 **Core React** → JSX, components, props, state, hooks, lifecycle.

 **Advanced React** → Context API, portals, error boundaries, Suspense.

 **Ecosystem** → React Query, Redux, Zustand, React Router, TanStack Table, etc.

 **Performance** → memoization, virtualization, code splitting.

 **Testing** → React Testing Library, Jest.

 **Real-world patterns** → lifting state up, container-presentational, custom hooks.

 **Deployment** → Vite/CRA, Next.js, build optimization.

**1. Core React**

* ✅ JSX → covered
* ✅ Components → functional & class
* ✅ Props → passing, defaultProps, children
* ✅ State → useState, class state
* ✅ Hooks → useState, useEffect, useRef, useContext, useReducer, custom hooks
* ✅ Lifecycle → mounting, updating, unmounting, with class methods and useEffect equivalents

💡 **Already fully covered.**

**2. Advanced React**

* ✅ Context API → avoid prop drilling
* ✅ Portals → modals, tooltips
* ✅ Error boundaries → class-based error handling
* ✅ Suspense → lazy loading components
* ✅ Lazy loading → can be part of Suspense discussion

💡 Covered. Only minor addition could be **Profiler API** for performance tracking in advanced apps, optional for interviews.

**3. Ecosystem Tools**

* ✅ React Query → server state, caching
* ✅ Redux → global state
* ✅ Zustand → lightweight state alternative
* ✅ React Router → routing & navigation
* ✅ TanStack Table → data tables, optional for complex apps

💡 Covered. Could optionally mention **Formik / React Hook Form** for forms, **React DnD** for drag-and-drop, but not mandatory.

**4. Performance**

* ✅ Memoization → React.memo, useMemo, useCallback
* ✅ Virtualization → react-window / react-virtualized
* ✅ Code splitting → React.lazy + Suspense

💡 Covered. Could also note **lazy loading images**, **bundle optimization**, but these are minor.

**5. Testing**

* ✅ React Testing Library → unit & integration tests
* ✅ Jest → test runner + assertions
* ✅ Optional: MSW (mocking API), Cypress (E2E), but not strictly mandatory

**6. Real-World Patterns**

* ✅ Lifting state up → child → parent communication
* ✅ Container-presentational → separating logic/UI
* ✅ Custom hooks → reusable logic

💡 Covered. Optional: HOC, render props, compound components.

**7. Deployment**

* ✅ Vite / CRA → build tools
* ✅ Next.js → SSR / SSG / full-stack React
* ✅ Build optimization → tree-shaking, minification, caching

💡 Covered. Optional: Netlify / Vercel deployment practices.

**✅ Conclusion**

* Your list is **complete for learning + interviews**.
* Minor additions could be **Profiler, Form libraries, HOC/render props**, but not essential unless aiming for **senior-level interviews**.

**Ultimate React Roadmap (Examples + When to Use + Interview Ready)**

**PART 1: CORE REACT**

**Focus:** JSX, Components, Props, State, Hooks, Lifecycle

**1. JSX**

const element = <h1>Hello React</h1>;

const user = "Veda";

const greeting = <h2>Hello, {user}!</h2>;

**When to use:** Always for rendering UI in React.

**2. Components**

**Functional Component**

function Welcome({ name }) {

return <h1>Hello {name}</h1>;

}

**Class Component**

class Welcome extends React.Component {

render() {

return <h1>Hello {this.props.name}</h1>;

}

}

**When to use:** Functional components for modern React; class components for legacy/lifecycle-heavy code.

**3. Props**

function Child({ message }) {

return <p>{message}</p>;

}

function Parent() {

return <Child message="Hello from parent" />;

}

**Key Notes:** Props are **immutable**, can pass **strings, numbers, objects, functions**, and **children**.

**Children Prop:**

**What is children? (Very simple)**

**children is a special prop that holds whatever you put *between* a component’s opening and closing tags.**

**<MyBox>**

**Hello!**

**</MyBox>**

**Inside MyBox, React gives you those inner bits as the children prop:**

**function MyBox(props) {**

**// props.children is "Hello!"**

**return <div className="box">{props.children}</div>;**

**}**

**1) Think of it like a container slot**

**A component is a box. children is the stuff you put inside the box.**

* **<Box>apple</Box> → children is "apple".**
* **<Box><p>apple</p><p>banana</p></Box> → children is two <p> elements.**

