

Frontend Developer Interview Questions and Answers

Frontend | Intern to senior level

Frontend developer interview questions and answers covering JavaScript, React, the browser, performance and state management, from fresher to senior level.

47 questions Read online: <https://www.fredevtemplates.com/interview-questions/role/frontend>

JavaScript

1. What is the difference between var, let and const in JavaScript?

Level: intern JavaScript

var is function-scoped and can be redeclared; let and const are block-scoped, and const cannot be reassigned after it is set.

var is function-scoped, is hoisted and initialised to undefined, and can be redeclared in the same scope. let and const are block-scoped: they only exist inside the nearest pair of curly braces.

let and const are also hoisted, but they stay in the temporal dead zone until the line where they are declared, so using them earlier throws a ReferenceError instead of silently giving undefined.

const prevents reassigning the variable, not mutating the value. A const array or object can still be changed by pushing to it or setting a property. Use const by default, let when you need to reassign, and avoid var in new code.

Key points

var: function scope, hoisted as undefined, can be redeclared

let: block scope, temporal dead zone, cannot be redeclared in the same scope

const: like let, but the binding cannot be reassigned

```
if (true) {  
  var a = 1;  
  let b = 2;  
}  
console.log(a); // 1 (var ignores the block)  
console.log(b); // ReferenceError: b is not defined  
  
const list = [1, 2];  
list.push(3); // allowed, the array is mutated  
list = []; // TypeError: Assignment to constant variable
```

2. What is the difference between == and === in JavaScript?

Level: intern JavaScript

== compares values after type coercion, while === compares both value and type with no conversion, so it is the safer default.

The loose equality operator == converts the operands to a common type before comparing, which leads to surprising results such as 0 == "" and null == undefined both being true. The strict equality operator === returns false immediately if the types differ.

Most teams enforce === with a lint rule. The one commonly accepted use of == is value == null, which checks for both null and undefined in one expression.

Neither operator compares objects by content. Two separate objects or arrays with identical contents are never equal, because both operators compare references.

Key points

`==` checks type and value, `===` coerces first

Prefer `===` everywhere except deliberate null checks

Objects are compared by reference, not by content

```
0 == ""; // true (both coerced to 0)
0 === ""; // false (number vs string)
null == undefined; // true
null === undefined; // false
NaN === NaN; // false (use Number.isNaN)
[1] === [1]; // false (different references)
```

3. What is hoisting in JavaScript?

Level: intern JavaScript

Hoisting is JavaScript moving declarations to the top of their scope during compilation, so functions and var variables can be referenced before the line where they are written.

Before running code, the engine processes declarations in each scope. Function declarations are hoisted with their full body, so you can call them before they appear. var variables are hoisted and set to undefined, but their assignment stays where you wrote it.

let, const and class declarations are hoisted too, but not initialised. Accessing them before the declaration throws a ReferenceError (the temporal dead zone). Function expressions assigned to a variable follow the variable's rules, not the function declaration rules.

Key points

Declarations are hoisted, assignments are not

Function declarations are available before their definition

let and const exist but cannot be used until declared

```
sayHi(); // "hi" - function declaration is fully hoisted
console.log(x); // undefined - var is hoisted, value is not
console.log(y); // ReferenceError - let is in the temporal dead zone

function sayHi() { console.log("hi"); }
var x = 5;
let y = 10;

notYet(); // TypeError: notYet is not a function
var notYet = function () {};
```

4. What is the difference between null and undefined in JavaScript?

Level: intern JavaScript

undefined means a variable has been declared but no value was assigned; null is an explicit value meaning 'no value' that you assign on purpose.

JavaScript uses undefined automatically: for uninitialised variables, missing function arguments, missing object properties and functions with no return statement. null is something a developer sets deliberately to say that a value is intentionally empty.

typeof undefined is 'undefined', while typeof null is 'object', a long-standing bug in the language that cannot be fixed without breaking the web. Use value == null to test for both, or optional chaining and nullish coalescing (user?.name ?? 'Guest') to handle them safely.

```
let a; // undefined
const b = null; // null
typeof a; // "undefined"
typeof b; // "object" (historical bug)
a == b; // true
a === b; // false
const name = user?.name ?? "Guest";
```

5. What is the difference between map, filter and reduce in JavaScript?

Level: intern JavaScript

map transforms every item into a new array of the same length, filter keeps only items that pass a test, and reduce folds the whole array into a single value.

All three are array methods that do not mutate the original array (when your callback is pure). map returns an array with the same number of items. filter returns a shorter or equal array. reduce takes an accumulator and returns anything: a number, an object, even another array.

They chain well: filter first to reduce the work, then map, then reduce. Forgetting the initial value in reduce is a common bug: on an empty array it throws a `TypeError`.

Key points

map: one output per input

filter: keeps items where the callback returns truthy

reduce: always pass an initial value

```
const orders = [{ price: 20, paid: true }, { price: 35, paid: false }, { price: 15, paid: true }];

const prices = orders.map(o => o.price); // [20, 35, 15]
const paid = orders.filter(o => o.paid); // 2 orders
const total = orders
  .filter(o => o.paid)
  .reduce((sum, o) => sum + o.price, 0); // 35
```

6. What is a closure in JavaScript and why is it useful?

Level: junior JavaScript

A closure is a function that keeps access to the variables of the scope where it was created, even after that outer function has finished running.

