JavaScript, React & Next.js Interview Preparation Guide

A detailed expansion of core interview questions for Senior/Mid-level Front End Engineers.

🟡 Part 1: JavaScript Fundamentals & Runtime

1. Explain the Event Loop simply.

The Concept: JavaScript is single-threaded, meaning it has one **Call Stack**. The Event Loop is the mechanism that allows JS to perform non-blocking I/O operations (like fetching data) despite being single-threaded.

How it works:

1. **Call Stack:** Executes synchronous code (LIFO - Last In, First Out).
2. **Web APIs:** When an async operation (like `setTimeout` or `fetch`) is encountered, it is offloaded to the browser's Web APIs.
3. **Queues:** Once the async operation finishes, its callback is moved to a queue.
   - **Macrotask Queue:** `setTimeout`, `setInterval`, UI rendering.
   - **Microtask Queue:** Promises (`.then`), `queueMicrotask`, MutationObserver.
4. **The Loop:** The Event Loop constantly checks: *"Is the Call Stack empty?"* If yes, it pushes items from the **Microtask Queue** first (until empty), then pushes **one** item from the **Macrotask Queue**, then repeats.

2. Microtask Queue vs. Macrotask Queue

- **Microtasks (High Priority):** These run *immediately* after the current script executes and before the event loop continues to the next task.
  - *Examples:* `Promise.resolve().then()`, `queueMicrotask`, `process.nextTick` (Node.js).
- **Macrotasks (Standard Priority):** These represent discrete, independent units of work.
  - *Examples:* `setTimeout`, `setInterval`, `setImmediate`, I/O, UI rendering.
- **Key Difference:** The Event Loop will process **all** microtasks before moving on to the next macrotask. This means a recursive microtask loop can block the UI/macrotasks.

3. How do async/await and Promises map to the Event Loop?

- **Promises:** When a Promise resolves, its `.then()` callback is pushed to the **Microtask Queue**.
- **Async/Await:** This is syntactic sugar for Promises.
  - When the engine sees `await`, it suspends the execution of that specific function.
  - The "rest" of the function (code after `await`) is effectively wrapped in a `.then()` block and pushed to the **Microtask Queue** once the awaited value resolves.
  - The engine goes back to execute other synchronous code outside that async function.

4. Call Stack Overflow vs. Memory Leak

- **Stack Overflow:** Occurs when there are too many function calls on the stack, usually due to **unbounded recursion**.
  - *Debug:* Check the browser console for `RangeError: Maximum call stack size exceeded` and review the stack trace to find the recursive function.
- **Memory Leak:** Occurs when objects are no longer needed but are still referenced by the root, preventing Garbage Collection (GC).
  - *Causes:* Global variables, detached DOM nodes, uncleared intervals, closures holding large scopes.
  - *Debug:* Use Chrome DevTools > **Memory Tab**. Take a **Heap Snapshot** and look for "Detached DOM trees" or objects that should have been collected.

5. How does `this` work?

The value of `this` is determined by **how the function is called** (execution context), not where it was defined (except for arrow functions).

- **Implicit Binding:** `obj.method()` -> `this` is `obj`.
- **Explicit Binding:** `call`, `apply`, `bind` -> `this` is manually defined.
- **New Binding:** `new Constructor()` -> `this` is the new instance.
- **Default Binding:** Plain function call `func()` -> `this` is global (window) or `undefined` (in strict mode).
- **Arrow Functions:** They do **not** have their own `this`. They inherit `this` from the enclosing lexical scope at the time of definition.

6. Explain Hoisting.

Hoisting is the behavior where variable and function declarations are moved to the top of their scope during the compilation phase.

- `function` **declarations:** Fully hoisted. You can call them before they are defined in the code.
- `var`**:** Hoisted but initialized as `undefined`. Accessing it before assignment gives `undefined`.
- `let` **/** `const`**:** Hoisted but placed in the **Temporal Dead Zone (TDZ)**. Accessing them before declaration throws a `ReferenceError`.

7. let / const / var differences.

Feature	`var`	`let`	`const`
Scope	Function Scope	Block Scope (`{}`)	Block Scope (`{}`)
Hoisting	Yes (undefined)	Yes (TDZ Error)	Yes (TDZ Error)
Reassign?	Yes	Yes	No
Redeclare?	Yes	No	No

- **Best Practice:** Default to `const`. Use `let` only when re-assignment is needed. Avoid `var`.

8. Closures: What and Why?

A closure is a function bundled together with references to its surrounding state (the lexical environment). It gives you access to an outer function's scope from an inner function, even after the outer function has finished executing.

