

# React FE Engineer Interview

A plain-language guide to advanced frontend concepts — grouped by theme, explained simply, with concrete examples.

## Table of Contents

1. [Rendering & Hydration](#)
2. [React Internals](#)
3. [JavaScript Runtime & Memory](#)
4. [Browser Rendering Pipeline](#)
5. [Performance & Loading](#)
6. [Build Tools & Bundling](#)
7. [Web APIs & Browser Primitives](#)
8. [Web Components & Shadow DOM](#)
9. [Concurrency & Async Patterns](#)
10. [Caching & Networking](#)
11. [Security](#)
12. [State Management & Architecture](#)
13. [Real-Time & Collaboration](#)
14. [Performance Metrics \(Core Web Vitals\)](#)
15. [Observability & Debugging](#)
16. [Accessibility](#)

## 1. Rendering & Hydration

### Hydration

**What it is:** After a server sends pre-rendered HTML to the browser, hydration is the process of attaching JavaScript event handlers to that static HTML so it becomes interactive.

**Analogy:** The server ships a printed photo of a button. Hydration is when JS “wakes it up” so clicking it actually does something.

**Example:** Next.js sends `<button>Click me</button>` as raw HTML. React then runs in the browser, finds that button, and attaches an `onClick` handler — that’s hydration.

**Pitfall:** If the HTML the server sent doesn’t match what React would render client-side, you get a hydration mismatch error.

## Partial Hydration

**What it is:** Only hydrate (activate) the parts of the page that actually need interactivity. Leave static parts as plain HTML.

**Analogy:** A newspaper page — most of it is just text you read. Only the crossword puzzle widget needs to be interactive. Why load JS for the article text?

**Example:** A blog post with a comment form. The article body stays as static HTML; only the comment form component gets hydrated with React.

## Islands Architecture

**What it is:** A page is mostly static HTML “ocean” with small interactive “islands” of JavaScript. Each island is independently hydrated.

**Example (Astro framework):**

```
←!— Static content – zero JS —→  
<h1>My Blog Post</h1>  
<p>Some text...</p>  
  
←!— Island – hydrated with React only when visible —→  
<CommentBox client:visible />
```

Each island boots independently. A crash in one doesn’t affect others.

# Streaming SSR

**What it is:** Instead of waiting for the entire page to be ready before sending it, the server streams HTML in chunks as each piece is ready. The browser can start rendering immediately.

**Analogy:** A restaurant that brings each dish as it's cooked, rather than making you wait until every dish for the whole table is done.

**Example:** React 18's `renderToPipeableStream`. The shell (header, nav) arrives in milliseconds. The data-heavy product list streams in later, replacing a skeleton loader.

## Selective Hydration

**What it is:** React can prioritize which parts of the page to hydrate first based on user interaction. If you click a component that hasn't been hydrated yet, React hydrates it immediately — skipping the queue.

**Example:** Page has 3 lazy sections. User clicks section 3 before sections 1 and 2 are hydrated. React jumps to hydrate section 3 first so the click isn't lost.

## Server Components

**What it is:** Components that run only on the server and send their output as serialized data — never ship their code to the browser. They can directly access databases without an API layer.

**Example (Next.js App Router):**

```
// This runs only on the server - DB call, no client JS shipped
async function ProductList() {
  const products = await db.query('SELECT * FROM products');
  return <ul>{products.map(p => <li>{p.name}</li>)}/<ul>;
}
```

The browser receives the rendered `<ul>`, not the component code.

## Edge Rendering

**What it is:** Running server-side rendering at CDN edge nodes close to the user, rather than in a central data center. Reduces latency dramatically.

**Example:** A user in Vienna hits a Vercel edge node in Frankfurt instead of a US origin server. Time-to-first-byte drops from ~200ms to ~20ms. Cloudflare Workers and Vercel Edge Functions enable this.

## Speculative Prerendering

**What it is:** The browser (or framework) prerenders a page the user is likely to navigate to next — before they click.

**Example:** Chrome's `<speculation-rules>` API:

```
<script type="speculationrules">
  { "prerender": [{ "urls": ["/checkout"] }] }
</script>
```

When the user clicks “Go to Checkout,” the page is already fully rendered. Navigation feels instant.

## 2. React Internals

### Reconciliation Algorithm

**What it is:** When state changes, React needs to figure out the minimum set of DOM changes to make. Reconciliation is the diffing process that compares the old virtual DOM tree with the new one.

**Rules React uses:**

- Elements of different types → tear down and rebuild subtree
- Same type → update props in place
- Lists use `key` to match old and new items

**Example:** Changing `<div>` to `<span>` destroys all children. Changing a prop on `<div className="a">` to `<div className="b">` just patches the attribute.

