React Interview: Detailed Q&A; with Code Examples

Curated practice pack mixing React fundamentals and React + System Design. Each section includes concise theory, pitfalls, and practical code.

Q: Functional Components vs Class Components

Functional components are plain JS functions returning JSX. Since React 16.8, they can manage state and side-effects using Hooks. Class components extend React.Component, use this.state and lifecycle methods (componentDidMount/Update/Unmount). Modern React favors functional components for less boilerplate, easier testing, and better composition with hooks.

Functional Component with Hooks

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

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);

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

Class Component Equivalent

```
import React from "react";

class CounterClass extends React.Component {
  state = { count: 0 };

  componentDidMount() {
    document.title = `Count: ${this.state.count}`;
  }

  componentDidUpdate() {
    document.title = `Count: ${this.state.count}`;
  }

  render() {
    return (
      <button onClick={() => this.setState(({ count }) => ({ count: count + 1 }))}>
        Clicked {this.state.count} times
      </button>
    );
  }
}
```

Tips:

• *Prefer function components + hooks for new code.*

• *Class components are still needed for legacy codebases and Error Boundaries.*

Q: useEffect vs useLayoutEffect — when to use which?

useEffect runs after the paint (non-blocking). useLayoutEffect runs synchronously after DOM mutations but before paint (blocking). Use useEffect for data fetching/subscriptions. Use useLayoutEffect for layout reads/writes that must happen before the user sees the frame.

useEffect (non-blocking)

```
useEffect(() => {
  const id = setInterval(() => console.log("tick"), 1000);
  return () => clearInterval(id);
}, []);
```

useLayoutEffect (measure & sync layout)

```
const boxRef = useRef(null);
useLayoutEffect(() => {
  const { height } = boxRef.current.getBoundingClientRect();
  boxRef.current.style.minHeight = `${Math.ceil(height)}px`;
}, []);
```

Tips:

• *Default to useEffect; reach for useLayoutEffect only when visual flicker must be avoided.*

• *Overusing useLayoutEffect can hurt performance.*

Q: What are React keys and why are they important?

Keys let React match items between renders for efficient reconciliation. Use stable unique IDs from data; avoid array indices in dynamic lists.

Correct

```
{todos.map(todo => (
  <Todo key={todo.id} todo={todo} />
))}
```

Problem with index keys (reorder/removal)

```
{todos.map((todo, index) => (
  <Todo key={index} todo={todo} />
))}
// Reordering can mix internal state between items.
```

Q: Controlled vs Uncontrolled Components

Controlled inputs get their value from React state and update via onChange; great for validation, conditional logic. Uncontrolled inputs keep their own value in the DOM; access via refs—useful for simple forms or integration with non-React code.

Controlled Input

```
const [name, setName] = useState("");
<input value={name} onChange={e => setName(e.target.value)} />
```

Uncontrolled Input

```
const ref = useRef();
<form onSubmit={e => { e.preventDefault(); alert(ref.current.value); }}>
  <input ref={ref} />
  <button>Submit</button>
</form>
```

Q: React Reconciliation & Diffing

React compares the new virtual DOM to the previous one. If element types differ, it destroys/recreates nodes. For same-type elements, it updates changed props. For lists, it uses keys to map children.

Tips:

• *Avoid unnecessary parent state changes that cascade re-renders.*

• *Use memoization to keep props stable.*

Q: React.memo vs useMemo vs useCallback

React.memo memoizes a component output if props are shallow-equal. useMemo memoizes a computation result. useCallback memoizes a function reference. They prevent unnecessary work and reduce child re-renders when used appropriately.

React.memo

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

useMemo

```
const expensive = useMemo(() => heavyCalc(items), [items]);
```

useCallback + child

```
const onSelect = useCallback((id) => setSelected(id), []);
<List onSelect={onSelect} />  // avoids new fn each render
```

Tips:

• *Always measure before and after optimizing (React Profiler).*

Q: What is React Fiber? How does it enable concurrent features?

