Frontend Developer Interview Preparation Guide

1. Self-introduction (Projects worked on)

2. How to set up a React.js project?

3. What software do you use for React Hooks? What are their use cases? When do you use them?

4. What are the main use cases for `React.memo`?

5. Besides the above, have you used anything else? Please describe in detail.

6. What React-related software have you used?

7. Have you encountered any problems using React?

8. When encountering these problems (No. 7), what

methods do you use to optimize?

---

9. What are cookies and code storage? Have you used them?

---

10. What is the difference between `localStorage` and `sessionStorage`?

---

11. What are the results of opening a new window using `localStorage` and `sessionStorage` respectively?

---

12. Do you know about sessions? (Backend-biased)

---

13. Are you familiar with frontend performance optimization?

---

14. If you were to implement TIV centering (center on screen horizontally and vertically), how would you do it? How would you draw a triangle?

---

15. What are common HTTP request status codes?

(301, 304, 401, 404, 504, 502, ...)

---

16. What are common array methods?

---

17. What is the difference between `for each`, `array`, and `some`? How would you generate a random number from 0 to 100?

---

18. How many people are currently on the frontend development team? How many people are in the entire team?

---

19. Has your previously developed web page been integrated into an APP?

---

20. How do you collaborate with other colleagues? Is there a specific process from data requirements to development? How do you interface, etc.?

---

21. Do you have a concept of versioning? How often

are versions released? Is it fixed or unfixed?

22. How do you collaborate with the backend team?

23. In your previous projects/tasks, which one gave you a sense of accomplishment? What problems did you encounter at that time?

24. If you were to redo a project you've worked on, which part do you think could be done better?

25. Do you frequently use GitHub?

26. What are your hobbies and interests?

27. What are your strengths and weaknesses? How do your friends evaluate you?

2. How to set up a React.js project?

Setting up a React.js project can be done in a few ways, depending on your needs and preferences. The official React documentation recommends starting with a framework for new applications, as these frameworks provide integrated features for deployment, scaling, and leverage React's architecture. Examples of such frameworks include Next.js, React Router, and Expo.

However, if you have specific constraints not met by existing frameworks, prefer to build your own, or simply want to understand the fundamentals, you can set up a

React app from scratch using build tools. Popular build tools for this purpose include Vite, Parcel, and RSbuild.

Using a Framework (Recommended for new apps):

Frameworks like Next.js, React Router, and Expo offer a more opinionated and complete solution for building React applications. They often come with pre-configured routing, data fetching, and server-side rendering capabilities. For instance, to start a new Next.js project, you would typically use a command like:

```
npx create-next-app@latest
```

Building from Scratch with Build Tools (e.g., Vite):

For a more flexible setup or to learn the basics, build tools are an excellent choice. Vite is a modern build tool that has gained popularity for its speed and simplicity. It provides a fast development server and optimizes your code for production. To set up a React project with Vite, you would generally follow these steps:

1. **Install Node.js and npm (or Yarn/pnpm):** Ensure you have Node.js installed on your system, which includes npm (Node Package Manager). You can download it from the official Node.js website [1].

2. **Create a new Vite project:** Open your terminal or command prompt and run the following command:

   `bash npm create vite@latest`

   This command will prompt you to name your project, select a framework (choose `react`), and then choose whether to use JavaScript or TypeScript.

3. **Navigate to your project directory:**

   `bash cd your-project-name`

4. **Install dependencies:**

   `bash npm install`

5. **Start the development server:**

   `bash npm run dev`

This will start a local development server, and you can view your React application in your browser, usually at `http://localhost:5173` (or a similar port).

Explanation of Technical Terms:

- **Framework:** In software development, a framework is a standardized, reusable set of libraries, tools, and conventions that provides a foundation for building applications. It dictates the overall structure and flow of an application, offering pre-built components and solutions for common tasks. This helps developers build applications more efficiently and consistently.

- **Build Tool:** A build tool automates the process of converting source code and other assets into a deployable application. For frontend development, build tools like Vite, Webpack, and Parcel handle tasks such as bundling JavaScript and CSS files, transpiling modern JavaScript (e.g., JSX, ES6+) into browser-compatible code, optimizing assets, and running development servers.

- **Node.js:** An open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser. It's commonly used for building backend services, command-line tools, and for managing frontend development workflows (e.g., running npm).

- **npm (Node Package Manager):** The default package manager for Node.js. It's used to install, manage, and share JavaScript libraries and tools. When you run `npm install`, it reads the `package.json` file in your project and downloads all the listed dependencies.

- **`npx`:** A command-line tool that comes with npm (since npm 5.2). It allows you to execute Node.js package executables directly without having to install them globally. This is particularly useful for running one-off commands or scaffolding new projects, like `create-next-app` or `create-vite`.

- **Development Server:** A local web server that serves your application files during development. It typically includes features like hot module replacement (HMR), which automatically updates your browser with code changes without requiring a full page refresh, significantly speeding up the development process.

References:

[1] Node.js Official Website: https://nodejs.org/en/download

3. What software do you use for React Hooks? What are their use cases? When do you use them?

React Hooks are not a separate software or library that you install; they are built-in functions provided by the React library itself, introduced in React 16.8. They allow you to use state and other React features in functional components, eliminating the need for class components in many scenarios. This makes component logic more reusable, readable, and easier to test.

Here are some of the most commonly used built-in React Hooks, their use cases, and when to use them:

`useState`

- **Purpose:** `useState` is a Hook that lets you add React state to function components. State is data that changes over time and affects how a component renders.
- **Use Cases:**
    - Managing form input values.
    - Toggling UI elements (e.g., showing/hiding a modal, expanding/collapsing a section).
    - Storing data fetched from an API that needs to be displayed and updated in the UI.
- **When to Use:** Use `useState` whenever your component needs to remember some data that might change and cause the component to re-render. It returns a pair: the current state value and a function that lets you update it.
- **Example:**

  ```jsx import React, { useState } from 'react';

  function Counter() { const [count, setCount] = useState(0); // Initialize count to 0

  return (

  You clicked {count} times

  setCount(count + 1)}> Click me
  ```

); } ```

`useEffect`

- **Purpose:** `useEffect` is a Hook that lets you perform side effects in function components. Side effects are operations that interact with the outside world, such as data fetching, subscriptions, or manually changing the DOM.
- **Use Cases:**
  - Fetching data from an API when a component mounts or when certain dependencies change.
  - Setting up event listeners (e.g., for window resize, scroll) and cleaning them up when the component unmounts.
  - Directly manipulating the DOM (e.g., changing the document title).
  - Setting up subscriptions (e.g., to a WebSocket) and cleaning them up.
- **When to Use:** Use `useEffect` when your component needs to do something after rendering that doesn't directly affect the render output, or when it needs to interact with external systems. It runs after every render by default, but you can control when it re-runs by providing a dependency array.
- **Example:**

  ```jsx import React, { useState, useEffect } from 'react';

  function TitleUpdater() { const [count, setCount] = useState(0);

  useEffect(() => { // Update the document title using the browser API document.title = `You clicked ${count} times`;

  ```
  // Cleanup function (optional): runs when the component unmounts or
  before the effect re-runs
  return () => {
    console.log('Cleanup effect');
  };
  ```

  }, [count]); // Re-run the effect only if count changes

  return (

  You clicked {count} times
  ```

setCount(count + 1)}> Click me

); } ```

`useContext`

- **Purpose:** `useContext` is a Hook that lets you read and subscribe to a React context from your component. Context provides a way to pass data through the component tree without having to pass props down manually at every level.
- **Use Cases:**
  - Sharing global data like theme (light/dark mode), user authentication status, or preferred language across many components.
  - Avoiding

prop-drilling (passing props through many intermediate components that don't directly use them). * **When to Use:** Use `useContext` when you have data that needs to be accessible by many components at different nesting levels, and you want to avoid explicitly passing that data down as props. * **Example:**

~~~
```jsx
import React, { useContext, createContext } from 'react';

// Create a Context
const ThemeContext = createContext("light");

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

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

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme === "dark" ? "#333" : "#eee", color:
theme === "dark" ? "white" : "black" }}>
      I am a {theme} button
    </button>
  );
}
```
~~~

`useRef`

- **Purpose:** `useRef` is a Hook that lets you reference a value that’ s not needed for rendering. It returns a mutable ref object whose `.current` property is initialized to the passed argument (`initialValue`). The returned object will persist for the full lifetime of the component.
- **Use Cases:**
  - Accessing and interacting with DOM elements directly (e.g., focusing an input, playing/pausing media).
  - Storing a mutable value that doesn’ t trigger a re-render when it changes (e.g., a timer ID, a previous state value).
- **When to Use:** Use `useRef` when you need a persistent, mutable value that doesn't cause re-renders when updated, or when you need to directly interact with a DOM element.

- **Example:**

  ```jsx import React, { useRef } from 'react';

  function TextInputWithFocusButton() { const inputEl = useRef(null); const onButtonClick = () => { // `current` points to the mounted text input element inputEl.current.focus(); }; return ( <> Focus the input </> ); } ```

`useCallback`

- **Purpose:** `useCallback` is a Hook that lets you cache a function definition between re-renders. It returns a memoized callback function. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary re-renders.
- **Use Cases:**
  - Optimizing performance by preventing unnecessary re-creation of callback functions, especially when passing them as props to child components that are memoized (e.g., with `React.memo`).
  - When a function is a dependency of another Hook (like `useEffect` or `useMemo`) and you want to prevent that Hook from re-running unnecessarily.
- **When to Use:** Use `useCallback` when you have a function that is being passed down to a child component, and that child component is optimized (e.g., using `React.memo`) to prevent re-renders if its props haven't changed. It helps maintain referential equality of the function across renders.
- **Example:**

  ```jsx import React, { useState, useCallback } from 'react';

  function ParentComponent() { const [count, setCount] = useState(0); const [text, setText] = useState('');

  // This function will only be re-created if `count` changes const handleClick = useCallback(() => { setCount(count + 1); }, [count]);

  return (

{text} setText(e.target.value)} />

Count: {count}

); }

// ChildComponent is memoized, so it only re-renders if its props change const ChildComponent = React.memo(({ onClick }) => { console.log("ChildComponent rendered"); return Increment Count; }); ```

## `useMemo`

- **Purpose:** `useMemo` is a Hook that lets you cache the result of a calculation between re-renders. It returns a memoized value. This is useful for optimizing performance by avoiding expensive calculations on every render.
- **Use Cases:**
    - Performing expensive calculations (e.g., filtering a large list, complex data transformations) and memoizing the result so it's only re-calculated when its dependencies change.
    - Optimizing child components by ensuring that complex objects or arrays passed as props don't cause unnecessary re-renders due to new object references on every render.
- **When to Use:** Use `useMemo` when you have a computationally expensive function whose result doesn't need to be re-calculated on every render, but only when its dependencies change. It helps prevent unnecessary re-renders and improves performance.
- **Example:**

  ```jsx import React, { useState, useMemo } from 'react';

  function ExpensiveCalculationComponent() { const [count, setCount] = useState(0); const [multiplier, setMultiplier] = useState(2);

  // This calculation will only re-run if `count` or `multiplier` changes const expensiveResult = useMemo(() => { console.log("Performing expensive calculation..."); return count * multiplier; }, [count, multiplier]);

  return (
  ```

Count: {count}

setCount(count + 1)}>Increment Count

Multiplier: {multiplier}

setMultiplier(multiplier + 1)}>Increment Multiplier

Result of expensive calculation: {expensiveResult}

); } ` ` `

Explanation of Technical Terms:

- **State:** In React, state refers to data that a component can manage and that can change over time. When state changes, React re-renders the component and its children to reflect the updated data.

- **Side Effects:** Operations that interact with the outside world and are not directly related to rendering the UI. Examples include data fetching, subscriptions, manually changing the DOM, timers, and logging.

- **DOM (Document Object Model):** A programming interface for web documents. It represents the page structure as a tree of objects, allowing programs to change the document structure, style, and content.