## Fiber Architecture

**What it is:** React's internal reimplementation (React 16+) that represents every component as a "fiber" — a unit of work that can be paused, resumed, or aborted. This enables concurrent features.

**Analogy:** Old React was a phone call you couldn't interrupt. Fiber is like a to-do list where you can pause after each task and handle something urgent.

**Example:** A large render task (1000 list items) is split into fibers. React processes some fibers, checks if the user did something, handles that first, then continues.

## Virtual DOM Diffing Complexity

**What it is:** Naively diffing two trees is  $O(n^3)$ . React uses heuristics to reduce this to  $O(n)$  — but those heuristics have edge cases.

### The heuristics:

1. Different element types → full subtree replacement (no comparison)
2. Use `key` props to match list items across renders

**Pitfall:** Using array index as `key` breaks diffing when items are reordered. React matches by key, so `key=0` after a sort still maps to the first item in the new order.

## Concurrent Rendering

**What it is:** React can work on rendering multiple versions of the UI simultaneously, and can interrupt low-priority work to handle urgent updates.

**Example:** User types in a search box (high priority). React is also rendering a massive filtered list (low priority). With concurrent rendering, keystrokes remain responsive; the list render is deferred.

## Time Slicing

**What it is:** Breaking up a large synchronous render into small chunks (“slices”), yielding to the browser between each slice so the page stays responsive.

**Example:** Rendering 10,000 rows. Without time slicing: browser freezes for 500ms. With time slicing: React renders 50 rows, yields for 16ms (one frame), renders next 50, etc.

## Suspense Boundaries

**What it is:** A React component that catches “not ready yet” signals from child components and shows a fallback UI (like a spinner) until they resolve.

**Example:**

```
<Suspense fallback={<Spinner />}>
  <UserProfile /> {/* throws a Promise if data isn't ready */}
</Suspense>
```

UserProfile suspends while fetching. <Spinner> shows. When data arrives, UserProfile renders.

## Render Waterfalls

**What it is:** A chain of sequential data fetches where each fetch can only start after the previous one completes — because child components only mount after parents finish rendering.

**Example:**

```
App fetches user → renders UserProfile
  → UserProfile fetches posts → renders PostList
    → PostList fetches comments → renders Comments
```

Total wait = time(user) + time(posts) + time(comments). Fix: fetch all in parallel at the top level or use server components.

## Tearing in Concurrent UI

**What it is:** In concurrent mode, React can pause mid-render. If external state (e.g., a Zustand store) changes during that pause, different parts of the UI may read different values of the same data — rendering an inconsistent snapshot.

**Example:** A price display reads `$100` at the start of render. React pauses. Price updates to `$200`. React resumes and another component reads `$200`. UI shows both `$100` and `$200` simultaneously.

**Fix:** Use `useSyncExternalStore` which forces synchronous reads.

## Scheduler Priorities

**What it is:** React's internal scheduler assigns priorities to work. Higher priority tasks preempt lower priority ones.

### Priority levels (approx):

- `Immediate` — synchronous, can't be interrupted (e.g., controlled input)
- `UserBlocking` — user interactions (click, keypress)
- `Normal` — data fetching renders
- `Low` — analytics, non-visible updates
- `Idle` — background work

## 3. JavaScript Runtime & Memory

### Event Loop (Macro vs Microtasks)

**What it is:** JavaScript is single-threaded. The event loop is the mechanism that decides what runs next. Tasks are split into:

- **Macrotasks:** `setTimeout`, `setInterval`, I/O callbacks — run one per loop iteration
- **Microtasks:** `Promise.then`, `queueMicrotask`, `MutationObserver` — all run before the next macrotask

**Example:**

```
console.log('1');
setTimeout(() => console.log('2'), 0); // macrotask
Promise.resolve().then(() => console.log('3')); // microtask
console.log('4');
// Output: 1, 4, 3, 2
```

## Task Starvation

**What it is:** If microtasks keep scheduling more microtasks, the event loop never gets to process macrotasks (including rendering) — the browser freezes.

### Example:

```
function infiniteMicrotasks() {
  Promise.resolve().then(infiniteMicrotasks); // Never yields
}
infiniteMicrotasks(); // Browser hangs – rendering never happens
```

**Fix:** Use `setTimeout` or `scheduler.yield()` to intentionally yield to the browser.

## Stale Closure Problem

**What it is:** A function closes over a variable's value at the time it was created. If the variable updates later, the function still holds the old ("stale") value.

