React Interview Questions - Complete Guide (100+ Questions)

Fundamental Concepts (25 questions)

1. What is React? React is a JavaScript library for building user interfaces, especially single-page applications (SPAs). It's based on reusable components and the concept of Virtual DOM for efficient updates.

2. What is JSX? JSX (JavaScript XML) is a syntax extension that allows writing HTML-like code within JavaScript. It gets transpiled to `React.createElement()` calls.

```javascript
const element = <h1>Hello, World!</h1>;
// Transpiles to:
const element = React.createElement('h1', null, 'Hello, World!');
```

3. What is the Virtual DOM? An in-memory representation of the real DOM. React uses the Virtual DOM to:

- Compare previous state with new state (diffing)
- Update only the parts that changed (reconciliation)
- Improve performance by minimizing DOM manipulations

4. What is reconciliation in React? Reconciliation is the algorithm React uses to diff one tree of elements against another to determine what needs to be changed. It's the process of comparing the new Virtual DOM tree with the previous Virtual DOM tree and updating the real DOM efficiently.

```javascript
// When state changes, React:
// 1. Creates new Virtual DOM tree
// 2. Compares with previous tree (diffing)
// 3. Calculates minimum changes needed
// 4. Updates real DOM (reconciliation)
```

5. What's the difference between mount and render?

- **Render**: The process of creating React elements and the Virtual DOM tree from components
- **Mount**: The process of creating DOM nodes and inserting them into the DOM for the first time

```javascript
// Render happens every time component updates
function MyComponent() {
  const [count, setCount] = useState(0); // Re-renders when count changes
  return <div>{count}</div>; // This JSX is "rendered" each time
}

// Mount happens only once when component is first added to DOM
// Unmount happens when component is removed from DOM
```

6. What's the difference between elements and components?

- **Element**: Plain object that describes what you want to see on screen
- **Component**: Function or class that accepts props and returns elements

7. What are functional components vs class components?

- **Functional**: Functions that receive props and return JSX (preferred with hooks)
- **Class**: ES6 classes that extend React.Component (legacy approach)

```javascript
// Functional
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

// Class
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

8. What are props? Arguments passed to components, similar to function parameters. They are immutable and flow from parent to child.

9. What are props.children? A special prop that contains the content between the opening and closing tags of a component. It allows components to be composable and flexible.

```javascript
function Card({ children }) {
  return (
    <div className="card">
      <div className="card-content">
        {children}
      </div>
    </div>
  );
}

// Usage
<Card>
  <h2>Title</h2>
  <p>This content becomes props.children</p>
  <button>Click me</button>
</Card>

// children can be:
// - String: "Hello World"
// - Element: <div>Hello</div>
// - Array: [<p key="1">Para 1</p>, <p key="2">Para 2</p>]
// - Function: {(data) => <div>{data}</div>}
```

10. What is state? Private data of a component that can change over time. When it changes, the component re-renders.

11. What's the difference between props and state?

- **Props**: Data passed from parent, immutable
- **State**: Internal component data, mutable

12. What does "unidirectional data flow" mean? Data flows in one direction: from parent components to child components through props. Children cannot directly modify parent props.

13. What is prop drilling? The problem of passing props through multiple levels of components that don't need them, just to reach a component that does need them.

14. How can prop drilling be avoided?

- Context API
- State management libraries (Redux, Zustand)

- Component composition
- Custom hooks

15. What is React Fragment? A wrapper that allows grouping elements without adding extra DOM nodes.

```javascript
return (
  <React.Fragment>
    <h1>Title</h1>
    <p>Description</p>
  </React.Fragment>
);

// Short syntax
return (
  <>
    <h1>Title</h1>
    <p>Description</p>
  </>
);
```

16. What are controlled vs uncontrolled components?

- **Controlled**: Input value is controlled by React state
- **Uncontrolled**: Value is handled by DOM, accessed via refs

```javascript
// Controlled
function ControlledInput() {
  const [value, setValue] = useState('');
  return (
    <input
      value={value}
      onChange={e => setValue(e.target.value)}
    />
  );
}

// Uncontrolled
function UncontrolledInput() {
  const inputRef = useRef();
  return <input ref={inputRef} />;
}
```

17. What is synthetic event? React's event system that wraps native DOM events, providing consistent API across browsers and additional features.

18. What are the JSX rules?

- Only one root element (or Fragment)
- Close all tags (self-closing for empty elements)
- `className` instead of `class`
- `camelCase` for properties
- JavaScript expressions in curly braces `{}`

19. What is React.StrictMode? A component that activates additional checks and warnings in development to detect potential problems.

20. What is React DevTools? Browser extension that provides debugging tools for React applications:

- Component tree inspection

- Props and state examination
- Performance profiling
- Hook debugging
- Time-travel debugging

