1. What are the key differences between functional and class components? Which would you prefer today?

Answer:

- **Class Components:** Used lifecycle methods (componentDidMount, etc.), have this, and required more boilerplate.
- **Functional Components:** Use **Hooks** (useState, useEffect, etc.), more concise, better readability, and encourage functional programming.
- **Today:** Functional components are preferred because they are lighter, integrate well with hooks, and are more future-proof since class components are legacy.

---

2. How does React's reconciliation (diffing algorithm) work?

Answer:

- React uses the **Virtual DOM** and a **diffing algorithm** (O(n)) instead of $O(n^3)$.
- It compares new and old trees:
    - Different element type → re-render subtree.
    - Same element type → update attributes & recurse children.
    - Keys are used to track elements in lists.
- This makes UI updates efficient and predictable.

---

3. How would you handle performance optimization in a large React app?

Answer:

- Use **React.memo** for memoizing components.
- Use **useMemo / useCallback** to avoid unnecessary recalculations or re-creations of functions.
- Optimize list rendering with **virtualization** (e.g., react-window).
- Lazy load routes/components using **React.lazy + Suspense**.
- Avoid unnecessary re-renders by using **keys** properly.

- Split bundles with **Webpack code splitting**.
- Use **profiling tools** (React DevTools Profiler, Lighthouse).

---

4. Explain Context API vs Redux. When would you use each?

Answer:

- **Context API:** Best for **simple, global state** like theme, auth user, or localization. Lightweight, built-in, no extra library.
- **Redux:** Best for **complex state management**, especially with large teams, asynchronous logic (Redux Thunk, Redux Saga), and when predictability and debugging tools (Redux DevTools) are needed.
  👉 As Tech Lead: Use Redux (or Zustand/Recoil) for complex state; Context API for light global sharing.

---

5. What are React Hooks rules and why do they exist?

Answer:

- Rules:
  1. Only call hooks at the **top level** of a component.
  2. Only call hooks inside **React functions** (components/custom hooks).
- Reason: Hooks rely on **consistent call order**. If called conditionally, React can't map hook state correctly → bugs.

---

6. How do you handle error boundaries in React?

Answer:

- **Error boundaries** are React components that catch JS errors in children and render fallback UI.
- Only class components can be error boundaries (using componentDidCatch).
- For functional components, wrap with error boundary libraries (react-error-boundary).

Example:

```
class ErrorBoundary extends React.Component {
 state = { hasError: false };

 static getDerivedStateFromError() {
  return { hasError: true };
 }

 componentDidCatch(error, info) {
  logError(error, info);
 }

 render() {
  return this.state.hasError ? <h1>Something went wrong</h1> : this.props.children;
 }
}
```

---

7. How would you architect a large React project?

Answer:

- Use a **feature-based folder structure** (not just components/services).
- Adopt a **design system** (Storybook, styled-components, Tailwind).
- Use **TypeScript** for type safety.
- Apply **state management** strategy: Redux/Context/Zustand.
- Follow **SOLID principles** for component design.
- Add **linting & formatting** (ESLint, Prettier).
- Implement **unit/integration tests** (Jest, React Testing Library, Cypress).

- CI/CD pipelines + code review guidelines.

---

8. What is Server-Side Rendering (SSR) vs Client-Side Rendering (CSR) vs Static Site Generation (SSG)?

Answer:

- **CSR (default React):** App renders in browser → slower first load, but faster navigation.
- **SSR (Next.js):** HTML generated on server for each request → better SEO & initial performance.
- **SSG (Next.js/Gatsby):** HTML prebuilt at build time → fastest load, but less dynamic.
  👉 usually suggest **Next.js** for SEO-heavy apps.

---

9. How do you secure a React application?

Answer:

- Never store **tokens** in localStorage (prefer HttpOnly cookies).
- Sanitize user input to prevent **XSS**.
- Use **Content Security Policy (CSP)** headers.
- Implement **role-based access control** in both frontend & backend.
- Avoid exposing sensitive data in frontend bundles.
- Regular dependency audits (npm audit, Snyk).

---

10. How do you handle micro-frontends in React?

Answer:

- Micro-frontends = splitting a large frontend into independently deployable apps.
- Approaches:
  - **Module Federation** (Webpack 5).
  - **Single-SPA** framework.

    - **iFrame-based** (not common now).
- Useful for large orgs with multiple teams working independently.
- As Tech Lead: Define boundaries (shared components, state sync, auth handling).

🔹 **Scenario-Based React Questions for Tech Lead**

1. Your team's React app is becoming slow. Users complain about laggy interactions. How would you diagnose and fix performance issues?

