React Study Notes for Interview Prep

After double-checking against the original document, I've added several missing key concepts and refined explanations to make this more comprehensive:

Core Concepts (Episodes 1-3)

React Basics

- **React**: A JavaScript library for building user interfaces using components
- **React DOM**: Connects React to the browser's DOM
- **JSX**: HTML-like syntax within JavaScript that gets converted to React elements
- **Component**: A reusable piece of UI (function that returns JSX)
- **Virtual DOM**: React's lightweight copy of the real DOM for efficient updates
- **Reconciliation**: Process of comparing previous and current Virtual DOM to make minimal real DOM updates
- **React Fiber**: Re-implementation of React's core algorithm for better UI responsiveness

Development Setup

- **CDN**: Content Delivery Network - can load React via script tags but not recommended for projects
- **NPM vs NPX**:
  - NPM: Package manager to install and manage dependencies
  - NPX: Package runner to execute packages without global installation
- **Package.json**: Configuration file that lists project dependencies and scripts
- **Package-lock.json**: Ensures consistent installations across environments
- **Bundlers**: Tools like Parcel/Webpack that combine code files for production
  - Hot Module Replacement (HMR): Updates code without full page refresh
  - **Browserslist**: Configuration for browser compatibility

Component Creation

- Components must start with capital letters
- Components are just JavaScript functions that return JSX
- **Component Composition**: Using components inside other components
- **{} syntax**: Used to embed JavaScript expressions in JSX

Working with Components (Episodes 4-7)

State and Props

- **Props**: Data passed from parent to child components (read-only)
- **State**: Data managed within a component using useState hook
- **useState**: Hook that returns a state value and update function
- const [count, setCount] = useState(0);
- Never create state variables in conditionals, loops, or outside component

Event Handling

- Add event handlers using camelCase attributes
- <button onClick={() => setCount(count + 1)}>Click</button>
- Different ways to handle events:
  - onClick={handleClick} - No arguments
  - onClick={() => handleClick(arg)} - With arguments
  - onClick={handleClick(arg)} - Incorrect, calls immediately

Effects and Lifecycle

- **useEffect**: Hook for handling side effects (API calls, subscriptions)
- useEffect(() => { // Effect code return () => {/* Cleanup code */}}, [dependencies]);
- Empty dependency array ([]) means "run once after initial render"
- No dependency array means "run after every render"
- With dependencies means "run when dependencies change"
- Can't use async directly in useEffect (must define async function inside)

Data Fetching

- Two approaches:
  1. Load → API call → Render
  2. Load → Render skeleton → API call → Re-render with data (preferred)
- Use async/await with useEffect for cleaner API calls
- **Shimmer UI**: Placeholder UI shown while content loads (better than spinners)
- **Optional Chaining**: ?. safely accesses nested properties without errors

Routing and Advanced Components (Episodes 7-8)

React Router

- **createBrowserRouter**: Configures routes
- **RouterProvider**: Applies routing configuration
- **Outlet**: Renders child routes
- **useParams**: Accesses URL parameters
- **createHashRouter**: Uses URL hash for navigation (works without server config)
- **createMemoryRouter**: In-memory router for testing
- Client-side routing doesn't reload the page (unlike server-side)

Class Components

- Traditional way to build React components before hooks
- Need to extend React.Component
- Use this.state and this.setState() for state management
- Lifecycle methods: constructor, render, componentDidMount, etc.
- Always need super(props) in constructor to use this.props
- Execution order: Parent constructor → Parent render → Child constructor → Child render → Child componentDidMount → Parent componentDidMount

Optimization Techniques (Episodes 9-10)

Code Splitting & Lazy Loading

- **lazy()**: Loads components only when needed
- **Suspense**: Shows fallback UI while components load
- const LazyComponent = lazy(() => import('./LazyComponent'));<Suspense fallback={<Shimmer />}> <LazyComponent /></Suspense>
- Benefits: Reduces initial load time, better performance

Custom Hooks

- Extract reusable logic into custom hooks (functions starting with "use")
- Follow single responsibility principle
- function useOnlineStatus() { const [isOnline, setIsOnline] = useState(true); // Logic to detect online status return isOnline;}
- Custom hooks can use other hooks inside them

CSS Approaches

1. **Inline CSS**: Style objects in JavaScript
2. **External CSS**: Import .css files
3. **CSS Modules**: Scoped class names (.module.css)
4. **Styled Components**: CSS-in-JS solution
5. **Tailwind**: Utility-first CSS framework
    - Uses PostCSS with plugins
    - Configuration in tailwind.config.js (content, theme, extend, plugins)

State Management (Episodes 11-12)

Component Communication

- **Lifting State Up**: Moving state to common parent
- **Prop Drilling**: Passing props through many layers (problematic)
- **Controlled Components**: Parent manages child's state through props
- **Uncontrolled Components**: Component manages its own state

Context API

- Provides way to share values without prop drilling
- // Create contextconst MyContext = createContext(defaultValue);// Provide context<MyContext.Provider value={data}> <App /></MyContext.Provider>// Consume contextconst data = useContext(MyContext);
- Good for mid-sized applications, not recommended for large apps

Redux

- **Store**: Central state container
- **Actions**: Objects describing what happened
- **Reducers**: Functions that update state based on actions
- **Slices**: Logical portions of Redux store
- **Dispatch**: Method to send actions
- **Selectors**: Functions to extract data from store

Redux Toolkit flow:

1. Configure store with `configureStore`
2. Create slices with `createSlice` (name, initialState, reducers)
3. Export actions and reducer
4. Use `useSelector()` to read state
5. Use `useDispatch()` to dispatch actions
6. Modern Redux allows state mutation (uses Immer behind scenes)

Testing (Episode 13)

Testing Types

- **Unit Testing**: Testing individual functions/components
- **Integration Testing**: Testing interactions between components
- **End-to-End Testing**: Testing complete user flows

Testing Tools

- **Jest**: Testing framework
- **React Testing Library**: Testing utilities for React
- **JSDOM**: Browser-like environment for tests

Testing Components

```
test('renders header', () => {
  render(<Header />);
  const headingElement = screen.getByRole('heading');
  expect(headingElement).toBeInTheDocument();
});
```

Testing Events

```
test('button click', () => {
  render(<Button />);
  const button = screen.getByRole('button');
  fireEvent.click(button);
  expect(screen.getByText('Clicked')).toBeInTheDocument();
});
```

Mocking

- Mock API calls using Jest mock functions
- Test asynchronous code with act() and await
- Helper functions: beforeAll, afterAll, beforeEach, afterEach
- Group related tests with describe()

Architecture Patterns

Monolithic vs. Microservices

- **Monolithic**: Single, unified codebase with all features
    - Easier initial development but harder to scale
- **Microservices**: Independent services with specific functions
    - More complex setup but better scalability

Performance Hooks (Bonus)

- **useMemo**: Caches calculated values between renders
- const expensiveValue = useMemo(() => computeExpensive(a, b), [a, b]);
- **useCallback**: Caches function definitions between renders
- const handleClick = useCallback(() => doSomething(a, b), [a, b]);
- **useRef**: Stores values that persist between renders without causing re-renders
- const countRef = useRef(0);
- // countRef.current++ doesn't cause re-render

Higher Order Components (HOC)

- Functions that take a component and return an enhanced component
- Used for code reuse, logic abstraction, and cross-cutting concerns
- Pure functions that don't modify the input component

Remember: Be prepared to explain concepts with examples, discuss pros and cons of different approaches, and demonstrate understanding of when to use specific React features.