```javascript
// Install as browser extension or standalone app
// Features:
// - Components tab: inspect component hierarchy
// - Profiler tab: analyze performance
// - Highlight renders option
// - Show props/state/hooks
```

21. How to count renders in React? Several methods to track component re-renders:

```javascript
// Method 1: useRef counter
function MyComponent() {
  const renderCount = useRef(0);
  renderCount.current += 1;

  console.log(`Component rendered ${renderCount.current} times`);
  return <div>Render count: {renderCount.current}</div>;
}

// Method 2: Custom hook
function useRenderCount() {
  const renderCount = useRef(0);
  renderCount.current += 1;
  return renderCount.current;
}

// Method 3: React DevTools Profiler
// Method 4: why-did-you-render library
```

22. What is React.translate and internationalization? React doesn't have built-in translation features, but common libraries include:

```javascript
```

```
// react-i18next (most popular)
import { useTranslation } from 'react-i18next';

function MyComponent() {
  const { t, i18n } = useTranslation();

  return (
    <div>
      <h1>{t('welcome_message')}</h1>
      <button onClick={() => i18n.changeLanguage('es')}>
        Español
      </button>
    </div>
  );
}

// react-intl (by FormatJS)
import { FormattedMessage } from 'react-intl';

function MyComponent() {
  return (
    <FormattedMessage
      id="welcome_message"
      defaultMessage="Welcome {name}!"
      values={{ name: 'John' }}
    />
  );
}

// Key considerations:
// - Pluralization rules
// - Date/number formatting
// - RTL (Right-to-Left) support
// - Dynamic imports for translations
// - Context-aware translations
```

23. What are keys in React and why are they important? Keys are special string attributes that help React identify which items have changed, been added, or removed in lists.

```
javascript

// ✗ Bad: using index as key
items.map((item, index) =>
  <li key={index}>{item.name}</li>
);

// ✅ Good: using unique, stable identifier
items.map(item =>
  <li key={item.id}>{item.name}</li>
);

// Why keys are important:
// 1. Optimize reconciliation process
// 2. Prevent bugs in stateful components
// 3. Improve performance in dynamic lists
// 4. Help React track component identity
```

24. When are keys not necessary? Keys are not required when:

- Rendering static lists that never change
- List items have no state or user input
- The list never reorders, adds, or removes items
- You're not rendering a list (single element)

```
javascript
```

```
// Keys not needed - static content
function StaticMenu() {
 return (
  <ul>
   <li>Home</li>    {/* No key needed */}
   <li>About</li>   {/* No key needed */}
   <li>Contact</li>  {/* No key needed */}
  </ul>
 );
}
```

25. What are the benefits of using Context API?

- **Avoid prop drilling**: Pass data through component tree without manual prop passing
- **Global state**: Share data across many components
- **Theme management**: Easy to implement theming systems
- **Authentication**: Share user authentication state
- **Localization**: Provide translation context
- **Configuration**: App-wide settings and preferences
- **Performance**: When used correctly, reduces unnecessary re-renders
- **Simplicity**: Simpler than Redux for many use cases

Hooks (30 questions)

26. What are React Hooks? Functions that let you use state and other React features in functional components. Introduced in React 16.8 to provide a more direct API to React concepts.

```javascript
// Definition characteristics:
// - Functions that start with "use"
// - Allow state and lifecycle in functional components
// - Enable logic reuse between components
// - Provide more granular control than class lifecycle methods
```

27. What are the rules of hooks?

- Only call hooks at the top level (not in loops, conditions, nested functions)
- Only call hooks from React functional components or custom hooks

28. What is useState? Hook that allows adding local state to functional components.

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

 return (
  <div>
   <p>Count: {count}</p>
   <button onClick={() => setCount(count + 1)}>
    Increment
   </button>
   <button onClick={() => setCount(prev => prev + 1)}>
    Increment (functional update)
   </button>
  </div>
 );
}
```

29. How does batching work in useState? React batches multiple state updates into a single re-render for performance.

```javascript
function handleClick() {
  setCount(count + 1);
  setFlag(!flag);
  // Only one re-render happens
}
```

30. What is useEffect? Hook for performing side effects in functional components (equivalent to componentDidMount, componentDidUpdate, componentWillUnmount).

```javascript
useEffect(() => {
  // Side effect
  document.title = `Count: ${count}`;

  // Cleanup (optional)
  return () => {
    document.title = 'React App';
  };
}, [count]); // Dependency array
```

