not throw an error
      expect(() => render(<TodoApp />)).not.toThrow();

      expect(
        screen.getByText("No todos yet. Add one above!")
      ).toBeInTheDocument();
    });
  });
```

```
describe("Accessibility", () => {
  beforeEach(() => {
    const savedTodos = [
      { id: 1, text: "Test todo", completed: false, createdAt: "2024-01-01" },
    ];
    localStorageMock.getItem.mockReturnValue(JSON.stringify(savedTodos));
  });

  it("has proper ARIA labels for form controls", () => {
    render(<TodoApp />);

    expect(screen.getByLabelText("New todo")).toBeInTheDocument();
    expect(
      screen.getByLabelText('Mark "Test todo" as complete')
    ).toBeInTheDocument();
    expect(screen.getByLabelText('Delete "Test todo"')).toBeInTheDocument();
  });

  it("supports keyboard navigation", async () => {
    const user = userEvent.setup();

    render(<TodoApp />);

    const input = screen.getByLabelText("New todo");

    // Focus should start on input
    await user.tab();
    expect(input).toHaveFocus();

    // Tab to Add button
    await user.tab();
    expect(screen.getByText("Add")).toHaveFocus();

    // Tab to filter buttons
    await user.tab();
    expect(screen.getByText("All (1)")).toHaveFocus();
  });
});

describe("Edge Cases", () => {
  it("handles very long todo text", async () => {
    const user = userEvent.setup();
    localStorageMock.getItem.mockReturnValue(null);

    const longText = "A".repeat(1000);

    render(<TodoApp />);

    const input = screen.getByPlaceholderText("What needs to be done?");

    await user.type(input, longText);
    await user.click(screen.getByText("Add"));
```

```
      expect(screen.getByText(longText)).toBeInTheDocument();
    });

    it("handles rapid todo additions", async () => {
      const user = userEvent.setup();
      localStorageMock.getItem.mockReturnValue(null);

      render(<TodoApp />);

      const input = screen.getByPlaceholderText("What needs to be done?");

      // Add multiple todos quickly
      for (let i = 1; i <= 5; i++) {
        await user.type(input, `Todo ${i}`);
        await user.click(screen.getByText("Add"));
      }

      expect(screen.getByText("All (5)")).toBeInTheDocument();
      expect(screen.getByText("Todo 1")).toBeInTheDocument();
      expect(screen.getByText("Todo 5")).toBeInTheDocument();
    });
  });
});
```

🎯 Best Practices for Testing

1. Test Structure and Organization

```
// ✅ Good test structure
describe("ComponentName", () => {
  // Setup and teardown
  beforeEach(() => {
    // Reset mocks, clear localStorage, etc.
  });

  describe("Rendering", () => {
    it("renders with default props", () => {});
    it("renders with custom props", () => {});
  });

  describe("User Interactions", () => {
    it("handles click events", () => {});
    it("handles form submission", () => {});
  });

  describe("Edge Cases", () => {
    it("handles error states", () => {});
    it("handles loading states", () => {});
  });
});
```

2. Query Priority

```
// ✅ Query priority (in order of preference)

// 1. Accessible to everyone (screen readers, etc.)
screen.getByRole("button", { name: /submit/i });
screen.getByLabelText(/email/i);
screen.getByPlaceholderText(/search/i);
screen.getByText(/hello world/i);

// 2. Semantic queries
screen.getByAltText(/profile picture/i);
screen.getByTitle(/close/i);

// 3. Test IDs (last resort)
screen.getByTestId("submit-button");
```

3. Async Testing Patterns

```
// ✅ Async testing best practices

// Use waitFor for async operations
await waitFor(() => {
  expect(screen.getByText("Data loaded")).toBeInTheDocument();
});

// Use findBy for elements that will appear
const element = await screen.findByText("Async content");

// Use act for state updates
act(() => {
  result.current.updateState();
});

// Set reasonable timeouts
await waitFor(
  () => expect(screen.getByText("Slow content")).toBeInTheDocument(),
  { timeout: 5000 }
);
```

4. Mock Management

```
// ✅ Mock management best practices

// Clear mocks between tests
beforeEach(() => {
  jest.clearAllMocks();
});
```

```
// Use specific mocks for specific tests
it("handles API error", () => {
  mockApiCall.mockRejectedValueOnce(new Error("API Error"));
  // Test error handling
});

// Restore original implementations
afterAll(() => {
  jest.restoreAllMocks();
});
```

Next up: We'll explore performance optimization techniques and learn how to build fast, efficient React applications! ⚡

18. Performance Optimization ⚡

> "Premature optimization is the root of all evil. But when the time comes, optimize like your users' experience depends on it." - Performance Philosophy

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- How to identify performance bottlenecks in React apps
- Core optimization techniques and when to use them
- Bundle optimization and code splitting strategies
- Image and asset optimization
- Memory leak prevention and cleanup
- Performance monitoring and measurement tools
- Real-world optimization patterns and case studies

🔍 Understanding React Performance

The React Rendering Process

```
// React's rendering phases
1. Trigger (state change, props change, parent re-render)
2. Render (create virtual DOM, run components)
3. Commit (update real DOM, run effects)
4. Browser Paint (layout, paint, composite)
```

Common Performance Issues

1. **Unnecessary Re-renders**
2. **Large Bundle Sizes**

3. **Inefficient State Updates**
4. **Memory Leaks**
5. **Blocking Operations**
6. **Poor Image/Asset Loading**

🛠️ Performance Measurement Tools

React DevTools Profiler

```
// ✅ Using React DevTools Profiler
// 1. Install React DevTools browser extension
// 2. Open DevTools → Profiler tab
// 3. Click record, interact with app, stop recording
// 4. Analyze flame graph and ranked chart

// Profiler API for programmatic measurement
import { Profiler } from "react";

function onRenderCallback(
  id,
  phase,
  actualDuration,
  baseDuration,
  startTime,
  commitTime
) {
  console.log("Profiler:", {
    id, // Component tree identifier
    phase, // "mount" or "update"
    actualDuration, // Time spent rendering
    baseDuration, // Estimated time without memoization
    startTime, // When React began rendering
    commitTime, // When React committed the update
  });

  // Send to analytics in production
  if (process.env.NODE_ENV === "production") {
    analytics.track("Component Render", {
      componentId: id,
      renderTime: actualDuration,
      phase,
    });
  }
}

function App() {
  return (
    <Profiler id="App" onRender={onRenderCallback}>
      <Header />
      <Main />
      <Footer />
    </Profiler>
```

```
  );
}
```

Custom Performance Hooks

```
// ✅ Custom hooks for performance monitoring
import { useEffect, useRef, useState } from "react";

// Hook to measure component render time
function useRenderTime(componentName) {
  const renderStart = useRef(performance.now());

  useEffect(() => {
    const renderEnd = performance.now();
    const renderTime = renderEnd - renderStart.current;

    console.log(`${componentName} render time: ${renderTime.toFixed(2)}ms`);

    // Reset for next render
    renderStart.current = performance.now();
  });
}

// Hook to track re-render count
function useRenderCount(componentName) {
  const renderCount = useRef(0);

  useEffect(() => {
    renderCount.current += 1;
    console.log(`${componentName} rendered ${renderCount.current} times`);
  });

  return renderCount.current;
}

// Hook to detect unnecessary re-renders
function useWhyDidYouUpdate(name, props) {
  const previousProps = useRef();

  useEffect(() => {
    if (previousProps.current) {
      const allKeys = Object.keys({ ...previousProps.current, ...props });
      const changedProps = {};

      allKeys.forEach((key) => {
        if (previousProps.current[key] !== props[key]) {
          changedProps[key] = {
            from: previousProps.current[key],
            to: props[key],
          };
        }
```

```
      });

      if (Object.keys(changedProps).length) {
        console.log("[why-did-you-update]", name, changedProps);
      }
    }

    previousProps.current = props;
  });
}

// Usage example
function ExpensiveComponent({ data, config, onUpdate }) {
  useRenderTime("ExpensiveComponent");
  const renderCount = useRenderCount("ExpensiveComponent");
  useWhyDidYouUpdate("ExpensiveComponent", { data, config, onUpdate });

  return (
    <div>
      <h3>Expensive Component (Render #{renderCount})</h3>
      {/* Component content */}
    </div>
  );
}
```

Performance Monitoring Component

```
// ✅ Comprehensive performance monitoring
function PerformanceMonitor({ children, threshold = 16 }) {
  const [metrics, setMetrics] = useState({
    fps: 0,
    memoryUsage: 0,
    slowRenders: 0
  });

  useEffect(() => {
    let frameCount = 0;
    let lastTime = performance.now();
    let animationId;

    const measureFPS = () => {
      frameCount++;
      const currentTime = performance.now();

      if (currentTime >= lastTime + 1000) {
        const fps = Math.round((frameCount * 1000) / (currentTime - lastTime));

        setMetrics(prev => ({
          ...prev,
          fps,
          memoryUsage: (performance as any).memory?.usedJSHeapSize || 0
```