Every function in JavaScript carries a reference to the lexical environment it was defined in. When an inner function is returned or passed elsewhere, that environment stays alive for as long as the function can be called, so the inner function can still read and update the outer variables.

Closures give you private state without classes, power patterns such as memoization, debounce and once-only functions, and make callbacks and React hooks work. The trade-off is memory: anything captured by a closure cannot be garbage collected while the closure is reachable.

Key points

A closure = a function plus its surrounding scope

Used for private state, factories, callbacks and memoization

Captured variables stay in memory while the closure is reachable

```
function createCounter() {
  let count = 0; // private, not reachable from outside
  return function () {
    count += 1;
    return count;
  };
}
```

```
const next = createCounter();
next(); // 1
next(); // 2
```

7. How does the this keyword work in JavaScript?

Level: junior JavaScript

this is decided by how a function is called, not where it is written: as a method it is the object before the dot, with new it is the new object, and arrow functions inherit it from the enclosing scope.

For a normal function there are four rules, applied in this order: new Foo() makes this the newly created object; call, apply or bind set it explicitly; obj.method() makes this the object before the dot; otherwise it is undefined in strict mode (or the global object in sloppy mode).

Arrow functions do not have their own this. They use the this of the scope they were defined in, which is why they are handy for callbacks inside methods. The classic bug is passing obj.method as a callback: the call site changes, so this is lost.

Key points

this depends on the call site

call, apply and bind set this explicitly

Arrow functions capture this lexically and cannot be rebound

```
const user = {
  name: "Asha",
  regular() { return this.name; },
  arrow: () => typeof this, // "this" is NOT user here
};

user.regular(); // "Asha"
const detached = user.regular;
detached(); // undefined (or TypeError) - lost context
detached.call({ name: "Ravi" }) // "Ravi"
user.regular.bind(user)(); // "Asha"
```

8. What is the difference between promises and async/await in JavaScript?

Level: junior JavaScript

async/await is syntax built on top of promises that lets you write asynchronous code that reads like synchronous code; under the hood every async function returns a promise.

A promise represents a value that will be available later and is in one of three states: pending, fulfilled or rejected. You consume it with .then() and .catch(). async/await lets you pause an async function with await until a promise settles, and handle errors with ordinary try/catch.

They are equivalent in power, but await makes sequential logic and error handling much easier to read. A common mistake is awaiting independent calls one after another. Use Promise.all to run them in parallel.

Key points

async functions always return a promise

await only pauses the async function, not the whole thread

Use Promise.all for independent work

```
// promises
fetch("/api/user").then(r => r.json()).then(showUser).catch(handleError);

// async/await
async function load() {
  try {
    const res = await fetch("/api/user");
    showUser(await res.json());
  } catch (err) {
    handleError(err);
  }
}

// run independent calls in parallel
const [user, posts] = await Promise.all([getUser(), getPosts()]);
```

9. What is event delegation in JavaScript?

Level: junior JavaScript

Event delegation means attaching one listener to a parent element and using event bubbling to handle events from its children, instead of adding a listener to each child.

When you click a child element, the event bubbles up through its ancestors. A single listener on the parent can inspect `event.target` (or use `event.target.closest(selector)`) to find which child was clicked and react accordingly.

This uses less memory for long lists and also works for elements added later, because the listener lives on the parent. It does not work for events that do not bubble, such as focus and blur (use `focusin` and `focusout` instead).

```
document.querySelector("#todo-list").addEventListener("click", (event) => {
  const item = event.target.closest("li[data-id]");
  if (!item) return;
  removeTodo(item.dataset.id);
});
```

10. What is the event loop in JavaScript and how does it work?

Level: mid JavaScript

The event loop is the mechanism that lets single-threaded JavaScript handle asynchronous work: it runs the call stack to empty, then runs all queued microtasks (promises), then takes the next task (timers, events) from the queue.

JavaScript runs one piece of code at a time on the call stack. Async operations such as timers, network requests and DOM events are handled by the browser (or Node.js), which places a callback in a queue when they finish. The event loop waits until the call stack is empty and then moves the next callback onto it.

There are two queues that matter. The macrotask queue holds timers, I/O and UI events. The microtask queue holds promise callbacks and `queueMicrotask`. After every task, the engine drains the entire microtask queue before taking the next macrotask, which is why a resolved promise runs before a `setTimeout` of 0.

A long synchronous task blocks everything, including rendering, because nothing else can run until the stack is empty. Splitting heavy work into chunks, using web workers, or yielding to the event loop keeps the page responsive.

Key points

One call stack, many queues

Microtasks (promises) run before the next macrotask (`setTimeout`)

Blocking the stack blocks rendering and input

Synchronous code first, then microtasks, then timers.

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

11. What is prototypal inheritance in JavaScript?

Level: mid JavaScript

In JavaScript every object has an internal link to another object called its prototype, and property lookups walk up this prototype chain until the property is found or the chain ends.

When you read `obj.name`, the engine checks `obj` first, then `obj`'s prototype, then that object's prototype, and so on until it reaches `null`. Methods shared by all instances live on the prototype instead of being copied onto each object, which saves memory.

The `class` keyword is syntax over this mechanism. `class B extends A` sets up `B.prototype` to inherit from `A.prototype`. You can create an object with a chosen prototype using `Object.create(proto)`, and inspect it with `Object.getPrototypeOf(obj)`.

