React is a JavaScript library for building interfaces. Its virtual DOM and component based architecture let it efficiently only send the minimal necessary updates to the DOM.

Components
- The most important concept of React is components. These are the building blocks of React apps.
- Components are reusable, self contained pieces of code that return markup, a description of what should appear on the screen written in JSX that React will turn into actual changes in the DOM when rendering.

JSX
- JSX is a syntax extension that is pretty much HTML, but it lets you write JavaScript code inside the curly braces.
- All tags must be closed
- JS expressions can let you conditionally render elements
- Comments are placed inside curly braces with /* */

Props
- Props are arguments to components.
- They are passed down from parent to child, and are read only, creating a one way data flow. Props are a way to pass data, not change it.
- Any JavaScript value can be passed as props, including objects or functions.
- “Prop drilling” is when you need to pass props through several levels of child components to get to the one you want. This is cumbersome and is often a sign you should use a Context API.

Context API
- Context API is React’s solution to state variables that need to be accessed by many components at different nesting levels without prop drilling. It is a react hook useContext.
- The ideal use case is on data that should be considered ‘global’ for a tree of components, eg language, current user.
- Advantages are avoiding prop drilling, centralizing state variables, easier refactoring without having to rewire prop chains.

State
- State is the internal data within a component that can change over time.
- Each component is responsible for its own state.
- The ‘useState’ hook is the main way to add state variables. Eg: const [count, setCount] = useState(0);
- State updates trigger rerenders.
- State updates are asynchronous. If the state update relies on previous state value, use the functional update form.
Eg: setCount(prevCount => prevCount + 1);
- Multiple ‘useState’ hooks can track multiple variables in the same component.

- When multiple components need to access the same state, it should live in the closest common ancestor, or use a Context API.

Hooks
- Hooks are functions that let you "hook into" React features like state and lifecycle methods.
- Hooks should only be called at the top level. Not in conditionals, loops, or nested functions.
- Only call them from React components or custom hooks.
- Hooks and only hooks should start with ‘use’.
- ‘useState’ is the simplest way to store state variables in components
- ‘useEffect’ executes code when dependencies change.
- ‘useContext’ lets you manage state globally

Async/Await
- A JavaScript feature that React components can use to handle asynchronous operations
- Without async/await, you'd use promises with .then() chains
- Has better readability, uses familiar try/catch blocks, has a more linear execution flow thats easier to debug.
- Chaining then() can make more sense for sequential or parallel operations.

Virtual DOM
- The Virtual DOM is a lightweight JavaScript representation of the actual DOM (Document Object Model) in the browser.
- It's essentially a tree of JavaScript objects that mirrors the structure of your HTML elements.
- When your React app first loads, React creates a virtual representation of the entire UI.
- When state or props change in your component React does this:
  * creates a new Virtual DOM tree
  * compares this new tree with the previous one (diffing)
  * calculates the minimal set of changes needed to update the real DOM
  * applies only those specific changes to the actual browser DOM

Common qustions

What are the core principles of React?
- Virtual DOM: React makes a lightweight copy of the actual DOM (Document Object Model) in memory. When state changes occur, React updates the virtual DOM, and only updates the necessary parts of the actual DOM.
- Component-based: React uses reusable, self-contained pieces of UI that manage their own state and behavior.
- Declarative: Instead of manipulating the DOM directly, you say what you want to happen based on the current state. React handles all the DOM stuff to match what you said. For example, instead of writing imperative code to show/hide elements, you simply declare {isVisible && <Component />}.

How does React handle state?

- React apps are made up of components, and state is an object that holds data about a component. This data can change over time.
- Components are all responsible for their own state, and can manage their state variable using React hooks like useState.
- This data can be shared with other components as props, but will be read only and form a one-way parent child relationship.
- Whenever state changes React automatically rerenders affected components.

What's the difference between state and props?
- Props are like read only args passed from parent to child
- State is mutable and managed internally within components, gets passed down to props
- Changes to state cause a rerender

Could you explain the React component lifecycle?
- Mounting: when a component is first created and added to the DOM, uses UseEffect()
- Updating: when state or props update, causing a rerender
- Unmounting: when a component is removed from the DOM
- State initialization - mounting
- Data fetching - mounting or updating

What are hooks, what are the rules, and why do they exist?
- React hooks are functions that let you 'hook' into core React features from function components.
- Only Call Hooks at the Top Level - React relies on the hooks being called in the same order every render to maintain state correctly
- Only Call Hooks from React Functions - Hooks are designed to work with React's component lifecycle and state management system. They need the context that React components provide.
- Custom hooks should start with 'use' - its a good, consistent naming convention and is used by linters.

What is UseEffect()?
- useEffect is a React Hook that lets you perform side effects in function components.
- you can make it trigger every frame, just on mount, or when specific values change

Other important React hooks:
useContext - Subscribes to React context without nesting
    * Enables sharing values between components without prop drilling
    * Great for global state like themes, user data, or localization
useReducer - Alternative to useState for complex state logic
    * Inspired by Redux's pattern for predictable state updates
    * Better for when state logic involves multiple sub-values or when next state depends on previous
useRef - Creates a mutable reference that persists across renders
    * Access DOM elements directly

    * Store values that don't trigger re-renders when changed

useMemo - Memoizes expensive calculations between renders

    * Only recalculates when dependencies change

useCallback - Returns a memoized callback function

    * Prevents unnecessary renders in child components that rely on callback functions

useLayoutEffect - Similar to useEffect but fires synchronously after DOM mutations

    * Generally, prefer useEffect unless you specifically need synchronous execution