```
        }));

        frameCount = 0;
        lastTime = currentTime;
      }

      animationId = requestAnimationFrame(measureFPS);
    };

    measureFPS();

    return () => {
      if (animationId) {
        cancelAnimationFrame(animationId);
      }
    };
  }, []);

  const onRender = useCallback((id, phase, actualDuration) => {
    if (actualDuration > threshold) {
      setMetrics(prev => ({
        ...prev,
        slowRenders: prev.slowRenders + 1
      }));

      console.warn(`Slow render detected: ${id} took
${actualDuration.toFixed(2)}ms`);
    }
  }, [threshold]);

  return (
    <>
      <Profiler id="PerformanceMonitor" onRender={onRender}>
        {children}
      </Profiler>

      {process.env.NODE_ENV === 'development' && (
        <div className="performance-monitor">
          <div>FPS: {metrics.fps}</div>
          <div>Memory: {(metrics.memoryUsage / 1024 / 1024).toFixed(2)} MB</div>
          <div>Slow Renders: {metrics.slowRenders}</div>
        </div>
      )}
    </>
  );
}
```

🚀 Core Optimization Techniques

1. Memoization Strategies

```jsx
// ✅ Strategic use of React.memo
const ExpensiveListItem = React.memo(
  function ListItem({ item, onUpdate }) {
    console.log("Rendering ListItem:", item.id);

    return (
      <div className="list-item">
        <h3>{item.title}</h3>
        <p>{item.description}</p>
        <button onClick={() => onUpdate(item.id)}>Update</button>
      </div>
    );
  },
  (prevProps, nextProps) => {
    // Custom comparison function
    return (
      prevProps.item.id === nextProps.item.id &&
      prevProps.item.title === nextProps.item.title &&
      prevProps.item.description === nextProps.item.description &&
      prevProps.onUpdate === nextProps.onUpdate
    );
  }
);

// ✅ Optimized parent component
function OptimizedList({ items }) {
  const [selectedId, setSelectedId] = useState(null);

  // Memoize callback to prevent unnecessary re-renders
  const handleUpdate = useCallback((id) => {
    setSelectedId(id);
    // Perform update logic
  }, []);

  // Memoize filtered items
  const visibleItems = useMemo(() => {
    return items.filter((item) => item.visible);
  }, [items]);

  return (
    <div className="optimized-list">
      {visibleItems.map((item) => (
        <ExpensiveListItem key={item.id} item={item} onUpdate={handleUpdate} />
      ))}
    </div>
  );
}
```

2. State Structure Optimization

```
// ❌ Poor state structure - causes unnecessary re-renders
function BadExample() {
  const [state, setState] = useState({
    user: { name: "", email: "" },
    posts: [],
    comments: [],
    ui: { loading: false, error: null },
  });

  // Any state change re-renders everything
  const updateLoading = (loading) => {
    setState((prev) => ({
      ...prev,
      ui: { ...prev.ui, loading },
    }));
  };

  return (
    <div>
      <UserProfile user={state.user} /> {/* Re-renders on loading change */}
      <PostList posts={state.posts} /> {/* Re-renders on loading change */}
      <CommentList comments={state.comments} />{" "}
      {/* Re-renders on loading change */}
    </div>
  );
}

// ✅ Optimized state structure - isolated state changes
function GoodExample() {
  const [user, setUser] = useState({ name: "", email: "" });
  const [posts, setPosts] = useState([]);
  const [comments, setComments] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  return (
    <div>
      <UserProfile user={user} /> {/* Only re-renders when user changes */}
      <PostList posts={posts} /> {/* Only re-renders when posts change */}
      <CommentList comments={comments} />{" "}
      {/* Only re-renders when comments change */}
      {loading && <LoadingSpinner />} {/* Only re-renders when loading changes */}
      {error && <ErrorMessage error={error} />}
    </div>
  );
}
```

3. Component Splitting and Lazy Loading

```jsx
// ✅ Code splitting with React.lazy
const LazyDashboard = React.lazy(() => import("./Dashboard"));
const LazyProfile = React.lazy(() => import("./Profile"));
const LazySettings = React.lazy(() => import("./Settings"));

// Custom hook for lazy loading with error handling
function useLazyComponent(importFunc, fallback = null) {
  const [Component, setComponent] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const loadComponent = useCallback(async () => {
    if (Component) return Component;

    setLoading(true);
    setError(null);

    try {
      const module = await importFunc();
      const LoadedComponent = module.default || module;
      setComponent(() => LoadedComponent);
      return LoadedComponent;
    } catch (err) {
      setError(err);
      console.error("Failed to load component:", err);
    } finally {
      setLoading(false);
    }
  }, [importFunc, Component]);

  useEffect(() => {
    loadComponent();
  }, [loadComponent]);

  if (error) {
    return fallback || <div>Failed to load component</div>;
  }

  if (loading || !Component) {
    return fallback || <div>Loading...</div>;
  }

  return Component;
}

// Route-based code splitting
function App() {
  return (
    <Router>
      <Suspense fallback={<PageLoader />}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/dashboard" element={<LazyDashboard />} />
```

```
            <Route path="/profile" element={<LazyProfile />} />
            <Route path="/settings" element={<LazySettings />} />
          </Routes>
        </Suspense>
      </Router>
    );
}

// Component-based lazy loading
function ConditionalFeature({ shouldLoad, children }) {
  const [showFeature, setShowFeature] = useState(false);

  const LazyFeature = useMemo(() => {
    if (!shouldLoad) return null;

    return React.lazy(() => import("./ExpensiveFeature"));
  }, [shouldLoad]);

  if (!shouldLoad) {
    return <button onClick={() => setShowFeature(true)}>Load Feature</button>;
  }

  return (
    <Suspense fallback={<div>Loading feature...</div>}>
      {showFeature && LazyFeature && <LazyFeature />}
    </Suspense>
  );
}
```

4. Virtual Scrolling for Large Lists

```
// ✅ Virtual scrolling implementation
import { useState, useEffect, useRef, useMemo } from "react";

function VirtualList({
  items,
  itemHeight,
  containerHeight,
  renderItem,
  overscan = 5,
}) {
  const [scrollTop, setScrollTop] = useState(0);
  const scrollElementRef = useRef();

  const { visibleItems, totalHeight, offsetY } = useMemo(() => {
    const containerItemCount = Math.ceil(containerHeight / itemHeight);
    const startIndex = Math.floor(scrollTop / itemHeight);
    const endIndex = Math.min(
      startIndex + containerItemCount + overscan,
      items.length
    );
```

```
    const visibleItems = items.slice(
      Math.max(0, startIndex - overscan),
      endIndex
    );

    return {
      visibleItems,
      totalHeight: items.length * itemHeight,
      offsetY: Math.max(0, startIndex - overscan) * itemHeight,
    };
  }, [items, itemHeight, scrollTop, containerHeight, overscan]);

  const handleScroll = (e) => {
    setScrollTop(e.target.scrollTop);
  };

  return (
    <div
      ref={scrollElementRef}
      style={{ height: containerHeight, overflow: "auto" }}
      onScroll={handleScroll}
    >
      <div style={{ height: totalHeight, position: "relative" }}>
        <div
          style={{
            transform: `translateY(${offsetY}px)`,
            position: "absolute",
            top: 0,
            left: 0,
            right: 0,
          }}
        >
          {visibleItems.map((item, index) => (
            <div key={item.id || index} style={{ height: itemHeight }}>
              {renderItem(item, index)}
            </div>
          ))}
        </div>
      </div>
    </div>
  );
}

// Usage example
function LargeDataList() {
  const [data] = useState(() =>
    Array.from({ length: 10000 }, (_, i) => ({
      id: i,
      name: `Item ${i}`,
      description: `Description for item ${i}`,
    }))
  );
```

```
  const renderItem = useCallback(
    (item) => (
      <div className="list-item">
        <h4>{item.name}</h4>
        <p>{item.description}</p>
      </div>
    ),
    []
  );

  return (
    <VirtualList
      items={data}
      itemHeight={80}
      containerHeight={400}
      renderItem={renderItem}
    />
  );
}
```

Bundle Optimization

Webpack Bundle Analysis