Key points

Lookups follow the prototype chain

`class` is syntactic sugar over prototypes

Shared methods live on the prototype, not on each instance

```
const animal = { speak() { return this.name + " makes a sound"; } };
const dog = Object.create(animal);
dog.name = "Rex";
dog.speak(); // "Rex makes a sound"
Object.getPrototypeOf(dog) === animal; // true

class Cat {
  constructor(name) { this.name = name; }
  speak() { return this.name + " meows"; } // lives on Cat.prototype
}
```

12. What is the difference between debounce and throttle in JavaScript?

Level: mid JavaScript

Debounce waits until the user stops triggering an event for a set time before running the function; throttle runs the function at most once per time interval no matter how often the event fires.

Use `debounce` when only the final value matters, such as a search box that should call the API after the user pauses typing. Use `throttle` when you want regular updates during continuous activity, such as scroll position tracking or window resize handling.

Both are built with closures and timers. In `React`, make sure the debounced function is created once (for example with `useMemo` or `useRef`), otherwise every render creates a new timer and the debounce does nothing.

Key points

Debounce: run after the burst ends

Throttle: run at a steady maximum rate

Both rely on closures to keep the timer between calls

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}

function throttle(fn, limit) {
  let waiting = false;
  return (...args) => {
    if (waiting) return;
    fn(...args);
    waiting = true;
    setTimeout(() => { waiting = false; }, limit);
  };
}
```

13. How do you find and fix memory leaks in JavaScript?

Level: senior JavaScript

Take heap snapshots in Chrome DevTools before and after the suspect action, compare what grew, and fix the retaining reference, which is usually a forgotten listener, timer, global variable or unbounded cache.

JavaScript uses garbage collection based on reachability: an object is freed only when nothing reachable still references it. A leak is memory that your app no longer needs but something still references.

The usual causes are event listeners that are never removed, setInterval callbacks that are never cleared, detached DOM nodes held in a variable, accidental globals, closures that capture large objects, and caches or maps that only grow. In React, a missing cleanup function in useEffect is the most common source.

To diagnose, open the DevTools Memory tab, take a snapshot, repeat the action several times, take another and use the comparison view to see which object types keep growing. The retainers panel shows the chain of references keeping an object alive. Fixes are to remove listeners and timers in cleanup code, use WeakMap or WeakRef for caches keyed by objects, and put a size limit or eviction policy on caches.

Key points

Leak = unneeded but still reachable

Compare heap snapshots to see what keeps growing

Always pair listeners and timers with cleanup

```
useEffect(() => {
  const onResize = () => setWidth(window.innerWidth);
  window.addEventListener("resize", onResize);
  const id = setInterval(refresh, 5000);

  return () => { // cleanup prevents the leak
    window.removeEventListener("resize", onResize);
    clearInterval(id);
  };
}, []);
```

React

14. What is the virtual DOM in React and how does it work?

Level: intern React

The virtual DOM is a lightweight JavaScript copy of the UI that React compares with the previous version after each update, so it only changes the parts of the real DOM that actually differ.

When state or props change, React re-runs your component to produce a new tree of React elements (the virtual DOM). It then compares that tree with the previous one in a process called reconciliation, works out the minimal set of changes, and applies only those to the real DOM.

Touching the real DOM is comparatively expensive, and batching updates lets React do it once per update instead of once per change. The virtual DOM is not automatically faster than hand-written DOM code, but it gives you a declarative model: you describe what the UI should look like for the current state, and React figures out how to get there.

Key points

Virtual DOM = tree of plain objects describing the UI

Reconciliation compares old and new trees (diffing)

Only the differences are applied to the real DOM

15. What is the difference between props and state in React?

Level: intern React

Props are read-only inputs passed from a parent component, while state is data a component owns and can change over time, which triggers a re-render when updated.

Props make a component reusable: the parent decides the values, and the child must not modify them. State is private memory for a component, created with `useState` or `useReducer`. Updating state tells React to re-render that component with the new value.

A good rule is to keep state as low as possible in the tree, and lift it up to the closest common parent only when two components need to share it. Data that can be calculated from props or state should be computed during render and not stored in extra state.

Key points

Props flow down, are read-only and come from the parent

State is owned by the component and changes cause re-renders

Derived values should be calculated, not stored in state

```
function Counter({ step }) { // step is a prop (read-only)
  const [count, setCount] = useState(0); // count is state (owned here)
  return <button onClick={() => setCount(count + step)}>{count}</button>;
}

<Counter step={5} />
```

16. What are React hooks and what are the rules of hooks?

Level: junior React

Hooks are functions such as `useState` and `useEffect` that let function components use state, lifecycle behaviour and other React features; they must be called at the top level of a component or custom hook, never inside loops, conditions or nested functions.

Before hooks, state and lifecycle methods required class components. Hooks let you do the same in function components and, importantly, let you extract and reuse stateful logic as custom hooks without wrapper components or render props.

React tracks hooks by the order in which they are called during each render. If a hook is skipped in one render because of an if statement, every hook after it would read the wrong state. That is why the two rules exist: only call hooks at the top level, and only call them from React function components or other custom hooks (whose names start with `use`). The `eslint-plugin-react-hooks` package enforces both.

Key points

Call hooks at the top level only

Call hooks only from components or custom hooks

React relies on call order to match state to hooks