31. What are the types of useEffect?

- No dependency array: runs on every render
- Empty array ([]): runs only on mount
- With dependencies ([dep]): runs when dependencies change

32. How to make async calls in useEffect and why? You cannot make useEffect async directly, but you can use async functions inside:

```javascript
```

```
// ✗ Wrong: useEffect cannot be async
useEffect(async () => {
  const data = await fetchData(); // This won't work
  setData(data);
}, []);

// ✅ Correct approaches:

// Method 1: Async function inside useEffect
useEffect(() => {
  const fetchDataAsync = async () => {
    try {
      const data = await fetchData();
      setData(data);
    } catch (error) {
      setError(error);
    }
  };

  fetchDataAsync();
}, []);

// Method 2: IIFE (Immediately Invoked Function Expression)
useEffect(() => {
  (async () => {
    const data = await fetchData();
    setData(data);
  })();
}, []);

// Method 3: Promise with .then()
useEffect(() => {
  fetchData()
    .then(setData)
    .catch(setError);
}, []);

// Why this pattern:
// 1. useEffect expects a cleanup function or nothing
// 2. Async functions return promises, not cleanup functions
// 3. Prevents memory leaks and race conditions
// 4. Allows proper error handling
```

33. What is cleanup in useEffect? Function that runs to clean up effects (cancel subscriptions, clear timers, etc.) before the component unmounts or before the next effect.

34. What is useContext? Hook that allows consuming a Context without needing a Consumer component.

javascript

```
const ThemeContext = React.createContext();

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Themed Button</button>;
}
```

35. What is useReducer? Hook for managing complex state using a reducer pattern similar to Redux.

javascript

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>
        +
      </button>
    </div>
  );
}
```

36. When to use useState vs useReducer?

- **useState**: Simple state, few updates
- **useReducer**: Complex state, multiple sub-values, complex update logic

37. What is useRef? Hook that returns a mutable ref object whose `.current` property persists across renders.

```
javascript

function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

38. What are the uses of useRef?

- Accessing DOM elements
- Storing mutable values that don't cause re-renders
- Keeping references to values across renders
- Previous values storage

39. What is useMemo? Hook that memoizes the result of an expensive computation and only recalculates when dependencies change.

```
javascript
```

```
function ExpensiveComponent({ items }) {
  const expensiveValue = useMemo(() => {
    return items.reduce((sum, item) => sum + item.value, 0);
  }, [items]);

  return <div>Total: {expensiveValue}</div>;
}
```

40. What is useCallback? Hook that memoizes a function and only creates a new instance when dependencies change.

```
javascript

function Parent({ items }) {
  const [filter, setFilter] = useState('');

  const handleClick = useCallback((id) => {
    // handle click logic
  }, []);

  return (
    <div>
      {items.map(item =>
        <Child
          key={item.id}
          item={item}
          onClick={handleClick}
        />
      )}
    </div>
  );
}
```

41. When to use useMemo vs useCallback?

- **useMemo**: Memoizes computed values/objects
- **useCallback**: Memoizes functions

42. When to avoid useMemo and useCallback? Avoid when:

- The computation is not expensive
- Dependencies change frequently
- The memoized value is used only once
- The component rarely re-renders
- Premature optimization without measuring

```
javascript

// ❌ Unnecessary memoization
const simpleValue = useMemo(() => x + y, [x, y]); // Simple arithmetic
const simpleCallback = useCallback(() => setValue(x), [x]); // Called once

// ✅ Beneficial memoization
const expensiveValue = useMemo(() => {
  return heavyComputation(largeDataSet);
}, [largeDataSet]);

const stableCallback = useCallback((id) => {
  // Function passed to many children
}, []);
```