### Example:

```

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

  useEffect(() => {
    const id = setInterval(() => {
      // 'count' is always 0 – stale closure!
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
} // empty deps = closure captures count=0 forever

```

**Fix:** Use functional update `setCount(c => c + 1)` or include `count` in deps array.

## Memoization Pitfalls

**What it is:** `useMemo` / `useCallback` / `React.memo` cache values to avoid recomputation. But they come with costs and foot-guns.

### Pitfalls:

- Every memoized value adds memory overhead
- Deps arrays are manual — easy to miss a dep → stale data
- `React.memo` compares props shallowly — passing a new object literal each render still breaks it
- Over-memoizing increases complexity without real gains for cheap operations

### Example:

```

// Pointless – creating new object each render defeats React.memo on Child
<Child config={{ theme: 'dark' }} />

// Fix: memoize the object
const config = useMemo(() => ({ theme: 'dark' }), []);
<Child config={config} />

```

# Referential Equality

**What it is:** In JavaScript, two objects/arrays are only equal (`==`) if they are the exact same reference in memory, not if they have the same contents.

## Example:

```
const a = { x: 1 };
const b = { x: 1 };
a == b; // false – different references
a === a; // true – same reference
```

This is why `useEffect(() => {}, [obj])` re-runs every render if `obj` is recreated each time — it's always a new reference.

# Immutable Data Patterns

**What it is:** Never mutate existing data; always create new copies with changes. This makes change detection trivial (reference check) and prevents bugs from shared mutable state.

## Example:

```
// Mutable (bad) – React won't detect this change
state.items.push(newItem);

// Immutable (good) – new reference = React knows it changed
setState(prev => ({ ...prev, items: [...prev.items, newItem] }));
```

# Structural Sharing

**What it is:** When creating immutable updates, unchanged subtrees share the same memory reference — only the changed path gets new objects. Immer and immutable.js implement this.

**Example:** A state tree with 1000 nodes. You change one leaf. Structural sharing creates ~10 new objects (only the path from root to the changed leaf). The other 990 nodes are reused.

# Browser Memory Leak Detection

**What it is:** Finding memory that is allocated but never freed, causing the browser tab to use more and more RAM over time.

**Common causes:** forgotten event listeners, closures holding DOM references, timers not cleared, detached DOM nodes still referenced in JS.

**Detection:** Chrome DevTools → Memory → Take Heap Snapshot before and after an action. Look for objects that accumulate across snapshots.

## Detached DOM Nodes

**What it is:** A DOM element that has been removed from the document but is still referenced in JavaScript — so the garbage collector can't free it.

### Example:

```
let cache = [];
function addElement() {
  const el = document.createElement('div');
  document.body.appendChild(el);
  cache.push(el); // holds reference
  document.body.removeChild(el); // removed from DOM but not from cache
}
// el is now a detached DOM node - leaking memory
```

## Garbage Collection Timing

**What it is:** The JS engine's GC runs at unpredictable times. A GC pause can cause a visible frame drop (jank). You can't control when GC runs, but you can reduce allocation frequency.

**Best practice:** Avoid allocating objects in hot paths (animation loops, event handlers). Reuse objects instead of creating new ones.

# 4. Browser Rendering Pipeline

## Critical Rendering Path

**What it is:** The sequence of steps the browser must complete before anything appears on screen:  
Parse HTML → Build DOM → Parse CSS → Build CSSOM → Combine into Render Tree → Layout → Paint → Composite.

**Key insight:** Anything that blocks any of these steps delays first paint.

## Render Blocking Resources

**What it is:** CSS and synchronous `<script>` tags block the browser from rendering. The browser won't paint until all blocking resources are loaded and processed.

### Example:

```
←!— Blocks rendering until stylesheet downloads —→  
<link rel="stylesheet" href="styles.css">  
  
←!— Blocks HTML parsing + rendering —→  
<script src="app.js"></script>  
  
←!— Non-blocking: async/defer —→  
<script src="app.js" defer></script>
```

## Layout Thrashing

**What it is:** Alternating between reading and writing layout properties forces the browser to recalculate layout repeatedly (once per read-after-write). This can happen dozens of times in a loop.

### Example:

```

// THRASHING – forces layout recalculation every iteration
elements.forEach(el => {
  const height = el.offsetHeight; // READ – forces layout
  el.style.height = height + 10 + 'px'; // WRITE – invalidates layout
});

// FIX – batch reads, then writes
const heights = elements.map(el => el.offsetHeight); // all reads
elements.forEach((el, i) => el.style.height = heights[i] + 10 + 'px'); // all writes

```

## Paint vs Composite vs Layout

**What it is:** Three distinct phases of rendering with different costs:

- **Layout (Reflow):** Calculate position and size of every element. Most expensive.
- **Paint:** Fill pixels – colors, shadows, text. Medium cost.
- **Composite:** Layer the painted layers in the right order on the GPU. Cheapest.