```
function useWindowWidth() { // custom hook
const [width, setWidth] = useState(window.innerWidth);
useEffect(() => {
const onResize = () => setWidth(window.innerWidth);
window.addEventListener("resize", onResize);
return () => window.removeEventListener("resize", onResize);
}, []);
return width;
}
```

17. How does useEffect work in React and what is the dependency array?

Level: junior React

useEffect runs a function after the component renders; the dependency array tells React when to re-run it: with no array it runs after every render, with an empty array only after the first, and with values only when those values change.

Effects are for synchronising your component with something outside React, such as subscriptions, timers, network requests or the document title. The function you return from an effect is the cleanup, and React runs it before the next run of the effect and when the component unmounts.

Every value from the component that the effect uses should be listed in the dependency array, otherwise the effect works with stale values. In development with Strict Mode, React deliberately mounts, unmounts and re-mounts components once to help you find missing cleanup. Do not use effects for things you can calculate during render or handle in an event handler.

Key points

No array: every render. [] : once. [a, b]: when a or b change

Return a function to clean up

Do not use effects for derived data or user-triggered actions

```
useEffect(() => {
const controller = new AbortController();
fetch("/api/users/" + userId, { signal: controller.signal })
.then(r => r.json())
.then(setUser)
.catch(err => { if (err.name !== "AbortError") setError(err); });

return () => controller.abort(); // cleanup when userId changes or unmounts
}, [userId]);
```

18. Why do we need keys in React lists?

Level: junior React

Keys give each list item a stable identity so React can match items between renders and correctly add, remove or reorder them without recreating or mixing up their state.

When React diffs two lists it needs to know which new item corresponds to which old one. Without keys it falls back to matching by position, so inserting an item at the top makes every item look changed, and local state such as an input's text can end up attached to the wrong row.

A key must be unique among siblings and stable across renders. Use an ID from your data. Using the array index is acceptable only for static lists that are never reordered, filtered or inserted into. Never use `Math.random()`, since a new key on every render makes React remount every item.

```
{todos.map(todo => (  
  <TodoItem key={todo.id} todo={todo} /> // stable ID, not the index  
))}
```

19. What is the difference between controlled and uncontrolled components in React?

Level: junior React

A controlled component keeps the input's value in React state and updates it through onChange, while an uncontrolled component lets the DOM hold the value and you read it with a ref when needed.

Controlled inputs make React the single source of truth: the value comes from state and every keystroke calls a handler that updates state. This makes validation, formatting and conditional disabling easy. The cost is a re-render on each keystroke.

Uncontrolled inputs use defaultValue and ref, so the browser manages the value and you read it on submit. They are simpler for basic forms and file inputs (which are always uncontrolled). Form libraries often use uncontrolled inputs under the hood to avoid re-rendering the whole form while typing.

```
// controlled  
const [email, setEmail] = useState("");  
<input value={email} onChange={e => setEmail(e.target.value)} />  
  
// uncontrolled  
const ref = useRef(null);  
<input defaultValue="" ref={ref} />  
const onSubmit = () => console.log(ref.current.value);
```

20. What is prop drilling and how do you avoid it in React?

Level: junior React

Prop drilling is passing props through several layers of components that do not use them just to reach a deeply nested child; you avoid it with composition, the Context API or a state management library.

First try composition: pass the child component itself, or pass children, so intermediate components never need to know about the data. This is often enough and keeps components easy to test.

If many distant components need the same value, such as the current user or theme, use the Context API. Context re-renders every consumer when its value changes, so keep contexts small and focused, and split frequently changing values from stable ones. For complex client state, a store such as Redux Toolkit or Zustand lets components subscribe to just the slice they need.

```
const ThemeContext = createContext("light");  
  
function App() {  
  return (  
    <ThemeContext.Provider value="dark">  
      <Layout /> {/* no theme prop needed */}  
    </ThemeContext.Provider>  
  );  
}  
  
function Button() {  
  const theme = useContext(ThemeContext);  
  return <button className={theme}>Save</button>;  
}
```

21. What is the difference between useMemo and useCallback in React?

Level: mid React

useMemo caches the result of a calculation, while useCallback caches a function definition; both only recompute when their dependencies change.

useMemo(() => compute(a, b), [a, b]) returns the cached value. useCallback(fn, deps) is shorthand for useMemo(() => fn, deps): it returns the same function instance between renders.

Neither makes your app faster by default; each adds its own cost. Use useMemo for genuinely expensive calculations, and use useCallback mainly when the function is passed to a child wrapped in React.memo or is a dependency of another hook. A stable function reference is useless if the child is not memoized, since the child re-renders with its parent anyway.

Key points

useMemo returns a value, useCallback returns a function

Both depend on a dependency array

Measure before optimising; memoisation has a cost

```
const visible = useMemo(
  () => todos.filter(t => t.status === filter),
  [todos, filter]
);

const handleSelect = useCallback((id) => setSelected(id), []);

const Row = React.memo(function Row({ item, onSelect }) { /* ... */ });
<Row item={item} onSelect={handleSelect} />
```

22. What is the difference between useRef and useState in React?

Level: mid React

Updating state triggers a re-render, while changing a ref's .current value does not; use state for data shown on screen and refs for DOM access or values that must persist without causing renders.

useRef returns a mutable object whose .current property lives for the whole life of the component. Typical uses are holding a DOM node (to focus an input or measure an element), storing a timer ID, or remembering the previous value of a prop.

Because React does not know when you change a ref, the UI will not update. If a value affects what the user sees, it belongs in state. Avoid reading or writing ref.current during rendering, except for lazy initialisation.