```
// webpack.config.js - Bundle analysis setup
const BundleAnalyzerPlugin =
  require("webpack-bundle-analyzer").BundleAnalyzerPlugin;

module.exports = {
  // ... other config
  plugins: [
    // Analyze bundle in production
    process.env.ANALYZE &&
      new BundleAnalyzerPlugin({
        analyzerMode: "static",
        openAnalyzer: false,
        reportFilename: "bundle-report.html",
      }),
  ].filter(Boolean),

  optimization: {
    splitChunks: {
      chunks: "all",
      cacheGroups: {
        vendor: {
          test: /[\\/]node_modules[\\/]/,
          name: "vendors",
          chunks: "all",
        },
        common: {
          name: "common",
```

```
          minChunks: 2,
          chunks: "all",
          enforce: true,
        },
      },
    },
  },
};
```

Tree Shaking and Dead Code Elimination

```
// ✅ Tree-shaking friendly imports
// Instead of importing entire library
import _ from "lodash"; // ❌ Imports entire lodash

// Import only what you need
import debounce from "lodash/debounce"; // ✅ Tree-shakable
import { debounce } from "lodash"; // ✅ Also tree-shakable with modern bundlers

// ✅ Conditional imports for development tools
const DevTools =
  process.env.NODE_ENV === "development"
    ? React.lazy(() => import("./DevTools"))
    : null;

function App() {
  return (
    <div>
      <MainApp />
      {DevTools && (
        <Suspense fallback={null}>
          <DevTools />
        </Suspense>
      )}
    </div>
  );
}

// ✅ Dynamic imports for feature flags
function FeatureComponent({ featureEnabled }) {
  const [FeatureModule, setFeatureModule] = useState(null);

  useEffect(() => {
    if (featureEnabled) {
      import("./AdvancedFeature").then((module) => {
        setFeatureModule(() => module.default);
      });
    }
  }, [featureEnabled]);

  if (!featureEnabled || !FeatureModule) {
```

```
    return <BasicFeature />;
  }

  return <FeatureModule />;
}
```

🖼️ Image and Asset Optimization

Responsive Images and Lazy Loading

```
// ✅ Optimized image component
import { useState, useRef, useEffect } from "react";

function OptimizedImage({
  src,
  alt,
  width,
  height,
  className,
  lazy = true,
  placeholder = "blur",
}) {
  const [isLoaded, setIsLoaded] = useState(false);
  const [isInView, setIsInView] = useState(!lazy);
  const [error, setError] = useState(false);
  const imgRef = useRef();

  // Intersection Observer for lazy loading
  useEffect(() => {
    if (!lazy || isInView) return;

    const observer = new IntersectionObserver(
      ([entry]) => {
        if (entry.isIntersecting) {
          setIsInView(true);
          observer.disconnect();
        }
      },
      { threshold: 0.1 }
    );

    if (imgRef.current) {
      observer.observe(imgRef.current);
    }

    return () => observer.disconnect();
  }, [lazy, isInView]);

  const handleLoad = () => {
    setIsLoaded(true);
  };
```

```
  const handleError = () => {
    setError(true);
  };

  // Generate responsive image sources
  const generateSrcSet = (baseSrc) => {
    const sizes = [480, 768, 1024, 1280, 1920];
    return sizes.map((size) => `${baseSrc}?w=${size} ${size}w`).join(", ");
  };

  if (error) {
    return (
      <div className={`image-error ${className}`} style={{ width, height }}>
        <span>Failed to load image</span>
      </div>
    );
  }

  return (
    <div
      ref={imgRef}
      className={`image-container ${className}`}
      style={{ width, height }}
    >
      {/* Placeholder */}
      {!isLoaded && (
        <div className={`image-placeholder ${placeholder}`}>
          {placeholder === "blur" && <div className="blur-placeholder" />}
          {placeholder === "skeleton" && (
            <div className="skeleton-placeholder" />
          )}
        </div>
      )}

      {/* Actual image */}
      {isInView && (
        <img
          src={src}
          srcSet={generateSrcSet(src)}
          sizes="(max-width: 768px) 100vw, (max-width: 1024px) 50vw, 33vw"
          alt={alt}
          width={width}
          height={height}
          onLoad={handleLoad}
          onError={handleError}
          style={{
            opacity: isLoaded ? 1 : 0,
            transition: "opacity 0.3s ease",
          }}
        />
      )}
    </div>
  );
```

```
}

// ✅ Image gallery with progressive loading
function ImageGallery({ images }) {
  const [loadedCount, setLoadedCount] = useState(0);

  const handleImageLoad = useCallback(() => {
    setLoadedCount((prev) => prev + 1);
  }, []);

  return (
    <div className="image-gallery">
      <div className="loading-progress">
        Loaded: {loadedCount} / {images.length}
      </div>

      <div className="gallery-grid">
        {images.map((image, index) => (
          <OptimizedImage
            key={image.id}
            src={image.src}
            alt={image.alt}
            width={300}
            height={200}
            lazy={index > 6} // Load first 6 immediately
            onLoad={handleImageLoad}
          />
        ))}
      </div>
    </div>
  );
}
```

Asset Preloading Strategy

```
// ✅ Strategic asset preloading
function useAssetPreloader() {
  const [preloadedAssets, setPreloadedAssets] = useState(new Set());

  const preloadImage = useCallback(
    (src) => {
      if (preloadedAssets.has(src)) return Promise.resolve();

      return new Promise((resolve, reject) => {
        const img = new Image();
        img.onload = () => {
          setPreloadedAssets((prev) => new Set([...prev, src]));
          resolve(img);
        };
        img.onerror = reject;
        img.src = src;
```

```
      });
    },
    [preloadedAssets]
  );

  const preloadRoute = useCallback(
    async (routePath) => {
      try {
        // Preload route component
        const routeModule = await import(`./routes${routePath}`);

        // Preload route-specific assets
        const assets = routeModule.preloadAssets || [];
        await Promise.all(assets.map(preloadImage));

        return routeModule;
      } catch (error) {
        console.error("Failed to preload route:", routePath, error);
      }
    },
    [preloadImage]
  );

  return { preloadImage, preloadRoute, preloadedAssets };
}

// ✅ Intelligent preloading based on user behavior
function NavigationWithPreloading() {
  const { preloadRoute } = useAssetPreloader();
  const [hoveredRoute, setHoveredRoute] = useState(null);

  // Preload on hover with debounce
  const debouncedPreload = useMemo(
    () =>
      debounce((route) => {
        if (route) {
          preloadRoute(route);
        }
      }, 300),
    [preloadRoute]
  );

  useEffect(() => {
    debouncedPreload(hoveredRoute);
  }, [hoveredRoute, debouncedPreload]);

  return (
    <nav>
      <Link
        to="/dashboard"
        onMouseEnter={() => setHoveredRoute("/dashboard")}
        onMouseLeave={() => setHoveredRoute(null)}
      >
        Dashboard
```

```
        </Link>

        <Link
          to="/profile"
          onMouseEnter={() => setHoveredRoute("/profile")}
          onMouseLeave={() => setHoveredRoute(null)}
        >
          Profile
        </Link>
      </nav>
    );
}
```

🧠 Memory Management

Preventing Memory Leaks

```
// ✅ Proper cleanup patterns
function ComponentWithCleanup() {
  const [data, setData] = useState(null);
  const abortControllerRef = useRef();
  const intervalRef = useRef();
  const observerRef = useRef();

  useEffect(() => {
    // Create abort controller for fetch requests
    abortControllerRef.current = new AbortController();

    const fetchData = async () => {
      try {
        const response = await fetch("/api/data", {
          signal: abortControllerRef.current.signal,
        });
        const result = await response.json();
        setData(result);
      } catch (error) {
        if (error.name !== "AbortError") {
          console.error("Fetch error:", error);
        }
      }
    };

    fetchData();

    // Set up interval
    intervalRef.current = setInterval(() => {
      fetchData();
    }, 30000);

    // Set up intersection observer
    observerRef.current = new IntersectionObserver((entries) => {
```

```
        // Handle intersection
      });

      const element = document.getElementById("target");
      if (element) {
        observerRef.current.observe(element);
      }

      // Cleanup function
      return () => {
        // Abort ongoing requests
        if (abortControllerRef.current) {
          abortControllerRef.current.abort();
        }

        // Clear interval
        if (intervalRef.current) {
          clearInterval(intervalRef.current);
        }

        // Disconnect observer
        if (observerRef.current) {
          observerRef.current.disconnect();
        }
      };
    }, []);

    return (
      <div>
        {data ? (
          <pre>{JSON.stringify(data, null, 2)}</pre>
        ) : (
          <div>Loading...</div>
        )}
      </div>
    );
  }