**2) Types of children (what can be inside)**

**children can be:**

* **strings or numbers ("Hi", 123)**
* **React elements (<div/>, <UserCard />)**
* **arrays of elements ([<A/>, <B/>])**
* **null, undefined, or false (nothing renders)**
* **functions (if you use render-props pattern)**

**3) Basic usage patterns**

**a) Simple wrapper**

**function Card({ children }) {**

**return <div className="card">{children}</div>;**

**}**

**// usage**

**<Card>**

**<h2>Title</h2>**

**<p>Body text</p>**

**</Card>**

**b) With additional props (header + children)**

**function Page({ title, children }) {**

**return (**

**<div>**

**<h1>{title}</h1>**

**<main>{children}</main>**

**</div>**

**);**

**}**

**// usage**

**<Page title="Users">**

**<UserList />**

**</Page>**

**why children props exist in React and why we use them, step by step, like a baby learning to stack blocks. 🍼**

**1️⃣ Components as Containers**

**React components are like boxes. Often you want a box that can hold any content, not just fixed content.**

**Without children, your component has to decide everything internally:**

**function Card() {**

**return (**

**<div className="card">**

**<h3>Title</h3>**

**<p>Some fixed content</p>**

**</div>**

**);**

**}**

**// You can’t change the content easily**

**With children, your component becomes flexible:**

**function Card({ children }) {**

**return <div className="card">{children}</div>;**

**}**

**// Usage:**

**<Card>**

**<h3>Dynamic Title</h3>**

**<p>Dynamic content</p>**

**</Card>**

**✅ This makes Card reusable — it can hold anything.**

**2️⃣ Separation of Layout and Content**

**children lets your component define layout/structure, but not content.**

**Example: PageLayout defines structure:**

**function PageLayout({ children }) {**

**return (**

**<div>**

**<header>My App Header</header>**

**<main>{children}</main>**

**<footer>Footer</footer>**

**</div>**

**);**

**}**

**// Usage:**

**<PageLayout>**

**<Dashboard />**

**</PageLayout>**

**<PageLayout>**

**<UserList />**

**</PageLayout>**

* **Layout is fixed (header/footer)**
* **Content is dynamic (Dashboard, UserList, etc.)**

**4️⃣ Conditional & Dynamic UI**

**children allows conditional rendering dynamically:**

**<Panel>**

**{isLoading ? <Spinner /> : <DataTable />}**

**</Panel>**

**Without children, Panel would need props for every possible case, which is messy.**

**5️⃣ Render Prop Pattern**

**Sometimes children is a function, letting the parent pass logic or data to child:**

**<DataLoader url="/users">**

**{({ data, loading }) => (loading ? <Spinner /> : <UserList users={data} />)}**

**</DataLoader>**

* **Parent provides logic**
* **Child provides layout**
* **children makes this possible elegantly**

**4. State**

function Counter() {

const [count, setCount] = React.useState(0);

return <button onClick={() => setCount(count+1)}>{count}</button>;

}

**When to use:** For dynamic data inside a component.

**5. Hooks**

* useState → state
* useEffect → side effects / lifecycle
* useContext → global data without prop drilling
* useRef → DOM access / mutable values
* useReducer → complex state logic
* useMemo / useCallback → performance

**6. Lifecycle (Class → Functional)**

| **Class** | **Functional (useEffect)** |
| --- | --- |
| componentDidMount | useEffect(()=>{}, []) |
| componentDidUpdate | useEffect(()=>{}, [deps]) |
| componentWillUnmount | useEffect(()=>{ return cleanup }, []) |

**PART 2: ADVANCED REACT**

**Focus:** Context API, Portals, Error Boundaries, Suspense

**1. Context API**

const ThemeContext = React.createContext();

function App() {

return <ThemeContext.Provider value="dark"><Toolbar /></ThemeContext.Provider>;

