Mern Stack Interview prep

React

What is React and why do we use it?

React is a JavaScript library for building user interfaces. It's mainly used for creating single-page applications because it updates the UI dynamically without reloading the entire page. We use React because it provides a component-based structure, uses a virtual DOM for efficient rendering, and helps build scalable, fast, and maintainable applications

What are components in React? Difference between class and function components?

Components in React are reusable building blocks of UI. They can be written as either class components or function components. Class components are ES6 classes that can hold state and lifecycle methods, while function components are plain JavaScript functions that take props and, with hooks, can also manage state and side effects. In modern React, function components are preferred because they are simpler and more concise.

Explain JSX. Why can't the browser read JSX directly?

JSX is a syntax extension in React that lets us write HTML-like code inside JavaScript. Browsers can't read JSX directly because they only understand plain JavaScript, so JSX must be transpiled (by Babel) into `React.createElement` calls before execution.

What are props and state? Difference between them.

Props are plain JavaScript objects that are passed from a parent component to a child component as inputs. The child uses them to render the UI. Props are immutable, meaning the child cannot change them — they are read-only.
State, on the other hand, is an internal object managed inside a component. It holds data that can change over time, and whenever state changes, React re-renders the component to update the UI.

What are controlled vs uncontrolled components?

A controlled component is an input element (like text field, checkbox, etc.) whose value is controlled by React state. Whenever the user types something, React updates the state, and the state becomes the single source of truth. So React is in charge.
An uncontrolled component is when the input manages its own state inside the DOM, and React doesn't control the value. We usually access it with a ref when needed.

What are keys in React? Why are they important in lists?

Keys in React are unique identifiers you give to elements in a list when rendering them. They help React identify which items have changed, been added, or removed, so it can update the UI efficiently without re-rendering the entire list.

What is reconciliation in React?

Reconciliation is the process React uses to update the real DOM efficiently. When a component's state or props change, React creates a new Virtual DOM and compares it with the previous Virtual DOM using the diffing algorithm. It then updates only the parts that actually changed in the real DOM, making rendering fast and efficient.

What are React hooks? Why were they introduced?

React Hooks are special functions that let you use state and other React features (like lifecycle methods) inside function components. They were introduced to make function components more powerful and avoid using class components for state or side effects, making code simpler and easier to manage

Difference between useState and useReducer . When would you use which?

useState:

- `useState` is a React hook that lets you store and update **simple state values** in a component (like numbers, strings, or small objects).
- Example use case: a counter, a toggle button, or a form input.
- Easy to use when the state logic is simple.

  useReducer:
- `useReducer` is a React hook used for **complex state logic** or when the next state depends on the previous state.
- It works like Redux — you define a **reducer function** to handle actions and update state.

Explain useEffect . What are dependencies?

useEffect:

- `useEffect` is a React hook used to handle **side effects** in function components.
- Side effects are operations that happen **outside the normal render cycle**, like fetching data from an API, updating the DOM manually, or setting up subscriptions.
- `useEffect` lets you perform these actions at the right time in the component's lifecycle.

  Dependencies:
- The **dependency array** controls when the effect runs:
  1. **No dependency array:** Effect runs **after every render**.
  2. **Empty array (`[]`):** Effect runs **only once**, after the component mounts.

3. **Array with variables (`[var1, var2]`):** Effect runs **first on mount**, and **every time the variables change**.

How to prevent infinite loops in `useEffect`?

Answer:
"To prevent infinite loops, always include a dependency array.

- Empty array (`[]`) → runs only once on mount.
- Include only the necessary variables → runs when those change.
  Without a dependency array, the effect runs after every render, causing an infinite loop."

What is useRef and where is it used?

useRef:
useRef is a React hook that lets you create a reference to a DOM element or a value that persists across renders. Unlike state, updating a ref ***doesn't trigger a re-render****. It's commonly used for directly accessing or manipulating DOM elements, storing mutable values, or keeping previous values between renders."*
Quick examples of use:

- Accessing input fields (`ref.current.value`)
- Storing timers or socket connections
- Keeping track of previous state without re-rendering

Explain custom hooks. Have you written any in your projects?

Custom hooks are reusable functions in React that let you extract and share logic between components. They always start with `use`, and inside them you can use other hooks like `useState`, `useEffect`, etc. I've written custom hooks in my projects to handle things like API calls, form handling, and local storage, so the code is cleaner and reusable.

> ⓘ **Custom hooks are reusable hooks I create to share logic across components.**

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

// Custom Hook
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(prev => prev + 1);
  const decrement = () => setCount(prev => prev - 1);
  const reset = () => setCount(initialValue);

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

```
}
export default useCounter;
```

uses

```
import React from "react";
import useCounter from "./useCounter";

function CounterComponent() {
  const { count, increment, decrement, reset } = useCounter(5);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>+1</button>
      <button onClick={decrement}>-1</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
export default CounterComponent;
```

What causes a React component to re-render?