- **Prop-drilling:** The process of passing data (props) from a parent component down to a deeply nested child component, through intermediate components that don't actually need the data themselves. This can make code harder to maintain and understand.

- **Context:** A React feature that allows you to share values (like themes or user authentication status) that are considered

“global” for a tree of React components, without having to explicitly pass the prop through every level of the tree.

- **Memoization:** An optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. In React, `useMemo` and `useCallback` are forms of memoization.

- **Referential Equality:** In JavaScript, two objects or functions are referentially equal if they point to the exact same location in memory. When comparing props in React, if a prop is an object or a function, React will re-render the component if the reference changes, even if the content of the object/function is the same. Memoization helps maintain referential equality for functions and objects to prevent unnecessary re-renders.

9. What are cookies and code storage? Have you used them?

Cookies

Definition: An HTTP cookie (also known as a web cookie, browser cookie, or simply cookie) is a small piece of data that a server sends to a user's web browser. The browser may store it and send it back with the next request to the same server. Cookies are essentially small text files stored on your computer or device by websites you visit.

Purpose and Use Cases:

Cookies are primarily used for three main purposes:

1. **Session Management:** They allow websites to remember information about a user's activity on the site, such as items added to a shopping cart, login status, or game scores. This enables a seamless browsing experience without requiring users to re-authenticate or re-enter information on every page.
2. **Personalization:** Websites can use cookies to remember user preferences, such as language settings, theme choices (dark/light mode), or customized content. This helps in providing a more tailored and user-friendly experience.
3. **Tracking and Analytics:** Cookies are widely used to track user behavior across websites. This data is then used for analytics (understanding website traffic and user engagement) and targeted advertising. For example, if you view a product on an e-commerce site, you might later see ads for that product on other websites.

How they work: When you visit a website, the server sends a `Set-Cookie` header in its HTTP response. Your browser then stores this cookie. On subsequent requests to the same server, the browser automatically includes the cookie in the `Cookie` header of its HTTP request. The server can then read the cookie to identify the user or retrieve their preferences.

Have I used them? Yes, as a frontend developer, I have interacted with cookies, primarily for:

- **Authentication:** Storing session tokens or user IDs to maintain login state across different pages of an application.
- **User Preferences:** Saving user-specific settings like language or display preferences.
- **Tracking (indirectly):** While I haven't directly implemented complex tracking systems, I've worked on applications that utilize third-party analytics tools (like Google Analytics) which rely on cookies for data collection.

Code Storage (Web Storage API)

Definition: "Code storage" in the context of web development typically refers to the Web Storage API, which provides mechanisms for web applications to store data locally within the user's browser. This API includes `localStorage` and `sessionStorage`.

Purpose and Use Cases:

The Web Storage API offers a more robust and larger-capacity alternative to cookies for client-side data storage. It's designed for storing key-value pairs.

- **`localStorage`:** Stores data with no expiration date. The data persists even when the browser window is closed and reopened. It's available across all windows and tabs from the same origin.
- **`sessionStorage`:** Stores data for the duration of a single browser session. The data is cleared when the browser tab or window is closed. It's specific to the tab it was created in and is not shared across different tabs or windows.

How they work: Both `localStorage` and `sessionStorage` are accessible through the `Window` object in JavaScript. They provide simple methods like `setItem()`, `getItem()`, `removeItem()`, and `clear()` to manage data.

Have I used them? Yes, I have extensively used both `localStorage` and `sessionStorage` in my frontend development work:

- **`localStorage` Use Cases:**
    - **Persisting User Preferences:** Storing user settings (e.g., theme, layout preferences) that should remain even after the user closes and reopens the browser.
    - **Caching Data:** Storing frequently accessed but not highly sensitive data (e.g., product lists, application configuration) to reduce API calls and improve performance.
    - **Offline Capabilities:** Storing data for progressive web applications (PWAs) to enable some functionality even when offline.
- **`sessionStorage` Use Cases:**
    - **Temporary Session Data:** Storing data that is only relevant for the current browsing session, such as form data that needs to be preserved across page navigations within the same tab, or temporary user input before submission.
    - **Shopping Cart Data:** For a shopping cart that should be cleared when the user closes the tab.
    - **Multi-step Forms:** Storing data for each step of a multi-step form, ensuring data is not lost if the user navigates back and forth within the form.

10. What is the difference between `localStorage` and `sessionStorage`?

Both `localStorage` and `sessionStorage` are part of the Web Storage API and provide a way to store key-value pairs in the client's browser. They share a similar API (`setItem`, `getItem`, `removeItem`, `clear`), but their primary differences lie in their **lifespan** and **scope**.

Here's a table summarizing their key distinctions:

Feature	`localStorage`	`sessionStorage`
Lifespan	Persistent; data remains until explicitly deleted by the user, by the application, or through browser settings. It has no expiration date.	Session-based; data is cleared when the browser tab or window is closed. It is designed for the duration of a single browsing session.
Scope	Data is accessible across all windows and tabs from the same origin (same protocol, host, and port).	Data is isolated to the specific tab or window where it was created. It is not shared across different tabs or windows, even if they are from the same origin.
Capacity	Typically 5MB to 10MB per origin.	Typically 5MB to 10MB per origin.
Accessibility	Accessible via `window.localStorage`	Accessible via `window.sessionStorage`
Security	Less secure for sensitive data as it persists and can be accessed by any script on the same origin.	More secure for temporary sensitive data as it is cleared on tab/window close.

Key Differences Explained:

- **Persistence:** The most significant difference is persistence. `localStorage` is designed for long-term storage, making it suitable for user preferences, cached application data, or offline capabilities. `sessionStorage`, on the other hand, is for temporary data that should only exist for the current user session within a specific tab.
- **Scope:** `localStorage` is shared across all open tabs and windows of the same origin. If you set an item in `localStorage` in one tab, you can access it in another tab from the same website. `sessionStorage` is strictly confined to the tab in which it was created. If you open a new tab to the same website, it will have its own independent `sessionStorage`.

11. What are the results of opening a new window using `localStorage` and `sessionStorage` respectively?

Let's consider the behavior of `localStorage` and `sessionStorage` when a new window or tab is opened:

When opening a new window/tab with `localStorage`:

If you have data stored in `localStorage` in one browser tab/window and then open a new tab or window to the **same origin** (same protocol, host, and port), the data stored in `localStorage` will be **accessible and shared** across both. This is because `localStorage` is persistent and scoped to the origin, not to individual tabs or sessions.

Example Scenario:

1. You open `https://example.com` in Tab A.
2. You execute `localStorage.setItem('username', 'JohnDoe');` in Tab A.
3. You open `https://example.com` in Tab B (a new tab or window).
4. In Tab B, if you execute `localStorage.getItem('username');`, it will successfully retrieve `'JohnDoe'`.

This behavior makes `localStorage` suitable for data that needs to persist across multiple browsing sessions and be available globally within the same application origin.

When opening a new window/tab with `sessionStorage`:

If you have data stored in `sessionStorage` in one browser tab/window and then open a new tab or window to the **same origin**, the data stored in `sessionStorage` will **not be accessible or shared** in the new tab/window. Each new tab or window gets its own independent `sessionStorage` object.

Important Note: There's a specific edge case: if you duplicate a tab (e.g., right-click on a tab and select

13. Are you familiar with frontend performance optimization?

Yes, I am familiar with frontend performance optimization. It is a crucial aspect of web development that focuses on improving the speed and responsiveness of web applications from the user's perspective. The goal is to deliver content to the user as quickly as possible, provide a smooth and interactive experience, and ultimately enhance user satisfaction and engagement.

Why is Frontend Performance Optimization Important?

- **User Experience (UX):** Faster loading times and smoother interactions lead to a better user experience. Users are more likely to stay on a site that responds quickly and less likely to abandon a slow-loading page.
- **Search Engine Optimization (SEO):** Search engines like Google consider page speed as a ranking factor. Faster websites tend to rank higher in search results, leading to increased visibility and organic traffic.
- **Conversion Rates:** For e-commerce sites or applications with specific goals (e.g., sign-ups, purchases), improved performance often translates to higher conversion rates.
- **Accessibility:** Optimized websites are generally more accessible, especially for users on slower networks or with older devices.
- **Cost Savings:** Efficiently loaded and rendered pages can reduce server load and bandwidth consumption, leading to lower hosting costs.

Common Frontend Performance Optimization Techniques:

Frontend performance optimization involves a wide range of techniques that can be broadly categorized into several areas:

1. **Code Optimization (HTML, CSS, JavaScript):**

    - **Minification:** Removing unnecessary characters (whitespace, comments, line breaks) from HTML, CSS, and JavaScript files without changing their functionality. This reduces file size and download time.
    - **Compression (Gzip/Brotli):** Using compression algorithms (like Gzip or Brotli) on web servers to reduce the size of text-based assets before sending

them to the browser. The browser then decompresses them.

- **Tree Shaking:** A form of dead code elimination that removes unused code from JavaScript bundles. This is particularly effective in modern JavaScript applications built with module bundlers like Webpack or Rollup.
- **Code Splitting:** Dividing JavaScript bundles into smaller chunks that can be loaded on demand. This reduces the initial load time by only loading the code necessary for the current view.
- **Efficient CSS:** Writing concise and efficient CSS, avoiding overly complex selectors, and minimizing layout recalculations.
- **Optimizing JavaScript Execution:** Avoiding long-running JavaScript tasks that block the main thread, using web workers for intensive computations, and debouncing/throttling event handlers.

2. **Image and Media Optimization:**

   - **Image Compression:** Compressing images to reduce their file size without significant loss of quality. Tools like TinyPNG or ImageOptim can be used.
   - **Responsive Images:** Serving different image sizes based on the user's device and screen resolution using `srcset` and `sizes` attributes in HTML, or CSS media queries.
   - **Lazy Loading Images and Videos:** Deferring the loading of off-screen images and videos until they are about to enter the viewport. This reduces initial page load time and bandwidth usage.
   - **Using Modern Image Formats:** Utilizing modern image formats like WebP or AVIF, which offer better compression and quality compared to older formats like JPEG or PNG.

3. **Network Optimization:**

   - **Reducing HTTP Requests:** Minimizing the number of requests a browser needs to make to render a page. This can be achieved through techniques like CSS sprites, combining small files, and inlining critical CSS/JS.
   - **Browser Caching:** Leveraging browser caching to store static assets (images, CSS, JS) locally on the user's device. This means that on subsequent visits, the browser can load these assets from cache instead of re-downloading them from the server.

- **Content Delivery Networks (CDNs):** Using CDNs to serve static assets from servers geographically closer to the user. This reduces latency and speeds up content delivery.
- **HTTP/2 and HTTP/3:** Utilizing newer HTTP protocols that offer features like multiplexing (allowing multiple requests/responses over a single connection) and server push, which can significantly improve loading performance.
- **Preloading, Preconnecting, Prefetching:** Using `<link rel=

"preload">`, `, and` ` to optimize resource loading by telling the browser to fetch or connect to resources earlier.

1. **Rendering Performance:**
   - **Optimizing Critical Rendering Path:** Prioritizing the loading and rendering of content that is immediately visible to the user (above-the-fold content). This involves inlining critical CSS and JavaScript.
   - **Avoiding Layout Thrashing:** Minimizing forced synchronous layouts and style recalculations by batching DOM read and write operations.
   - **Using `requestAnimationFrame`:** For animations and visual updates, using `requestAnimationFrame` ensures that changes are performed just before the browser's next repaint, leading to smoother animations.
2. **Performance Monitoring and Tools:**
   - **Browser Developer Tools:** Using browser developer tools (e.g., Chrome DevTools, Firefox Developer Tools) to analyze network requests, performance timelines, and identify bottlenecks.
   - **Lighthouse:** An open-source, automated tool for improving the quality of web pages. It audits performance, accessibility, progressive web apps, SEO, and more.
   - **Web Vitals:** A set of standardized metrics (Largest Contentful Paint, First Input Delay, Cumulative Layout Shift) that aim to quantify the user experience of a web page. Optimizing for these metrics is crucial for good performance.