**Performance rule:** Stick to properties that only trigger composite ( `transform` , `opacity` ). Avoid properties that trigger layout ( `width` , `height` , `top` , `margin` ).

## GPU Acceleration in CSS

**What it is:** Certain CSS properties promote an element to its own GPU layer, where animations can run without involving the CPU or triggering layout/paint.

**How to trigger:**

```

/* Explicit GPU layer promotion */
:animated {
  will-change: transform;
  transform: translateZ(0); /* classic "null transform hack" */
}

```

**Pitfall:** Too many GPU layers consume large amounts of GPU memory. Use sparingly.

# CSS Containment

**What it is:** Tells the browser that a component's subtree is independent — changes inside it can't affect layout, paint, or style outside it. Enables rendering optimizations.

## Example:

```
.card {  
  contain: layout style paint; /* or shorthand: contain: strict */  
}
```

Now changing content inside `.card` never forces the browser to recalculate layout for the rest of the page.

# Browser Compositing Layers

**What it is:** The browser splits the page into layers (like Photoshop layers), paints each separately, then composites them together on the GPU. Layers that move can be recomposed without repainting.

**Example:** A fixed navigation bar is on its own layer. When you scroll, the browser just moves the content layer — the nav layer isn't repainted, just repositioned during composite.

# Subpixel Rendering

**What it is:** Monitors have three sub-pixels per pixel (R, G, B). The browser can use these to render text and borders at sub-pixel precision, making them appear sharper.

**Relevance:** Fractional pixel values (e.g., `width: 100.5px`) can cause blurry rendering or inconsistent 1px borders across different screens. Always prefer whole pixel values for borders and positioned elements.

# 5. Performance & Loading

## Preload vs Prefetch vs Preconnect

**What it is:** Three resource hints that tell the browser to prepare resources ahead of time:

Hint	When	Use for
preload	Current page definitely needs it soon	Fonts, hero images, critical CSS
prefetch	Next navigation might need it	Next page's JS bundle
preconnect	Will connect to this origin soon	API domains, CDN origins

### Example:

```
<link rel="preload" href="hero.jpg" as="image">
<link rel="prefetch" href="/checkout.js">
<link rel="preconnect" href="https://api.example.com">
```

## Priority Hints

**What it is:** The `fetchpriority` attribute lets you tell the browser how important a resource is, overriding its default heuristics.

### Example:

```
←!— Boost LCP image – don't let browser think it's low priority —→


←!— Deprioritize below-the-fold images —→

```

## Largest Contentful Paint (LCP)

**What it is:** The time until the largest visible element (image, video, or text block) is fully rendered. Core Web Vital. Target: under 2.5 seconds.

**Common causes of poor LCP:** render-blocking resources, slow server response, unoptimized images, no `preload` on hero image.

## First Input Delay (FID) / Interaction to Next Paint (INP)

**FID:** Time from first user interaction (click, keypress) to when the browser can respond. Replaced by INP.

**INP:** Measures all interactions throughout the page lifetime, not just the first. Worst-case interaction latency percentile. Target: under 200ms.

**Common cause:** Long tasks on the main thread blocking input processing.

## Cumulative Layout Shift (CLS)

**What it is:** Measures unexpected layout shifts — elements that jump around as the page loads. Score is the sum of all unexpected shifts. Target: under 0.1.

**Classic cause:** Images without explicit `width` / `height` — browser doesn't reserve space, so content shifts when the image loads.

**Fix:**

```
>
```

## Tree Shaking Internals

**What it is:** Bundlers (Webpack, Rollup, esbuild) statically analyze ES module `import` / `export` graphs and remove code that is never imported. Dead code elimination.

**Requirement:** Must use ES modules (`import` / `export`). CommonJS (`require`) is not statically analyzable.

### Example:

```
// utils.js - only 'add' is imported, 'subtract' is tree-shaken out
export function add(a, b) { return a + b; }
export function subtract(a, b) { return a - b; }

// app.js
import { add } from './utils'; // subtract never makes it to bundle
```

## Code Splitting Strategies

**What it is:** Splitting the JS bundle into smaller chunks that are loaded on demand, rather than loading everything upfront.