Answer:
A React component re-renders mainly in three cases:

1. **State changes** inside the component.
2. **Props change** that are passed from the parent.
3. **Parent re-renders**, which can cause child components to re-render as well."

How do you prevent unnecessary re-renders?

Unnecessary re-renders can be prevented using `React.memo`, which memorizes a component and prevents it from re-rendering if its props haven't changed. Additionally, we can use `useCallback` and `useMemo` to memoize functions and computed values passed to child components, avoiding re-renders caused by changing references.

What is code splitting? How do you implement it in React?

Code splitting is a technique to split your React application into smaller chunks so the browser only loads what is needed for the initial render. This improves performance by reducing the bundle size. In React, we implement code splitting using `React.lazy()` for dynamic imports and `Suspense` to show a fallback while the component loads.

What is lazy loading in React?

Lazy loading is a technique to load parts of a React application only when they are needed, instead of loading the entire app at once. This improves performance by reducing the initial bundle size. In React, we use `React.lazy()` and `Suspense` to implement lazy loading for components.

How would you optimize a React app's performance?

Answer:
I optimize a React app by:

1. **Preventing unnecessary re-renders** using `React.memo`, `useMemo`, and `useCallback`.
2. **Lazy loading components** with `React.lazy` and `Suspense` so only needed parts load initially.
3. **Optimizing long lists** with unique `keys` and techniques like windowing (e.g., `react-window`).
4. **Splitting code** into smaller chunks to reduce initial bundle size and improve load time."

How do you manage state in React?

In React, state can be managed at different levels:

1. Component-level state:** Using `useState` or `useReducer` for local state inside a component.
2. Shared state across components: **Using** props** to pass data from parent to child.
3. Global state:Using `Context API` for small to medium apps, or **Redux / Zustand** for larger apps to manage state across the application efficiently.

What is Context API? When would you use it?

Context API is a React feature that lets you share data across components without prop drilling. Instead of passing props through multiple nested components, you can provide the data at a global level and consume it anywhere using `useContext`.
I usually use it in small to medium apps for things like themes, user info, or authentication state.

Explanation of Redux Flow

1. Purpose:

Redux (and Redux Toolkit) is used for **global state management** in React applications, especially when state needs to be shared across multiple components. Redux Toolkit simplifies setup by reducing boilerplate.

---

2. Core Concepts:

1. **Store**
   - The **single source of truth** for your application state.
   - Created once, usually in a `store.js` file.
   - Holds the state tree for the app.
2. **Slice** (Redux Toolkit)
   - A "slice" is a piece of state along with its reducers and actions.
   - You define initial state, reducers (functions that update state), and actions in one place.
   - Example: `userSlice`, `cartSlice`, etc.
3. **Action**
   - An object that describes **what happened**.
   - Usually has a `type` and optional `payload`.
   - In Redux Toolkit, you don't manually define `type`; `createSlice` auto-generates actions.
4. **Reducer**
   - A function that takes the **current state and an action**, and returns the **new state**.
   - Redux Toolkit lets you write reducers that **mutate state directly** using Immer internally.
5. **Dispatch**
   - Used to **send an action** to the store.
   - Example: `dispatch(addItem(item))`.
6. **Selector**
   - A function to **read state** from the store.
   - Example: `useSelector(state => state.cart.items)`.

Flow Summary in Words (Interview-ready)

1. The **store** holds the app's global state.
2. A **slice** defines a part of the state, its actions, and reducers.
3. You **dispatch actions** to tell Redux what to change.
4. The **reducer** updates the state based on the action.
5. Components use **useSelector** to read state and **useDispatch** to send actions.

28. Difference between Redux and Context API

Both are used for state management, but the difference is mainly in scale and features:

Feature	Context API	Redux / Redux Toolkit
Scope	Small to medium apps	Medium to large apps

Feature	Context API	Redux / Redux Toolkit
Boilerplate	Minimal	More structured, needs slices/actions/store
Performance	Can cause unnecessary re-renders if not optimized	Optimized with reducers and predictable state flow
State Updates	Directly with `useContext`	Through actions and reducers, more predictable
Best Use	Themes, user info, auth state	Complex apps with multiple state slices and async logic"*

29. What are actions, reducers, and store in Redux

Redux has three main building blocks:

1. Action: A plain JavaScript object that describes what happened (type + optional payload).
2. Reducer: A function that takes the current state and an action and returns the new state.
3. Store: The single source of truth that holds the global state and allows components to dispatch actions and subscribe to changes."*

30. What is middleware in Redux? Example of custom middleware

Middleware in Redux is a function that sits between dispatching an action and the moment it reaches the reducer. It's used for logging, handling async operations, or modifying actions.

Example:** Custom logging middleware

```
const loggerMiddleware = store => next => action => {
console.log('Dispatching:', action);   let result = next(action);
console.log('Next state:', store.getState());   return result; };
```

Here, every time an action is dispatched, the middleware logs the action and the updated state before passing it to the reducer.

How do you fetch data in React?

