

Software Engineer Interview Questions and Answers

Intern to senior | Frontend, backend and full stack

All questions across every technology, from intern to senior level.

115 questions Read online: <https://www.fredevtemplates.com/interview-questions>

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.

`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}Id]`). 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;
```

SQL

48. What is the difference between INNER JOIN and LEFT JOIN in SQL?

Level: intern SQL

INNER JOIN returns only rows that have a match in both tables, while LEFT JOIN returns every row from the left table and fills the right table's columns with NULL when there is no match.

Use INNER JOIN when you only want records that exist on both sides, such as orders that have a customer. Use LEFT JOIN when the left table is the main list and the related data is optional, such as all customers including those who have never placed an order.

A common trick is to find rows with no match by left joining and filtering on `WHERE right.id IS NULL`. Be careful: putting a condition on the right table in the WHERE clause turns a left join back into an inner join, so put it in the ON clause instead. RIGHT JOIN is the mirror image of LEFT JOIN, and FULL OUTER JOIN keeps unmatched rows from both sides.

Key points

INNER: matches only. LEFT: all left rows plus matches

Unmatched right-side columns are NULL

Filter the right table in ON, not WHERE, to keep left-join behaviour

```
-- every customer, with order count (0 if none)
SELECT c.id, c.name, COUNT(o.id) AS orders
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name;

-- customers who never ordered
SELECT c.*
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.id IS NULL;
```

49. What is the difference between WHERE and HAVING in SQL?

Level: intern SQL

WHERE filters individual rows before grouping, while HAVING filters groups after GROUP BY, so only HAVING can use aggregate functions like COUNT or SUM.

SQL processes a query in a logical order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY.

Because WHERE runs before grouping, the aggregates do not exist yet. HAVING runs after grouping, so it can test them.

Put every condition that does not need an aggregate in WHERE. It reduces the rows before grouping, which is faster than filtering the groups afterwards.

```
SELECT department, COUNT(*) AS staff, AVG(salary) AS avg_salary
FROM employees
WHERE status = 'active' -- filters rows first
GROUP BY department
HAVING COUNT(*) >= 5; -- filters groups after aggregation
```

50. What is the difference between DELETE, TRUNCATE and DROP in SQL?

Level: intern SQL

DELETE removes selected rows and can be rolled back, TRUNCATE quickly removes all rows and keeps the table structure, and DROP removes the entire table including its structure.

DELETE is a DML command: it can use a WHERE clause, fires row-level triggers, and is logged row by row, so it is slower on big tables but fully transactional. TRUNCATE is a DDL command that empties the whole table, usually resets identity counters and is much faster because it deallocates data pages instead of logging each row. DROP TABLE deletes the table definition, indexes and permissions as well.

Transaction behaviour differs by database: in PostgreSQL and SQL Server TRUNCATE can be rolled back inside a transaction, while in MySQL and Oracle it causes an implicit commit. Always check your database's documentation before relying on it.

```
DELETE FROM orders WHERE created_at < '2024-01-01'; -- some rows
TRUNCATE TABLE temp_imports; -- all rows, keep table
DROP TABLE temp_imports; -- table is gone
```

51. What is the difference between a primary key and a foreign key?

Level: intern SQL

A primary key uniquely identifies each row in its own table and cannot be NULL, while a foreign key is a column that references the primary key of another table to link the two.

A table has one primary key (which may be made of several columns). It guarantees uniqueness and is indexed automatically. A table can have many foreign keys. A foreign key enforces referential integrity: you cannot insert an order for a customer that does not exist, and you cannot delete a customer that still has orders unless you define a cascade rule such as ON DELETE CASCADE.

A foreign key column may contain duplicates and, unless declared NOT NULL, may be NULL. A unique key also enforces uniqueness but, unlike a primary key, allows NULL (the exact number of NULLs allowed depends on the database).

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);

CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE RESTRICT
);
```

52. How do you find the second highest salary in SQL?

Level: junior SQL

Use a subquery that takes the MAX salary below the overall MAX, or use DENSE_RANK() and pick rank 2, which also handles ties and generalises to the Nth highest.

The subquery version works on every database. The window function version is the one interviewers like best, because changing 2 to N gives you the Nth highest salary, and DENSE_RANK correctly treats equal salaries as the same rank. Use DISTINCT so tied rows return one value.

Another common answer is ORDER BY salary DESC LIMIT 1 OFFSET 1 (MySQL, PostgreSQL) or OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY (SQL Server), but you need SELECT DISTINCT salary first or ties break it. If there is no second salary, wrap the query so it returns NULL rather than an empty result when the question asks for that.

Key points

Subquery with MAX is the simplest answer

DENSE_RANK handles ties and any N

ROW_NUMBER would be wrong when salaries tie

```
-- works everywhere
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Nth highest with window function (here N = 2)
SELECT DISTINCT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
) ranked
WHERE rnk = 2;
```

53. How do you find and delete duplicate rows in SQL?

Level: junior SQL

Find duplicates with GROUP BY and HAVING COUNT(*) > 1, then delete them by numbering rows within each duplicate group with ROW_NUMBER() and removing every row after the first.