### Strategies:

1. **Route-based:** Each route is a separate chunk — most common
2. **Component-based:** Lazy-load heavy components (`React.lazy`)
3. **Vendor splitting:** Separate `node_modules` from app code for better caching

## Dynamic Import Chunking

**What it is:** Using `import()` to dynamically load a module at runtime. The bundler creates a separate chunk for it automatically.

### Example:

```
// Modal only loads when user clicks "Open"
async function handleClick() {
  const { HeavyModal } = await import('./HeavyModal');
  // render modal
}
```

The `HeavyModal` code is in its own chunk — not downloaded until needed.

## Module Federation

**What it is:** A Webpack 5 feature that allows multiple independently deployed JavaScript applications to share modules at runtime — the foundation of many micro-frontend architectures.

**Example:** Shell app at `shell.example.com` dynamically loads the `Header` component from `nav-app.example.com` without bundling it. Both apps can be deployed independently.

## 6. Build Tools & Bundling

(Covered within Section 5 — *Tree Shaking, Code Splitting, Dynamic Imports, Module Federation*)

## 7. Web APIs & Browser Primitives

### IntersectionObserver Internals

**What it is:** Asynchronously observes when an element enters or leaves the viewport (or a specified root). Runs off the main thread — far cheaper than scroll event listeners.

**Example:**

```
const observer = new IntersectionObserver(entries => {
  entries.forEach(entry => {
    if (entry.isIntersecting) lazyLoadImage(entry.target);
  });
}, { threshold: 0.1 }); // fires when 10% visible

observer.observe(document.querySelector('.lazy-image'));
```

## ResizeObserver Loop Limits

**What it is:** `ResizeObserver` fires when an element's size changes. If your callback itself causes a resize (e.g., by setting `height`), you get an infinite loop. Browsers detect this and throw a `ResizeObserver loop limit exceeded` error.

**Fix:** Debounce the callback or use `requestAnimationFrame` to defer the write.

## MutationObserver Cost

**What it is:** `MutationObserver` watches for DOM changes (attribute changes, child additions, etc.). Watching large, frequently-changing subtrees is expensive.

**Best practice:** Be as specific as possible — narrow your `subtree`, limit `attributeFilter` to only the attributes you care about. Disconnect the observer when done.

## IndexedDB

**What it is:** A low-level, async, transactional key-value/object store in the browser. Can store large amounts of structured data (files, blobs, JSON). Survives page reloads and browser restarts.

**Use cases:** Offline data, draft saving, caching API responses locally.

**Note:** The raw API is verbose — use libraries like `idb` (a tiny Promise wrapper).

## Web Workers vs Service Workers

**What it is:** Two different kinds of background threads in the browser:

	Web Worker	Service Worker
Purpose	CPU-intensive computation	Network proxy / offline caching
Lifecycle	Lives while page is open	Persists after tab closes

	<b>Web Worker</b>	<b>Service Worker</b>
<b>DOM access</b>	None	None
<b>Network</b>	Can fetch	Intercepts all fetches
<b>Use case</b>	Image processing, crypto	PWA offline, push notifications

## SharedArrayBuffer

**What it is:** A fixed-length raw binary buffer shared between the main thread and workers without copying. Enables true shared memory between threads.

**Requirement:** Requires Cross-Origin-Isolated headers ( COOP + COEP ) due to Spectre mitigations.

**Example:** A Web Worker and the main thread both read/write the same `SharedArrayBuffer` , synchronized via `Atomics` .

## Transferable Objects

**What it is:** When sending data to a Web Worker, you normally copy it (expensive for large data). Transferable objects are transferred — ownership moves to the worker, the original becomes unusable. Zero-copy.

**Example:**

```
const buffer = new ArrayBuffer(1024 * 1024 * 100); // 100MB
worker.postMessage(buffer, [buffer]); // transferred – buffer is now empty here
```

## OffscreenCanvas

**What it is:** A `Canvas` that can be used off the main thread (in a Web Worker). Rendering-heavy canvas operations no longer block the UI.

**Example:** A game renders its WebGL scene in a Worker using `OffscreenCanvas`. The main thread handles input. UI never janks during heavy rendering.

## WebAssembly Integration

**What it is:** A binary instruction format that runs at near-native speed in the browser. You compile C/C++/Rust to `.wasm` and call it from JavaScript.

**Use cases:** Video codecs, image processing, physics engines, encryption.

**Example:**

```
const wasmModule = await WebAssembly.instantiateStreaming(fetch('decoder.wasm'));
const result = wasmModule.instance.exports.decodeFrame(frameData);
```

## WebRTC

**What it is:** Browser API for peer-to-peer real-time communication (video, audio, data) directly between browsers, without a media server.