43. What are custom hooks? JavaScript functions that use other hooks and allow extracting component logic for reuse.

```javascript
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = useCallback(() => setCount(c => c + 1), []);
  const decrement = useCallback(() => setCount(c => c - 1), []);
  const reset = useCallback(() => setCount(initialValue), [initialValue]);

  return { count, increment, decrement, reset };
}

// Usage
function CounterComponent() {
  const { count, increment, decrement, reset } = useCounter(10);

  return (
    <div>
      <span>{count}</span>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

44. How to share logic between components?

- Custom hooks
- Higher-Order Components (HOCs)
- Render props
- Context API

45. What is useLayoutEffect? Similar to useEffect but fires synchronously after all DOM mutations, before the browser paints.

46. When to use useLayoutEffect vs useEffect?

- **useLayoutEffect**: When you need to read/modify DOM before paint
- **useEffect**: For most cases (asynchronous)

47. What is useImperativeHandle? Hook that customizes the instance value exposed when using `ref` with `forwardRef`.

48. What is useDebugValue? Hook to display a label in React DevTools for custom hooks.

49. How would you handle global state with hooks?

- Context API + useContext
- Custom hooks that encapsulate state logic
- Libraries like Zustand that use hooks

50. What are dependencies in hooks? Array that specifies which values the hook should "watch" to decide when to execute or recalculate.

51. What happens if you omit dependencies in useEffect? The effect may use stale values (stale closures) or not execute when it should.

52. How to simulate lifecycle methods with hooks?

- `componentDidMount`: `useEffect(() => {}, [])`
- `componentDidUpdate`: `useEffect(() => {})`
- `componentWillUnmount`: cleanup function in useEffect

53. What are the benefits of hooks over class components?

- Simpler code and less boilerplate
- Better logic reuse with custom hooks
- No `this` binding confusion
- Easier testing
- Better tree shaking
- Gradual adoption possible

54. What are the common mistakes with hooks?

- Breaking the rules of hooks
- Missing dependencies in dependency arrays
- Overusing useMemo/useCallback
- Not understanding closure behavior
- Infinite loops in useEffect

55. How do you debug hooks?

- React DevTools hooks section
- useDebugValue for custom hooks
- Console.log in effects
- ESLint plugin for hooks rules
- React DevTools Profiler

State Management and Context API (20 questions)

56. What is the Context API? A mechanism to pass data through the component tree without having to pass props manually at every level.

```javascript
const ThemeContext = React.createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemedButton />;
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Themed Button</button>;
}
```

57. When to use Context API?

- Data needed by many components (theme, auth, language)
- Avoiding prop drilling
- Not for data that changes frequently (can cause re-renders)

58. What are Context API best practices?

```javascript
```

```jsx
// 1. Split contexts by concern
const UserContext = createContext();
const ThemeContext = createContext();

// 2. Provide default values
const ThemeContext = createContext({
  theme: 'light',
  toggleTheme: () => {}
});

// 3. Use custom hooks for consuming context
function useTheme() {
  const context = useContext(ThemeContext);
  if (context === undefined) {
    throw new Error('useTheme must be used within a ThemeProvider');
  }
  return context;
}

// 4. Memoize context values
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');

  const value = useMemo(() => ({
    theme,
    toggleTheme: () => setTheme(t => t === 'light' ? 'dark' : 'light')
  }), [theme]);

  return (
    <ThemeContext.Provider value={value}>
      {children}
    </ThemeContext.Provider>
  );
}

// 5. Split providers for frequently changing data
const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  // User data (changes less frequently)
};

const NotificationProvider = ({ children }) => {
  const [notifications, setNotifications] = useState([]);
  // Notifications (change frequently)
};

// 6. Use composition to avoid deeply nested providers
function AppProviders({ children }) {
  return (
    <ThemeProvider>
      <UserProvider>
        <NotificationProvider>
          {children}
        </NotificationProvider>
      </UserProvider>
    </ThemeProvider>
  );
}
```

59. What are the problems with Context API?

- Re-renders of all consumers when value changes
- Difficult optimizations
- No time-travel debugging like Redux

60. How to optimize Context to avoid re-renders?

- Split contexts (separate frequently changing data)
- Memoize the value with useMemo
- Use React.memo on consumers

```javascript
const value = useMemo(() => ({
  user,
  login,
  logout
}), [user]);

<AuthContext.Provider value={value}>
```

61. What is Redux and why use it? A predictable state container for JavaScript applications. Benefits:

- Single source of truth
- Immutable state
- Time-travel debugging
- Middleware ecosystem

62. What are Redux principles?

- Single source of truth (single store)
- State is read-only (modified only via actions)
- Changes are made with pure functions (reducers)

63. What are actions in Redux? Plain objects that describe what happened in the application. Have a type and optionally payload.

```javascript
const addTodo = (text) => ({
  type: 'ADD_TODO',
  payload: {
    id: Date.now(),
    text,
    completed: false
  }
});
```

64. What are reducers? Pure functions that take current state and an action, and return new state.