👉 Look for:

- Use **React Profiler** & Chrome DevTools.
- Identify unnecessary re-renders (React.memo, useMemo, useCallback).
- Optimize lists (react-window, pagination, virtualization).
- Lazy load routes/components.
- Split bundles via Webpack/Next.js.
- Monitor real-time performance with tools like **NewRelic, Sentry**.

---

2. You are asked to migrate a legacy app with many class components to functional components. How would you approach it?

👉 Look for:

- Migrate **incrementally**, not in one go.
- Prioritize critical or frequently updated components.
- Replace lifecycle methods with appropriate hooks:
    - componentDidMount → useEffect.
    - componentDidUpdate → useEffect with dependencies.
    - componentWillUnmount → cleanup in useEffect.
- Write tests before refactor to ensure stability.

- Educate team on hooks best practices.

---

3. The backend team exposes a GraphQL API. How do you integrate it into your React application?

👉 Look for:

- Use **Apollo Client** or **Relay**.
- Handle caching & local state with Apollo.
- Use code generation tools (graphql-codegen) for TypeScript types.
- Optimize queries using fragments.
- Handle errors globally with Apollo link.

---

4. You notice that the bundle size of your React app is growing too large. How do you reduce it?

👉 Look for:

- Code splitting with **React.lazy + Suspense**.
- Dynamic imports (import() syntax).
- Tree shaking unused code.
- Use a **CDN for static assets**.
- Compress images & use SVGs.
- Split vendor libraries into separate bundles.
- Analyze bundle size with **Webpack Bundle Analyzer**.

---

5. Your app requires authentication and role-based access control. How would you design it in React?

👉 Look for:

- Store JWT in **HttpOnly cookies** (not localStorage).
- Create a central **Auth Context/Provider**.

- Protect routes with **HOC / ProtectedRoute component**.
- Role-based access with **permissions config** (e.g., canEdit, canView).
- Refresh tokens & silent re-authentication.
- Consider OAuth2 / OIDC for enterprise apps.

---

6. How would you design a React app to support multiple themes (light/dark mode)?

👉 Look for:

- Use **Context API** for theme state.
- CSS variables or styled-components for dynamic styles.
- Persist user choice in localStorage.
- Ensure accessibility (contrast ratios).
- Consider system preference (prefers-color-scheme).

---

7. Your React app must support offline usage. How would you implement it?

👉 Look for:

- Use **Service Workers** (via Workbox or CRA PWA support).
- Cache static assets.
- Use IndexedDB/localStorage for offline data.
- Sync changes when online (background sync).
- Show offline indicators & fallback UIs.

---

8. You are leading a migration from React Router v5 to v6. What are the key considerations?

👉 Look for:

- API changes:
    - Switch → Routes.

    - Route props replaced with useParams, useNavigate.
    - Nested routes improvements.
- Update auth guards & lazy loading strategy.
- Regression testing after migration.
- Ensure team is trained on new APIs.

---

9. You have a global state that multiple teams need to access and update. Would you use Context API or Redux (or something else)? Why?

👉 Look for:

- Context API: lightweight, good for **static/global config** (theme, locale, user).
- Redux (or Zustand, Recoil): best for **complex, shared, mutable state**.
- Tech Lead answer: "I'd evaluate the complexity. For small apps, Context + useReducer is enough. For larger apps with async workflows, Redux Toolkit provides structure and devtools."

---

10. Your React app suddenly shows memory leaks when navigating between pages. How do you debug it?

👉 Look for:

- Use Chrome DevTools **Memory tab**.
- Ensure cleanup in useEffect (remove event listeners, cancel subscriptions).
- Verify no dangling timers/intervals.
- Avoid storing large objects in state unnecessarily.
- Use React DevTools to inspect retained components.

---

11. Your team needs to integrate a micro-frontend strategy. How would you implement it in React?

👉 Look for:

- Options:
    - **Webpack 5 Module Federation**.
    - **Single-SPA** framework.
    - Team-defined shared libraries.
- Challenges:
    - Shared dependencies (React versions).
    - Routing between micro-frontends.
    - Consistent authentication and theming.
- Ensure CI/CD pipelines for independent deployments.

---

12. How would you ensure accessibility (a11y) in a React application?

👉 Look for:

- Use semantic HTML (<button>, <nav>, etc.).
- Add aria-* attributes for screen readers.
- Keyboard navigation support.
- Color contrast checks.
- Automated tools (axe-core, Lighthouse).
- Educate team on inclusive design.