My Approach to Optimization:

When approaching frontend performance optimization, I typically follow a systematic process:

1. **Identify Bottlenecks:** Use tools like Lighthouse, WebPageTest, and browser developer tools to identify the slowest parts of the application and understand where the most significant performance gains can be made.
2. **Prioritize:** Focus on optimizing the critical rendering path first to ensure a fast initial load. Then, address other areas like image optimization, code splitting, and caching.
3. **Implement and Measure:** Apply optimization techniques and continuously measure the impact of changes. It's important to have a baseline and track improvements over time.
4. **Automate:** Integrate performance checks and optimizations into the build process (e.g., using Webpack plugins for minification, compression, and image optimization) to ensure consistent performance.
5. **Educate:** Share best practices with the team to foster a performance-aware development culture.

By applying these techniques and maintaining a performance-first mindset, I aim to deliver fast, responsive, and enjoyable web experiences for users.

14. If you were to implement TIV centering (screen top-bottom-left-right centering), how would you do it? How would you draw a triangle?

TIV Centering (Centering an Element Horizontally and Vertically)

"TIV centering" refers to the common task of centering an element both horizontally and vertically within its parent container or the viewport. There are several robust and widely used methods to achieve this in CSS, each with its own advantages and ideal use cases. The most modern and flexible approaches involve Flexbox and CSS Grid.

Here are the primary methods:

1. Using Flexbox (Recommended for single-item centering)

Flexbox is a one-dimensional layout system that makes it easy to align and distribute space among items in a container. It's excellent for centering a single item or a group of items.

How to do it:

Apply `display: flex`, `justify-content: center`, and `align-items: center` to the parent container of the element you want to center.

CSS:

```css
.parent-container {
  display: flex;
  justify-content: center; /* Centers horizontally */
  align-items: center;    /* Centers vertically */
  height: 100vh; /* Example: Centers in the full viewport height */
  width: 100vw;  /* Example: Centers in the full viewport width */
  /* Or set a fixed height/width for the parent */
}

.centered-element {
  /* No specific styles needed for the child for centering */
}
```

HTML:

```html
<div class="parent-container">
  <div class="centered-element">
    This element is centered.
  </div>
</div>
```

Explanation:

- `display: flex;`: Turns the parent into a flex container.
- `justify-content: center;`: Aligns flex items along the main axis (horizontally by default) in the center.
- `align-items: center;`: Aligns flex items along the cross axis (vertically by default) in the center.

2. Using CSS Grid (Recommended for more complex layouts or multiple items)

CSS Grid is a two-dimensional layout system that offers even more powerful alignment capabilities. It's particularly useful when you need to center items within a grid structure or when dealing with multiple items.

How to do it:

Apply `display: grid`, `place-items: center` to the parent container.

CSS:

```css
 .parent-container {
   display: grid;
   place-items: center; /* Centers both horizontally and vertically */
   height: 100vh;
   width: 100vw;
}

.centered-element {
  /* No specific styles needed for the child for centering */
}
```

HTML:

```html
 <div class="parent-container">
   <div class="centered-element">
     This element is centered.
   </div>
</div>
```

Explanation:

- `display: grid;`: Turns the parent into a grid container.
- `place-items: center;`: This is a shorthand for `align-items: center` and `justify-items: center`. It centers grid items both vertically and horizontally within their grid area.

3. Using `position: absolute` with `transform` (Legacy but still useful)

This method is older but still effective, especially when you need to center an element that is absolutely positioned.

How to do it:

Set the child element to `position: absolute`, `top: 50%`, `left: 50%`, and then use `transform: translate(-50%, -50%)` to adjust its position based on its own size.

CSS:

```
.parent-container {
  position: relative; /* Parent must have a defined position */
  height: 100vh;
  width: 100vw;
}

.centered-element {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%); /* Adjusts for the element's own size */
}
```

HTML:

```
<div class="parent-container">
  <div class="centered-element">
    This element is centered.
  </div>
</div>
```

Explanation:

- `position: relative;` on the parent establishes a positioning context for the absolutely positioned child.
- `position: absolute;` on the child takes it out of the normal document flow.
- `top: 50%;` and `left: 50%;` move the top-left corner of the element to the exact center of the parent.
- `transform: translate(-50%, -50%);` then shifts the element back by half of its own width and half of its own height, effectively centering it perfectly.

Drawing a Triangle with CSS

Drawing a triangle purely with CSS is a classic technique that leverages the way borders behave when elements have zero width and height. When an element has no content and its width and height are set to 0, its borders meet at angles. By making some borders transparent and giving others a color, you can create the illusion of a triangle.

How to do it:

Create an element with `width: 0` and `height: 0`. Then, apply borders to it, making three of the four borders transparent and giving the remaining border a color. The direction of the colored border will determine the direction of the triangle.

Example: Upward-pointing Triangle

CSS:

```css
.triangle-up {
  width: 0;
  height: 0;
  border-left: 50px solid transparent; /* Left side of the triangle */
  border-right: 50px solid transparent; /* Right side of the triangle */
  border-bottom: 100px solid blue; /* Base of the triangle (colored) */
}
```

HTML:

```html
<div class="triangle-up"></div>
```

Explanation:

- `width: 0; height: 0;`: This creates a point in the center of where the element would normally be.
- `border-left: 50px solid transparent;` and `border-right: 50px solid transparent;`: These create two transparent triangular halves that meet at the top point. The `50px` determines half the base width of the triangle.
- `border-bottom: 100px solid blue;`: This creates the visible base of the triangle. The `100px` determines the height of the triangle. Because the left and right borders are transparent, only the bottom border is visible, forming an upward-pointing triangle.

Variations for other directions:

- **Downward-pointing Triangle:** Make `border-top` colored and `border-left` / `border-right` transparent.
- **Leftward-pointing Triangle:** Make `border-right` colored and `border-top` / `border-bottom` transparent.

- **Rightward-pointing Triangle:** Make `border-left` colored and `border-top` / `border-bottom` transparent.

Technical Terms Explained:

- **Viewport:** The visible area of a web page in the browser window. `100vh` refers to 100% of the viewport height, and `100vw` refers to 100% of the viewport width.
- **Flexbox (Flexible Box Layout):** A CSS3 layout module that provides an efficient way to lay out, align, and distribute space among items in a container, even when their size is unknown or dynamic. It's a one-dimensional layout system.
- **CSS Grid Layout:** A CSS3 layout module that offers a two-dimensional grid-based layout system, capable of handling both columns and rows. It's ideal for designing major page areas or small user interface elements.
- **`position: absolute;`:** A CSS positioning property that removes an element from the normal document flow and positions it relative to its closest positioned ancestor (or the initial containing block if no such ancestor exists). This allows precise placement using `top`, `right`, `bottom`, and `left` properties.
- **`transform: translate();`:** A CSS function that allows you to move an element from its current position. `translate(-50%, -50%)` moves the element up by 50% of its own height and left by 50% of its own width, which is key for perfect centering when combined with `top: 50%; left: 50%;`.
- **Borders:** In CSS, borders are lines that surround an element. When an element has zero width and height, its borders meet at sharp angles, forming triangular shapes. By selectively coloring and making borders transparent, you can create visible triangles.

15. What are common HTTP request status codes? (301, 304, 401, 404, 504, 502, ...)

HTTP (Hypertext Transfer Protocol) status codes are three-digit numbers returned by a server in response to a client's HTTP request. They indicate whether a specific HTTP request has been successfully completed. Status codes are grouped into five classes, each representing a different type of response:

- **1xx Informational responses:** The request was received, continuing process.

- **2xx Success:** The request was successfully received, understood, and accepted.
- **3xx Redirection:** Further action needs to be taken in order to complete the request.
- **4xx Client errors:** The request contains bad syntax or cannot be fulfilled.
- **5xx Server errors:** The server failed to fulfill an apparently valid request.

Here are some common HTTP status codes, including those you specifically mentioned:

200 OK

- **Meaning:** The request has succeeded. This is the most common status code for a successful HTTP request.
- **Use Case:** A web page loaded successfully, an API call returned data as expected.

301 Moved Permanently

- **Meaning:** The requested resource has been permanently moved to a new URL. The client should use the new URL for all future requests.
- **Use Case:** When a website changes its domain name, a page's URL changes permanently, or when consolidating duplicate content. It's crucial for SEO as it passes link equity to the new URL.

302 Found (Previously "Moved Temporarily")

- **Meaning:** The requested resource has been temporarily moved to a different URL. The client should continue to use the original URL for future requests.
- **Use Case:** During website maintenance, A/B testing, or when a temporary redirect is needed for a specific campaign. Unlike 301, it does not pass link equity.

304 Not Modified

- **Meaning:** The requested resource has not been modified since the version specified by the request headers (`If-Modified-Since` or `If-None-Match`). The client can use its cached copy of the resource.

- **Use Case:** Used for caching. When a browser requests a resource it already has in its cache, it sends a conditional request. If the server determines the resource hasn't changed, it sends a 304, telling the browser to use its local cached version, saving bandwidth and speeding up page load.

400 Bad Request

- **Meaning:** The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).
- **Use Case:** Invalid input in a form, missing required parameters in an API request, or a corrupted request body.

401 Unauthorized

- **Meaning:** The request has not been applied because it lacks valid authentication credentials for the target resource. The client needs to authenticate itself to get the requested response.
- **Use Case:** Attempting to access a protected resource without being logged in, or with invalid login credentials (e.g., wrong username/password).

403 Forbidden

- **Meaning:** The client does not have access rights to the content; i.e., it is unauthorized, so the server is refusing to give the requested resource. Unlike 401, authentication will not help.
- **Use Case:** A logged-in user tries to access content or functionality they don't have permission for (e.g., a regular user trying to access an admin panel).

404 Not Found

- **Meaning:** The server cannot find the requested resource. This is one of the most common errors encountered on the web.
- **Use Case:** A user tries to access a page that doesn't exist, a broken link, or a mistyped URL.

500 Internal Server Error

- **Meaning:** The server encountered an unexpected condition that prevented it from fulfilling the request. This is a generic error message when no more specific message is suitable.
- **Use Case:** A bug in the server-side application code, a misconfigured server, or a database error.

502 Bad Gateway

- **Meaning:** The server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.
- **Use Case:** Often seen when a web server (like Nginx or Apache) is acting as a reverse proxy for an application server (like Node.js or Python Flask), and the application server crashes or returns an invalid response.

503 Service Unavailable

- **Meaning:** The server is not ready to handle the request. Common causes are a server that is down for maintenance or overloaded.
- **Use Case:** During scheduled server maintenance, or when a server is experiencing very high traffic and cannot handle new requests.

504 Gateway Timeout

- **Meaning:** The server, while acting as a gateway or proxy, did not receive a timely response from an upstream server it needed to access to complete the request.
- **Use Case:** Similar to 502, but specifically indicates a timeout. This can happen if the upstream server is too slow to respond, or if there are network issues between the proxy and the upstream server.

Understanding these status codes is essential for debugging web applications, optimizing performance, and ensuring a good user experience.

16. What are common array methods?

JavaScript arrays come with a rich set of built-in methods that facilitate various operations, from adding/removing elements to transforming and iterating over data. These methods make working with collections of data much more efficient and readable compared to traditional `for` loops.