- **Common Gotcha (Loops):** Using `var` in a `for` loop with `setTimeout`. Because `var` is function-scoped, all timeouts share the *same reference* to the loop variable `i`.
    - *Fix:* Use `let` (block scoped) which creates a new binding for `i` in each iteration.

9. Prototypal Inheritance vs. Class Sugar

- **Prototypal Inheritance:** Objects can link to other objects. When accessing a property, if the object doesn't have it, the engine looks up the "prototype chain" until it finds it or hits null.
- **Classes:** In JS, `class` is primarily syntactic sugar over this existing prototypal inheritance. Under the hood, it still uses functions and prototypes (`__proto__`), but provides a cleaner, more familiar syntax for developers coming from OOP languages.

10. Event Delegation

Instead of attaching an event listener to every single child element (e.g., every row in a table), you attach **one** listener to a common parent.

- **How:** inside the handler, check `event.target` to see which child triggered the event.
- **Why:** Saves memory (fewer function objects) and handles dynamic content (new children added later automatically "have" the listener).

11. Debouncing vs. Throttling

- **Debouncing:** "Wait until the user stops."
  - *Logic:* Group multiple sequential calls into a single execution. The timer resets on every event.
  - *Use Case:* Search bar inputs (wait for typing to finish before API call), window resizing.
- **Throttling:** "Execute at most once every X milliseconds."
  - *Logic:* Ensure the function runs regularly, but not more often than the limit.
  - *Use Case:* Scroll event handlers (checking scroll position), button click spam prevention.

12. Memory & Performance Tools

- **Performance Tab:** Record a session to see FPS, main thread blocking, and function execution time (Flame Charts).
- **Memory Tab:**
  - *Heap Snapshot:* Find memory leaks (detached nodes).
  - *Allocation instrumentation:* See memory spikes in real-time.
- **Lighthouse:** Automated auditing for performance, accessibility, and SEO.

13. `==` vs `===`

- `==` (Loose Equality): Performs **type coercion** before comparing. (e.g., `5 == "5"` is true). This often leads to bugs.
- `===` (Strict Equality): Checks both **value** and **type**. (e.g., `5 === "5"` is false). Always use this.

14. Web Workers

JavaScript is single-threaded on the main thread (UI thread). Web Workers allow you to run scripts in background threads.

- **Capability:** Perfect for CPU-intensive tasks (image processing, sorting massive arrays) without freezing the UI.
- **Limitation:** Workers **cannot access the DOM**. Communication happens via `postMessage`.

15. How to cancel Fetch/XHR?

- **Fetch:** Create an `AbortController`. Pass its `signal` to the fetch options (`fetch(url, { signal })`). Call `controller.abort()` to cancel.
- **XHR:** Call `xhr.abort()` on the instance.

💻 Part 2: Coding Questions (Practice Strategy)

Tips for implementation questions:

- **16. Reverse String:** Avoid `.split().reverse().join()` if performance matters; use a two-pointer approach (swap start/end) for O(1) space.
- **17. Find Duplicate:** Use a `Set` for O(N) time complexity. If space is tight, sort first O(N log N).
- **18. Debounce Implementation:** Return a function that holds a `timer` variable in its closure. Clear `timer` on every call.
- **19. Deep Clone:** `JSON.parse(JSON.stringify(obj))` fails on Dates, Functions, and `undefined`. Use `structuredClone()` (modern) or recursion for a robust solution.
- **20. Flatten Array:** Use `Array.flat(Infinity)` or write a recursive reduce function.
- **21. Pub/Sub:** Maintain an object `events = { 'event_name': [callbacks] }`.
- **23. LRU Cache:** Use a `Map`. It preserves insertion order. When accessing an item, delete and re-set it to move it to the "end" (most recently used).

⚛️ Part 3: React Core & Hooks

24. Why React?

React solves the problem of keeping the UI in sync with the state.

- **Declarative:** You describe *what* the UI should look like for a given state, not *how* to change it step-by-step (imperative).
- **Component-Based:** Encourages reusable, composable code.
- **Virtual DOM:** Optimizes rendering performance by minimizing direct DOM manipulations (which are slow).

25. Reconciliation and Keys

- **Reconciliation:** The process of syncing the Virtual DOM with the real DOM.
- **Keys:** Special string attributes you must pass to lists. React uses keys to identify which items have changed, are added, or are removed.

    - *Risk:* Using `index` as a key can cause bugs if the list order changes (React thinks the item is the same because the index matches, preserving old state).