First decide what makes two rows duplicates (for example the same email) and which row you want to keep (usually the oldest, so the lowest id). Always run the matching SELECT before the DELETE, and take a backup or work inside a transaction.

ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) numbers rows inside each group of identical emails, so any row with a number above 1 is a duplicate. In MySQL older than 8.0 window functions are unavailable, and you cannot select from the same table in a DELETE subquery directly, so you wrap the subquery in a derived table or self-join instead. Afterwards, add a unique constraint so the duplicates cannot come back.

```
-- 1. find
SELECT email, COUNT(*) AS copies
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- 2. delete, keeping the lowest id (PostgreSQL / SQL Server / MySQL 8+)
WITH ranked AS (
SELECT id, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) AS rn
FROM users
)
DELETE FROM users WHERE id IN (SELECT id FROM ranked WHERE rn > 1);

-- 3. prevent it happening again
ALTER TABLE users ADD CONSTRAINT uq_users_email UNIQUE (email);
```

54. What is the difference between UNION and UNION ALL in SQL?

Level: junior SQL

UNION combines two result sets and removes duplicate rows, while UNION ALL combines them and keeps every row, which makes it faster.

UNION has to sort or hash the combined result to find duplicates, which costs time and memory. If you know the two queries cannot return the same row, or you want the duplicates, use UNION ALL. Both require the same number of columns with compatible types, and the column names come from the first query.

```
SELECT email FROM newsletter_subscribers
UNION -- one row per distinct email
SELECT email FROM customers;

SELECT email FROM newsletter_subscribers
UNION ALL -- keeps duplicates, faster
SELECT email FROM customers;
```

55. What is database normalization and what are the normal forms?

Level: junior SQL

Normalization is organising tables to reduce redundancy and update anomalies: 1NF means atomic values, 2NF removes partial dependencies on part of a composite key, and 3NF removes dependencies between non-key columns.

In 1NF every column holds a single value (no comma-separated lists) and each row is unique. In 2NF the table is in 1NF and every non-key column depends on the whole primary key, which matters for composite keys. In 3NF the table is in 2NF and non-key columns depend only on the key, not on other non-key columns (a city and zip_code pair stored with an order is an example of what to move out).

Normalisation avoids storing the same fact in many places, so updates and deletes cannot leave inconsistent data. The trade-off is more joins. Reporting databases and read-heavy systems often denormalise on purpose, accepting some redundancy for faster reads.

Key points

1NF: atomic values. 2NF: no partial dependency. 3NF: no transitive dependency

Goal: remove insert, update and delete anomalies

Denormalise deliberately for read performance

56. What are ACID properties in a database?

Level: mid SQL

ACID stands for Atomicity, Consistency, Isolation and Durability: a transaction either fully succeeds or fully fails, keeps the data valid, does not interfere with other transactions, and survives crashes once committed.

Atomicity: all statements in a transaction are applied or none are, so a bank transfer never debits one account without crediting the other. Consistency: a transaction moves the database from one valid state to another, respecting constraints. Isolation: concurrent transactions behave as if they ran in some order, to the degree set by the isolation level. Durability: once committed, data is stored permanently, typically via a write-ahead log.

Relational databases provide ACID by default. Many NoSQL systems relax some of these guarantees to gain availability or scale, which is the trade-off behind the BASE and CAP discussions.

```
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT; -- or ROLLBACK; if anything failed
```

57. What is an index in SQL and how does it improve performance?

Level: mid SQL

An index is a separate sorted data structure (usually a B-tree) that lets the database find rows without scanning the whole table, at the cost of extra storage and slower writes.

Without an index, a query like `WHERE email = 'a@b.com'` reads every row (a full table scan). With an index on email the database walks the tree in a handful of steps and jumps straight to the matching rows. Indexes also speed up JOIN, ORDER BY and GROUP BY on the indexed columns.

Every INSERT, UPDATE and DELETE must also update every index on the table, so do not index everything. In a composite index the column order matters: an index on (last_name, first_name) helps a search on last_name alone but not on first_name alone. An index is also skipped when you wrap the column in a function (`WHERE LOWER(email) = ...`) or use a leading wildcard (`LIKE '%smith'`), and it is of little use on low-cardinality columns such as a boolean.

Key points

Faster reads, slower writes, more storage

Column order in composite indexes matters

Functions on the column and leading wildcards prevent index use

```
CREATE INDEX idx_orders_customer_date ON orders (customer_id, created_at);

-- uses the index
SELECT * FROM orders WHERE customer_id = 42 AND created_at >= '2026-01-01';
-- cannot use it efficiently (function on the column)
SELECT * FROM orders WHERE DATE(created_at) = '2026-01-01';
```

58. What is the difference between a clustered and a non-clustered index?

Level: mid SQL

A clustered index defines the physical order in which the table's rows are stored, so there can be only one per table, while a non-clustered index is a separate structure that points back to the rows.