```
function SearchBox() {
  const inputRef = useRef(null);
  const [query, setQuery] = useState(""); // changes the UI

  return (
    <>
      <input ref={inputRef} value={query} onChange={e => setQuery(e.target.value)} />
      <button onClick={() => inputRef.current.focus()}>Focus</button>
    </>
  );
}
```

23. What is the difference between SSR, SSG and CSR in React?

Level: mid React

CSR renders in the browser after JavaScript loads, SSR renders the HTML on the server for each request, and SSG renders the HTML once at build time; SSR and SSG give crawlers and users content

faster, which helps SEO.

With client-side rendering the server sends an almost empty HTML file and the browser builds the page after downloading and running JavaScript. It is simple to host but the first paint is slower and search engines must run JavaScript to see the content.

Server-side rendering generates HTML per request, so content is visible immediately and is always fresh, at the cost of server work and a slower time to first byte. Static site generation builds HTML ahead of time and serves it from a CDN, which is the fastest option for content that does not change per user. Incremental static regeneration (as used by Next.js) rebuilds static pages in the background on a schedule.

In every case React then hydrates the HTML in the browser, attaching event handlers so the page becomes interactive. Mismatches between server and client output cause hydration errors.

Key points

CSR: render in browser. SSR: render per request. SSG: render at build time

SSR and SSG improve first paint and SEO

Hydration makes server-rendered HTML interactive

24. Why does a React component re-render and how do you prevent unnecessary re-renders?

Level: senior React

A component re-renders when its state changes, its parent re-renders, or a context it consumes changes; you prevent wasted renders by moving state down, using React.memo, and keeping props referentially stable.

By default, when a component renders, all of its children render too, even if their props did not change. React then compares the output and only updates the DOM where something is different, so a re-render is not necessarily a DOM update, but the work of calling your component still costs time.

Start by fixing the structure: keep state close to where it is used, and split a component so a frequently changing part (such as an input) does not drag a heavy sibling along. Passing children as props also helps, because the same element object does not re-render. If that is not enough, wrap pure components in React.memo, stabilise object and function props with useMemo and useCallback, and split large contexts. Use the React DevTools Profiler to confirm what actually rendered and why before optimising.

Key points

Triggers: own state change, parent render, context change

Move state down and compose with children first

React.memo only helps if props are stable

25. How do you optimize performance in a React application?

Level: senior React

Measure first with the Profiler and Lighthouse, then reduce unnecessary renders, split code with lazy loading, virtualise long lists, and optimise images and bundle size.

Rendering: avoid wasted re-renders by colocating state, using React.memo, useMemo and useCallback where profiling shows a benefit, and avoiding creating new objects in props for memoised children. For slow updates triggered by typing or filtering, useTransition and useDeferredValue keep the interface responsive.

Loading: split the bundle by route with React.lazy and Suspense (or Next.js dynamic imports), remove unused dependencies, and analyse the bundle with a tool such as webpack-bundle-analyzer. Serve properly sized, modern-format images and lazy-load below-the-fold content.

Data and lists: virtualise long lists with a library such as react-window so only visible rows are in the DOM, cache server data with React Query or SWR, and debounce expensive handlers. Track real-user metrics such as Largest Contentful Paint and Interaction to Next Paint, not just lab scores.

Key points

Profile before optimising

Reduce renders, reduce bundle size, reduce DOM size

Use real-user metrics to verify improvements

```
const Reports = React.lazy(() => import("./Reports"));

<Suspense fallback={<Spinner />}>
  <Reports />
</Suspense>
```

HTML & CSS

26. What are semantic HTML elements and why do they matter?

Level: intern HTML & CSS

Semantic elements describe their meaning in the name, such as header, nav, main, article and footer, so browsers, assistive technology and search engines can understand the page structure.

A <div> or says nothing about the content. <header>, <nav>, <main>, <article>, <section>, <aside> and <footer> tell the browser what that region is for. Screen readers use them as landmarks, and search engines use them as a hint about what is primary content.

Use one <main> per page, put the site chrome in <header> and <footer>, and wrap a self-contained story or card in <article>. Keep using <div> when you only need a styling hook. Pair landmarks with a clear heading order (h1 then h2) rather than skipping levels.

Key points

Semantic tags describe meaning, not just layout

They help accessibility, SEO and maintainability

div and span are still correct when there is no meaning to express

27. What is the CSS box model?

Level: intern HTML & CSS

Every element is a box made of content, padding, border and margin. The default content-box size counts only the content; border-box includes padding and border in the width you set.

From the inside out: content (the text or image), padding (space inside the border), border, then margin (space outside the border). Adjacent vertical margins collapse into the larger of the two, which surprises people building stacked cards.

With box-sizing: content-box (the default), width: 200px is the content only, so padding and border make the element wider. box-sizing: border-box makes width include padding and border, which is why most teams set *, ::before, ::after { box-sizing: border-box; }.