Fiber is the reimplementation of React's reconciliation as a cooperative scheduler. It splits rendering into units of work, allowing React to pause, resume, or abort rendering. This enables interruptible rendering, prioritization (user input > background), and features like Suspense and startTransition.

startTransition (mark non-urgent updates)

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

function Search({ query }) {
  const [text, setText] = useState("");
  const [results, setResults] = useState([]);

  const onChange = (e) => {
    const q = e.target.value;
    setText(q); // urgent (keeps typing responsive)
    startTransition(() => {
      setResults(expensiveFilter(q)); // non-urgent
    });
  };

  return <input value={text} onChange={onChange} />;
}
```

Q: React Suspense (data & code loading)

Suspense lets you show a fallback while components or data are loading. For code-splitting: React.lazy. For data: use libraries that support Suspense (Relay, React Query experimental).

Code-splitting with React.lazy

```
const ProductDetails = React.lazy(() => import("./ProductDetails"));
```

```
<Suspense fallback={<Skeleton />}>
  <ProductDetails />
</Suspense>
```

Tips:

• *Wrap only the parts that load; avoid suspense waterfalls by co-locating boundaries.*

Q: Context API vs Redux — when to choose each?

Context is great for simple, low-frequency global values (theme, auth). Redux excels for complex state, predictable updates, DevTools, middleware, and large teams.

Context example

```
const AuthContext = createContext(null);
function App() {
  const [user, setUser] = useState(null);
  return (
    <AuthContext.Provider value={{ user, setUser }}>
      <Routes />
    </AuthContext.Provider>
  );
}
```

Redux slice (RTK)

```
// using @reduxjs/toolkit
const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    inc: (s) => { s.value += 1; },
    dec: (s) => { s.value -= 1; }
  }
});
export const { inc, dec } = counterSlice.actions;
```

Q: CSR vs SSR vs SSG (+ISR)

CSR: render in browser after JS loads (best interactivity, slower first paint). SSR: server renders HTML per request, client hydrates (better SEO/TTFB). SSG: build-time HTML for fast static delivery. ISR (Incremental Static Regeneration) updates static pages on a schedule or when revalidated.

Next.js examples

```
// SSR
export async function getServerSideProps() {
  const data = await fetchAPI();
  return { props: { data } };
}

// SSG
export async function getStaticProps() {
  const data = await fetchAPI();
  return { props: { data }, revalidate: 60 }; // ISR: rebuild at most once/min
}
```

Q: What is hydration?

Hydration attaches event handlers to server-rendered HTML so it becomes interactive without re-rendering from scratch. In React 18+, use hydrateRoot.

Client entry (React 18)

```
import { hydrateRoot } from "react-dom/client";
import App from "./App";

hydrateRoot(document.getElementById("root"), <App />);
```

Q: Error Boundaries — what can they catch?

They catch errors during rendering, in lifecycle methods, and constructors of child components. They do not catch errors in event handlers, async code, SSR, or outside React.

Class-based Error Boundary

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() { return { hasError: true }; }
  componentDidCatch(error, info) { console.error(error, info); }
  render() { return this.state.hasError ? <Fallback /> : this.props.children; }
}
```

Q: Code Splitting — how and why?

Split bundles to improve initial load time. Use dynamic import, React.lazy + Suspense, and route-based splitting. Preload/Prefetch strategic chunks.

Route-based splitting (React Router)

```
const Home = lazy(() => import("./pages/Home"));
const Product = lazy(() => import("./pages/Product"));

<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/p/:id" element={<Product />} />
</Routes>
```

Q: Higher-Order Components (HOCs) — use cases

HOCs are functions that take a component and return an enhanced component, ideal for cross-cutting concerns like auth, feature flags, analytics.

withAuth HOC

```
function withAuth(Component) {
  return function Wrapped(props) {
    const user = useUser();
    if (!user) return <LoginPrompt />;
    return <Component {...props} user={user} />;
  };
}
const Dashboard = withAuth(AnalyticsDashboard);
```

Q: Custom Hooks — reuse stateful logic

Encapsulate complex logic (state + effects) into reusable functions prefixed with 'use'.

useFetch example

```
function useFetch(url) {
  const [data, setData] = useState();
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let active = true;
    setLoading(true);
    fetch(url)
      .then(r => r.json())
      .then(d => { if (active) { setData(d); setLoading(false); } })
      .catch(e => { if (active) { setError(e); setLoading(false); } });
    return () => { active = false; };
  }, [url]);

  return { data, loading, error };
}
```

Q: useState vs useReducer — when to prefer which?