26. The Virtual DOM

A JavaScript object representation of the real DOM. When state changes:

1. React creates a new Virtual DOM tree.
2. It "diffs" this new tree against the previous one.
3. It calculates the minimal set of changes (patches).
4. It updates the real DOM in a batch.

27. Class vs. Function Components

- **Class:** Heavy syntax, requires `this` binding, lifecycle methods (`componentDidMount`) are split by *when* they run, not *what* they do.
- **Function + Hooks:** Lighter syntax, no `this` . Hooks (`useEffect`) allow grouping related logic (e.g., subscribing and unsubscribing) together.

28. `useState` vs. `useReducer`

- **`useState` :** Best for independent, primitive state (boolean toggles, inputs).
- **`useReducer` :** Best for complex state logic, where the next state depends on the previous one, or when multiple sub-values change together. It mimics the Redux pattern.

29. `useEffect` Dependencies & Cleanup

- **Dependencies Array:** Controls when the effect runs.
    - `[]` : Runs once on mount.
    - `[prop]` : Runs on mount and whenever `prop` changes.
    - *No array:* Runs on every render (dangerous).
- **Cleanup Function:** Return a function from `useEffect` to clean up side effects (remove event listeners, cancel subscriptions) before the component unmounts or re-runs.

30. Stale Closures in Hooks

Because functional components are just functions, variables form closures. If you use `useEffect` without including a variable in the dependency array, the effect captures the *old* value of that variable from the initial render.

- **Fix:** Always include dependencies in the array, or use the "functional update" form of setState: `setCount(prev => prev + 1)`.

31. `useMemo` vs. `useCallback`

- `useMemo`: Caches the **result** of a calculation. Use it to avoid expensive computations on every render.
- `useCallback`: Caches the **function definition** itself. Use it to prevent child components (wrapped in `React.memo`) from re-rendering unnecessarily because a function prop "changed" reference.

32. Context vs. Redux vs. Props

- **Props Drilling:** Passing data down 3+ levels. Bad for maintenance.
- **Context API:** Great for global, low-frequency updates (Theme, User Auth, Language). Not optimized for high-frequency updates (typing in a text input) as it re-renders all consumers.
- **State Managers (Redux/Zustand):** Better for complex global state where performance matters (selectors prevent unnecessary re-renders).

33. Optimizing React Performance

1. **Code Splitting:** `React.lazy` and `Suspense` to load bundles only when needed.
2. **Memoization:** `React.memo`, `useMemo`, `useCallback` to prevent wasted renders.
3. **Virtualization:** Use libraries like `react-window` for long lists (render only what is visible).
4. **State Colocation:** Move state down to the lowest common ancestor (don't keep everything global).

34. Suspense & Error Boundaries

- **Suspense:** Allows components to "wait" for something (like data fetching or lazy-loaded code) before rendering. Shows a fallback (spinner) in the meantime.
- **Error Boundary:** A **Class Component** (currently) that wraps part of the tree. It catches errors in the child tree, logs them, and displays a fallback UI instead of crashing the whole app.

35. Testing Components

- **Unit/Integration:** Use **React Testing Library**. Focus on *behavior*, not implementation details.
  - *Good:* `screen.getByText('Submit').click()`
  - *Bad:* `component.state.isOpen === true`
- **E2E:** Playwright or Cypress for full user flows.

36. Controlled vs. Uncontrolled

- **Controlled:** React State (`useState`) drives the input value. You have total control over the data.
- **Uncontrolled:** The DOM handles the data. You access it via a Ref (`useRef`). Easier for integrating non-React libraries or simple forms.

⚡ Part 4: Next.js (Essentials & Production)

44. Why Next.js?

- **Hybrid Rendering:** Offers flexibility per page (Static, Server-Side, Client-Side).
- **Routing:** File-system based routing is intuitive.
- **Performance:** Automatic image optimization, script optimization, and font loading.
- **Full Stack:** API Routes allow building backend logic within the same project.
- *(Interview Tip: When mentioning PanditAI, explain how Next.js specifically solved a problem there—e.g., "We needed SEO for public pages (SSG) but real-time data for the dashboard (Client/SSR).")*

45. App Router vs. Pages Router

- **Pages Router:** The "classic" Next.js. `pages/` directory. Uses `getServerSideProps` / `getStaticProps` .
- **App Router (New):** `app/` directory. Built on **React Server Components (RSC)**.
  - *Key Diff:* Components are Server Components by default. Data fetching happens inside the component (async/await). Supports Layouts and Streaming naturally.

