Front-End Developer

Interview Guide

Prepared from the perspective of a Hiring Manager at a Top IT Company

Covers HTML, CSS, JavaScript, React, Performance, Accessibility, System Design & Behavioral Questions

1. HTML Fundamentals

Q1. What is semantic HTML and why does it matter?

Semantic HTML uses tags that convey meaning about their content (e.g., <header>, <article>, <nav>, <footer>) instead of generic tags like or . It improves accessibility for screen readers, helps SEO because search engines understand page structure better, and makes code more maintainable for other developers.

Q2. What is the difference between <script>, <script async>, and <script defer>?

- <script>: Blocks HTML parsing until the script downloads and executes.
- <script async>: Downloads in parallel with HTML parsing and executes as soon as it's ready (order not guaranteed with other async scripts).
- <script defer>: Downloads in parallel but executes only after HTML parsing completes, in order of appearance.

Q3. What are data attributes and when would you use them?

data-* attributes let you store custom data directly on HTML elements (e.g., data-user-id="123"). They're useful for attaching metadata to DOM elements that JavaScript can read via element.dataset.userId, without polluting class names or IDs.

Q4. What is the difference between localStorage, sessionStorage, and cookies?

- localStorage: Persists indefinitely, ~5-10MB, accessible only via JavaScript, not sent to server.
- sessionStorage: Persists only for the tab session, cleared on tab close, same size limits.
- Cookies: Small (~4KB), sent with every HTTP request, can have expiration dates, used for authentication/session tracking, and can be marked HttpOnly for security.

2. CSS Fundamentals

Q5. Explain the CSS Box Model.

Every element is a rectangular box composed of, from inside out: content, padding, border, and margin. By default (box-sizing: content-box), width/height apply only to content, so padding and border add to total size. box-sizing: border-box makes width/height include padding and border, which is why most developers set it globally.

Q6. What is the difference between flexbox and CSS grid?

Flexbox is one-dimensional (row OR column) — ideal for aligning items in a single line, like navbars or button groups. Grid is two-dimensional (rows AND columns simultaneously) — ideal for full page layouts or complex component structures. Many real layouts use both together.

Q7. Explain CSS specificity and how conflicts are resolved.

Specificity determines which CSS rule applies when multiple rules target the same element. Rough order (lowest to highest): element selectors < class/attribute/pseudo-class selectors < ID selectors < inline styles < !important. Equal specificity means the last rule declared wins.

Q8. What is the difference between relative, absolute, fixed, and sticky positioning?

- relative: Positioned relative to its normal position; still occupies original space.
- absolute: Positioned relative to the nearest positioned ancestor (or the document if none); removed from normal flow.
- fixed: Positioned relative to the viewport; stays in place on scroll.
- sticky: Acts as relative until a scroll threshold is met, then behaves like fixed.

Q9. What are CSS pre-processors (SASS/LESS) and why use them?

They extend CSS with variables, nesting, mixins, functions, and imports, which are compiled down to plain CSS. They improve maintainability and reduce repetition, though native CSS now supports variables (--custom-property) and nesting too.

Q10. How do you make a website responsive?

Using fluid grids/flexbox/grid layouts, relative units (%, em, rem, vw/vh) instead of fixed pixels, media queries to adapt layout at breakpoints, and a proper <meta name="viewport"> tag. Mobile-first design (writing base styles for mobile, then adding complexity for larger screens) is the modern best practice.

3. JavaScript Core Concepts

Q11. Explain the difference between var, let, and const.

- var: Function-scoped, hoisted with undefined initialization, can be redeclared.
- let: Block-scoped, hoisted but not initialized (temporal dead zone), can be reassigned but not redeclared in the same scope.
- const: Block-scoped, cannot be reassigned (though object/array contents can still be mutated).

Q12. What is a closure in JavaScript? Give an example.

A closure is a function that retains access to variables from its enclosing lexical scope even after that outer function has returned.

```
function counter() {
  let count = 0;
  return function () {
    count++;
    return count;
  };
}
const increment = counter();
increment(); // 1
increment(); // 2
```

The inner function "remembers" count even after counter() finishes executing.

Q13. Explain the JavaScript Event Loop.

JavaScript is single-threaded and uses a call stack to execute code. Asynchronous operations (timers, network requests) are handed off to Web APIs, and their callbacks go into a task queue (or microtask queue for Promises). The event loop continuously checks: if the call stack is empty, it pushes the next queued callback onto the stack. Microtasks (Promises) are processed before macrotasks (setTimeout).

Q14. What is the difference between == and ===?