```javascript
function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, action.payload];
    case 'TOGGLE_TODO':
      return state.map(todo =>
        todo.id === action.payload.id
          ? { ...todo, completed: !todo.completed }
          : todo
      );
    default:
      return state;
  }
}
```

65. What is Redux Toolkit? Official toolset for efficient Redux usage including:

- configureStore
- createSlice
- createAsyncThunk
- Immer for immutability

66. How do you handle async actions in Redux?

- Redux Thunk: allows action creators that return functions
- Redux Saga: uses generators for side effects
- createAsyncThunk from Redux Toolkit

```javascript
const fetchUser = createAsyncThunk(
  'users/fetchById',
  async (userId) => {
    const response = await api.fetchUser(userId);
    return response.data;
  }
);
```

67. What is middleware in Redux? Functions that execute between dispatching an action and when it reaches the reducer.

68. What are modern alternatives to Redux?

- Zustand: Simple and lightweight
- Jotai: Atomic state management
- Valtio: Proxy-based state
- SWR/React Query: For server state

69. What is server state vs client state?

- **Server state**: Data from server (async, shared, cached)
- **Client state**: Local UI state (sync, private)

70. How to handle server state in React?

- React Query (TanStack Query)
- SWR
- Apollo Client (GraphQL)
- Custom hooks with fetch

71. What is React Query? Library for fetching, caching, synchronizing and updating server state.

```javascript
function Profile() {
  const { data, isLoading, error } = useQuery(
    ['profile'],
    fetchProfile
  );

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>Welcome {data.name}</div>;
}
```

72. When to use local state vs global state?

- **Local state**: Component-specific data, temporary UI state
- **Global state**: Shared data, user authentication, theme, app settings

73. What are the benefits of using Context API over prop drilling?

- Cleaner component APIs
- Less repetitive code
- Easier refactoring
- Better separation of concerns
- Improved maintainability

74. How do you decide between Context API and Redux?

- **Context API**: Simpler state needs, less boilerplate, built into React
- **Redux**: Complex state logic, time-travel debugging, extensive middleware ecosystem

75. What is the compound component pattern with Context? Pattern where multiple components work together, sharing state through Context internally.

```javascript
const AccordionContext = createContext();

function Accordion({ children, ...props }) {
  const [openIndex, setOpenIndex] = useState(null);
  return (
    <AccordionContext.Provider value={{ openIndex, setOpenIndex }}>
      <div {...props}>{children}</div>
    </AccordionContext.Provider>
  );
}

function AccordionItem({ index, children }) {
  const { openIndex, setOpenIndex } = useContext(AccordionContext);
  const isOpen = openIndex === index;

  return (
    <div>
      <button onClick={() => setOpenIndex(isOpen ? null : index)}>
        Toggle
      </button>
      {isOpen && <div>{children}</div>}
    </div>
  );
}

// Usage
<Accordion>
  <AccordionItem index={0}>Content 1</AccordionItem>
  <AccordionItem index={1}>Content 2</AccordionItem>
</Accordion>
```

Performance Optimization (15 questions)

76. How do you optimize React performance?

- React.memo for functional components
- useMemo for expensive computations
- useCallback for stable functions
- Lazy loading with React.lazy
- Code splitting
- Virtualization for large lists

77. What is React.memo? Higher-order component that memoizes functional components, similar to PureComponent for classes.

```javascript
const MemoizedComponent = React.memo(function MyComponent({ name }) {
  return <div>Hello {name}</div>;
});

// With custom comparison
const MemoizedComponent = React.memo(MyComponent, (prevProps, nextProps) => {
  return prevProps.name === nextProps.name;
});
```

78. When to use React.memo?

- Components that render frequently
- Components with expensive renders
- Components that receive same props often
- When memoization cost is less than re-render cost

79. What causes unnecessary re-renders?

- Creating objects/arrays inline in props
- Passing inline functions as props
- Context value changing on every render
- Not using React.memo when needed

80. How would you profile a React application?

- React DevTools Profiler
- Browser DevTools Performance tab
- React.Profiler API
- Libraries like why-did-you-render

81. What is React.lazy? Function that enables dynamic component loading (code splitting).

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

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

82. What is Suspense? Component that lets you specify the loading UI while waiting for some code to load.

83. What are render props? Technique for sharing code between components using a prop whose value is a function.

```javascript
```

```
function DataProvider({ render }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchData().then(setData);
  }, []);

  return render(data);
}

<DataProvider render={data => (
  <div>{data ? <p>{data.message}</p> : <p>Loading...</p>}</div>
)} />
```

84. What are Higher-Order Components (HOCs)? Functions that take a component and return a new component with additional functionality.

javascript

```
function withLoading(Component) {
  return function WithLoadingComponent({ isLoading, ...props }) {
    if (isLoading) return <div>Loading...</div>;
    return <Component {...props} />;
  };
}

const EnhancedComponent = withLoading(MyComponent);
```

85. When to use HOCs vs hooks vs render props?

- **Hooks**: Preferred for most cases, cleaner composition
- **HOCs**: For wrapping logic, cross-cutting concerns
- **Render props**: When you need maximum render flexibility

86. What is virtualization and when to use it? Technique that renders only visible items in large lists to improve performance.

javascript