Here are some of the most common and frequently used array methods:

- **`push()`**: Adds one or more elements to the end of an array and returns the new length of the array. `javascript const fruits = ["apple", "banana"]; fruits.push("orange"); // fruits is now ["apple", "banana", "orange"]`
- **`pop()`**: Removes the last element from an array and returns that element. `javascript const fruits = ["apple", "banana", "orange"]; const lastFruit = fruits.pop(); // lastFruit is "orange", fruits is now ["apple", "banana"]`
- **`shift()`**: Removes the first element from an array and returns that element. `javascript const fruits = ["apple", "banana", "orange"]; const firstFruit = fruits.shift(); // firstFruit is "apple", fruits is now ["banana", "orange"]`
- **`unshift()`**: Adds one or more elements to the beginning of an array and returns the new length of the array. `javascript const fruits = ["banana", "orange"]; fruits.unshift("grape"); // fruits is now ["grape", "banana", "orange"]`
- **`concat()`**: Used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. `javascript const arr1 = [1, 2]; const arr2 = [3, 4]; const newArr = arr1.concat(arr2); // newArr is [1, 2, 3, 4]`
- **`slice()`**: Returns a shallow copy of a portion of an array into a new array object selected from `start` to `end` (end not included). The original array will not be modified. `javascript const animals = ["ant", "bison", "camel", "duck", "elephant"]; const slicedAnimals = animals.slice(2, 4); // slicedAnimals is ["camel", "duck"]`

- **`splice()`:** Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. `javascript const months = ["Jan", "Feb", "March", "April"]; months.splice(2, 0, "May"); // Inserts "May" at index 2. months is ["Jan", "Feb", "May", "March", "April"] months.splice(3, 1); // Removes 1 element at index 3. months is ["Jan", "Feb", "May", "April"]`

- **`indexOf()`:** Returns the first index at which a given element can be found in the array, or -1 if it is not present. `javascript const numbers = [1, 2, 3, 2]; const index = numbers.indexOf(2); // index is 1`

- **`includes()`:** Determines whether an array includes a certain value among its entries, returning `true` or `false` as appropriate. `javascript const numbers = [1, 2, 3]; const hasTwo = numbers.includes(2); // hasTwo is true`

- **`find()`:** Returns the value of the first element in the provided array that satisfies the provided testing function. Otherwise, `undefined` is returned. `javascript const ages = [3, 10, 18, 20]; const adult = ages.find(age => age >= 18); // adult is 18`

- **`findIndex()`:** Returns the index of the first element in the array that satisfies the provided testing function. Otherwise, -1 is returned. `javascript const ages = [3, 10, 18, 20]; const adultIndex = ages.findIndex(age => age >= 18); // adultIndex is 2`

- **`sort()`:** Sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values. ```javascript const numbers = [3, 1, 4, 1, 5, 9]; numbers.sort(); // numbers is [1, 1, 3, 4, 5, 9] (for numbers, often needs a compare function)

  const words = ["banana", "apple", "cherry"]; words.sort(); // words is ["apple", "banana", "cherry"] ```

- **`reverse()`:** Reverses an array in place. The first array element becomes the last, and the last array element becomes the first. `javascript const numbers = [1, 2, 3]; numbers.reverse(); // numbers is [3, 2, 1]`

17. What is the difference between `for each` and `array` and `some`? How would you generate a random number from 0 to 100?

Let's clarify the terms first. "`array`" refers to the data structure itself, while `forEach` and `some` are methods that operate on arrays. It seems the question might be implicitly asking about `forEach`, `map`, `filter`, `reduce`, and `some` as common array iteration/transformation methods, and then contrasting them with traditional `for` loops.

`Array` (Data Structure)

An `Array` in JavaScript is a global object that is used in the construction of array-like objects. Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. They are ordered collections of values, where each value is called an element, and each element has a numerical index.

```
const myArray = [1, "hello", true, { key: "value" }];
```

`forEach()`

- **Purpose:** The `forEach()` method executes a provided function once for each array element. It is primarily used for iterating over an array and performing a side effect (e.g., logging to the console, modifying an external variable).
- **Return Value:** `forEach()` does not return a new array; it always returns `undefined`. It is not chainable.
- **Mutability:** It does not modify the original array unless the callback function explicitly mutates it.
- **Use Case:** When you need to iterate over an array and perform an action for each element, but you don't need to create a new array or accumulate a single result.

  `javascript const numbers = [1, 2, 3]; numbers.forEach(number => { console.log(number * 2); // Outputs: 2, 4, 6 }); // numbers remains [1, 2, 3]`

`map()`

- **Purpose:** The `map()` method creates a **new array** populated with the results of calling a provided function on every element in the calling array.
- **Return Value:** It returns a new array of the same length as the original, with each element being the result of the callback function.
- **Mutability:** It does not modify the original array.
- **Use Case:** When you need to transform each element in an array into a new value and collect these new values into a new array.

  `javascript const numbers = [1, 2, 3]; const doubledNumbers = numbers.map(number => number * 2); // doubledNumbers is [2, 4, 6] // numbers remains [1, 2, 3]`

`filter()`

- **Purpose:** The `filter()` method creates a **new array** with all elements that pass the test implemented by the provided function.
- **Return Value:** It returns a new array containing only the elements for which the callback function returned `true`. If no elements pass the test, an empty array is returned.
- **Mutability:** It does not modify the original array.
- **Use Case:** When you need to select a subset of elements from an array based on a certain condition.

  `javascript const numbers = [1, 2, 3, 4, 5]; const evenNumbers = numbers.filter(number => number % 2 === 0); // evenNumbers is [2, 4] // numbers remains [1, 2, 3, 4, 5]`

`reduce()`

- **Purpose:** The `reduce()` method executes a reducer callback function on each element of the array, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

- **Return Value:** It returns a single value (which can be a number, string, object, or another array).
- **Mutability:** It does not modify the original array.
- **Use Case:** When you need to compute a single value from an array, such as a sum, average, or to flatten an array of arrays, or to group objects by a property.

  ```
  javascript const numbers = [1, 2, 3, 4, 5]; const sum =
  numbers.reduce((accumulator, currentValue) => accumulator +
  currentValue, 0); // sum is 15 // numbers remains [1, 2, 3, 4, 5]
  ```

`some()`

- **Purpose:** The `some()` method tests whether at least one element in the array passes the test implemented by the provided function.
- **Return Value:** It returns `true` if the callback function returns a truthy value for at least one element in the array. Otherwise, it returns `false`.
- **Mutability:** It does not modify the original array.
- **Use Case:** When you need to check if any element in an array satisfies a certain condition, and you only care about whether at least one does, not which one.

  ```
  javascript const numbers = [1, 2, 3, 4, 5]; const hasEven =
  numbers.some(number => number % 2 === 0); // hasEven is true const
  hasSix = numbers.some(number => number === 6); // hasSix is false //
  numbers remains [1, 2, 3, 4, 5]
  ```

How to generate a random number from 0 to 100?

To generate a random integer between 0 and 100 (inclusive) in JavaScript, you can use `Math.random()` in combination with `Math.floor()`.

- **`Math.random()`:** Returns a floating-point, pseudo-random number in the range `[0, 1)` (inclusive of 0, but not 1). This means it can be 0, but never quite 1.
- **`Math.floor()`:** Returns the largest integer less than or equal to a given number.

Formula:

To get a random integer between `min` (inclusive) and `max` (inclusive), the formula is:

`Math.floor(Math.random() * (max - min + 1)) + min;`

For a random number from 0 to 100:

- `min = 0`
- `max = 100`

So the formula becomes:

`Math.floor(Math.random() * (100 - 0 + 1)) + 0;` `Math.floor(Math.random() * 101);`

Example:

```
 function getRandomNumber0To100() {
   return Math.floor(Math.random() * 101);
}

const randomNumber = getRandomNumber0To100();
console.log(randomNumber); // Outputs a random integer between 0 and 100
```

Explanation:

1. `Math.random()` generates a number like `0.12345`, `0.98765`, etc.
2. Multiplying by `101` (`max - min + 1`) scales this range. So, `Math.random() * 101` will produce a number in the range `[0, 101)` (e.g., `0.0` to `100.999...`).
3. `Math.floor()` then rounds this number down to the nearest whole integer. This ensures that `0` is included, and `100` is the maximum possible integer, as `100.999...` becomes `100`.

4. What are the main use cases for `React.memo`?

`React.memo` is a higher-order component (HOC) in React that is used for performance optimization. It memoizes a functional component, meaning it prevents the component from re-rendering if its props have not changed. This can be particularly useful in applications with many components, or components that render frequently with the same props, to avoid unnecessary re-renders and improve overall application performance.

How `React.memo` works:

When a component is wrapped with `React.memo`, React will perform a shallow comparison of its props with the previous props. If the shallow comparison determines that the props are the same, React skips rendering the component and reuses the last rendered result. If the props are different, the component will re-render.

By default, `React.memo` performs a shallow comparison of props. For more complex comparisons (e.g., deep comparison of objects or arrays), you can provide a custom comparison function as the second argument to `React.memo`.

Main Use Cases for `React.memo`:

`React.memo` is most effective in the following scenarios:

1. **Pure Functional Components:** Components that always render the same output given the same props and state. If a component is "pure," its output is solely determined by its props, and it doesn't have any side effects that would change its output without a prop change.

2. **Components with Expensive Renders:** If a component's rendering logic is computationally intensive or involves rendering a large number of child components, memoizing it can significantly reduce the performance overhead.

3. **Components that Re-render Frequently:** In applications where certain components re-render very often (e.g., due to parent component state changes or context updates), but their own props don't change frequently, `React.memo` can prevent these unnecessary re-renders.

4. **Components Receiving Stable Props:** When a component receives props that are guaranteed to be stable (e.g., primitive values, or memoized functions/objects using `useCallback` or `useMemo`), `React.memo` can effectively prevent re-renders.

    - **Example with `useCallback`:** If a parent component passes a callback function to a child component, and that child component is wrapped with `React.memo`, the child will re-render every time the parent re-renders because functions are re-created on every render, leading to a new reference. By wrapping the callback with `useCallback`, you ensure the function reference remains stable, allowing `React.memo` to work effectively.

```jsx import React, { useState, useCallback, memo } from 'react';

// Memoized child component const MyButton = memo(({ onClick, label }) => { console.log(`MyButton (${label}) rendered`); return {label}; });

function ParentComponent() { const [count, setCount] = useState(0); const [text, setText] = useState('');

// Memoize the handleClick function using useCallback const handleClick = useCallback(() => { setCount(prevCount => prevCount + 1); }, []); // Empty dependency array means this function is created only once

return (

Count: {count}

{text} setText(e.target.value)} placeholder="Type something..." />

); } `` In this example, `MyButton` will only re-render when its `onClick` or `label` props actually change. Since `handleClick` is memoized with `useCallback` (and its dependencies are stable), `MyButton` won't re-render when `text` state changes in `ParentComponent`.

**When NOT to use `React.memo`:**

While `React.memo` can be beneficial, it's not a silver bullet and should be used judiciously. Overusing it can sometimes lead to more overhead than benefit. Avoid `React.memo` if:

- **Components Re-render Infrequently:** If a component rarely re-renders, the overhead of the prop comparison might outweigh the benefits of skipping a render.
- **Components Have Constantly Changing Props:** If a component's props are always changing (e.g., a component displaying real-time data), `React.memo` will always find differences and re-render, adding unnecessary comparison overhead.
- **Components Have Simple Renders:** If a component's render logic is very simple and fast, the performance gain from memoization might be negligible.
- **Components with Complex Props without Custom Comparison:** If a component receives complex objects or arrays as props, and you don't provide a
```

custom comparison function, the shallow comparison will often lead to unnecessary re-renders because object references change on every render, even if their content is the same.

In summary, `React.memo` is a powerful tool for optimizing React applications by preventing unnecessary re-renders of functional components. It's most effective when applied to pure components that re-render frequently with stable props or have expensive rendering logic. However, it's important to profile your application and identify actual performance bottlenecks before applying `React.memo` to ensure it provides a real benefit.

5. Besides the above, have you used anything else? Please describe in detail.

Beyond the core React Hooks (`useState`, `useEffect`, `useContext`, `useRef`, `useCallback`, `useMemo`) and `React.memo` for optimization, I have experience with several other important React concepts and related libraries that are crucial for building robust and scalable frontend applications.