== performs type coercion before comparison (e.g., "5" == 5 is true). === (strict equality) compares both value and type without coercion ("5" === 5 is false). Best practice is to always use === unless coercion is explicitly intended.

Q15. Explain this keyword behavior in JavaScript.

this refers to the object that is executing the current function, and its value depends on how a function is called:

- In a regular function call, this is undefined (strict mode) or the global object.
- As an object method, this refers to the object.
- Arrow functions don't have their own this — they inherit it from the enclosing lexical scope.
- With call, apply, bind, this can be explicitly set.

Q16. What are Promises and how do they differ from callbacks/async-await?

A Promise represents an eventual result of an asynchronous operation, with states: pending, fulfilled, or rejected. They solve "callback hell" by allowing chaining via .then()/.catch(). async/await is syntactic sugar over Promises, letting asynchronous code read like synchronous code.

```
async function getUser() {
  try {
    const response = await fetch('/api/user');
    const data = await response.json();
    return data;
  } catch (error) {
```

```
    console.error(error);
  }
}
```

Q17. What is event delegation and why is it useful?

Event delegation attaches a single event listener to a parent element instead of individual listeners on each child, leveraging event bubbling. It's useful for performance (fewer listeners) and for handling dynamically added elements.

```
document.getElementById('list').addEventListener('click', (e) => {
  if (e.target.tagName === 'LI') {
    console.log('Clicked:', e.target.textContent);
  }
});
```

Q18. Explain hoisting in JavaScript.

Hoisting is JavaScript's behavior of moving variable and function declarations to the top of their scope before code execution. var declarations are hoisted and initialized as undefined. let/const are hoisted but remain uninitialized (temporal dead zone) until their declaration line executes. Function declarations are fully hoisted (including their body); function expressions are not.

Q19. What is the difference between deep copy and shallow copy?

A shallow copy duplicates only the top-level properties; nested objects/arrays still reference the same memory (e.g., Object.assign({}, obj) or spread {...obj}). A deep copy duplicates all nested levels recursively, so no shared references remain (e.g., structuredClone(obj) or JSON.parse(JSON.stringify(obj)), though the latter loses functions/dates).

Q20. Explain debouncing vs throttling.

- Debouncing: Delays function execution until a certain time has passed since the last call — useful for search-as-you-type inputs.
- Throttling: Ensures a function runs at most once in a specified time interval, regardless of how many times it's triggered — useful for scroll or resize handlers.

4. React (or Framework-Specific) Questions

Q21. What is the Virtual DOM and how does it improve performance?

The Virtual DOM is a lightweight in-memory representation of the real DOM. When state changes, React creates a new virtual tree, diffs it against the previous one (reconciliation), and updates only the changed parts of the real DOM — avoiding costly full re-renders.

Q22. Explain the difference between state and props.

- Props: Read-only data passed from a parent component to a child; the child cannot modify them.
- State: Data managed internally within a component that can change over time and triggers re-renders when updated.

Q23. What are React Hooks? Name a few commonly used ones.

Hooks let you use state and other React features in functional components without writing classes.

- useState: manage local state.
- useEffect: handle side effects (data fetching, subscriptions, DOM manipulation).
- useContext: consume context values without prop drilling.
- useMemo/useCallback: memoize values/functions for performance.
- useRef: persist mutable values across renders without causing re-renders.

Q24. What is the difference between controlled and uncontrolled components?

A controlled component has its form data managed by React state (value + onChange). An uncontrolled component manages its own state internally in the DOM, accessed via a ref when needed. Controlled components give more predictable, testable behavior.

Q25. What is prop drilling and how do you avoid it?

Prop drilling is passing data through multiple layers of components that don't need it, just to reach a deeply nested child. It can be avoided using the Context API, or state management libraries like Redux, Zustand, or Recoil.

Q26. Explain the React component lifecycle (or useEffect equivalent).

Class components have lifecycle methods: componentDidMount, componentDidUpdate, componentWillUnmount. In functional components, useEffect replicates all three:

```
useEffect(() => {
  // runs like componentDidMount + componentDidUpdate
  return () => {
    // cleanup, like componentWillUnmount
  };
}, [dependencies]);
```

Q27. What is memoization in React, and when should you use useMemo/React.memo?

Memoization caches the output of expensive computations or component renders so they aren't recalculated unnecessarily. useMemo memoizes a computed value; useCallback memoizes a function reference; React.memo prevents a component from re-rendering if its props haven't changed. Overusing them can add unnecessary complexity, so they should be used only when profiling shows a real performance issue.

Q28. What are keys in React lists and why are they important?

Keys help React identify which items have changed, been added, or removed in a list, enabling efficient re-rendering. Keys should be stable and unique (like an ID), not array indexes, especially if the list order can change.