```
// Using react-window
import { FixedSizeList as List } from 'react-window';

function VirtualizedList({ items }) {
  const Row = ({ index, style }) => (
    <div style={style}>
      {items[index].name}
    </div>
  );

  return (
    <List
      height={600}
      itemCount={items.length}
      itemSize={50}
    >
      {Row}
    </List>
  );
}
```

87. How do you implement code splitting in React?

- React.lazy for component-level splitting
- Dynamic imports for utility functions

- Route-based splitting
- Webpack bundle splitting

88. What is the difference between client-side and server-side rendering performance?

- **CSR**: Faster subsequent navigation, slower initial load
- **SSR**: Faster initial load, better SEO, slower navigation

89. How do you optimize bundle size?

- Tree shaking
- Code splitting
- Import only what you need
- Bundle analyzer
- Compress assets
- Remove unused dependencies

90. What are the performance implications of Context?

- All consumers re-render when context value changes
- Can cause performance issues with frequently changing data
- Solution: split contexts, memoize values

Forms and Validation (10 questions)

91. What are popular form libraries for React?

```javascript
```

```
// React Hook Form (most popular)
import { useForm } from 'react-hook-form';

function MyForm() {
  const { register, handleSubmit, formState: { errors } } = useForm();

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

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName", { required: true })} />
      {errors.firstName && <span>This field is required</span>}
      <input type="submit" />
    </form>
  );
}

// Formik (established library)
import { Formik, Form, Field } from 'formik';

function MyForm() {
  return (
    <Formik
      initialValues={{ email: '' }}
      onSubmit={(values) => console.log(values)}
    >
      <Form>
        <Field type="email" name="email" />
        <button type="submit">Submit</button>
      </Form>
    </Formik>
  );
}

// React Final Form
// Unform
// Ant Design Forms
```

92. What are popular validation libraries?

```
javascript
```

```javascript
// Yup (schema validation)
import * as yup from 'yup';

const schema = yup.object({
  email: yup.string().email().required(),
  age: yup.number().positive().integer().required(),
});

// Joi
const Joi = require('joi');

const schema = Joi.object({
  email: Joi.string().email().required(),
  password: Joi.string().min(8).required()
});

// Zod (TypeScript-first)
import { z } from 'zod';

const UserSchema = z.object({
  email: z.string().email(),
  age: z.number().min(18)
});

// React Hook Form with Yup
import { yupResolver } from '@hookform/resolvers/yup';

const { register, handleSubmit } = useForm({
  resolver: yupResolver(schema)
});

// Custom validation
const validateEmail = (email) => {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
};
```

93. How do you handle form state in React?

```javascript
```

```
// Controlled components (React manages state)
function ControlledForm() {
  const [formData, setFormData] = useState({
    email: '',
    password: ''
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
  };

  return (
    <form>
      <input
        name="email"
        value={formData.email}
        onChange={handleChange}
      />
    </form>
  );
}

// Uncontrolled components (DOM manages state)
function UncontrolledForm() {
  const formRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    const formData = new FormData(formRef.current);
    console.log(Object.fromEntries(formData));
  };

  return (
    <form ref={formRef} onSubmit={handleSubmit}>
      <input name="email" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

94. What are the pros and cons of different form approaches?

- **Controlled**: Better validation, React-like, but more boilerplate
- **Uncontrolled**: Less boilerplate, better performance, but less control
- **Libraries**: Less boilerplate, built-in validation, but additional dependency

95. How do you handle complex form validation?

```
javascript
```

```
// Field-level validation
const validateEmail = (value) => {
  if (!value) return 'Email is required';
  if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value)) {
    return 'Invalid email format';
  }
};

// Form-level validation
const validateForm = (values) => {
  const errors = {};

  if (!values.email) errors.email = 'Required';
  if (!values.password) errors.password = 'Required';
  if (values.password !== values.confirmPassword) {
    errors.confirmPassword = 'Passwords must match';
  }

  return errors;
};

// Async validation
const validateUsernameAsync = async (username) => {
  const response = await api.checkUsername(username);
  return response.available ? undefined : 'Username taken';
};
```

96. How do you optimize form performance?

- Use uncontrolled components when possible
- Debounce validation
- Memo expensive validation functions
- Use React Hook Form for better performance
- Validate on blur instead of change for heavy validations