1. State Management Libraries (e.g., Redux, Zustand, React Query)

While `useState` and `useContext` are sufficient for local and simple global state, complex applications often benefit from dedicated state management solutions. I have experience with:

- **Redux (with Redux Toolkit):** Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. Redux Toolkit is the official, opinionated, batteries-included toolset for efficient Redux development. It simplifies common Redux patterns and reduces boilerplate.
    - **Use Case:** Managing complex global application state, such as user authentication, shopping cart data, or data fetched from multiple API endpoints that needs to be accessible across many components. It provides a centralized store, making state changes predictable and debuggable.
    - **How I use it:** I typically use Redux Toolkit to define slices (reducers, actions, and initial state), configure the store, and use `useSelector` to read state

    and `useDispatch` to dispatch actions from React components. This approach streamlines state management, especially for large applications.

- **Zustand:** A small, fast, and scalable bearbones state-management solution using simplified flux principles. It's known for its simplicity and minimal boilerplate.
    - **Use Case:** For applications where Redux might be overkill, but a global state management solution is still needed. It's great for smaller to medium-sized applications or specific features within a larger app where a lightweight solution is preferred.
    - **How I use it:** I define stores using `create` function, and then use the custom hook generated by Zustand to access and update the state directly in components. Its simplicity makes it very quick to set up and use.
- **React Query (TanStack Query):** While not strictly a state management library in the traditional sense, React Query is a powerful data-fetching library that manages server state. It handles caching, background re-fetching, data synchronization, and error handling out of the box.
    - **Use Case:** Managing asynchronous data (server state) in React applications. It's ideal for scenarios involving frequent data fetching, mutations, and complex caching requirements. It significantly reduces the amount of boilerplate code needed for data fetching and provides a much better developer experience compared to `useEffect` for data fetching.
    - **How I use it:** I use `useQuery` for fetching data, `useMutation` for sending data to the server, and its powerful caching mechanisms to ensure data freshness and optimal performance. It integrates seamlessly with React components and provides hooks for various data-fetching lifecycle events.

2. React Router

- **Purpose:** React Router is a collection of navigational components that compose declaratively with your application. It enables client-side routing in React applications, allowing users to navigate between different views or pages without full page reloads.
- **Use Case:** Building Single Page Applications (SPAs) where different URLs correspond to different components or views. It handles URL synchronization, nested routes, and programmatic navigation.

- **How I use it:** I use components like `BrowserRouter`, `Routes`, and `Route` to define the application's routing structure. Hooks like `useNavigate` for programmatic navigation, `useParams` for accessing URL parameters, and `useLocation` for getting the current URL location are frequently used.

3. Form Libraries (e.g., React Hook Form, Formik)

Managing forms in React can become complex, especially with validation, submission, and state management. I have experience with:

- **React Hook Form:** A performant, flexible, and extensible forms library with easy-to-use validation. It leverages uncontrolled components and refs to reduce re-renders and improve performance.
    - **Use Case:** Building forms of any complexity, from simple login forms to multi-step forms with intricate validation rules. Its focus on performance and minimal re-renders makes it a strong choice.
    - **How I use it:** I use the `useForm` hook to register inputs, handle submission, and manage validation errors. It integrates well with UI libraries and provides a clean API for form management.

4. Component Libraries/UI Frameworks (e.g., Material-UI, Ant Design, Chakra UI)

- **Purpose:** These libraries provide pre-built, customizable UI components that adhere to design systems (like Material Design or Ant Design principles). They accelerate development by offering ready-to-use buttons, inputs, cards, navigation elements, etc.
- **Use Case:** Rapidly building consistent and aesthetically pleasing user interfaces without having to style every element from scratch. They ensure design consistency and often come with accessibility features built-in.
- **How I use them:** I integrate these libraries into projects to quickly assemble UIs. I customize components using their theming capabilities to match the project's brand guidelines. This significantly speeds up the UI development process.

5. Testing Libraries (e.g., React Testing Library, Jest)

- **Purpose:** Essential for ensuring the quality and reliability of React applications. Jest is a JavaScript testing framework, and React Testing Library is a set of utilities that helps you test React components in a way that resembles how users interact with your application.
- **Use Case:** Writing unit and integration tests for React components and application logic. React Testing Library encourages testing components by querying the DOM in a way that a user would, promoting more robust and maintainable tests.
- **How I use them:** I write tests to cover component rendering, user interactions, and state updates. Jest is used for running tests and assertions, while React Testing Library provides methods to render components and simulate user events.

6. CSS-in-JS Libraries (e.g., Styled Components, Emotion)

- **Purpose:** These libraries allow you to write CSS directly within your JavaScript components, enabling dynamic styling based on component props or state, and providing scoped styles to prevent conflicts.
- **Use Case:** When you want to colocate your styles with your components, leverage JavaScript for dynamic styling, and ensure that styles are encapsulated and don't leak globally. They are particularly useful in component-driven development.
- **How I use them:** I define styled components using template literals, passing props to dynamically change styles. This approach enhances component reusability and maintainability by keeping styles tightly coupled with the components they affect.

My experience with these tools and concepts allows me to choose the right solution for different project requirements, ensuring that the application is not only functional but also performant, maintainable, and scalable.

6. What React-related software have you used?

When working with React, "software" can refer to a few different categories: the development environment, build tools, libraries/frameworks built on top of React, and testing utilities. Here's a breakdown of the React-related software and tools I commonly use:

1. Development Environment & Tools:

- **Node.js and npm/Yarn/pnpm:** These are fundamental for any modern JavaScript development, including React. Node.js provides the runtime environment, and npm (Node Package Manager), Yarn, or pnpm are used for managing project dependencies and running scripts.
- **Code Editor (e.g., VS Code):** My primary code editor is Visual Studio Code. It offers excellent support for React development with features like:
    - **Syntax Highlighting:** For JSX, JavaScript, TypeScript, and CSS.
    - **IntelliSense:** Autocompletion and smart suggestions for React components, props, and methods.
    - **ESLint and Prettier Integration:** For code linting and formatting, ensuring code quality and consistency across the team.
    - **Debugger:** Built-in debugging capabilities for React applications.
    - **Extensions:** A vast ecosystem of extensions for React development (e.g., React Developer Tools, ES7 React/Redux/GraphQL/React-Native snippets).
- **Browser Developer Tools:** Essential for debugging, inspecting components, monitoring network requests, and analyzing performance. The React Developer Tools extension for Chrome/Firefox is particularly invaluable for inspecting React component trees, props, state, and performance.

2. Build Tools & Bundlers:

- **Vite:** As mentioned earlier, Vite is my preferred build tool for starting new React projects due to its speed and simplicity. It uses native ES modules for development, leading to very fast hot module replacement (HMR).
- **Webpack (and Babel):** While I prefer Vite for new projects, I have extensive experience with Webpack, especially in older or larger enterprise projects.

Webpack is a powerful module bundler that processes your application's assets (JavaScript, CSS, images) into optimized bundles for deployment. Babel is often used alongside Webpack to transpile modern JavaScript (like JSX and ES6+) into browser-compatible JavaScript.

3. React Libraries & Frameworks (beyond core React):

- **React Router:** For handling client-side routing in Single Page Applications (SPAs), allowing navigation between different views without full page reloads.
- **State Management Libraries:**
  - **Redux Toolkit:** For predictable and scalable global state management in larger applications.
  - **Zustand:** For lightweight and fast global state management in smaller to medium-sized applications.
  - **React Query (TanStack Query):** For managing server state, including data fetching, caching, and synchronization.
- **Form Libraries:**
  - **React Hook Form:** For efficient and performant form management and validation.
- **UI Component Libraries:**
  - **Material-UI (MUI):** A popular React UI framework that implements Google's Material Design. It provides a comprehensive set of pre-built, customizable components.
  - **Ant Design:** Another enterprise-class UI design system and React UI library, offering a rich set of components and a strong design philosophy.
  - **Chakra UI:** A simple, modular, and accessible component library that gives you the building blocks you need to build your React applications.
- **CSS-in-JS Libraries:**
  - **Styled Components / Emotion:** For writing component-scoped CSS directly within JavaScript, enabling dynamic styling and better component encapsulation.

4. Testing Frameworks & Utilities:

- **Jest:** A delightful JavaScript Testing Framework with a focus on simplicity. It's widely used for unit and integration testing of React components.
- **React Testing Library:** A set of utilities that encourages good testing practices by focusing on testing components in a way that resembles how users interact with them, rather than testing implementation details.

5. Version Control:

- **Git:** Essential for source code management, collaboration, and tracking changes. I use Git daily for committing code, branching, merging, and resolving conflicts.

This comprehensive set of tools and libraries allows me to efficiently develop, debug, test, and deploy high-quality React applications.

7. Have you encountered any problems using React?

Yes, like any technology, React has its challenges, and I have encountered several common problems during my development experience. Understanding these issues is crucial for building robust and performant applications.

Here are some of the key problems I've faced:

1. **Unnecessary Re-renders and Performance Issues:**
    - **Problem:** This is perhaps the most frequent performance bottleneck in React applications. Components re-render not only when their own state or props change but also when their parent components re-render. If not managed properly, this can lead to a cascade of unnecessary re-renders, especially in complex component trees, impacting application performance and user experience.
    - **Example:** A parent component updates a piece of state that doesn't directly affect a deeply nested child component, but the child still re-renders because its parent re-rendered, even if its own props haven't changed in value (though their reference might have).
2. **Prop Drilling:**

- **Problem:** As applications grow, passing data from a parent component down to deeply nested child components through many intermediate components (that don't actually use the data themselves) becomes cumbersome. This pattern, known as "prop drilling," makes the code harder to read, maintain, and refactor.
- **Example:** Passing a user object from the `App` component down to a `UserProfile` component, which is nested several levels deep (e.g., `App -> Layout -> Header -> UserInfo -> UserProfile`).

3. **Managing Asynchronous Operations (Data Fetching, Race Conditions):**

    - **Problem:** Handling asynchronous data fetching (e.g., API calls) within `useEffect` can be tricky. Issues like race conditions (where a faster, older request might return after a slower, newer request, leading to outdated data being displayed) or memory leaks (trying to update state on an unmounted component) are common.
    - **Example:** A user quickly types into a search bar, triggering multiple API calls. If the responses come back out of order, the UI might display results from an older search query.

4. **Complex State Logic and Side Effects:**

    - **Problem:** For components with complex state transitions or multiple interdependent pieces of state, `useState` can become unwieldy, leading to verbose and hard-to-manage code. Similarly, managing multiple side effects within `useEffect` can make the component logic difficult to follow and debug.
    - **Example:** A component that manages a multi-step form with various validations, conditional fields, and external API interactions, all handled within a single `useEffect` or multiple `useState` calls.

5. **Bundle Size and Initial Load Time:**

    - **Problem:** Large React applications, especially those with many dependencies, can result in large JavaScript bundle sizes. This directly impacts the initial load time of the application, leading to a poor first impression for users, especially on slower networks.

    - **Example:** A large e-commerce application with many features and third-party libraries might have a JavaScript bundle size of several megabytes, causing a noticeable delay before the page becomes interactive.

6. **Debugging Challenges:**

    - **Problem:** While React Developer Tools are excellent, debugging complex state flows, re-render cycles, or issues originating from third-party libraries can still be challenging. Understanding why a component re-rendered or why a specific state update occurred can sometimes require significant investigation.

7. **SEO for Single Page Applications (SPAs):**

    - **Problem:** Traditional SPAs render content on the client-side, meaning the initial HTML served to search engine crawlers might be largely empty. This can negatively impact SEO, as crawlers might not fully index the content.
    - **Example:** A blog built as an SPA might struggle to rank well in search results if its content is not rendered on the server or pre-rendered.

These problems are common in modern frontend development, and addressing them effectively requires a good understanding of React's core principles and available optimization techniques.

8. When encountering these (No. 7) problems, what methods do you use to optimize?

Addressing the problems mentioned above involves a combination of best practices, built-in React features, and external libraries. My approach to optimizing and solving these issues typically includes:

1. **Optimizing Unnecessary Re-renders:**

    - **`React.memo`:** For functional components, I use `React.memo` to memoize components and prevent re-renders if their props haven't changed. This is particularly effective for pure components or components with expensive rendering logic.