**Components:**

- `RTCPeerConnection` — manages the P2P connection
- `RTCDataChannel` — arbitrary data (like WebSockets, but P2P)
- Signaling (via your server) — how peers find each other

**Use cases:** Video calls (Zoom-like), collaborative editing, file sharing.

## AbortController

**What it is:** A standard way to cancel async operations (fetch, event listeners, any async logic).

**Example:**

```
const controller = new AbortController();

fetch('/api/data', { signal: controller.signal });

// Cancel the request
controller.abort();
```

The fetch will reject with an `AbortError`. Works with `addEventListener` too.

## Backpressure in Streams API

**What it is:** When a data producer is faster than the consumer, backpressure signals the producer to slow down, preventing memory overflow.

**Example:** Streaming a 4GB file: if you read faster than you process, you buffer everything in RAM. The Streams API's `WritableStream` uses `desiredSize` to signal the readable side to pause.

## Streaming Fetch Response Handling

**What it is:** Instead of waiting for the full response, you can process `response.body` as a `ReadableStream` — chunk by chunk.

### Example:

```
const response = await fetch('/large-data');
const reader = response.body.getReader();

while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  processChunk(value); // Uint8Array chunk
}
```

Useful for LLM streaming responses, large file downloads, real-time logs.

## PerformanceObserver API

**What it is:** Observe performance entries (LCP, FID, layout shifts, long tasks, resource timings) as they happen, in production.

### Example:

```
new PerformanceObserver(list => {
  list.getEntries().forEach(entry => {
    if (entry.entryType === 'largest-contentful-paint') {
      console.log('LCP:', entry.startTime);
    }
  });
}).observe({ type: 'largest-contentful-paint', buffered: true });
```

## Long Tasks API

**What it is:** Notifies you whenever a task on the main thread takes longer than 50ms — these are the tasks that cause jank and poor INP scores.

### Example:

```
new PerformanceObserver(list => {
  list.getEntries().forEach(task => {
    console.log('Long task:', task.duration + 'ms', task.attribution);
  });
}).observe({ type: 'longtask', buffered: true });
```

## 8. Web Components & Shadow DOM

### Shadow DOM

**What it is:** An encapsulated DOM subtree attached to an element — its CSS and structure are isolated from the main document. Styles from outside don't leak in; styles inside don't leak out.

### Example:

```

const shadow = element.attachShadow({ mode: 'open' });
shadow.innerHTML =
  `
```

- Event bubbling across shadow DOM boundaries behaves differently
- SSR of shadow DOM requires Declarative Shadow DOM ( `<template shadowrootmode="open">` )

## 9. Concurrency & Async Patterns

### Race Conditions in UI State

**What it is:** When two async operations run concurrently and the slower one finishes last, overwriting the result of the faster one — leaving stale data in the UI.

#### Example:

```
// User types "A" → fetch for "A" starts
// User types "AB" → fetch for "AB" starts
// "AB" fetch finishes first, shows results
// "A" fetch finishes, overwrites with wrong results!
```

**Fix:** Cancel previous request with `AbortController` when a new one starts.

### Priority Inversion in Async Code

**What it is:** A high-priority task is blocked waiting for a low-priority task to complete.

**Example:** A critical UI update is awaiting a low-priority background data sync, which holds a lock. The UI freezes even though the critical work is “ready.” Use separate queues and avoid shared locks across priority levels.

### Deterministic Rendering

**What it is:** Given the same input/state, the component always produces the exact same output — no randomness, no side effects during render, no dependency on external mutable state.

**Why it matters:** Required for SSR hydration (server and client must produce identical HTML), testing, and time-travel debugging.

**Anti-pattern:** Reading `Date.now()` or `Math.random()` during render.

## Idempotent UI Actions

**What it is:** An action that can be safely repeated multiple times and always produces the same result. Especially critical for network requests.

**Example:** “Mark as read” — calling it 5 times should have the same effect as calling it once. Use `PUT` or `PATCH` (idempotent) instead of `POST` (not idempotent) for such actions.

# 10. Caching & Networking

## Service Worker Lifecycle Traps

**What it is:** Service Workers have a specific lifecycle (installing → waiting → activating → activated) that causes common bugs:

**Traps:**

- A new SW won’t activate while old tabs are still open (it waits)
- Calling `skipWaiting()` activates immediately — but risks breaking in-flight requests from the old SW
- `clients.claim()` takes control of existing pages without reload

## Cache Invalidation Strategies

**What it is:** Deciding when cached data is stale and should be refreshed. The hardest problem in caching.

**Common strategies:**

- **TTL (Time To Live):** Expire after N seconds
- **Versioning:** Change the URL (`app.v2.js`) — old URL is always “fresh”
- **ETag validation:** Ask server “is this still valid?” cheaply
- **Stale-while-revalidate:** Serve stale immediately, refresh in background

## Stale-While-Revalidate

**What it is:** A caching strategy where you immediately serve cached (possibly stale) content, while simultaneously fetching a fresh version in the background.