  // ✅ Memory-efficient event listeners
  function useEventListener(eventName, handler, element = window) {
    const savedHandler = useRef();

    useEffect(() => {
      savedHandler.current = handler;
    }, [handler]);

    useEffect(() => {
      const isSupported = element && element.addEventListener;
      if (!isSupported) return;

      const eventListener = (event) => savedHandler.current(event);
      element.addEventListener(eventName, eventListener);

      return () => {
```

```
      element.removeEventListener(eventName, eventListener);
    };
  }, [eventName, element]);
}

// ✅ Debounced state updates to prevent excessive re-renders
function useDebouncedState(initialValue, delay) {
  const [value, setValue] = useState(initialValue);
  const [debouncedValue, setDebouncedValue] = useState(initialValue);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return [debouncedValue, setValue];
}
```

🎯 Mini-Challenge: Performance Optimization Dashboard

Challenge: Build a Performance-Optimized Data Dashboard

Create a dashboard that efficiently handles:

- Large datasets (10,000+ items)
- Real-time updates
- Multiple chart types
- Filtering and searching
- Export functionality

```
// Your task: Optimize this dashboard for performance
function DataDashboard() {
  const [data, setData] = useState([]);
  const [filters, setFilters] = useState({
    category: "",
    dateRange: { start: null, end: null },
    searchTerm: "",
  });
  const [chartType, setChartType] = useState("line");
  const [isExporting, setIsExporting] = useState(false);

  // Generate large dataset
  useEffect(() => {
    const generateData = () => {
      return Array.from({ length: 10000 }, (_, i) => ({
        id: i,
```

```
        category: ["A", "B", "C", "D"][Math.floor(Math.random() * 4)],
        value: Math.random() * 1000,
        date: new Date(2024, 0, 1 + Math.floor(Math.random() * 365)),
        name: `Item ${i}`,
        description: `Description for item ${i}`,
        tags: ["tag1", "tag2", "tag3"].slice(
          0,
          Math.floor(Math.random() * 3) + 1
        ),
      }));
    };

    setData(generateData());
  }, []);

  // Filter data
  const filteredData = data.filter((item) => {
    if (filters.category && item.category !== filters.category) return false;
    if (
      filters.searchTerm &&
      !item.name.toLowerCase().includes(filters.searchTerm.toLowerCase())
    )
      return false;
    if (filters.dateRange.start && item.date < filters.dateRange.start)
      return false;
    if (filters.dateRange.end && item.date > filters.dateRange.end)
      return false;
    return true;
  });

  // Calculate statistics
  const stats = {
    total: filteredData.length,
    average:
      filteredData.reduce((sum, item) => sum + item.value, 0) /
      filteredData.length,
    categories: filteredData.reduce((acc, item) => {
      acc[item.category] = (acc[item.category] || 0) + 1;
      return acc;
    }, {}),
  };

  const handleExport = async () => {
    setIsExporting(true);
    // Simulate export process
    await new Promise((resolve) => setTimeout(resolve, 2000));
    setIsExporting(false);
  };

  return (
    <div className="dashboard">
      <header>
        <h1>Data Dashboard</h1>
        <button onClick={handleExport} disabled={isExporting}>
```

```
          {isExporting ? "Exporting..." : "Export Data"}
        </button>
      </header>

      <div className="filters">
        <select
          value={filters.category}
          onChange={(e) =>
            setFilters((prev) => ({ ...prev, category: e.target.value }))
          }
        >
          <option value="">All Categories</option>
          <option value="A">Category A</option>
          <option value="B">Category B</option>
          <option value="C">Category C</option>
          <option value="D">Category D</option>
        </select>

        <input
          type="text"
          placeholder="Search..."
          value={filters.searchTerm}
          onChange={(e) =>
            setFilters((prev) => ({ ...prev, searchTerm: e.target.value }))
          }
        />
      </div>

      <div className="stats">
        <div>Total: {stats.total}</div>
        <div>Average: {stats.average.toFixed(2)}</div>
        <div>
          Categories:{" "}
          {Object.entries(stats.categories)
            .map(([cat, count]) => `${cat}: ${count}`)
            .join(", ")}
        </div>
      </div>

      <div className="chart-controls">
        <button
          className={chartType === "line" ? "active" : ""}
          onClick={() => setChartType("line")}
        >
          Line Chart
        </button>
        <button
          className={chartType === "bar" ? "active" : ""}
          onClick={() => setChartType("bar")}
        >
          Bar Chart
        </button>
        <button
          className={chartType === "pie" ? "active" : ""}
```

```
            onClick={() => setChartType("pie")}
          >
            Pie Chart
          </button>
        </div>

        <div className="chart">
          {chartType === "line" && <LineChart data={filteredData} />}
          {chartType === "bar" && <BarChart data={filteredData} />}
          {chartType === "pie" && <PieChart data={filteredData} />}
        </div>

        <div className="data-table">
          <table>
            <thead>
              <tr>
                <th>ID</th>
                <th>Name</th>
                <th>Category</th>
                <th>Value</th>
                <th>Date</th>
              </tr>
            </thead>
            <tbody>
              {filteredData.map((item) => (
                <tr key={item.id}>
                  <td>{item.id}</td>
                  <td>{item.name}</td>
                  <td>{item.category}</td>
                  <td>{item.value.toFixed(2)}</td>
                  <td>{item.date.toLocaleDateString()}</td>
                </tr>
              ))}
            </tbody>
          </table>
        </div>
      </div>
    );
}

// Chart components (simplified)
function LineChart({ data }) {
  return <div>Line Chart with {data.length} points</div>;
}

function BarChart({ data }) {
  return <div>Bar Chart with {data.length} bars</div>;
}

function PieChart({ data }) {
  return <div>Pie Chart with {data.length} slices</div>;
}
```

Solution: Optimized Performance Dashboard

```
// ✅ Optimized dashboard with performance techniques
import { useState, useEffect, useMemo, useCallback, memo } from "react";
import { useDebouncedState, useVirtualList } from "./hooks";

// Memoized chart components
const LineChart = memo(function LineChart({ data }) {
  console.log("Rendering LineChart with", data.length, "points");

  // Only re-render if data reference changes
  return (
    <div className="chart line-chart">
      <h3>Line Chart</h3>
      <p>{data.length} data points</p>
      {/* Actual chart implementation would go here */}
    </div>
  );
});

const BarChart = memo(function BarChart({ data }) {
  console.log("Rendering BarChart with", data.length, "bars");

  return (
    <div className="chart bar-chart">
      <h3>Bar Chart</h3>
      <p>{data.length} data points</p>
    </div>
  );
});

const PieChart = memo(function PieChart({ data }) {
  console.log("Rendering PieChart with", data.length, "slices");

  return (
    <div className="chart pie-chart">
      <h3>Pie Chart</h3>
      <p>{data.length} data points</p>
    </div>
  );
});

// Memoized table row component
const TableRow = memo(function TableRow({ item }) {
  return (
    <tr>
      <td>{item.id}</td>
      <td>{item.name}</td>
      <td>{item.category}</td>
      <td>{item.value.toFixed(2)}</td>
      <td>{item.date.toLocaleDateString()}</td>
    </tr>
```

```
  );
});

// Virtualized table component
function VirtualizedTable({ data }) {
  const renderRow = useCallback(
    (item, index) => <TableRow key={item.id} item={item} />,
    []
  );

  return (
    <div className="data-table">
      <table>
        <thead>
          <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Category</th>
            <th>Value</th>
            <th>Date</th>
          </tr>
        </thead>
      </table>

      <VirtualList
        items={data}
        itemHeight={40}
        containerHeight={400}
        renderItem={renderRow}
      />
    </div>
  );
}

// Custom hook for data filtering with memoization
function useDataFilter(data, filters) {
  return useMemo(() => {
    console.log("Filtering data...", { dataLength: data.length, filters });

    return data.filter((item) => {
      if (filters.category && item.category !== filters.category) return false;
      if (
        filters.searchTerm &&
        !item.name.toLowerCase().includes(filters.searchTerm.toLowerCase())
      )
        return false;
      if (filters.dateRange.start && item.date < filters.dateRange.start)
        return false;
      if (filters.dateRange.end && item.date > filters.dateRange.end)
        return false;
      return true;
    });
  }, [data, filters]);
}
```

```
// Custom hook for statistics calculation
function useDataStats(data) {
  return useMemo(() => {
    console.log("Calculating stats for", data.length, "items");

    if (data.length === 0) {
      return {
        total: 0,
        average: 0,
        categories: {},
      };
    }

    const total = data.length;
    const sum = data.reduce((acc, item) => acc + item.value, 0);
    const average = sum / total;

    const categories = data.reduce((acc, item) => {
      acc[item.category] = (acc[item.category] || 0) + 1;
      return acc;
    }, {});

    return { total, average, categories };
  }, [data]);
}

// Main optimized dashboard component
function OptimizedDataDashboard() {
  // State management
  const [data, setData] = useState([]);
  const [filters, setFilters] = useState({
    category: "",
    dateRange: { start: null, end: null },
    searchTerm: "",
  });
  const [chartType, setChartType] = useState("line");
  const [isExporting, setIsExporting] = useState(false);