97. How do you handle dynamic forms?

javascript

```
function DynamicForm() {
  const [fields, setFields] = useState([{ id: 1, value: '' }]);

  const addField = () => {
    setFields([...fields, { id: Date.now(), value: '' }]);
  };

  const removeField = (id) => {
    setFields(fields.filter(field => field.id !== id));
  };

  const updateField = (id, value) => {
    setFields(fields.map(field =>
      field.id === id ? { ...field, value } : field
    ));
  };

  return (
    <form>
      {fields.map(field => (
        <div key={field.id}>
          <input
            value={field.value}
            onChange={(e) => updateField(field.id, e.target.value)}
          />
          <button onClick={() => removeField(field.id)}>Remove</button>
        </div>
      ))}
      <button onClick={addField}>Add Field</button>
    </form>
  );
}
```

98. How do you handle file uploads in forms?

```
javascript
```

```
function FileUploadForm() {
  const [file, setFile] = useState(null);
  const [preview, setPreview] = useState(null);

  const handleFileChange = (e) => {
    const selectedFile = e.target.files[0];
    setFile(selectedFile);

    // Create preview for images
    if (selectedFile && selectedFile.type.startsWith('image/')) {
      const reader = new FileReader();
      reader.onloadend = () => setPreview(reader.result);
      reader.readAsDataURL(selectedFile);
    }
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    const formData = new FormData();
    formData.append('file', file);

    await fetch('/api/upload', {
      method: 'POST',
      body: formData
    });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="file" onChange={handleFileChange} />
      {preview && <img src={preview} alt="Preview" />}
      <button type="submit">Upload</button>
    </form>
  );
}
```

99. What are form accessibility best practices?

- Use proper labels and `htmlFor` attributes
- Provide clear error messages
- Use ARIA attributes for screen readers
- Ensure keyboard navigation works
- Use semantic HTML elements
- Provide form instructions

100. How do you test forms in React?

```
javascript
```

```
import { render, screen, fireEvent } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test('should submit form with valid data', async () => {
  const onSubmit = jest.fn();
  render(<MyForm onSubmit={onSubmit} />);

  await userEvent.type(screen.getByLabelText(/email/i), 'test@example.com');
  await userEvent.type(screen.getByLabelText(/password/i), 'password123');

  fireEvent.click(screen.getByRole('button', { name: /submit/i }));

  expect(onSubmit).toHaveBeenCalledWith({
    email: 'test@example.com',
    password: 'password123'
  });
});
```

Testing (10 questions)

101. How do you test React components?

- React Testing Library (recommended)
- Enzyme (legacy)
- Jest for unit tests
- Cypress/Playwright for E2E

102. What is React Testing Library? Library that provides utilities for testing React components by focusing on how users interact with the application.

```javascript
import { render, screen, fireEvent } from '@testing-library/react';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

103. What are testing philosophies?

- Test behavior, not implementation
- Test from the user's perspective
- Write tests that give confidence

104. How do you test hooks?

```javascript
import { renderHook, act } from '@testing-library/react-hooks';

test('should increment counter', () => {
  const { result } = renderHook(() => useCounter());

  act(() => {
    result.current.increment();
  });

  expect(result.current.count).toBe(1);
});
```

105. How do you test components with Context? Create custom render that includes providers:

```javascript
const CustomProvider = ({ children }) => (
  <ThemeProvider value="dark">
    <AuthProvider>
      {children}
    </AuthProvider>
  </ThemeProvider>
);

const renderWithProviders = (ui) => {
  return render(ui, { wrapper: CustomProvider });
};
```

106. How do you mock API calls in tests?

- Jest mocks
- Mock Service Worker (MSW)
- Axios mock adapter

107. What are snapshot tests? Tests that capture the rendered output of a component and compare it with saved snapshots.

108. What are testing best practices?

- Test user behavior, not implementation details
- Use meaningful test descriptions
- Arrange-Act-Assert pattern
- Mock external dependencies
- Test error states

109. How do you test async components?

```javascript
test('loads and displays greeting', async () => {
  render(<Greeting />);

  const greeting = await screen.findByText(/hello/i);
  expect(greeting).toBeInTheDocument();
});
```

110. What is the AAA pattern in testing?

- **Arrange**: Set up the test (render component, setup mocks)
- **Act**: Execute the action (click, type, etc.)
- **Assert**: Verify the result

Advanced React (15 questions)

111. What is Server-Side Rendering (SSR)? Technique where HTML is generated on the server instead of the client, improving SEO and initial performance.

112. What are the benefits of SSR?

- Better SEO
- Faster first contentful paint
- Better experience on slow connections
- Social media sharing (meta tags)

113. What is Static Site Generation (SSG)? Generating HTML pages at build time instead of request time. Faster than SSR for content that doesn't change frequently.

114. What's the difference between SSR, SSG, and CSR?

- **SSR**: Server rendering per request
- **SSG**: Generated at build time
- **CSR**: Client-side rendering

115. What is hydration? Process where React takes server-rendered HTML and makes it interactive by attaching event listeners and state.

116. What are React Server Components? Components that run on the server and allow:

- Direct backend access
- Smaller bundle sizes
- Better performance

117. What is Concurrent Mode? Set of features that help React apps stay responsive and adjust to the user's device capabilities and network speed.

118. What is Time Slicing? Feature of Concurrent Mode that allows React to interrupt rendering work to handle higher priority tasks.

119. What are transitions in React? Way to mark updates as non-urgent, allowing React to interrupt them for more urgent updates.