}

function Toolbar() {

const theme = React.useContext(ThemeContext);

return <button className={theme}>Click</button>;

}

**When to use:** Avoid prop drilling for theme, auth, user settings.

**2. Portals**

ReactDOM.createPortal(<Modal />, document.getElementById('modal-root'));

**When to use:** Modals, tooltips outside parent DOM hierarchy.

**3. Error Boundaries**

class ErrorBoundary extends React.Component {

state = { hasError: false };

componentDidCatch() { this.setState({ hasError: true }) }

render() { return this.state.hasError ? <h1>Error!</h1> : this.props.children; }

}

**When to use:** Catch runtime errors in UI components.

**4. Suspense & Lazy Loading**

const LazyComponent = React.lazy(() => import('./LazyComponent'));

<Suspense fallback={<p>Loading...</p>}><LazyComponent /></Suspense>

**When to use:** Code splitting and lazy loading large components.

**PART 3: ECOSYSTEM TOOLS**

**Focus:** React Query, Redux, Zustand, Router, TanStack Table

**1. React Query**

const { data, isLoading } = useQuery({

queryKey: ["todos"],

queryFn: () => fetch("/api/todos").then(res => res.json())

});

**When to use:** API fetching, caching, background refetch.

**React Query Features Explained in Detail**

**1️⃣ Caching**

**What it means:**  
Caching is storing the result of an API call locally so that if you request the same data again, you don’t have to hit the server — you can just use the stored copy.

**Old way:**  
Every time a component mounts, you fetch the API again, even if the data hasn’t changed.

**React Query:**

* Automatically **caches API responses** using a key you define (queryKey).
* If another component requests the same data, React Query **returns the cached data immediately**.
* You can configure cache lifetime with staleTime and cacheTime.

**Example:**

const { data } = useQuery({

queryKey: ["todos"],

queryFn: () => fetch("/api/todos").then(res => res.json()),

staleTime: 60000, // data stays fresh for 1 minute

});

✅ **Benefit:** Faster UI, less network traffic, better performance.

**2️⃣ Background Updates**

**What it means:**  
Even if cached data is used, React Query can **fetch the latest data in the background** to keep it fresh without the user noticing.

**Old way:**  
You had to manually write code to refetch on interval, window focus, or tab change.

**React Query:**

* Automatically refetches when:
  + User focuses back on the window
  + Interval (polling) is set
  + Manually triggered

**Example:**

const { data } = useQuery({

queryKey: ["todos"],

queryFn: fetchTodos,

refetchOnWindowFocus: true,

refetchInterval: 5000, // fetch every 5 seconds

});

✅ **Benefit:** Data is always fresh without extra code.

**3️⃣ Automatic Retries**

**What it means:**  
If an API call fails (network error or server issue), React Query can **automatically retry the request** a set number of times before giving up.

**Old way:**  
You had to manually write retry logic, often with try/catch and setTimeout.

**React Query:**

const { data, error } = useQuery({

queryKey: ["todos"],

queryFn: fetchTodos,

retry: 3, // retry up to 3 times automatically

});

✅ **Benefit:** More resilient apps, fewer manual error-handling lines.

**4️⃣ Simplified State Management for API Calls**

**What it means:**  
When fetching API data, you typically manage three states:

1. **Loading** → isLoading
2. **Error** → isError or error
3. **Success/Data** → data

**Old way:**  
You had to manually declare state variables for each component:

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

**React Query:**  
All of this comes **built-in** with useQuery:

const { data, isLoading, isError, error } = useQuery({

queryKey: ["todos"],

queryFn: fetchTodos,

});

* isLoading → automatically true while fetching
* isError → true if request fails
* data → automatically filled when request succeeds

✅ **Benefit:** Less boilerplate, less chance of bugs, very readable.