  // Debounced search to prevent excessive filtering
  const [debouncedSearchTerm, setDebouncedSearchTerm] = useDebouncedState(
    "",
    300
  );

  // Update filters when debounced search changes
  useEffect(() => {
    setFilters((prev) => ({ ...prev, searchTerm: debouncedSearchTerm }));
  }, [debouncedSearchTerm]);

  // Generate data once on mount
  useEffect(() => {
    console.log("Generating initial data...");
```

```
    const generateData = () => {
      const categories = ["A", "B", "C", "D"];
      const data = [];

      for (let i = 0; i < 10000; i++) {
        data.push({
          id: i,
          category: categories[Math.floor(Math.random() * categories.length)],
          value: Math.random() * 1000,
          date: new Date(2024, 0, 1 + Math.floor(Math.random() * 365)),
          name: `Item ${i}`,
          description: `Description for item ${i}`,
          tags: ["tag1", "tag2", "tag3"].slice(
            0,
            Math.floor(Math.random() * 3) + 1
          ),
        });
      }

      return data;
    };

    // Simulate async data loading
    const timer = setTimeout(() => {
      setData(generateData());
    }, 100);

    return () => clearTimeout(timer);
  }, []);

  // Memoized filtered data
  const filteredData = useDataFilter(data, filters);

  // Memoized statistics
  const stats = useDataStats(filteredData);

  // Memoized chart data (sample for performance)
  const chartData = useMemo(() => {
    // For charts, we might want to sample the data for better performance
    const maxChartPoints = 1000;
    if (filteredData.length <= maxChartPoints) {
      return filteredData;
    }

    // Sample data for chart performance
    const step = Math.floor(filteredData.length / maxChartPoints);
    return filteredData.filter((_, index) => index % step === 0);
  }, [filteredData]);

  // Memoized event handlers
  const handleCategoryChange = useCallback((e) => {
    setFilters((prev) => ({ ...prev, category: e.target.value }));
  }, []);
```

```
  const handleSearchChange = useCallback(
    (e) => {
      setDebouncedSearchTerm(e.target.value);
    },
    [setDebouncedSearchTerm]
  );

  const handleChartTypeChange = useCallback((type) => {
    setChartType(type);
  }, []);

  const handleExport = useCallback(async () => {
    setIsExporting(true);

    try {
      // Simulate export process with Web Workers for heavy processing
      await new Promise((resolve) => {
        const worker = new Worker(
          new URL("./exportWorker.js", import.meta.url)
        );
        worker.postMessage({ data: filteredData });
        worker.onmessage = () => {
          worker.terminate();
          resolve();
        };
      });
    } catch (error) {
      console.error("Export failed:", error);
    } finally {
      setIsExporting(false);
    }
  }, [filteredData]);

  // Memoized chart component
  const ChartComponent = useMemo(() => {
    switch (chartType) {
      case "line":
        return <LineChart data={chartData} />;
      case "bar":
        return <BarChart data={chartData} />;
      case "pie":
        return <PieChart data={chartData} />;
      default:
        return null;
    }
  }, [chartType, chartData]);

  if (data.length === 0) {
    return (
      <div className="dashboard loading">
        <h1>Loading Dashboard...</h1>
        <div className="loading-spinner">⏳</div>
      </div>
    );
```

```
  }

  return (
    <div className="dashboard optimized">
      <header className="dashboard-header">
        <h1>Optimized Data Dashboard</h1>
        <button
          onClick={handleExport}
          disabled={isExporting}
          className="export-button"
        >
          {isExporting ? "⏳ Exporting..." : "📊 Export Data"}
        </button>
      </header>

      <div className="filters">
        <select
          value={filters.category}
          onChange={handleCategoryChange}
          className="category-filter"
        >
          <option value="">All Categories</option>
          <option value="A">Category A</option>
          <option value="B">Category B</option>
          <option value="C">Category C</option>
          <option value="D">Category D</option>
        </select>

        <input
          type="text"
          placeholder="Search items..."
          onChange={handleSearchChange}
          className="search-input"
        />
      </div>

      <div className="stats">
        <div className="stat-item">
          <strong>Total:</strong> {stats.total.toLocaleString()}
        </div>
        <div className="stat-item">
          <strong>Average:</strong> {stats.average.toFixed(2)}
        </div>
        <div className="stat-item">
          <strong>Categories:</strong>{" "}
          {Object.entries(stats.categories)
            .map(([cat, count]) => `${cat}: ${count}`)
            .join(", ")}
        </div>
      </div>

      <div className="chart-controls">
        {["line", "bar", "pie"].map((type) => (
          <button
```

```
              key={type}
              className={chartType === type ? "active" : ""}
              onClick={() => handleChartTypeChange(type)}
            >
              {type.charAt(0).toUpperCase() + type.slice(1)} Chart
            </button>
          ))}
        </div>

        <div className="chart-container">
          {ChartComponent}
          {chartData.length !== filteredData.length && (
            <p className="chart-note">
              📊 Showing {chartData.length} of {filteredData.length} points for
              performance
            </p>
          )}
        </div>

        <VirtualizedTable data={filteredData} />
      </div>
    );
}

export default OptimizedDataDashboard;
```

```
// exportWorker.js - Web Worker for heavy export processing
self.onmessage = function (e) {
  const { data } = e.data;

  // Simulate heavy processing
  const processedData = data.map((item) => ({
    ...item,
    processed: true,
    exportTimestamp: new Date().toISOString(),
  }));

  // Simulate export delay
  setTimeout(() => {
    self.postMessage({ success: true, count: processedData.length });
  }, 1500);
};
```

🎯 Key Performance Takeaways

1. Measurement First

- Always measure before optimizing
- Use React DevTools Profiler
- Monitor real user metrics

- Set performance budgets

2. Optimization Hierarchy

1. **Prevent unnecessary work** (memoization, virtualization)
2. **Reduce work complexity** (efficient algorithms, data structures)
3. **Defer work** (lazy loading, code splitting)
4. **Parallelize work** (Web Workers, concurrent features)

3. Common Optimization Patterns

- Memoize expensive calculations with `useMemo`
- Memoize callbacks with `useCallback`
- Use `React.memo` for component memoization
- Implement virtual scrolling for large lists
- Split code at route boundaries
- Optimize images and assets
- Clean up resources properly

4. Performance Monitoring

- Set up continuous performance monitoring
- Track Core Web Vitals
- Monitor bundle sizes
- Use performance budgets in CI/CD

Next up: We'll explore deployment strategies and learn how to ship React applications to production! 🚀

19. Deployment & Production 🚀

> "Shipping is a feature. A feature that needs to be built, tested, and maintained like any other." - Production Philosophy

🎯 Learning Objectives

By the end of this chapter, you'll understand:

- Production build optimization and configuration
- Deployment strategies and platforms
- Environment management and configuration
- CI/CD pipelines for React applications
- Monitoring and error tracking in production
- Performance optimization for production
- Security best practices
- Scaling and maintenance strategies

🏗️ Production Build Preparation

Build Optimization

```
// package.json - Production scripts
{
  "scripts": {
    "build": "react-scripts build",
    "build:analyze": "npm run build && npx serve -s build",
    "build:profile": "react-scripts build --profile",
    "build:stats": "react-scripts build && npx webpack-bundle-analyzer
build/static/js/*.js",
    "preview": "npx serve -s build -l 3000",
    "test:e2e": "cypress run",
    "test:e2e:ci": "start-server-and-test preview http://localhost:3000 test:e2e"
  },
  "homepage": "https://myapp.com"
}
```

Environment Configuration

```
// .env.production
REACT_APP_API_URL=https://api.myapp.com
REACT_APP_ENVIRONMENT=production
REACT_APP_VERSION=$npm_package_version
REACT_APP_SENTRY_DSN=https://your-sentry-dsn
REACT_APP_ANALYTICS_ID=GA-XXXXXXXXX
GENERATE_SOURCEMAP=false

// .env.staging
REACT_APP_API_URL=https://staging-api.myapp.com
REACT_APP_ENVIRONMENT=staging
REACT_APP_VERSION=$npm_package_version
REACT_APP_SENTRY_DSN=https://your-staging-sentry-dsn
GENERATE_SOURCEMAP=true

// .env.development
REACT_APP_API_URL=http://localhost:8000
REACT_APP_ENVIRONMENT=development
REACT_APP_VERSION=dev
GENERATE_SOURCEMAP=true
```