```
*, ::before, ::after { box-sizing: border-box; }

.card {
  width: 320px; /* includes padding and border */
  padding: 16px;
  border: 1px solid #ddd;
  margin-bottom: 16px;
}
```

28. What is the difference between display none and visibility hidden in CSS?

Level: *intern HTML & CSS*

display: none removes the element from the layout so it takes no space; **visibility: hidden** hides it but leaves the empty space, and the element can still be read in the accessibility tree in some cases.

display: none is as if the element was not in the document for layout: later siblings move up. **visibility: hidden** keeps the box, so a 200px-tall banner still reserves 200px. **opacity: 0** is similar to hidden for layout, but the element can still receive pointer events unless you also set **pointer-events: none**.

For content that should not be announced to screen readers, prefer **display: none** or the HTML **hidden** attribute. To hide visually but keep it available to assistive technology (a skip link target, for example), use a visually-hidden class rather than **display: none**.

29. What is the difference between inline, block and inline-block in CSS?

Level: *intern HTML & CSS*

Block elements start on a new line and take the full available width; **inline** elements sit in the text flow and ignore width and height; **inline-block** sits in the flow but accepts width, height, padding and margin on all sides.

Headings, paragraphs and **div** are block by default. **span**, **a** and **strong** are inline: width, height and vertical margin do not apply the way you expect. **inline-block** is the middle ground used for buttons or icon-and-label pairs that should sit in a row but have a fixed size.

Today Flexbox or Grid usually replaces floats and long rows of **inline-block**, because **inline-block** still leaves gaps from the whitespace between tags. Use the **display** value that matches the layout job, not a habit.

30. What is the difference between relative, absolute, fixed and sticky positioning in CSS?

Level: *junior HTML & CSS*

Relative offsets an element from its normal place; **absolute** positions it against the nearest positioned ancestor; **fixed** is against the viewport; **sticky** sits in normal flow until a scroll threshold, then sticks.

position: relative with **top/left** moves the element but the original space is kept. **position: absolute** takes it out of flow; the containing block is the nearest ancestor with position other than static (or the initial containing block). That is why a modal overlay needs **position: relative** on its parent.

fixed stays put when the page scrolls, which is how sticky headers used to be built. **sticky** is a hybrid: the element scrolls with its container until **top: 0** (or similar) is reached, then it behaves like **fixed** within that container. **sticky** fails if any ancestor has **overflow: hidden**.

Key points

absolute is relative to the nearest positioned ancestor

fixed is relative to the viewport

sticky needs a threshold (**top/left**) and no **overflow-hidden** ancestors

31. What is the difference between Flexbox and CSS Grid?

Level: *junior HTML & CSS*

Flexbox is one-dimensional (a row or a column). **CSS Grid** is two-dimensional (rows and columns at once). Use **Flexbox** for alignment along a line; use **Grid** when the layout is a real 2D structure.

A nav bar, a cluster of buttons, or a card header with a title on the left and actions on the right is a **Flexbox** problem. A page with a sidebar, a main column and a footer, or a photo gallery with consistent tracks, is a **Grid** problem. They combine well: a **Grid** page whose cards use **Flexbox** internally.

Grid lets you name areas (grid-template-areas) and place items on explicit lines. Flexbox shines at distributing leftover space (justify-content, flex-grow) and wrapping. Neither replaces the other; interviews often ask you to pick one and say why.

```
.page {
  display: grid;
  grid-template-columns: 240px 1fr;
  grid-template-rows: auto 1fr auto;
  min-height: 100vh;
}

.toolbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 1rem;
}
```

32. How does CSS specificity work?

Level: junior HTML & CSS

Specificity is the score the browser uses to decide which rule wins: inline styles beat IDs, IDs beat classes/attributes/pseudo-classes, and those beat elements. When scores tie, the last rule in the source wins.

Count a tuple (inline, IDs, classes, elements). #nav .item is (0, 1, 1, 0). div.card p is (0, 0, 1, 2). !important jumps the queue and should be a last resort; too many of them mean the specificity war is already lost.

A more maintainable approach is low-specificity utilities or a method such as BEM, plus CSS layers (@layer) so you can control order without stacking IDs. Attribute selectors and pseudo-classes (.btn:hover) count as classes. The universal selector * and the :where() pseudo-class add zero specificity.

33. What is the difference between em, rem, %, vh and vw in CSS?

Level: junior HTML & CSS

em is relative to the parent font size, rem is relative to the root font size, % is relative to the parent's relevant size, and vh/vw are 1% of the viewport height or width.

rem is the default choice for type and spacing because it stays predictable when components nest. em compounds: a 1.2em padding inside a 1.2em heading is larger than you think. % on width is the parent's width; % on height only works if the parent has an explicit height.

100vh is the full viewport height, which on mobile can include the area hidden by the browser chrome and cause jumpiness. Newer units such as svh, lvh and dvh address that. For fluid type, clamp(1rem, 2vw, 2rem) is a common pattern.

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

Level: junior HTML & CSS

Cookies are sent to the server with every matching request and can expire; localStorage persists until cleared and stays in the browser; sessionStorage lasts for the current tab only.

Use cookies for authentication when the server must see the value (prefer HttpOnly, Secure and SameSite). Use localStorage for non-sensitive UI preferences that should survive a restart. Use sessionStorage for a wizard or draft that should die when the tab closes.

Both storage APIs are readable by JavaScript, so they are the wrong place for access tokens if you have XSS. Cookies set without HttpOnly have the same problem. All three are origin-scoped. localStorage has a typical

quota of about 5 MB; cookies should stay tiny because they travel on every request.

35. What is the difference between async and defer on script tags?

Level: mid HTML & CSS

Both download the script without blocking HTML parsing. defer waits until parsing finishes and runs scripts in document order; async runs a script as soon as it is downloaded, which can be before or after parsing and in any order.

A classic `<script src>` in the head blocks parsing until the file is downloaded and executed. defer is the right default for module-style application code that depends on the DOM and on other scripts. async is for independent tags such as analytics, where order does not matter.

`type="module"` scripts are deferred by default. async and defer have no effect on inline scripts. Put critical, tiny inline scripts only when you must paint something before the main bundle arrives.

36. How does the browser render a web page?

Level: mid HTML & CSS

The browser parses HTML into a DOM and CSS into a CSSOM, combines them into a render tree, computes layout (reflow), paints pixels and composites layers onto the screen.

This path is called the critical rendering path. Scripts without async/defer pause HTML parsing. CSS is render-blocking: the browser waits for stylesheets before first paint so it does not flash unstyled content. After the first paint, changing geometry (width, height, top) causes reflow; changing only color or transform can often skip to paint or composite.

Performance work targets each stage: reduce render-blocking CSS, defer non-critical JS, avoid layout thrashing (reading layout then writing styles in a loop), and keep animations on transform and opacity so they run on the compositor thread. Senior interviews often follow this with Core Web Vitals: LCP, INP and CLS.

Key points

DOM + CSSOM render tree layout paint composite

CSS blocks first paint; JS without async/defer blocks parsing

Animate transform and opacity to avoid layout

37. What are Core Web Vitals and how do you improve them?

Level: senior HTML & CSS

Core Web Vitals are Google's field metrics for real-user experience: LCP (loading), INP (interactivity) and CLS (visual stability). They influence search ranking and are measured on real devices, not only Lighthouse.

LCP (Largest Contentful Paint) should happen within about 2.5 seconds: compress the hero image, preload it, and cut render-blocking CSS/JS. INP (Interaction to Next Paint) replaced FID: keep event handlers short, break up long tasks, and show immediate feedback. CLS (Cumulative Layout Shift) should stay below 0.1: always set width and height (or aspect-ratio) on images and embeds, and avoid inserting banners above existing content.

Lab tools (Lighthouse, DevTools Performance) are for debugging. Rankings use field data from the Chrome User Experience Report. Fix the slowest template first, measure with web-vitals or Search Console, and do not chase a 100 Lighthouse score at the cost of features users need.

TypeScript

38. What is TypeScript and why do we use it?

Level: *intern TypeScript*

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It catches type errors in the editor and at build time, and it documents the shape of data for the next person who reads the code.

Every valid JavaScript file is already TypeScript, which is why teams can adopt it gradually. The compiler (tsc) erases types and emits JS that runs anywhere JavaScript runs. Types exist only at compile time: they are not available for runtime checks unless you add a library or generate them.

The usual wins are safer refactors, better editor autocomplete, and APIs that are harder to call wrongly. The cost is a build step and time spent modelling awkward JavaScript patterns. Interviews often follow this with "when would you still use JavaScript?" - small scripts, prototyping, or a codebase that is not yet worth converting.

39. What is the difference between interface and type in TypeScript?

Level: *intern TypeScript*

Both describe object shapes. interface can be merged and is preferred for public object contracts; type can express unions, intersections and primitives, and cannot be reopened later.

For a simple object, they look the same and teams pick one style. interface supports declaration merging: two interface User blocks become one, which is how many .d.ts files extend library types. type aliases are fixed once written.

Use type when you need a union (string | number), a mapped type, or a tuple. Use interface when you are describing an object that others might extend. You can implement either with a class. Do not waste interview time claiming one is "faster"; they compile to the same nothing.

Key points

interface: extendable, mergeable, object-shaped

type: unions, intersections, primitives, tuples

Pick one style for objects and stay consistent

```
interface User { id: string; name: string }
interface User { email?: string } // merged

type Id = string | number; // union: type only
type Admin = User & { role: "admin" }; // intersection
```

40. What is the difference between any, unknown and never in TypeScript?

Level: *junior TypeScript*

any turns off checking. unknown is a type-safe any: you must narrow it before use. never means a value cannot exist, used for exhaustive checks and functions that always throw.

A value typed any can be passed anywhere and you can read any property on it, so a typo compiles. unknown is what you give to untrusted input (JSON.parse, a network payload) until you prove its shape with typeof, in, or a type guard.

never is the empty set. A function with return type never does not return (throw or infinite loop). In a switch, assigning the leftover value to const _exhaustive: never = value makes the compiler fail when a new union member is added and not handled.

```
function handle(x: unknown) {
  if (typeof x === "string") return x.toUpperCase();
  throw new Error("expected string");
}

function fail(message: string): never {
  throw new Error(message);
}
```

41. What are generics in TypeScript?

Level: junior TypeScript

Generics are type parameters: they let a function, class or type work with many types while keeping the relationship between inputs and outputs, instead of falling back to any.

`identity<T>(value: T): T` returns the same type it received. Without `T` you would write `any` and lose that link. Constraints (`T extends { id: string }`) describe the minimum shape. Several parameters (`K`, `V`) model maps and API wrappers.

Common interview follow-ups: default parameters (`T = string`), `keyof T` for safe property access, and why `Array<T>` and `T[]` are the same. Prefer inference (`wrap(user)`) over forcing `wrap<User>(user)` at every call site.