There are a few common ways to fetch data in React

1. Fetch API: The built-in browser API to make HTTP requests.
2. Axios: A popular library for making HTTP requests with simpler syntax and better features like interceptors.
3. React Query: A library that manages server state, caching, background updates, and simplifies fetching, caching, and synchronizing data with the UI.

Have you used React Query? What are its benefits?

Yes, I have used React Query. It's very helpful for managing server-side state efficiently. React Query automatically caches server responses, avoids unnecessary network requests, keeps data in sync, and provides features like background updates, pagination, and retries. This makes fetching and managing data much easier and more performant

Difference between React Query and Redux

Redux and React Query serve different purposes:

- **Redux:** Used for managing **global client-side state**. It helps keep state consistent across components and is ideal for things like UI state, user info, or app settings.
- **React Query:** Focused on **server-side state**. It caches API responses, reduces network calls, keeps data in sync, and handles background updates, pagination, and retries.

What is the difference between CSR, SSR, and SSG?

The main differences are about **where and when the HTML is generated**:

1. **CSR (Client-Side Rendering):**
   - The browser downloads JavaScript and renders the UI on the client side.
   - Example: React apps.
   - Pros: Fast navigation after initial load, rich interactivity.
   - Cons: Slower initial load, SEO challenges.
2. **SSR (Server-Side Rendering):**
   - The server generates the full HTML for a page on each request and sends it to the client.
   - Example: Next.js.
   - Pros: Faster first-page load, better SEO.
   - Cons: Higher server load, slower page transitions compared to CSR.
3. **SSG (Static Site Generation):**
   - HTML pages are generated **at build time**, not on each request.
   - Example: Next.js static export.
   - Pros: Extremely fast, good SEO, minimal server load.
   - Cons: Content cannot change on every request unless revalidated.

What is hydration in React (Next.js)?

Hydration is the process where React attaches event listeners and makes a **server-rendered HTML page interactive on the client side**. In Next.js, SSR or SSG generates HTML on the server, and React hydrates it on the client to make it dynamic.

What are error boundaries? How do you create one?

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree and display a fallback UI instead of crashing the whole app.
You create one by defining a class component with `static getDerivedStateFromError()` and `componentDidCatch()` methods.

What is React Fiber?

React Fiber is the reimplementation of React's core rendering engine to enable incremental rendering, better handling of animations, gestures, and concurrency. Fiber breaks rendering work into small units, allowing React to pause, resume, or prioritize updates efficiently.

How does React handle concurrency? (React 18 concurrent features)

React 18 introduces concurrent rendering, which allows React to work on multiple tasks at once, pause low-priority updates, and keep the UI responsive. Features include automatic batching, `startTransition`, `Suspense` for data fetching, and interruptible rendering.

Difference between controlled vs uncontrolled forms

Controlled components have their form values managed by React state, meaning the value of inputs is tied to state and updated via `onChange`.
Uncontrolled components manage their own internal state in the DOM, and React reads the value using refs when needed.

How do you handle form validation in React?

1. Controlled components: Validate values stored in React state on events like `onChange` or `onSubmit`.
2. Libraries: Use libraries like Formik or React Hook Form to simplify validation, including synchronous and asynchronous rules.

Validation logic can include required fields, pattern matching, min/max length, or custom rules.

Difference between `onChange` and `onInput` **

- `onChange`: Triggered after the value of an input changes and loses focus in HTML; in React, it triggers on every value change.
- `onInput`: Triggered immediately whenever the value changes, even while typing.

Suppose a list of 10,000 items is rendering slowly — how do you optimize it?

Rendering 10,000 items at once can be slow. To optimize:

1. Use unique `key` props for list items so React can efficiently track changes.

2. **Implement pagination or infinite scrolling** to render only a small subset of items at a time.
3. **Virtualize the list** using libraries like `react-window` or `react-virtualized` to render only visible items in the viewport.
4. **Memoize components** with `React.memo` if list items are complex to avoid unnecessary re-renders.

How do you handle authentication in React?

Authentication in React is usually handled by combining frontend and backend:

1. The user logs in through a form.
2. The backend validates credentials and issues a **token** (JWT).
3. The frontend stores the token securely (preferably in **HTTP-only cookies**) and uses it to access protected routes or APIs.
4. For route protection, use **private routes** or route guards to restrict access to authenticated users.

How do you secure tokens in React (JWT, HTTP-only cookies)?

Tokens should **never be stored in localStorage or sessionStorage** because they are vulnerable to XSS attacks. The recommended approach:

1. **HTTP-only cookies:** Tokens are stored in cookies inaccessible to JavaScript, making them more secure.
2. **Secure and SameSite flags:** Prevent CSRF attacks.
3. **Backend validation:** Every API call checks the token to ensure the user is authenticated.

How do you implement role-based access control (RBAC) in React?

RBAC restricts access based on user roles. Implementation steps:

1. **Store user role** after login (from JWT or backend).
2. **Protect routes/components** by checking the role before rendering.
3. **Conditional rendering:** Show or hide UI elements based on role.