```
// config/environment.js
const config = {
  development: {
    apiUrl: process.env.REACT_APP_API_URL || "http://localhost:8000",
    enableDevTools: true,
    logLevel: "debug",
    enableMocking: true,
  },
```

```
  staging: {
    apiUrl: process.env.REACT_APP_API_URL,
    enableDevTools: true,
    logLevel: "info",
    enableMocking: false,
  },
  production: {
    apiUrl: process.env.REACT_APP_API_URL,
    enableDevTools: false,
    logLevel: "error",
    enableMocking: false,
  },
};

const environment = process.env.REACT_APP_ENVIRONMENT || "development";

export default {
  ...config[environment],
  environment,
  version: process.env.REACT_APP_VERSION || "unknown",
};
```

Advanced Webpack Configuration

```
// craco.config.js - Custom webpack config with CRACO
const path = require("path");
const { BundleAnalyzerPlugin } = require("webpack-bundle-analyzer");
const CompressionPlugin = require("compression-webpack-plugin");

module.exports = {
  webpack: {
    alias: {
      "@": path.resolve(__dirname, "src"),
      "@components": path.resolve(__dirname, "src/components"),
      "@utils": path.resolve(__dirname, "src/utils"),
      "@hooks": path.resolve(__dirname, "src/hooks"),
    },
    plugins: {
      add: [
        // Gzip compression
        new CompressionPlugin({
          algorithm: "gzip",
          test: /\.(js|css|html|svg)$/,
          threshold: 8192,
          minRatio: 0.8,
        }),

        // Bundle analysis in production
        ...(process.env.ANALYZE === "true"
          ? [
              new BundleAnalyzerPlugin({
```

```
                analyzerMode: "static",
                openAnalyzer: false,
                reportFilename: "bundle-report.html",
              }),
            ]
          : []),
      ],
    },
    configure: (webpackConfig, { env }) => {
      if (env === "production") {
        // Optimize chunks
        webpackConfig.optimization.splitChunks = {
          chunks: "all",
          cacheGroups: {
            vendor: {
              test: /[\\/]node_modules[\\/]/,
              name: "vendors",
              chunks: "all",
              enforce: true,
            },
            common: {
              name: "common",
              minChunks: 2,
              chunks: "all",
              enforce: true,
            },
          },
        };

        // Remove console logs in production

webpackConfig.optimization.minimizer[0].options.terserOptions.compress.drop_consol
e = true;
      }

      return webpackConfig;
    },
  },

  // PWA configuration
  plugins: [
    {
      plugin: require("craco-workbox"),
      options: {
        skipWaiting: true,
        clientsClaim: true,
        runtimeCaching: [
          {
            urlPattern: /^https:\/\/api\.myapp\.com/,
            handler: "NetworkFirst",
            options: {
              cacheName: "api-cache",
              expiration: {
                maxEntries: 50,
```

```
                maxAgeSeconds: 300, // 5 minutes
              },
            },
          },
          {
            urlPattern: /\.(?:png|jpg|jpeg|svg|gif)$/,
            handler: "CacheFirst",
            options: {
              cacheName: "image-cache",
              expiration: {
                maxEntries: 100,
                maxAgeSeconds: 86400, // 24 hours
              },
            },
          },
        ],
      },
    },
  ],
};
```

🌐 Deployment Platforms

Vercel Deployment

```
// vercel.json
{
  "version": 2,
  "builds": [
    {
      "src": "package.json",
      "use": "@vercel/static-build",
      "config": {
        "distDir": "build"
      }
    }
  ],
  "routes": [
    {
      "src": "/static/(.*)",
      "headers": {
        "cache-control": "public, max-age=31536000, immutable"
      }
    },
    {
      "src": "/(.*\\.(js|css|ico|png|jpg|jpeg|svg|gif|woff|woff2)$)",
      "headers": {
        "cache-control": "public, max-age=31536000, immutable"
      }
    },
    {
```

```
      "src": "/(.*)",
      "dest": "/index.html"
    }
  ],
  "headers": [
    {
      "source": "/(.*)",
      "headers": [
        {
          "key": "X-Content-Type-Options",
          "value": "nosniff"
        },
        {
          "key": "X-Frame-Options",
          "value": "DENY"
        },
        {
          "key": "X-XSS-Protection",
          "value": "1; mode=block"
        },
        {
          "key": "Referrer-Policy",
          "value": "strict-origin-when-cross-origin"
        }
      ]
    }
  ],
  "env": {
    "REACT_APP_ENVIRONMENT": "production"
  }
}
```

Netlify Deployment

```
# netlify.toml
[build]
  publish = "build"
  command = "npm run build"

[build.environment]
  REACT_APP_ENVIRONMENT = "production"
  NODE_VERSION = "18"

[[redirects]]
  from = "/api/*"
  to = "https://api.myapp.com/:splat"
  status = 200
  force = true

[[redirects]]
  from = "/*"
```

```
  to = "/index.html"
  status = 200

[[headers]]
  for = "/*"
  [headers.values]
    X-Frame-Options = "DENY"
    X-XSS-Protection = "1; mode=block"
    X-Content-Type-Options = "nosniff"
    Referrer-Policy = "strict-origin-when-cross-origin"
    Content-Security-Policy = "default-src 'self'; script-src 'self' 'unsafe-
inline' https://www.google-analytics.com; style-src 'self' 'unsafe-inline'; img-
src 'self' data: https:; font-src 'self' data:;"

[[headers]]
  for = "/static/*"
  [headers.values]
    Cache-Control = "public, max-age=31536000, immutable"
```

AWS S3 + CloudFront Deployment

```
# deploy-aws.yml (GitHub Actions)
name: Deploy to AWS

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: "18"
          cache: "npm"

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test -- --coverage --watchAll=false

      - name: Build application
        run: npm run build
        env:
          REACT_APP_API_URL: ${{ secrets.REACT_APP_API_URL }}
```

```
          REACT_APP_ENVIRONMENT: production

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Deploy to S3
        run: |
          aws s3 sync build/ s3://${{ secrets.S3_BUCKET }} --delete
          aws s3 cp build/index.html s3://${{ secrets.S3_BUCKET }}/index.html --
cache-control "no-cache"

      - name: Invalidate CloudFront
        run: |
          aws cloudfront create-invalidation --distribution-id ${{
secrets.CLOUDFRONT_DISTRIBUTION_ID }} --paths "/*"
```

Docker Deployment

```
# Dockerfile
# Multi-stage build for optimized production image
FROM node:18-alpine AS builder

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production && npm cache clean --force

# Copy source code
COPY . .

# Build application
RUN npm run build

# Production stage
FROM nginx:alpine AS production

# Copy custom nginx config
COPY nginx.conf /etc/nginx/nginx.conf

# Copy built application
COPY --from=builder /app/build /usr/share/nginx/html

# Add health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
```

```
  CMD curl -f http://localhost/ || exit 1

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

```
# nginx.conf
events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    # Gzip compression
    gzip on;
    gzip_vary on;
    gzip_min_length 1024;
    gzip_types text/plain text/css text/xml text/javascript application/javascript
application/xml+rss application/json;

    # Security headers
    add_header X-Frame-Options "DENY" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header Referrer-Policy "strict-origin-when-cross-origin" always;

    server {
        listen 80;
        server_name localhost;
        root /usr/share/nginx/html;
        index index.html;

        # Cache static assets
        location /static/ {
            expires 1y;
            add_header Cache-Control "public, immutable";
        }

        # Handle client-side routing
        location / {
            try_files $uri $uri/ /index.html;

            # Don't cache the main HTML file
            location = /index.html {
                add_header Cache-Control "no-cache, no-store, must-revalidate";
                add_header Pragma "no-cache";
                add_header Expires "0";
            }
        }
```

```
        # Health check endpoint
        location /health {
            access_log off;
            return 200 "healthy\n";
            add_header Content-Type text/plain;
        }
    }
}
```

```yaml
# docker-compose.yml
version: "3.8"

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:80"
    environment:
      - NODE_ENV=production
    restart: unless-stopped

  # Optional: Add monitoring
  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
```