**Example ( Cache-Control header):**

```
Cache-Control: max-age=60, stale-while-revalidate=3600
```

Fresh for 60s. From 60s to 3660s, serve stale but revalidate in background. After 3660s, must revalidate.

## ETag vs Cache-Control

**What it is:** Two mechanisms for controlling HTTP caching:

- **Cache-Control :** Defines for how long a response can be cached locally (no server contact needed)
- **ETag :** A fingerprint of the resource. Client sends `If-None-Match: <etag>`. Server returns `304 Not Modified` if unchanged — saves bandwidth, but still requires a network round-trip

## HTTP/3 and QUIC

**What it is:** HTTP/3 uses QUIC (UDP-based) instead of TCP. Key improvements:

- **No head-of-line blocking:** Multiple streams are independent; a lost packet only blocks its own stream
- **0-RTT connection establishment:** Reconnect with no round-trip delay
- **Built-in TLS:** Security is mandatory and faster

**Impact:** Especially beneficial on mobile/lossy networks where packet loss is common.

## CORS Preflight

**What it is:** Before a cross-origin request with custom headers or non-simple methods (PUT, DELETE, etc.), the browser sends an `OPTIONS` request to ask the server if it allows the real request.

### Example:

```
OPTIONS /api/data HTTP/1.1
Origin: https://app.com
Access-Control-Request-Method: PUT
```

Server must respond with appropriate `Access-Control-Allow-*` headers or the real request is blocked — never even sent.

**Optimization:** Use `Access-Control-Max-Age` to cache preflight results.

## 11. Security

### SameSite Cookie Modes

**What it is:** Controls when cookies are sent with cross-site requests:

Mode	Cookies sent on...
Strict	Only same-site requests
Lax (default)	Same-site + top-level GET navigations (e.g., clicking a link)
None	All cross-site requests (requires <code>Secure</code> )

### CSRF vs XSS Mitigation

**CSRF (Cross-Site Request Forgery):** Attacker tricks the user's browser into making authenticated requests to your site from a different origin. **Mitigation:** `SameSite=Lax/Strict` cookies, CSRF tokens, `Origin / Referer` header validation.

**XSS (Cross-Site Scripting):** Attacker injects malicious JavaScript into your page that runs in other users' browsers. **Mitigation:** Content Security Policy, output encoding, Trusted Types , HttpOnly cookies (so stolen cookies can't be read by JS).

## Content Security Policy (CSP)

**What it is:** An HTTP header that tells the browser which sources of content (scripts, styles, images) are allowed. Blocks injected malicious scripts even if XSS occurs.

### Example:

```
Content-Security-Policy: default-src 'self'; script-src 'self'  
https://cdn.example.com
```

Any script not from self or cdn.example.com is blocked — including injected <script> tags.

## Trusted Types

**What it is:** A browser API that prevents DOM XSS by requiring that all dangerous sinks ( innerHTML , eval , document.write ) receive a “Trusted Type” object rather than a raw string.

### Example:

```
// Without Trusted Types: XSS vector  
el.innerHTML = userInput; // dangerous  
  
// With Trusted Types policy:  
const policy = trustedTypes.createPolicy('default', {  
  createHTML: input => DOMPurify.sanitize(input)  
});  
el.innerHTML = policy.createHTML(userInput); // safe
```

# DOM Clobbering

**What it is:** An attacker injects HTML with `id` or `name` attributes that collide with global JavaScript variables or DOM properties, overwriting them.

## Example:

```
←!— Injected by attacker →  
<form id="getElementById">...</form>  
  
←!— Now this is broken – document.getElementById is the form element! →  
document.getElementById('app'); // TypeError
```

**Mitigation:** CSP, DOMPurify , avoid relying on global named element access.

# Prototype Pollution

**What it is:** An attacker manipulates `Object.prototype` by exploiting unsafe recursive merge/clone functions. Since all objects inherit from `Object.prototype` , this affects every object in the app.

## Example:

```
// Unsafe deep merge  
merge({}, JSON.parse('{"__proto__": {"isAdmin": true}}'));  
  
// Now every object in the app has isAdmin = true  
({}).isAdmin; // true – prototype was polluted
```

**Mitigation:** Use `Object.create(null)` for sensitive objects, `JSON.parse` with validation, or `Object.freeze(Object.prototype)` .