- **`useCallback` and `useMemo`:** When passing functions or objects as props to memoized child components, I use `useCallback` to memoize callback functions and `useMemo` to memoize values. This ensures referential equality of props, allowing `React.memo` to effectively prevent re-renders.
- **Context API with Selective Consumers:** Instead of passing the entire context value, I design contexts to provide smaller, more granular pieces of data. Consumers then only subscribe to the specific parts of the context they need, minimizing re-renders when other parts of the context change.
- **State Colocation:** I try to keep component state as close as possible to where it's used. Lifting state up only when necessary helps reduce the scope of re-renders.

2. **Solving Prop Drilling:**

    - **React Context API:** For data that needs to be accessed by many components at different nesting levels, the Context API is my primary solution. It allows me to provide data at a higher level in the component tree and consume it directly in any descendant component, avoiding prop drilling.
    - **State Management Libraries (Redux, Zustand):** For complex global state that needs to be managed and accessed across the entire application, libraries like Redux or Zustand provide a centralized store, eliminating the need for prop drilling.
    - **Component Composition:** Sometimes, restructuring components and using composition can reduce prop drilling. Instead of passing props down, you can pass children or render props, allowing the parent to control the rendering of its children more directly.

3. **Managing Asynchronous Operations and Race Conditions:**

    - **React Query (TanStack Query):** This is my go-to library for managing server state. It handles caching, background re-fetching, data synchronization, and automatically manages race conditions and stale data. It significantly simplifies data fetching logic and reduces boilerplate.
    - **Cleanup Functions in `useEffect`:** For simpler `useEffect` scenarios, I always include cleanup functions to prevent memory leaks (e.g., clearing

timers, unsubscribing from events, or canceling pending requests when a component unmounts or an effect re-runs).

  - **AbortController:** For manual API calls, I use `AbortController` to cancel pending fetch requests when a component unmounts or when a new request is made, preventing race conditions and unnecessary state updates.

4. **Simplifying Complex State Logic and Side Effects:**

   - **`useReducer`:** For complex state logic that involves multiple sub-values or when the next state depends on the previous one, `useReducer` is a powerful alternative to `useState`. It centralizes state logic in a reducer function, making it more predictable and testable.
   - **Custom Hooks:** To extract and reuse stateful logic and side effects across multiple components, I create custom Hooks. This promotes code reusability, improves readability, and makes components cleaner and more focused on rendering.
   - **Separation of Concerns:** I strive to keep components focused on their primary responsibility (rendering UI) and delegate complex logic or side effects to custom Hooks or utility functions.

5. **Reducing Bundle Size and Improving Initial Load Time:**

   - **Code Splitting (Lazy Loading):** I implement code splitting using `React.lazy()` and `Suspense` to dynamically import components only when they are needed. This reduces the initial bundle size and speeds up the first paint.
   - **Tree Shaking:** I ensure that my build process (e.g., Webpack with proper configuration) effectively performs tree shaking to eliminate unused code from the final bundles.
   - **Image Optimization:** I optimize images by compressing them, using modern formats (WebP), and implementing lazy loading for off-screen images.
   - **Minification and Compression:** I ensure that all production builds are minified (removing whitespace, comments) and compressed (Gzip/Brotli) to reduce file sizes.
   - **CDN Usage:** For static assets, I leverage Content Delivery Networks (CDNs) to serve them from geographically closer servers, reducing latency.

6. **Debugging Challenges:**

    - **React Developer Tools:** I heavily rely on the React DevTools browser extension to inspect component hierarchies, props, state, and identify re-render patterns. The profiler feature is particularly useful for pinpointing performance bottlenecks.
    - **Logging and Debugging Statements:** Strategic use of `console.log` or a dedicated debugging library can help trace state changes and component lifecycles.
    - **Linters and Type Checkers:** Using ESLint for code quality and TypeScript for type checking helps catch many potential errors early in the development process, reducing debugging time.

7. **Improving SEO for SPAs:**

    - **Server-Side Rendering (SSR) / Next.js:** For applications where SEO is critical, I advocate for using frameworks like Next.js, which provide built-in SSR capabilities. This allows the initial page content to be rendered on the server, making it crawlable by search engines.
    - **Pre-rendering / Static Site Generation (SSG):** For content that doesn't change frequently, SSG (also supported by Next.js and Gatsby) can generate HTML files at build time, which are then served directly to crawlers and users, offering excellent performance and SEO.

By proactively applying these optimization and problem-solving strategies, I aim to build efficient, maintainable, and user-friendly React applications.

12. Do you know about sessions? (Backend-biased)

Yes, I have a good understanding of sessions, even though they are primarily a backend concept. In web development, a **session** refers to a period of interaction between a user and a web application. Since HTTP is a stateless protocol (meaning each request from a client to the server is independent and contains all the information needed to fulfill the request, without relying on previous requests), sessions are crucial for maintaining state across multiple requests from the same user.

How Sessions Work:

1. **Session Creation:** When a user first interacts with a web application (e.g., logs in, visits a page), the server generates a unique **session ID**. This ID is a random, often long, string of characters.
2. **Session Storage (Server-Side):** The server then stores this session ID along with associated user data (e.g., user ID, login status, preferences, shopping cart contents) in a temporary storage mechanism. Common server-side session storage options include:
    - **Server Memory:** Simple but not scalable across multiple servers.
    - **Databases:** More persistent and scalable, allowing sessions to be shared across a cluster of servers.
    - **Dedicated Session Stores:** Such as Redis or Memcached, which are fast, in-memory data stores optimized for key-value storage.
3. **Session ID Transmission (Client-Side):** The session ID is then sent back to the client (the user's browser). The most common way to do this is by setting an **HTTP cookie** that contains the session ID. This cookie is typically a

session cookie, meaning it is deleted when the browser is closed. 4. **Subsequent Requests:** For every subsequent request the user makes to the server, the browser automatically sends the session cookie containing the session ID. The server receives this ID, looks it up in its session storage, retrieves the associated user data, and can then process the request in the context of that specific user and their ongoing session.

Why Sessions are Used:

- **Maintaining User State:** The primary reason is to maintain state for stateless HTTP requests. This allows the server to "remember" a user across multiple page views or interactions.
- **Authentication and Authorization:** Once a user logs in, their authenticated status and permissions can be stored in the session, allowing them to access protected resources without re-authenticating on every request.
- **Personalization:** User preferences, such as language, theme, or customized content, can be stored in the session.
- **Shopping Carts:** Items added to a shopping cart can be stored in the session until the user completes the purchase or the session expires.

Session vs. Cookies vs. Web Storage:

It's important to distinguish sessions from client-side storage mechanisms like cookies and Web Storage (`localStorage` , `sessionStorage`):

- **Session (Server-Side):** The actual data is stored on the server. Only a session ID (a small identifier) is sent to the client, usually via a cookie. This makes sessions more secure for sensitive data, as the data itself never leaves the server.
- **Cookies (Client-Side with Server Interaction):** Cookies can store small amounts of data directly on the client. They are sent with every HTTP request to the server, which can lead to performance overhead if too much data is stored. They can be persistent or session-based.
- **Web Storage (`localStorage` , `sessionStorage`) (Client-Side Only):** These store larger amounts of data directly in the client's browser. The data is *not* automatically sent with every HTTP request. This makes them suitable for client-side caching or storing non-sensitive user preferences. They are generally more performant for client-side data storage than cookies because they don't add overhead to every request.

Security Considerations:

- **Session Hijacking:** If a session ID is compromised (e.g., through XSS attacks), an attacker can impersonate the user. Measures like using secure cookies (HttpOnly, Secure flags), short session timeouts, and regenerating session IDs after login are crucial.
- **Session Fixation:** An attack where an attacker fixes a user's session ID before they log in, then uses that ID to impersonate the user after they authenticate.

In summary, while sessions are managed on the backend, understanding their role is vital for frontend developers, especially when dealing with authentication, user state, and data flow in a web application. It helps in designing robust and secure client-server interactions.

1. Self-introduction (Projects worked on)

As an AI, I don't have a personal history or projects in the traditional sense. However, if I were a human frontend developer preparing for this interview, I would structure my self-introduction to highlight my experience with React and related technologies, focusing on projects that demonstrate my skills in data analysis, user retention, and web development.

Example Self-Introduction (Hypothetical Human Developer):

"Hello! My name is [Your Name], and I'm a passionate frontend developer with [X years] of experience, specializing in building dynamic and data-driven web applications primarily using React.js. My work has largely focused on the gaming industry, where I've contributed to projects involving game data analysis, improving user retention for returning players, and maintaining complex data visualization dashboards.

One of my most impactful projects involved developing a real-time analytics dashboard for a popular mobile game. The goal was to provide game designers and product managers with immediate insights into player behavior, such as daily active users, in-game purchases, and retention rates. I was responsible for building the entire frontend, from designing the interactive UI using React and a component library like Ant Design, to integrating with various backend APIs that provided the raw game data. A key challenge was optimizing the data visualization components to handle large datasets efficiently without compromising performance, which I addressed by implementing memoization techniques (`React.memo`, `useMemo`) and leveraging a data-fetching library like React Query for intelligent caching and background updates. This project significantly improved our team's ability to make data-informed decisions, directly contributing to a [mention a positive outcome, e.g., 15% increase in player engagement].

Another project focused on enhancing the user experience for returning players. I developed a personalized 'welcome back' portal that presented tailored content and incentives based on their past game activity. This involved working closely with backend developers to define API contracts for user data and implementing complex state management using Redux Toolkit to ensure a seamless and responsive interface. The success of this initiative was measured by a [mention a positive outcome, e.g., 10% improvement in 7-day retention for returning users].

My experience has equipped me with a strong understanding of modern frontend development practices, including component-based architecture, state management, performance optimization, and effective collaboration with cross-functional teams. I'm excited about the opportunity to bring these skills to your team."

18. How many people are currently on the frontend development team? How many people are in the entire team?

As an AI, I don't belong to a human team. However, in an interview setting, this question aims to understand your team dynamics, the scale of the development effort, and your potential role within the company's structure. A good answer would be specific to your previous company.

Example Answer (Hypothetical Human Developer):

"In my previous role, our core frontend development team consisted of **[e.g., 4-5]** dedicated developers. We were part of a larger engineering department. The entire product development team, including backend developers, QA engineers, product managers, and designers, was approximately **[e.g., 25-30]** people. This structure allowed for focused frontend work while maintaining close collaboration with other disciplines."

19. Has your previously developed web page been integrated into an APP?

Based on your provided work content, "全是Web没有用App的" (all Web, no App), the answer would be no. This question checks your experience with hybrid or native app development contexts.

Example Answer (Based on your context):

"No, in my previous roles, my work has been exclusively focused on web-based applications. All the projects I've worked on were built as single-page applications (SPAs) or progressive web applications (PWAs) designed for browser environments, rather than being integrated into native mobile applications. While I haven't directly integrated web pages into mobile apps, I have experience building responsive web interfaces that provide a consistent user experience across various devices, including mobile browsers."

20. How do you collaborate with other colleagues? Is there a specific process from data requirements to development? How do you interface, etc.?

This question assesses your teamwork skills and understanding of the software development lifecycle. Emphasize communication, structured processes, and problem-solving.

Example Answer (Hypothetical Human Developer):

"Collaboration is key to successful project delivery, and I prioritize clear communication and structured workflows. Our typical process, from data requirements to development, followed these steps:

1. **Requirement Gathering & Clarification:** It usually starts with product managers or data analysts defining new features or improvements, often driven by game data analysis or user retention goals. We'd have initial meetings to understand the 'what' and 'why' of the feature. As a frontend developer, I'd actively participate, asking clarifying questions about data sources, expected user interactions, and potential edge cases to ensure a shared understanding.

2. **Design & API Definition:** Once requirements were clear, the design team would create mockups and prototypes. Simultaneously, the frontend and backend teams would collaborate closely to define the necessary API endpoints and data structures. This involved:

    - **Joint API Design Sessions:** We'd hold meetings with backend developers to discuss data models, request/response formats, and potential challenges. We'd use tools like Swagger/OpenAPI for documentation.
    - **Contract-First Development:** We aimed to agree on API contracts before full implementation, allowing frontend and backend teams to work in parallel with mock data if needed.