CI/CD Pipeline

GitHub Actions Workflow

```yaml
# .github/workflows/ci-cd.yml
name: CI/CD Pipeline

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  NODE_VERSION: "18"
  CACHE_KEY: node-modules
```

```yaml
jobs:
  test:
    name: Test
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: "npm"

      - name: Install dependencies
        run: npm ci

      - name: Run linting
        run: npm run lint

      - name: Run type checking
        run: npm run type-check

      - name: Run unit tests
        run: npm test -- --coverage --watchAll=false

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage/lcov.info
          flags: unittests
          name: codecov-umbrella

      - name: Run E2E tests
        run: |
          npm run build
          npm run test:e2e:ci

      - name: Upload E2E artifacts
        uses: actions/upload-artifact@v3
        if: failure()
        with:
          name: cypress-screenshots
          path: cypress/screenshots

  security:
    name: Security Scan
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3
```

```yaml
      - name: Run security audit
        run: npm audit --audit-level=high

      - name: Run Snyk security scan
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}

  build:
    name: Build
    runs-on: ubuntu-latest
    needs: [test, security]

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: "npm"

      - name: Install dependencies
        run: npm ci

      - name: Build application
        run: npm run build
        env:
          REACT_APP_VERSION: ${{ github.sha }}

      - name: Upload build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build-files
          path: build/
          retention-days: 30

  deploy-staging:
    name: Deploy to Staging
    runs-on: ubuntu-latest
    needs: build
    if: github.ref == 'refs/heads/develop'
    environment: staging

    steps:
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: build/

      - name: Deploy to staging
        run: |
```

```
          echo "Deploying to staging environment"
          # Add your staging deployment commands here

  deploy-production:
    name: Deploy to Production
    runs-on: ubuntu-latest
    needs: build
    if: github.ref == 'refs/heads/main'
    environment: production

    steps:
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: build/

      - name: Deploy to production
        run: |
          echo "Deploying to production environment"
          # Add your production deployment commands here

      - name: Notify deployment
        uses: 8398a7/action-slack@v3
        with:
          status: ${{ job.status }}
          channel: "#deployments"
          webhook_url: ${{ secrets.SLACK_WEBHOOK }}
        if: always()
```

📊 Production Monitoring

Error Tracking with Sentry

```
// src/utils/errorTracking.js
import * as Sentry from "@sentry/react";
import { BrowserTracing } from "@sentry/tracing";
import config from "../config/environment";

// Initialize Sentry
if (config.environment === "production") {
  Sentry.init({
    dsn: process.env.REACT_APP_SENTRY_DSN,
    environment: config.environment,
    release: config.version,
    integrations: [
      new BrowserTracing({
        // Set sampling rate for performance monitoring
        tracePropagationTargets: [config.apiUrl],
      }),
    ],
```

```
    tracesSampleRate: 0.1, // 10% of transactions
    beforeSend(event, hint) {
      // Filter out development errors
      if (config.environment === "development") {
        console.error("Sentry Event:", event, hint);
        return null;
      }
      return event;
    },
    beforeBreadcrumb(breadcrumb) {
      // Filter sensitive data from breadcrumbs
      if (breadcrumb.category === "console" && breadcrumb.level === "error") {
        return breadcrumb;
      }
      if (breadcrumb.category === "navigation") {
        return breadcrumb;
      }
      return null;
    },
  });
}

// Error boundary with Sentry integration
export const SentryErrorBoundary = Sentry.withErrorBoundary(
  ({ children }) => children,
  {
    fallback: ({ error, resetError }) => (
      <div className="error-boundary">
        <h2>Something went wrong</h2>
        <p>We've been notified about this error.</p>
        <button onClick={resetError}>Try again</button>
        {config.environment === "development" && (
          <details>
            <summary>Error details</summary>
            <pre>{error.toString()}</pre>
          </details>
        )}
      </div>
    ),
    beforeCapture: (scope, error, errorInfo) => {
      scope.setTag("errorBoundary", true);
      scope.setContext("errorInfo", errorInfo);
    },
  }
);

// Custom error logging
export const logError = (error, context = {}) => {
  if (config.environment === "production") {
    Sentry.withScope((scope) => {
      Object.keys(context).forEach((key) => {
        scope.setContext(key, context[key]);
      });
      Sentry.captureException(error);
```

```
    });
  } else {
    console.error("Error:", error, context);
  }
};

// Performance monitoring
export const startTransaction = (name, op = "navigation") => {
  if (config.environment === "production") {
    return Sentry.startTransaction({ name, op });
  }
  return {
    finish: () => {},
    setTag: () => {},
    setData: () => {},
  };
};
```

Analytics Integration

```
// src/utils/analytics.js
import config from "../config/environment";

class Analytics {
  constructor() {
    this.isEnabled = config.environment === "production";
    this.queue = [];

    if (this.isEnabled) {
      this.initializeGA();
    }
  }

  initializeGA() {
    // Google Analytics 4
    const script = document.createElement("script");
    script.async = true;
    script.src = `https://www.googletagmanager.com/gtag/js?
id=${process.env.REACT_APP_ANALYTICS_ID}`;
    document.head.appendChild(script);

    window.dataLayer = window.dataLayer || [];
    window.gtag = function () {
      window.dataLayer.push(arguments);
    };

    window.gtag("js", new Date());
    window.gtag("config", process.env.REACT_APP_ANALYTICS_ID, {
      page_title: document.title,
      page_location: window.location.href,
    });
```

```
  }

  track(eventName, properties = {}) {
    if (!this.isEnabled) {
      console.log("Analytics (dev):", eventName, properties);
      return;
    }

    // Google Analytics
    if (window.gtag) {
      window.gtag("event", eventName, {
        event_category: properties.category || "General",
        event_label: properties.label,
        value: properties.value,
        ...properties,
      });
    }

    // Custom analytics endpoint
    this.sendToCustomAnalytics(eventName, properties);
  }

  page(path, title) {
    if (!this.isEnabled) {
      console.log("Analytics Page (dev):", path, title);
      return;
    }

    if (window.gtag) {
      window.gtag("config", process.env.REACT_APP_ANALYTICS_ID, {
        page_path: path,
        page_title: title,
      });
    }
  }

  identify(userId, traits = {}) {
    if (!this.isEnabled) {
      console.log("Analytics Identify (dev):", userId, traits);
      return;
    }

    if (window.gtag) {
      window.gtag("config", process.env.REACT_APP_ANALYTICS_ID, {
        user_id: userId,
      });
    }
  }

  async sendToCustomAnalytics(eventName, properties) {
    try {
      await fetch(`${config.apiUrl}/analytics`, {
        method: "POST",
        headers: {
```

```
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            event: eventName,
            properties: {
              ...properties,
              timestamp: new Date().toISOString(),
              url: window.location.href,
              userAgent: navigator.userAgent,
              version: config.version,
            },
          }),
        });
      } catch (error) {
        console.error("Failed to send analytics:", error);
      }
    }
  }

  export default new Analytics();

  // React hook for analytics
  export function useAnalytics() {
    const track = useCallback((eventName, properties) => {
      analytics.track(eventName, properties);
    }, []);

    const page = useCallback((path, title) => {
      analytics.page(path, title);
    }, []);

    const identify = useCallback((userId, traits) => {
      analytics.identify(userId, traits);
    }, []);

    return { track, page, identify };
  }
```

Performance Monitoring

```
  // src/utils/performanceMonitoring.js
  import config from "../config/environment";

  class PerformanceMonitor {
    constructor() {
      this.metrics = new Map();
      this.isEnabled = config.environment === "production";

      if (this.isEnabled) {
        this.initializeWebVitals();
        this.setupPerformanceObserver();
```

```
    }
  }

  async initializeWebVitals() {
    const { getCLS, getFID, getFCP, getLCP, getTTFB } = await import(
      "web-vitals"
    );

    getCLS(this.sendMetric.bind(this));
    getFID(this.sendMetric.bind(this));
    getFCP(this.sendMetric.bind(this));
    getLCP(this.sendMetric.bind(this));
    getTTFB(this.sendMetric.bind(this));
  }

  setupPerformanceObserver() {
    // Long tasks observer
    if ("PerformanceObserver" in window) {
      const longTaskObserver = new PerformanceObserver((list) => {
        list.getEntries().forEach((entry) => {
          this.sendMetric({
            name: "long-task",
            value: entry.duration,
            id: `long-task-${Date.now()}`,
          });
        });
      });

      try {
        longTaskObserver.observe({ entryTypes: ["longtask"] });
      } catch (e) {
        console.warn("Long task observer not supported");
      }
    }
  }

  sendMetric(metric) {
    if (!this.isEnabled) {
      console.log("Performance Metric (dev):", metric);
      return;
    }

    // Send to analytics
    if (window.gtag) {
      window.gtag("event", metric.name, {
        event_category: "Web Vitals",
        value: Math.round(metric.value),
        metric_id: metric.id,
        metric_value: metric.value,
        metric_delta: metric.delta,
      });
    }

    // Send to custom endpoint
```

```
    this.sendToCustomEndpoint(metric);
  }

  async sendToCustomEndpoint(metric) {
    try {
      await fetch(`${config.apiUrl}/metrics`, {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          ...metric,
          timestamp: new Date().toISOString(),
          url: window.location.href,
          userAgent: navigator.userAgent,
          version: config.version,
        }),
      });
    } catch (error) {
      console.error("Failed to send performance metric:", error);
    }
  }

  // Custom performance measurements
  mark(name) {
    if (performance.mark) {
      performance.mark(name);
    }
  }

  measure(name, startMark, endMark) {
    if (performance.measure) {
      performance.measure(name, startMark, endMark);

      const measure = performance.getEntriesByName(name)[0];
      if (measure) {
        this.sendMetric({
          name: `custom-${name}`,
          value: measure.duration,
          id: `custom-${name}-${Date.now()}`,
        });
      }
    }
  }
}

export default new PerformanceMonitor();

// React hook for performance monitoring
export function usePerformanceMonitoring() {
  const mark = useCallback((name) => {
    performanceMonitor.mark(name);
  }, []);
```

```
  const measure = useCallback((name, startMark, endMark) => {
    performanceMonitor.measure(name, startMark, endMark);
  }, []);

  return { mark, measure };
}
```