```
function first<T>(items: T[]): T | undefined {
  return items[0];
}

first(["a", "b"]); // string | undefined
first<number>([1, 2, 3]); // number | undefined
```

42. What are union and intersection types in TypeScript?

Level: junior TypeScript

A union (`A | B`) is a value that is one of the members. An intersection (`A & B`) is a value that must satisfy both types at once.

Unions model "this or that": a result that is `{ ok: true, data: T } | { ok: false, error: string }`. Narrow with a discriminant field (`ok`) before you read `data`. Intersections model "this and that": `User & { role: "admin" }` has every `User` field plus `role`.

A union of objects is not an object with optional fields. `A | B` only lets you access properties common to both until you narrow. That is the usual bug when someone writes `user.role` on a `User | Admin` without checking.

43. What are utility types in TypeScript like Partial, Pick and Omit?

Level: junior TypeScript

Utility types are built-in helpers that transform an existing type: `Partial` makes every field optional, `Required` the opposite, `Pick` keeps a set of keys, `Omit` removes them, and `Record` builds a key/value map.

`Partial<User>` is useful for a PATCH body. `Pick<User, "id" | "name">` is a list-row type. `Omit<User, "password">` is what you send to the client. `Record<Id, User>` is a dictionary. `ReturnType<typeof fn>` and `Parameters<typeof fn>` pull types from functions so you do not repeat them.

You can write the same things with mapped types; the utilities are the readable version. Interviews sometimes ask you to implement `Partial` yourself: `{ [K in keyof T]?: T[K] }`.

```
type User = { id: string; name: string; password: string };
type PublicUser = Omit<User, "password">;
type UserPatch = Partial<Pick<User, "name" | "password">>;
```

44. What is type narrowing in TypeScript?

Level: mid TypeScript

Narrowing is when TypeScript shrinks a wide type (such as a union) to a more specific type after a runtime check like `typeof`, `instanceof`, `in`, or a custom type guard.

Control-flow analysis follows your `if` and `switch`. `typeof x === "string"` narrows `string | number` to `string` in that branch. `x instanceof Date` works for classes. `"width" in value` distinguishes object shapes. A user-defined guard is a function whose return type is `value` is `Fish`.

Equality checks against a discriminant (`event.type === "click"`) are the cleanest pattern for API payloads. Assertions (`value as User`) skip narrowing and are a common source of production bugs: the compiler believes you, the runtime does not.

```
function isUser(value: unknown): value is { id: string } {
  return typeof value === "object" && value !== null && "id" in value;
}

function label(value: string | number) {
  return typeof value === "string" ? value.toUpperCase() : value.toFixed(2);
}
```

45. What is the difference between enum and union types in TypeScript?

Level: mid TypeScript

A string union such as `"admin" | "user"` exists only in the type system and erases to plain strings. A TypeScript enum compiles to a JavaScript object (unless it is a `const` enum), which is extra runtime code and a weaker default.

Most codebases prefer union types plus `as const` objects when they also need a runtime list (`const Roles = ["admin", "user"] as const`). Numeric enums are especially surprising because they reverse-map (`Color[0] === "Red"`). `const` enum inlines values but is banned in some projects because it breaks isolated modules and bundlers.

If a library already exposes an enum, use it. When you design a new API, a union is simpler to tree-shake, serialise and explain.

46. How does TypeScript type inference work?

Level: mid TypeScript

TypeScript fills in types you omit: literals get their literal or widened type, function return types are inferred from return statements, and generic arguments are inferred from the values you pass.

`let count = 0` becomes `number`. `const role = "admin"` stays the literal `"admin"` because `const` cannot be reassigned. Contextual typing works the other way: a callback argument is inferred from the function you pass it to (`arr.map(item => ...)`).

Write types at module boundaries (function arguments, exported APIs, public class fields) and let inference handle locals. Over-annotating locals makes refactors noisy. When inference widens too far (`[]` is `any[]` without a hint), give the generic or a typed empty value (`[] as User[]` or `User[] = []`).

47. What are mapped types and conditional types in TypeScript?

Level: senior TypeScript

Mapped types build a new object type by looping over keys ({ [K in keyof T]: ... }). Conditional types pick result with `T extends U ? X : Y`, which is how many utility types are implemented.

a `Partial<T>` is a mapped type that adds `?` to every key. You can also remap keys with `as` (`[K in keyof T as ${K & string}!]: T[K]`). Conditional types power `Exclude`, `Extract` and `NonNullable`. They become interesting with `infer`, which pulls a piece out of a type (`T extends Promise<infer U> ? U : T`).

Senior interviews look for judgement: these features belong in shared utility types, not in every component file. If a conditional type is hard to read, a few well-named helpers beat one clever one-liner. Recursion has a depth limit; it is the wrong tool for modelling arbitrarily nested JSON unless you must.

```
type ReadonlyFields<T> = { readonly [K in keyof T]: T[K] };
type Awaited<T> = T extends Promise<infer U> ? Awaited<U> : T;
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/role/frontend>