useState is perfect for simple state. useReducer shines when updates are driven by events/actions and state has multiple fields or complex transitions.

useReducer counter

```
function reducer(state, action) {
  switch (action.type) {
    case "inc": return { count: state.count + 1 };
    case "dec": return { count: state.count - 1 };
    case "reset": return { count: 0 };
    default: return state;
  }
}
const [state, dispatch] = useReducer(reducer, { count: 0 });
```

Q: React Performance Techniques

Measure with React Profiler. Use memoization (React.memo/useMemo/useCallback), virtualize long lists, keep state local, avoid needless context churn, debounce expensive handlers, and ship smaller bundles (code splitting, tree-shaking).

Tips:

• *Prefer selective memoization where profiling shows wins.*

• *Split vendor bundles and lazy-load rarely used routes.*

Q: System Design: Fast product listing with 10,000+ items

Frontend: virtualized list (react-window), lazy image loading, skeletons, prefetch next page, stable keys. State: cart in Redux/Context; server data via React Query (caching, pagination, background revalidate). API: server-side filtering/sorting, pagination or cursor-based infinite scroll. Infra: CDN for images, Redis for cache, horizontal scaling behind a load balancer.

Virtualized grid with react-window

```
import { FixedSizeGrid as Grid } from "react-window";

function ProductGrid({ items, columnCount = 4, rowHeight = 300 }) {
  const rowCount = Math.ceil(items.length / columnCount);
  return (
    <Grid
      columnCount={columnCount}
      columnWidth={260}
      height={800}
      rowCount={rowCount}
      rowHeight={rowHeight}
      width={1080}
    >
      {({ columnIndex, rowIndex, style }) => {
        const index = rowIndex * columnCount + columnIndex;
        const item = items[index];
        if (!item) return null;
        return (
          <div style={style}>
            <img src={item.img} loading="lazy" alt={item.title} />
            <div>{item.title}</div>
            <div>{item.price}</div>
            <button>Add to Cart</button>
          </div>
        );
      }}
    </Grid>
  );
}
```

React Query for infinite scroll

```
import { useInfiniteQuery } from "@tanstack/react-query";

function useProducts(params) {
  return useInfiniteQuery({
    queryKey: ["products", params],
    queryFn: ({ pageParam = 0 }) => fetch(`/api/products?cursor=${pageParam}`).then(r => r.json())
    getNextPageParam: (last) => last.nextCursor ?? false
  });
}
```

Q: System Design: Real-time chat

Use WebSockets for duplex communication; fall back to SSE/long-polling if needed. Persist messages in DB, fan-out with Redis pub/sub. On the client, virtualize messages, batch updates, and keep input interactions high-priority with startTransition.

Socket.io basic client

```
import { io } from "socket.io-client";
const socket = io("https://chat.example.com");
socket.on("message", (msg) => setMessages(m => [...m, msg]));
function send(msg) { socket.emit("message", msg); }
```

Q: System Design: Authentication in React apps

Use OAuth/OIDC or username/password with JWT. Store access tokens in HttpOnly secure cookies to mitigate XSS. Use refresh tokens, rotate them, and protect routes with guards. Rate-limit login attempts and use CSRF protection for sensitive endpoints.

Protected Route (React Router v6)

```
function PrivateRoute({ children }) {
  const { user } = useAuth();
  return user ? children : <Navigate to="/login" replace />;
}
```

Q: Pitfall: setState inside useEffect without dependency array

If you call setState inside useEffect without a dependency array, it runs after every render, causing an infinite render loop. Fix by adding a dependency array or using a condition inside the effect.

Problem

```
useEffect(() => {
  setCount(c => c + 1); // runs after EVERY render -> infinite loop
});
```

Fix

```
useEffect(() => {
  setCount(c => c + 1);
}, []); // run once on mount
```

Q: Bonus: Complex forms with useReducer

For multi-field forms with validation, useReducer centralizes updates and makes logic predictable.

Login form reducer

```
function formReducer(state, action) {
  switch (action.type) {
    case "email": return { ...state, email: action.value };
    case "password": return { ...state, password: action.value };
    case "error": return { ...state, error: action.value };
    default: return state;
  }
}
const [state, dispatch] = useReducer(formReducer, { email: "", password: "", error: null });
```