```javascript
import { startTransition } from 'react';

function handleClick() {
 // Urgent update
 setInputValue(value);

 // Non-urgent update
 startTransition(() => {
  setSearchResults(results);
 });
}
```

120. What is useTransition? Hook that allows marking updates as non-urgent transitions.

```javascript
```

```
function SearchBox() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isPending, startTransition] = useTransition();

  const handleChange = (e) => {
    setQuery(e.target.value);
    startTransition(() => {
      setResults(search(e.target.value));
    });
  };

  return (
    <div>
      <input
        value={query}
        onChange={handleChange}
        placeholder="Search..."
      />
      {isPending && <div>Searching...</div>}
      <ul>
        {results.map(result => (
          <li key={result.id}>{result.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

121. What is useDeferredValue? Hook that allows deferring updates to values that are not critical, similar to debouncing.

```
javascript

function SearchResults({ query }) {
  const deferredQuery = useDeferredValue(query);
  const results = useMemo(() =>
    search(deferredQuery), [deferredQuery]
  );

  return (
    <div>
      {results.map(result => (
        <div key={result.id}>{result.name}</div>
      ))}
    </div>
  );
}
```

122. What is React.forwardRef? Function that allows a component to pass a ref to one of its child components.

```
javascript

const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="fancy-button">
    {props.children}
  </button>
));

// Usage
function App() {
  const ref = useRef();
  return <FancyButton ref={ref}>Click me!</FancyButton>;
}
```

123. What are portals in React? Way to render child components into a DOM node that exists outside the parent component's DOM hierarchy.

```javascript
import ReactDOM from 'react-dom';

function Modal({ children }) {
  return ReactDOM.createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById('modal-root')
  );
}
```

124. When to use portals?

- Modals and overlays
- Tooltips
- Dropdowns
- Notifications
- Any UI that needs to "escape" the parent container

125. What are React best practices?

- **Small, focused components**: Single responsibility
- **Composition over inheritance**: Use composition patterns
- **Avoid prop drilling**: Use Context or state management
- **Performance**: Use React.memo, useMemo, useCallback when needed
- **Naming**: Descriptive names for components and props
- **Error boundaries**: Handle errors gracefully
- **Testing**: Test user behavior, not implementation
- **Accessibility**: Use semantic HTML, ARIA attributes
- **File structure**: Organize by features/domains
- **Code splitting**: Use React.lazy for large components
- **State management**: Local state first, then global if needed
- **Custom hooks**: Extract reusable logic
- **TypeScript**: For better developer experience and fewer bugs

---

Interview Tips and Common Patterns

Questions you must master:

1. **Virtual DOM and Reconciliation**: How it works, why it's useful
2. **Hooks**: useState, useEffect, useContext, custom hooks
3. **Performance**: React.memo, useMemo, useCallback, when to use
4. **State management**: Context API, when to use Redux
5. **Component lifecycle**: With hooks and classes
6. **Testing**: React Testing Library, best practices
7. **Error handling**: Error boundaries, async error handling
8. **Modern features**: Suspense, Concurrent features

During the interview:

- **Explain your thought process** out loud
- **Ask clarifying questions** about requirements

- **Consider edge cases** and mention limitations
- **Talk about performance** when relevant
- **Mention testing** if appropriate
- **Relate to real experience** when possible

Example structured answer:

Question: "How would you optimize a component that renders frequently?"

Structured response:

1. **Identify the cause**: "First, I'd use React DevTools Profiler to identify why it's re-rendering"
2. **Technical solutions**: "Depending on the case, I'd use React.memo if props don't change, useMemo for expensive computations, useCallback for functions passed as props"
3. **Additional considerations**: "I'd also check if the parent is passing objects/arrays inline, and consider splitting state if it's too complex"
4. **Real experience**: "In my last project, I had a table component that rendered constantly, and I optimized it by implementing virtualization with react-window"
5. **Trade-offs**: "It's important to balance optimization with code complexity - not all optimizations are worth it"

This structure shows deep technical knowledge, practical experience, and critical thinking about trade-offs - exactly what interviewers look for.

Remember: Practice makes perfect. Implement these concepts in real projects, don't just memorize them for interviews.