**Visualizing it Together**

| **Feature** | **Old Way** | **React Query** |
| --- | --- | --- |
| **Cache** | None | Auto-cache by queryKey |
| **Refetch / Updates** | Manual useEffect + intervals | Auto refetch on window focus, interval, manual trigger |
| **Retry** | Manual try/catch + setTimeout | Auto-retry with retry config |
| **State Management** | Manual useState for loading, error, data | isLoading, isError, data built-in |

**✅ TL;DR**

React Query is essentially **a smart API manager for React**. It:

1. **Stores API data** in cache
2. **Keeps it fresh** with background refetch
3. **Retries failed requests** automatically
4. **Reduces boilerplate** for loading/error/data states

It **makes data fetching reactive, reliable, and simple**, which is why it’s so widely used in modern React apps.

**1. Similarities between RTK Query and React Query**

| **Feature** | **RTK Query** | **React Query** |
| --- | --- | --- |
| Purpose | Server state management | Server state management |
| Data fetching | Yes | Yes |
| Caching | Automatic | Automatic |
| Background updates | Automatic | Automatic |
| Loading/Error state | Automatic | Automatic |
| Auto-generated hooks | Yes (useGetXQuery) | Yes (useQuery, useMutation) |
| DevTools | Redux DevTools | React Query DevTools |

✅ Both handle **API calls, caching, retries, and background refresh** for you, so you don’t need to write boilerplate for fetching.

**2. Differences**

| **Feature** | **RTK Query** | **React Query** |
| --- | --- | --- |
| Dependency on Redux | Yes, part of Redux Toolkit | No, independent library |
| Global state integration | Stored in Redux store | Stored internally (React state) |
| Best for | Apps already using Redux | Apps without Redux / any frontend app |
| Boilerplate | Slightly more setup | Very minimal setup |
| Normalization | Supports via entityAdapter | Requires manual or optional normalization |

**3. When to use which**

* **If your app already uses Redux:**
  + RTK Query is ideal, because it integrates directly with your Redux store.
* **If your app does not use Redux:**
  + React Query is simpler and perfect for managing server state without Redux.
* **For large apps with both server + complex client state:**
  + Sometimes people **combine Redux (for client state) + React Query (for server state)**.
  + Or just use **RTK Query** for both server state and some client-side caching.

**2. Redux (Global State)**

const counterSlice = createSlice({name: "counter", initialState:{value:0}, reducers:{increment: state=>{state.value++}}});

**When to use:** Global UI/client state like auth, cart, theme.

**3. React Router**

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

**When to use:** SPA page navigation.

**4. Zustand**

const useStore = create(set => ({ count: 0, increment: () => set(state => ({ count: state.count+1 })) }));

**When to use:** Lightweight state management alternative to Redux.

**5. TanStack Table**

const table = useReactTable({ columns, data });

**When to use:** For complex table rendering, sorting, pagination.

**PART 4: PERFORMANCE + PATTERNS**

* **React.memo / useMemo / useCallback** → prevent unnecessary renders
* **Virtualization** → react-window/react-virtualized for huge lists
* **Code splitting** → React.lazy + Suspense

**Patterns**

* **Lifting state up** → child → parent communication
* **Container-Presentational** → separate logic/UI
* **Custom hooks** → reusable logic
* **HOC / Render Props** → share behavior across components

**PART 5: TESTING & DEPLOYMENT**

**1. Testing**

test("renders hello", () => {

render(<h1>Hello World</h1>);

expect(screen.getByText(/Hello/i)).toBeInTheDocument();

});

* Tools: **Jest**, **React Testing Library**, optionally **MSW**, **Cypress**

**2. Deployment**

* Tools: **Vite**, **CRA**, **Next.js**
* Steps: npm run build → deploy to **Netlify / Vercel / GitHub Pages**
* Build optimization: tree-shaking, caching, minification

**PART 6: INTERVIEW PREP**

**Topics to Master**

1. Props vs State, Controlled vs Uncontrolled
2. useEffect dependencies
3. Lifting state up
4. React Query vs Redux vs Context
5. Hooks differences, custom hooks
6. Patterns: HOC, Render Props, Container-Presentational
7. Performance optimization questions

**Example Coding Task**

* **Todo App:** add/remove/edit tasks, controlled inputs, global state with Redux or Context, API fetch with React Query, lazy load heavy components.