3. **Development & Daily Stand-ups:** We followed an Agile methodology, typically Scrum. Daily stand-ups were crucial for synchronizing progress, discussing blockers, and ensuring everyone was aligned. We used Jira for task management.

    - **Version Control (Git):** All code was managed using Git, with a feature-branching workflow. Pull Requests (PRs) were used for code reviews, ensuring code quality and knowledge sharing.
    - **Regular Demos:** We'd frequently demo work-in-progress to product owners and stakeholders to gather early feedback.
4. **Testing & QA:** After development, features went through a rigorous QA process. We'd write unit and integration tests for our React components, and QA engineers would perform manual and automated testing.
5. **Deployment & Monitoring:** Once approved, features were deployed. We'd monitor performance and error logs post-deployment to quickly identify and address any issues.

Interfacing: We primarily interfaced through:

- **Regular Meetings:** Daily stand-ups, sprint planning, and retrospectives.
- **Communication Tools:** Slack for quick questions and discussions, and Jira for formal task tracking.
- **Code Reviews:** Essential for ensuring code quality and understanding each other's implementations.
- **Shared Documentation:** Maintaining up-to-date API documentation and frontend architectural decisions.

This structured approach, combined with open communication, allowed us to deliver features efficiently and maintain a high-quality codebase."

21. Have you a concept of versioning? How often are versions released? Is it fixed or un-fixed?

This question delves into your understanding of software release cycles and version control.

Example Answer (Hypothetical Human Developer):

"Yes, I have a strong concept of versioning, as it's fundamental to managing software development and deployments. We primarily used **Semantic Versioning (SemVer)**, which follows the `MAJOR.MINOR.PATCH` format (e.g., `1.2.3`).

- **`MAJOR` version:** Incremented for incompatible API changes.
- **`MINOR` version:** Incremented for adding functionality in a backward-compatible manner.
- **`PATCH` version:** Incremented for backward-compatible bug fixes.

Regarding release frequency, it was generally **fixed but flexible**.

- **Fixed Cadence:** We typically aimed for a **bi-weekly (every two weeks)** release cycle for our main product updates. This allowed us to consistently deliver new features and bug fixes to users.
- **Flexibility:** However, this was not rigid. If a critical bug was discovered, we would perform an **emergency hotfix release** outside the regular cadence. Conversely, if a sprint's features weren't fully ready or required more testing, we might push the release back by a few days to ensure stability and quality. Major feature releases or significant architectural changes might also warrant a slightly longer cycle.

Our versioning and release process involved:

1. **Feature Branches:** Each new feature or bug fix was developed on a separate Git branch.
2. **Code Reviews:** All code was reviewed before merging into the `develop` branch.
3. **Staging Environment:** Features were deployed to a staging environment for thorough QA and stakeholder review.
4. **Release Branch:** For a release, a dedicated release branch was created from `develop`.
5. **Tagging:** Once tested and approved, the release branch was merged into `main` (or `master`), and a version tag (e.g., `v1.2.3`) was applied to the `main` branch. This tag marked the exact code state of the release.
6. **Deployment:** The tagged version from `main` was then deployed to production.

This approach ensured that we had a clear history of changes, could easily roll back if necessary, and maintained a predictable release schedule while retaining the agility to respond to urgent needs."

22. How do you collaborate with the backend team?

This expands on question 20, focusing specifically on frontend-backend interaction. Emphasize communication, clear contracts, and mutual understanding.

Example Answer (Hypothetical Human Developer):

"Collaboration with the backend team is paramount for building full-stack applications. My approach focuses on establishing clear communication channels and defining robust API contracts to ensure smooth integration.

Here's how we typically collaborated:

1. **Early Involvement in Feature Planning:** When a new feature was planned, both frontend and backend leads (and relevant developers) would be involved from the outset. This allowed us to discuss the feature's requirements from both perspectives, identify potential technical challenges, and agree on the overall architecture.
2. **API Contract Definition:** This is the most critical aspect. We would hold dedicated sessions to define the API endpoints, request methods (GET, POST, PUT, DELETE), request payloads, and expected response structures (JSON schemas). We often used tools like Postman or Swagger/OpenAPI to document these contracts. This 'contract-first' approach allowed both teams to work in parallel:
   - **Frontend:** Could start building UI components and integrate with mock data based on the agreed-upon API contract.
   - **Backend:** Could focus on implementing the API logic, knowing exactly what data format the frontend expected.
3. **Regular Sync-ups:** Beyond formal meetings, we had frequent informal sync-ups. This could be daily stand-ups where we'd mention any API-related blockers or questions, or quick ad-hoc calls to clarify specific data points or error handling.
4. **Shared Tools & Environments:**
   - **Version Control:** Both teams used Git, and we often had shared repositories for API documentation or common utility functions.

    - **Staging Environments:** We had dedicated staging environments where integrated frontend and backend code could be tested together before production deployment.
    - **Communication Channels:** Slack channels were used for real-time questions and updates, and Jira tickets were linked between frontend and backend tasks to track dependencies.

5. **Error Handling & Debugging:** When issues arose during integration, we'd collaborate closely. This involved:
    - **Clear Error Messages:** Backend was encouraged to provide descriptive error messages and appropriate HTTP status codes.
    - **Joint Debugging:** Sometimes, we'd pair program or share screens to debug issues that spanned both the frontend and backend, quickly pinpointing whether the problem was with the request, the response, or the data processing.
6. **Performance Considerations:** We also discussed performance implications, such as data payload sizes, caching strategies, and potential N+1 query issues, to ensure the APIs were efficient for frontend consumption.

This collaborative approach, centered around clear communication and well-defined API contracts, ensured that our frontend and backend systems integrated seamlessly and delivered a cohesive user experience."

23. Previous projects/tasks, which one gave you a sense of accomplishment? What problems did you encounter at that time?

This is an opportunity to showcase your problem-solving skills, resilience, and ability to learn from challenges. Refer back to the self-introduction if you mentioned a specific project.

Example Answer (Hypothetical Human Developer, building on Self-Introduction):

"The project that gave me the greatest sense of accomplishment was the **real-time analytics dashboard** for the mobile game. It was a challenging but incredibly rewarding experience because it directly impacted business decisions and improved our understanding of player behavior.

During that project, I encountered several significant problems:

1. **Handling Large Datasets and Performance:** The sheer volume of game data was immense. We were receiving millions of data points daily, and displaying this in real-time charts and tables on the frontend was a major performance hurdle. Initial attempts led to slow loading times and UI freezes.

    - **Problem:** Rendering complex charts with thousands of data points, and updating them frequently, caused significant re-renders and performance degradation.
    - **Solution:** I tackled this by implementing several optimization techniques:
        - **Data Aggregation on Backend:** Worked with the backend team to pre-aggregate data where possible, reducing the payload size sent to the frontend.
        - **Virtualization/Windowing:** For large tables, I used virtualization libraries (e.g., `react-window` or `react-virtualized`) to only render the visible rows, significantly improving scroll performance.
        - **Chart Library Optimization:** Leveraged the performance features of our charting library (e.g., WebGL rendering, data simplification for overview charts).
        - **`React.memo` and `useMemo`:** Extensively used `React.memo` for pure components and `useMemo` for expensive calculations within components to prevent unnecessary re-renders.

2. **Complex Data Synchronization and Caching:** The dashboard needed to display both real-time data and historical trends, and ensure data consistency across different views. Managing data freshness and avoiding stale data was a challenge.

    - **Problem:** Users would navigate between different reports, and the data might not always be up-to-date, or multiple requests for the same data would be made.
    - **Solution:** I introduced **React Query (TanStack Query)** to manage server state. This library provided powerful caching mechanisms, automatic background re-fetching, and invalidation strategies. It allowed us to:

        - **Cache API Responses:** Store fetched data in a cache, so subsequent requests for the same data were served instantly.
        - **Background Re-fetching:** Automatically update data in the background to ensure freshness without blocking the UI.
        - **Stale-While-Revalidate:** Display cached data immediately while re-fetching in the background, providing a fast initial load.

3. **User Experience for Data Exploration:** The initial design was functional but lacked intuitive ways for users to filter, sort, and drill down into the data.
    - **Problem:** Users found it difficult to customize their views or extract specific insights from the vast amount of data.
    - **Solution:** I collaborated closely with product designers to implement advanced filtering, sorting, and grouping options directly within the UI. This involved designing flexible component APIs and integrating with the backend to support dynamic query parameters. We also added features like downloadable reports and customizable chart types.

Overcoming these challenges, especially the performance and data synchronization aspects, was a huge learning curve and seeing the positive impact on the product team's ability to analyze game data made it incredibly satisfying."

24. If you were to redo a project you've worked on, which part do you think could be done better?

This question assesses your self-reflection, critical thinking, and continuous improvement mindset. Choose a specific area and explain *why* and *how* you would improve it.

Example Answer (Hypothetical Human Developer, building on previous answers):

"If I were to redo the real-time analytics dashboard project, one area I would definitely approach differently, or put more emphasis on from the start, is **initial architectural planning for scalability and modularity, particularly around data visualization components and their interaction with state management.**

Why it could be done better:

When we started, the focus was heavily on getting the core functionality out quickly. As the project grew, and more complex data visualizations and filtering options were added, we found ourselves retrofitting some of the state management and data flow for individual chart components. This led to:

- **Increased Complexity:** Some components became more tightly coupled than ideal, making them harder to reuse or modify independently.
- **Suboptimal Performance in Edge Cases:** While we optimized heavily, some very specific combinations of filters and large datasets still caused minor performance hiccups because the data transformation pipeline wasn't as streamlined as it could have been from day one.
- **Maintenance Overhead:** Debugging data flow issues across multiple interconnected charts became more time-consuming.

How I would do it better:

1. **More Granular State Management for Charts:** Instead of having a single large state object for all dashboard filters and data, I would have designed a more granular state structure, perhaps using `useReducer` or even a dedicated Zustand store for each major dashboard section or complex chart. This would allow for more isolated updates and prevent unnecessary re-renders of unrelated components.

2. **Standardized Data Transformation Layer:** I would have established a more explicit and standardized data transformation layer between the raw API responses and the data consumed by the charting libraries. This layer would be responsible for memoizing transformations (`useMemo`) and ensuring data was in the most optimal format for rendering, reducing redundant calculations across different components.

3. **Earlier Integration of a Data-Fetching Library:** While we eventually adopted React Query, integrating it even earlier in the project lifecycle would have prevented some initial struggles with `useEffect` for data fetching, race conditions, and manual caching. Its declarative nature would have simplified the data flow from the start.

4. **Component API Design:** I would have spent more time designing the APIs (props) for our reusable data visualization components to be even more flexible

and decoupled from the specific data source. This would make them truly plug-and-play for different types of reports.

This retrospective insight comes from learning how the application evolved and scaled. Starting with a more robust and modular architecture for these core data-driven parts would have saved development time and improved long-term maintainability and performance."

25. Have you often been on GitHub?

This question checks your familiarity with Git and GitHub as collaborative tools.

Example Answer (Hypothetical Human Developer):

"Yes, I'm on GitHub almost daily. It's an indispensable part of my development workflow. I use it for:

- **Version Control:** Pushing my code, creating branches for new features or bug fixes, merging pull requests, and resolving conflicts.
- **Code Reviews:** Participating in code reviews for my team members, providing feedback, and learning from their code.
- **Collaboration:** Collaborating on shared repositories, contributing to team projects, and tracking project progress.
- **Open Source:** Occasionally contributing to open-source projects or exploring repositories for learning and inspiration.
- **Portfolio:** My personal projects and contributions are also hosted on GitHub, serving as a public portfolio of my work.

I'm very comfortable with Git commands and the GitHub interface for managing repositories and collaborating effectively."

26. What are your hobbies and interests?

This is a common behavioral question to understand your personality and how you spend your time outside of work. It's an opportunity to show you're a well-rounded individual.