Security Best Practices

Content Security Policy

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <!-- Security headers -->
    <meta
      http-equiv="Content-Security-Policy"
      content="
    default-src 'self';
    script-src 'self' 'unsafe-inline' https://www.google-analytics.com
https://www.googletagmanager.com;
    style-src 'self' 'unsafe-inline' https://fonts.googleapis.com;
    font-src 'self' https://fonts.gstatic.com;
    img-src 'self' data: https: blob:;
    connect-src 'self' https://api.myapp.com https://www.google-analytics.com;
    frame-ancestors 'none';
    base-uri 'self';
    form-action 'self';
  "
    />

    <meta http-equiv="X-Content-Type-Options" content="nosniff" />
    <meta http-equiv="X-Frame-Options" content="DENY" />
    <meta http-equiv="X-XSS-Protection" content="1; mode=block" />
    <meta
      http-equiv="Referrer-Policy"
      content="strict-origin-when-cross-origin"
    />

    <title>My React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Environment Variable Security

```
// src/utils/security.js

// Sanitize environment variables
export const sanitizeEnvVars = () => {
  const allowedVars = [
    "REACT_APP_API_URL",
    "REACT_APP_ENVIRONMENT",
    "REACT_APP_VERSION",
  ];

  const sanitized = {};

  allowedVars.forEach((varName) => {
    if (process.env[varName]) {
      sanitized[varName] = process.env[varName];
    }
  });

  return sanitized;
};

// Validate API responses
export const validateApiResponse = (response, schema) => {
  // Implement response validation logic
  // Use libraries like Joi or Yup for schema validation
  return response;
};

// Sanitize user input
export const sanitizeInput = (input) => {
  if (typeof input !== "string") return input;

  return input.replace(/[<>"'&]/g, (match) => {
    const escapeMap = {
      "<": "&lt;",
      ">": "&gt;",
      '"': "&quot;",
      "'": "&#x27;",
      "&": "&amp;",
    };
    return escapeMap[match];
  });
};

// Rate limiting for API calls
class RateLimiter {
  constructor(maxRequests = 100, windowMs = 60000) {
    this.maxRequests = maxRequests;
    this.windowMs = windowMs;
    this.requests = new Map();
```

```
  }

  isAllowed(key = "default") {
    const now = Date.now();
    const windowStart = now - this.windowMs;

    if (!this.requests.has(key)) {
      this.requests.set(key, []);
    }

    const requests = this.requests.get(key);

    // Remove old requests
    const validRequests = requests.filter((time) => time > windowStart);

    if (validRequests.length >= this.maxRequests) {
      return false;
    }

    validRequests.push(now);
    this.requests.set(key, validRequests);

    return true;
  }
}

export const apiRateLimiter = new RateLimiter();
```

🎯 Production Checklist

Pre-Deployment Checklist

```
## 🚀 Production Deployment Checklist

### Code Quality

- [ ] All tests passing (unit, integration, e2e)
- [ ] Code coverage above threshold (80%+)
- [ ] No console.log statements in production code
- [ ] No TODO/FIXME comments for critical issues
- [ ] Code reviewed and approved
- [ ] TypeScript errors resolved
- [ ] ESLint warnings addressed

### Performance

- [ ] Bundle size analyzed and optimized
- [ ] Images optimized and compressed
- [ ] Lazy loading implemented for routes
- [ ] Code splitting configured
- [ ] Service worker configured (if PWA)
```

```
- [ ] Caching strategies implemented

### Security

- [ ] Environment variables properly configured
- [ ] No sensitive data in client-side code
- [ ] Content Security Policy configured
- [ ] HTTPS enforced
- [ ] Security headers configured
- [ ] Dependencies audited for vulnerabilities

### Monitoring

- [ ] Error tracking configured (Sentry)
- [ ] Analytics configured (Google Analytics)
- [ ] Performance monitoring setup
- [ ] Health check endpoints working
- [ ] Logging configured

### Infrastructure

- [ ] Domain and SSL certificate configured
- [ ] CDN configured for static assets
- [ ] Backup and disaster recovery plan
- [ ] Monitoring and alerting setup
- [ ] Load balancing configured (if needed)

### Documentation

- [ ] Deployment documentation updated
- [ ] API documentation current
- [ ] Environment setup documented
- [ ] Troubleshooting guide available
```

Post-Deployment Monitoring

```
// src/utils/healthCheck.js
export class HealthChecker {
  constructor() {
    this.checks = new Map();
    this.interval = null;
  }

  addCheck(name, checkFunction, interval = 30000) {
    this.checks.set(name, {
      fn: checkFunction,
      interval,
      lastRun: null,
      status: "unknown",
    });
  }
```

```
  async runCheck(name) {
    const check = this.checks.get(name);
    if (!check) return null;

    try {
      const result = await check.fn();
      check.status = "healthy";
      check.lastRun = new Date();
      return { name, status: "healthy", result };
    } catch (error) {
      check.status = "unhealthy";
      check.lastRun = new Date();
      return { name, status: "unhealthy", error: error.message };
    }
  }

  async runAllChecks() {
    const results = [];

    for (const [name] of this.checks) {
      const result = await this.runCheck(name);
      results.push(result);
    }

    return results;
  }

  startMonitoring() {
    this.interval = setInterval(async () => {
      const results = await this.runAllChecks();
      const unhealthy = results.filter((r) => r.status === "unhealthy");

      if (unhealthy.length > 0) {
        console.warn("Health check failures:", unhealthy);
        // Send alerts
      }
    }, 60000); // Check every minute
  }

  stopMonitoring() {
    if (this.interval) {
      clearInterval(this.interval);
      this.interval = null;
    }
  }
}

// Setup health checks
const healthChecker = new HealthChecker();

// API health check
healthChecker.addCheck("api", async () => {
  const response = await fetch(`${config.apiUrl}/health`);
```

```
    if (!response.ok) {
      throw new Error(`API health check failed: ${response.status}`);
    }
    return response.json();
  });

  // Local storage check
  healthChecker.addCheck("localStorage", () => {
    const testKey = "__test__";
    localStorage.setItem(testKey, "test");
    const value = localStorage.getItem(testKey);
    localStorage.removeItem(testKey);

    if (value !== "test") {
      throw new Error("localStorage not working");
    }

    return { available: true };
  });

  // Performance check
  healthChecker.addCheck("performance", () => {
    const navigation = performance.getEntriesByType("navigation")[0];
    const loadTime = navigation.loadEventEnd - navigation.fetchStart;

    if (loadTime > 5000) {
      throw new Error(`Slow page load: ${loadTime}ms`);
    }

    return { loadTime };
  });

  export default healthChecker;
```

🎯 Key Production Takeaways

1. Build Optimization

- Minimize bundle size with code splitting
- Optimize assets (images, fonts, etc.)
- Enable compression (gzip/brotli)
- Configure caching strategies

2. Deployment Strategy

- Use CI/CD pipelines for automated deployments
- Implement blue-green or rolling deployments
- Have rollback strategies in place
- Test deployments in staging first

3. Monitoring & Observability

- Set up error tracking and alerting
- Monitor performance metrics
- Track user analytics
- Implement health checks

4. Security

- Configure security headers
- Implement Content Security Policy
- Audit dependencies regularly
- Sanitize user inputs

5. Maintenance

- Keep dependencies updated
- Monitor for security vulnerabilities
- Regular performance audits
- Backup and disaster recovery plans

Congratulations! 🎉 You've completed the comprehensive React.js learning journey! You now have the knowledge and tools to build, test, optimize, and deploy production-ready React applications. Keep practicing, stay updated with the React ecosystem, and continue building amazing user experiences!