46. SSR vs. SSG vs. ISR

- **SSG (Static Site Generation):** HTML generated at **build time**. Fastest. Great for blogs, marketing pages.
- **SSR (Server-Side Rendering):** HTML generated on **every request**. Slower TTFB, but data is always fresh. Good for personalized dashboards.
- **ISR (Incremental Static Regeneration):** "Best of both." Generate static pages, but rebuild them in the background after a specific time interval (`revalidate` prop) or on-demand.

47. When is SSR slower than SSG?

Always, regarding Time to First Byte (TTFB). SSG serves a pre-computed file from a CDN (Edge). SSR requires the server to spin up, run logic, fetch DB data, generate HTML, and send it.

48. Data Fetching in App Router

- **Server Components:** Fetch data directly inside the component using `async/await`. No need for `useEffect` or `useState`. Secure (secrets don't leak to client).
- **Client Components:** Use standard React patterns (React Query / SWR / `useEffect`) or call an internal API Route.

49. Authentication Patterns

- **NextAuth.js (Auth.js):** The standard solution. Handles OAuth, sessions, and DB adapters easily.
- **Custom:**
    - *Session:* Store session ID in an `httpOnly` cookie. Middleware validates it.
    - *JWT:* Store access token (short-lived) in memory/JS and refresh token in `httpOnly` cookie.

50. Middleware & Edge Functions

- **Middleware:** Code that runs **before** a request completes.
    - *Use Cases:* Redirecting users based on auth status, rewriting paths for A/B testing, geolocation blocking.
- **Edge Functions:** Serverless functions running on the CDN edge (closer to user). Fast start-up but limited runtime (no Node.js FS access).

51. Securing API Routes

- **Rate Limiting:** Prevent abuse (use Redis or Upstash).
- **Validation:** Use libraries like **Zod** to validate incoming JSON bodies.
- **Methods:** Check `req.method` (only allow POST/GET as intended).
- **Auth:** Validate the session/token at the very top of the handler.

52. Image Optimization (`next/image`)

Automatically serves images in modern formats (WebP/AVIF) and resizes them based on the user's device size. Prevents **Cumulative Layout Shift (CLS)** by requiring dimensions.

53. ISR (Incremental Static Regeneration)

Allows you to update static content *without* rebuilding the entire site.

- *How:* You return `revalidate: 60` (seconds). The first user after 60s gets the stale page, but triggers a background rebuild. The *next* user gets the fresh page.

🛠️ Part 5: System Design & Practical Tasks

64. Design a Job Queue (ETA System)

If a task takes > 3-4 seconds (e.g., generating a PDF or AI response), don't make the user wait on a loading spinner via HTTP.

1. **Client** sends request -> **Server** creates a Job ID, puts task in **Redis/Queue**, returns Job ID immediately.
2. **Worker** picks up task, processes it, updates DB with result/status.
3. **Client** polls `/api/status/{jobId}` every few seconds OR server pushes update via **WebSockets**.

65. DB Sessions vs. JWT

- **DB Sessions:** The server stores the active session in a database.
  - *Pros:* Easy to revoke (ban user = delete row).
  - *Cons:* Database lookup on every request (latency).
- **JWT (Stateless):** The token contains the data signed by a secret.
  - *Pros:* Fast (no DB lookup needed to verify).
  - *Cons:* Hard to revoke immediately (must wait for expiry or build a "blocklist" which defeats the purpose).

66. Retrieval System for Embeddings (e.g., for eVakeel)

- *Scenario:* Searching through legal documents.

1. **Ingestion:** Chunk text -> OpenAI/HuggingFace Model -> Vectors -> Store in **Vector DB** (Pinecone/Milvus/pgvector).
2. **Retrieval:** User Query -> Generate Vector -> Cosine Similarity Search in Vector DB -> Return Top K chunks.
3. **Optimization:** Cache frequent queries in Redis. Use hybrid search (Keyword + Semantic) for better legal accuracy.

68-72. Behavioral Questions (STAR Method)

Use the **STAR** method for every answer:

- **S**ituation: "In my previous project, we had a memory leak..."
- **T**ask: "I needed to identify the cause without stopping production..."
- **A**ction: "I used Chrome Profiler to isolate the component and applied a fix using..."
- **R**esult: "Reduced memory usage by 40% and improved page load by 2s."

(Self-Correction/Note: When asked about AI tools (Q70), be honest but emphasize that YOU are the pilot. "I use Copilot to scaffold boilerplate tests, but I manually review every line for security and logic errors.")