5. Performance & Optimization

Q29. How would you improve the performance of a web application?

- Minify and bundle CSS/JS, use code-splitting and lazy loading.
- Optimize images (compression, modern formats like WebP, responsive srcset).
- Use a CDN for static assets.
- Reduce render-blocking resources; use async/defer.
- Implement caching strategies (browser cache, service workers).
- Avoid layout thrashing and minimize reflows/repaints.
- Use tools like Lighthouse, WebPageTest to identify bottlenecks.

Q30. What is code splitting and lazy loading?

Code splitting breaks a large JavaScript bundle into smaller chunks that load on demand rather than all at once. Lazy loading defers loading of non-critical resources (images, components, routes) until they're actually needed, improving initial page load time.

```
const LazyComponent = React.lazy(() => import('./MyComponent'));
```

Q31. What is the Critical Rendering Path?

It's the sequence of steps the browser takes to convert HTML, CSS, and JS into pixels on screen: parsing HTML into the DOM, parsing CSS into the CSSOM, combining them into the Render Tree, computing Layout, and then Painting. Optimizing this path (reducing blocking resources, minimizing DOM depth) speeds up perceived load time.

6. Accessibility & Best Practices

Q32. What is web accessibility (a11y) and why is it important?

Accessibility ensures websites are usable by people with disabilities (visual, auditory, motor, cognitive). It's important for inclusivity, legal compliance (e.g., WCAG, ADA), and often improves usability and SEO for everyone. Key practices:

semantic HTML, proper alt text, keyboard navigability, sufficient color contrast, and ARIA attributes when native semantics aren't enough.

Q33. What is CORS and why does it matter?

Cross-Origin Resource Sharing is a browser security mechanism that restricts web pages from making requests to a different domain than the one that served the page, unless the server explicitly allows it via response headers like Access-Control-Allow-Origin. It prevents malicious sites from making unauthorized requests using a user's credentials.

Q34. What is XSS and how do you prevent it?

Cross-Site Scripting is an attack where malicious scripts are injected into trusted websites, often through unsanitized user input. Prevention includes: escaping/sanitizing user input before rendering, using textContent instead of innerHTML, Content Security Policy (CSP) headers, and relying on frameworks (React, Angular) that escape output by default.

7. System Design / Practical Questions

Q35. How would you design a reusable component library?

Discuss: consistent design tokens (colors, spacing, typography), prop-based configurability, accessibility built-in, documentation (e.g., Storybook), versioning/publishing via npm, and testing (unit + visual regression).

Q36. How would you optimize a page with a large list (e.g., 10,000 items)?

Use virtualization/windowing (e.g., react-window or react-virtualized) to render only the visible items in the viewport, recycling DOM nodes as the user scrolls, instead of rendering all 10,000 elements at once.

Q37. How do you handle API error states and loading states in the UI?

Maintain explicit state variables (loading, error, data), show skeleton loaders or spinners during fetch, display user-friendly error messages with retry options, and avoid leaving the UI in an ambiguous state. Using patterns like React Query/SWR can simplify this with built-in caching, retries, and stale-while-revalidate behavior.

Q38. Walk me through what happens when you type a URL into the browser and press Enter.

1. Browser checks cache/DNS resolution to get the server's IP address.
2. TCP handshake (and TLS handshake if HTTPS) establishes a connection.
3. Browser sends an HTTP request; server responds with HTML.
4. Browser parses HTML into the DOM, fetching CSS/JS along the way.
5. CSSOM is built and combined with the DOM into a render tree.
6. Layout and paint occur, and the page becomes visible.
7. JavaScript executes, potentially fetching more data or altering the DOM.

8. Behavioral / Soft Skill Questions (Common in Top Companies)

Q39. Tell me about a challenging bug you fixed. How did you debug it?

(Answer should demonstrate a structured approach: reproducing the issue, isolating variables, using dev tools/logging, forming hypotheses, and validating the fix — plus what was learned.)

Q40. How do you keep your front-end skills up to date?

(Look for genuine engagement: following release notes for frameworks, reading engineering blogs, contributing to open source, building side projects, attending meetups/conferences.)

Q41. Describe a time you disagreed with a teammate on a technical approach. How did you resolve it?

(Look for collaboration, willingness to back decisions with data/prototypes, and respect for differing opinions — not just "I was right.")

Interview Tips for Candidates

- Always explain your reasoning out loud, not just the final answer.
- For coding questions, clarify requirements and edge cases before jumping into code.
- Be honest if you don't know something — offer how you'd find out.
- Relate answers to real projects you've worked on wherever possible.