# 12. State Management & Architecture

## Finite State Modeling

**What it is:** Modeling UI state as a finite state machine — a fixed set of states with explicit transitions between them. Eliminates impossible states.

**Example:** A fetch request has states: `idle → loading → success | error`. With FSM, you can never be in `loading` and `success` simultaneously. Libraries: XState.

## Event Sourcing in Frontend

**What it is:** Instead of storing current state, store every action/event that led to that state. State is derived by replaying events.

**Example:** A text editor stores `[{type:'insert', pos:0, char:'H'}, {type:'insert', pos:1, char:'i'}]` rather than just `"Hi"`. This enables undo/redo, audit logs, and time-travel debugging.

## Optimistic UI Rollback Strategy

**What it is:** Immediately update the UI as if an action succeeded, then roll back if the server request fails.

**Example:**

```
// 1. Immediately show the liked state
dispatch({ type: 'LIKE_POST', id: postId });

try {
  await api.likePost(postId);
} catch (e) {
  // 2. Server failed - roll back
  dispatch({ type: 'UNLIKE_POST', id: postId });
  showError('Like failed, please try again');
}
```

## Micro-Frontend Orchestration

**What it is:** Composing a single web application from multiple independently deployed frontend applications (micro-frontends), each owned by a separate team.

**Orchestration concerns:** Routing (which micro-frontend handles which URL), shared state (auth, user data), shared dependencies (avoid loading React twice), error isolation (one MFE crashing shouldn't break others), styling isolation.

## Offline Conflict Resolution

**What it is:** When a user edits data offline and syncs later, that data may conflict with changes others made while they were offline. You need a strategy to resolve conflicts.

### Strategies:

- **Last-write-wins:** Latest timestamp overrides (simple, lossy)
- **Server-wins / Client-wins:** One side always wins (simple but frustrating)
- **Three-way merge:** Compare base, server, and client versions (like Git)
- **CRDTs:** Data structures that merge automatically without conflicts

## CRDT Basics for Collaboration

**What it is:** Conflict-free Replicated Data Types — data structures designed to be merged automatically from any two states without conflicts, regardless of the order operations arrived.

**Example:** A “grow-only set” CRDT — you can only add items, never remove. Two clients can each add items offline; merging is simply the union of both sets — always correct.

**Real-world use:** Figma, Notion, and most collaborative editors use CRDTs or OT (Operational Transformation) for real-time sync.

## 13. Real-Time & Collaboration

(Covered: WebRTC, CRDT, Event Sourcing, Offline Conflict Resolution — see above)

# 14. Performance Metrics (Core Web Vitals)

(Covered: LCP, FID, INP, CLS — see Section 5)

## Speculative Prerendering

(Covered in Section 1)

# 15. Observability & Debugging

(Covered: PerformanceObserver, Long Tasks API, Memory Leak Detection, Detached DOM Nodes, GC Timing — see Sections 5 & 3)

# 16. Accessibility

## Accessibility Tree

**What it is:** A parallel tree the browser builds from the DOM, exposing semantic information (role, name, state) to assistive technologies like screen readers via the OS accessibility APIs.

**Example:** `<button>Submit</button>` becomes an accessibility node with `role=button`, `name="Submit"`, `state=enabled`. A `<div>` with `onclick` has no semantic role — invisible to screen readers unless you add `role="button"` and `tabindex`.

## ARIA Live Regions Internals

**What it is:** HTML regions marked with `aria-live` that instruct screen readers to announce content changes automatically, without the user navigating to them.

### Modes:

- `aria-live="polite"` — announces after current speech finishes

- `aria-live="assertive"` — interrupts immediately (use sparingly — very disruptive)

**Internals:** The browser watches the live region subtree with a `MutationObserver`-like mechanism and queues announcements.

**Pitfall:** Inserting new elements into a live region works. Updating `textContent` works. Moving an existing live region in the DOM may reset it — some screen readers stop tracking.

## Pointer Events

**What it is:** A unified input event model that handles mouse, touch, and stylus through a single event API (`pointerdown`, `pointermove`, `pointerup`, `pointercancel`).

**Why it matters:** Replaces the messy dual `mouse*` + `touch*` event handling with one consistent API.

### Example:

```
element.addEventListener('pointerdown', e => {
  console.log(e.pointerType); // "mouse", "touch", or "pen"
  console.log(e.pressure);   // 0-1 for stylus
});
```

Use `touch-action: none` on elements that handle pointer events to prevent the browser's default scroll/zoom behavior.

*End of document. — Generated for Frontend Engineer interviews.*