Example Answer (Hypothetical Human Developer):

"Outside of coding, I have a few interests that help me unwind and stay balanced. I'm quite keen on **[e.g., hiking/photography/cooking/playing a musical instrument]**. I find that [hiking] helps me clear my head and appreciate nature, which can be a nice contrast to screen time. I also enjoy **[e.g., playing video games/reading sci-fi novels/learning new languages]**. [Playing video games] not only aligns with my professional interest in the gaming industry but also helps me understand user experience from a player's perspective. I also try to stay active by [e.g., going to the gym/cycling]."

27. Strengths & Weaknesses? How do your friends evaluate you?

This is a classic behavioral question. For strengths, align them with the job requirements. For weaknesses, choose a genuine one and frame it as an area of growth, demonstrating self-awareness. For friends' evaluation, pick positive traits that align with professional qualities.

Example Answer (Hypothetical Human Developer):

"**Strengths:**

I believe my greatest strengths as a frontend developer are my **problem-solving abilities** and my **dedication to continuous learning**. I enjoy dissecting complex challenges, whether it's optimizing a slow-rendering component or debugging an intricate data flow, and finding efficient solutions. I'm also constantly exploring new technologies and best practices in the React ecosystem and broader frontend landscape to ensure my skills remain sharp and relevant. For instance, when we faced performance issues with large datasets, I proactively researched and implemented virtualization techniques and React Query, which significantly improved the dashboard's responsiveness.

Another strength is my **attention to detail and commitment to user experience**. I strive to build interfaces that are not only functional but also intuitive, accessible, and visually appealing. I pay close attention to responsiveness, animation, and overall user flow.

Weaknesses:

One area I'm actively working on is **delegation and trusting others with tasks that I feel I can do faster myself**. In the past, especially when under tight deadlines, I sometimes found myself taking on too much, which could lead to burnout or bottlenecking. I've learned that while my intention is to ensure quality and speed, effective teamwork requires distributing responsibilities. To address this, I've been consciously practicing delegating more, providing clear instructions, and trusting my teammates' capabilities. I've also focused on improving my mentoring skills to empower others to take on more complex tasks.

How my friends evaluate me:

My friends would probably describe me as **analytical and reliable**. They often come to me for advice when they need a logical approach to a problem, and they know that if I commit to something, I'll follow through. They might also say I'm **curious** and always eager to learn new things, whether it's a new programming concept or a new hobby."

onClick={() => setMultiplier(multiplier + 1)}>Increment Multiplier

Result: {expensiveResult}

); } ```

These Hooks, along with `React.memo` (discussed in Q4), form the core toolkit for building efficient, maintainable, and scalable React applications using functional components.

9. What are cookies and code storage? Have you used them?

In frontend development, **cookies** and **web storage** (which includes `localStorage` and `sessionStorage`) are client-side mechanisms used to store data directly within the user's web browser. They are essential for maintaining state, personalizing user experiences, and enabling various web functionalities.

Cookies

Cookies are small pieces of data (text files) that a web server sends to a user's web browser. The browser stores them and sends them back to the same server with every subsequent request. They are primarily used for:

- **Session Management:** Keeping a user logged in, remembering user preferences (e.g., language, theme).
- **Personalization:** Storing user-specific settings or tracking user behavior to provide tailored content.
- **Tracking:** Recording and analyzing user activity across websites (though this is increasingly restricted due to privacy concerns).

Key Characteristics of Cookies:

- **Small Size:** Typically limited to about 4KB per domain. This limit includes the cookie's name, value, expiration date, and other attributes.
- **Sent with Every HTTP Request:** This is a significant point. Every time the browser makes a request to the server for the domain that set the cookie, the cookie data is automatically included in the HTTP header. This can lead to increased network traffic if too much data is stored in cookies.
- **Expiration:** Can be set to expire after a certain duration (persistent cookies) or when the browser session ends (session cookies).
- **Domain-Specific:** Cookies are associated with a specific domain and are only sent to that domain.
- **Security Flags:** Can have flags like `HttpOnly` (prevents client-side JavaScript access, mitigating XSS attacks) and `Secure` (ensures the cookie is only sent over HTTPS).

Have I used them?

Yes, I have used cookies, primarily for session management (e.g., storing authentication tokens or session IDs) and remembering user preferences. While direct manipulation of cookies via JavaScript (`document.cookie`) is possible, I often interact with them indirectly through backend-set `HttpOnly` cookies for security reasons, or through libraries that abstract away the complexities of cookie management for specific use cases like authentication.

Web Storage (localStorage and sessionStorage)

Web Storage is a more modern and robust way to store data on the client-side compared to cookies. It provides larger storage capacity and is not sent with every HTTP request, making it more efficient for storing client-side data that doesn't need to be transmitted to the server on every interaction. Web Storage is divided into two types:

- **`localStorage`**
- **`sessionStorage`**

I will elaborate on the differences between `localStorage` and `sessionStorage` in the next question.

Key Characteristics of Web Storage:

- **Larger Capacity:** Typically offers 5MB to 10MB of storage per origin (domain), significantly more than cookies.
- **Not Sent with HTTP Requests:** Data stored in `localStorage` or `sessionStorage` is *not* automatically sent to the server with every HTTP request. This reduces network overhead.
- **Client-Side Only:** Data is accessible only via JavaScript on the client-side.
- **Origin-Specific:** Data is scoped to the origin (protocol, hostname, and port). Data stored by one origin cannot be accessed by another.

Have I used them?

Yes, I frequently use both `localStorage` and `sessionStorage` for various purposes:

- **`localStorage`:** For persistent data that needs to remain available even after the user closes and reopens the browser. Examples include:
  - Storing user preferences (e.g., theme settings, layout preferences).
  - Caching application data (e.g., frequently accessed lists, user profiles) to improve initial load times and reduce API calls.
  - Saving user input in forms to prevent data loss on accidental page refresh.
- **`sessionStorage`:** For temporary data that is only relevant for the duration of a single browser session. Examples include:

    - Storing data for multi-step forms, where the data is needed across different pages within the same session but not persistently.
    - Caching temporary state for a specific user journey.

Both cookies and web storage are valuable tools in a frontend developer's arsenal, each with its own strengths and ideal use cases. Choosing the right storage mechanism depends on the data's sensitivity, size, persistence requirements, and whether it needs to be sent to the server with every request.

10. What is the difference between `localStorage` and `sessionStorage`?

Both `localStorage` and `sessionStorage` are part of the Web Storage API, providing mechanisms for web applications to store data locally within the user's browser. They are similar in that they both offer a key-value store and are origin-bound (data stored by one origin cannot be accessed by another). However, their primary difference lies in their **persistence** and **scope**.

Here's a detailed comparison:

Feature	`localStorage`	`sessionStorage`
Persistence	**Persistent.** Data remains even after the browser window is closed and reopened, or the computer is restarted. It has no expiration date.	**Session-based.** Data is cleared when the browser tab or window is closed. It persists across page reloads and restores (e.g., using the browser's back/forward buttons) within the same tab.
Scope	**Global to the origin.** Data is accessible across all tabs and windows from the same origin.	**Limited to the tab/window.** Data is only accessible within the specific tab or window where it was set. If you open a new tab or window to the same origin, it will have its own separate `sessionStorage` instance.
Capacity	Typically 5MB to 10MB per origin.	Typically 5MB to 10MB per origin.
Data Type	Stores data as strings. Objects need to be serialized (e.g., using `JSON.stringify()`) before storing and deserialized (`JSON.parse()`) when retrieved.	Stores data as strings. Objects need to be serialized (e.g., using `JSON.stringify()`) before storing and deserialized (`JSON.parse()`) when retrieved.
Accessibility	Accessible via JavaScript (`window.localStorage`).	Accessible via JavaScript (`window.sessionStorage`).
Sent with HTTP Requests	No. Data is not automatically sent to the server with every HTTP request.	No. Data is not automatically sent to the server with every HTTP request.
Use Cases	- User preferences (theme, layout)	
- Caching application data (offline capabilities)		
- Saving user input in forms across sessions	- Data for multi-step forms (within a single session)	

Feature	`localStorage`	`sessionStorage`
- Temporary state for a specific user journey		
- Storing data that should not persist if the user closes the tab		

Key Differences Explained:

1. **Persistence:**
    - **`localStorage`:** Think of it as a permanent storage locker in your browser. Whatever you put in there stays until you explicitly remove it, or the user clears their browser data. This makes it ideal for long-term user preferences or cached data that enhances the user experience across multiple visits.
    - **`sessionStorage`:** This is like a temporary locker that gets emptied as soon as you close the specific browser tab or window where the data was stored. It's perfect for data that is only relevant for the current browsing session and should not carry over to new tabs or future visits.
2. **Scope:**
    - **`localStorage`:** If you open multiple tabs or windows to the same website (same origin), they all share the same `localStorage` data. Changes made in one tab are immediately reflected in others.
    - **`sessionStorage`:** Each new tab or window opened to the same origin gets its own independent `sessionStorage` instance. Data stored in `sessionStorage` in one tab is not accessible from another tab, even if they are visiting the same website. This isolation is crucial for certain application flows.

Common Methods for `localStorage` and `sessionStorage`:

Both `localStorage` and `sessionStorage` expose the same API methods:

- **`setItem(key, value)`**: Adds a key-value pair to storage. `javascript localStorage.setItem("username", "Alice"); sessionStorage.setItem("currentStep", "2");`
- **`getItem(key)`**: Retrieves the value associated with a given key. `javascript const username = localStorage.getItem("username"); // "Alice" const currentStep = sessionStorage.getItem("currentStep"); // "2"`
- **`removeItem(key)`**: Removes the key-value pair with the specified key. `javascript localStorage.removeItem("username");`
- **`clear()`**: Removes all key-value pairs from storage. `javascript sessionStorage.clear();`
- **`key(index)`**: Returns the name of the key at the specified index.
- **`length`**: Returns the number of key-value pairs currently stored.

Understanding these differences is crucial for choosing the appropriate storage mechanism for your application's needs, balancing persistence, scope, and performance considerations.

11. What are the results of opening a new window using `localStorage` and `sessionStorage` respectively?

This question directly tests your understanding of the scope differences between `localStorage` and `sessionStorage`.

Let's consider a scenario where you have a web page open at `https://example.com`.

Scenario 1: Opening a new window/tab with `localStorage`

If you store data in `localStorage` in the original tab/window:

```
// In original tab/window (https://example.com)
localStorage.setItem("userPreference", "darkTheme");
console.log(localStorage.getItem("userPreference")); // Output: "darkTheme"
```

Now, if you open a **new tab or a new window** to the **same origin** (`https://example.com`):

```
// In the new tab/window (https://example.com)
console.log(localStorage.getItem("userPreference")); // Output: "darkTheme"
```

Result for `localStorage`:

When you open a new window or tab to the same origin, the data stored in `localStorage` **will be accessible** in the new window/tab. This is because `localStorage` is scoped globally to the origin, meaning all tabs and windows from the same domain share the same `localStorage` data. Any changes made to `localStorage` in one tab will be immediately reflected in all other tabs/windows from the same origin.

Scenario 2: Opening a new window/tab with `sessionStorage`

If you store data in `sessionStorage` in the original tab/window:

```
// In original tab/window (https://example.com)
sessionStorage.setItem("currentFormStep", "3");
console.log(sessionStorage.getItem("currentFormStep")); // Output: "3"
```

Now, if you open a **new tab or a new window** to the **same origin** (`https://example.com`):

```
// In the new tab/window (https://example.com)
console.log(sessionStorage.getItem("currentFormStep")); // Output: null
```

Result for `sessionStorage`:

When you open a new window or tab to the same origin, the data stored in `sessionStorage` **will NOT be accessible** in the new window/tab. Instead, the new window/tab will have its own, empty `sessionStorage` instance. This is because `sessionStorage` is scoped to the specific browser tab or window where it was created. Each new tab or window represents a new session for `sessionStorage`.

Important Note on Duplicating Tabs:

Some browsers (like Chrome) have a