In SQL Server and MySQL InnoDB the primary key is the clustered index by default, so the table data is the leaf level of that tree. Looking up by the clustered key is the fastest possible read, and range scans on it are efficient because neighbouring keys are stored together.

A non-clustered index stores the indexed columns plus a pointer (the clustered key or a row locator). A lookup finds the entry and then does a second lookup to fetch the remaining columns, unless the index covers the query

by including all needed columns. Choose a narrow, ever-increasing clustered key, because wide or random keys (such as random GUIDs) are copied into every non-clustered index and cause page splits.

59. What are transaction isolation levels in SQL and what problems do they prevent?

Level: senior SQL

Isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable) control how much concurrent transactions can see of each other's changes, trading consistency against concurrency to prevent dirty reads, non-repeatable reads and phantom reads.

A dirty read means reading data another transaction has not committed. A non-repeatable read means reading the same row twice in one transaction and getting different values because someone committed in between. A phantom read means running the same query twice and getting a different set of rows because rows were inserted or deleted.

Read Uncommitted allows all three. Read Committed prevents dirty reads (the default in PostgreSQL, SQL Server and Oracle). Repeatable Read also prevents non-repeatable reads (the default in MySQL InnoDB, and it also prevents most phantoms there). Serializable prevents all three by making transactions behave as if they ran one after another, at the highest cost in locking or retries.

Choose the lowest level that is correct for the business rule, and use explicit locking (SELECT ... FOR UPDATE) or optimistic concurrency with a version column for specific hot spots such as inventory or balances.

Key points

Higher isolation = fewer anomalies but less concurrency

Defaults differ by database

Use row locks or version columns for critical updates

60. How do you optimize a slow SQL query?

Level: senior SQL

Read the execution plan with EXPLAIN to find full scans and expensive joins, add or fix indexes for the filter and join columns, select only the columns and rows you need, and rewrite predicates so indexes can be used.

Start by measuring, not guessing: run EXPLAIN (or EXPLAIN ANALYZE in PostgreSQL, or the actual execution plan in SQL Server) and look for sequential scans on large tables, big gaps between estimated and actual row counts, sorts that spill to disk, and nested loops over huge inputs.

Typical fixes: add an index matching the WHERE and JOIN columns (and ORDER BY where possible); avoid SELECT *; make predicates sargable (compare the raw column, not a function of it); replace OFFSET pagination on large offsets with keyset pagination (WHERE id > last_seen_id ORDER BY id LIMIT 50); avoid N+1 query patterns from the application; update table statistics; and archive or partition very large tables.

If the query is still too slow, consider a read replica, caching, a materialised view or precomputed summary table. Confirm every change with the plan and with timings on production-sized data.

Key points

Measure with the execution plan first

Indexes, sargable predicates, fewer columns and rows

Keyset pagination beats large OFFSET

```
EXPLAIN ANALYZE
SELECT o.id, o.total
FROM orders o
WHERE o.customer_id = 42
AND o.created_at >= '2026-01-01'
ORDER BY o.created_at DESC
LIMIT 50;

-- keyset pagination instead of a large OFFSET
SELECT id, total FROM orders
WHERE id < 98765
ORDER BY id DESC
LIMIT 50;
```

C#

61. What is the difference between a value type and a reference type in C#?

Level: intern C#

A value type variable holds its data directly and is copied on assignment, while a reference type variable holds a reference to an object, so assignment copies the reference and both variables see the same object.

Value types include int, double, bool, char, DateTime, enums and any struct. Reference types include class, string, arrays, delegates and object. A value type is stored inline where it is declared: on the stack for a local variable, or inside the containing object on the heap. A reference type's object always lives on the managed heap and is cleaned up by the garbage collector.

This changes behaviour when you pass values to methods. Passing a struct hands the method its own copy, so changes inside do not leak out. Passing a class passes a copy of the reference, so the method can modify the same object. Reference types can be null; value types cannot, unless you use Nullable<T> (int?).

Key points

Value types are copied, reference types share one object

Reference types can be null, value types need int? syntax

Large mutable structs are usually a bad idea

```
struct PointS { public int X; }
class PointC { public int X; }

var s1 = new PointS { X = 1 }; var s2 = s1; s2.X = 99;
Console.WriteLine(s1.X); // 1 (copy)

var c1 = new PointC { X = 1 }; var c2 = c1; c2.X = 99;
Console.WriteLine(c1.X); // 99 (same object)
```

62. What is the difference between string and StringBuilder in C#?

Level: intern C#

A string is immutable, so every change creates a new string object, while StringBuilder is a mutable buffer that modifies its contents in place, making it much faster for repeated concatenation.

Each time you do `s += "x"` in a loop, the runtime allocates a new string and copies the old content, which is $O(n^2)$ work overall and creates a lot of garbage. `StringBuilder` keeps an internal character buffer that grows as needed, so appending is cheap and you call `ToString()` once at the end.

For a handful of concatenations, a plain string or string interpolation is clearer and just as fast. Use `StringBuilder` when building text in a loop or from many pieces.

```
var sb = new StringBuilder();
for (int i = 0; i < 10_000; i++)
{
    sb.Append(i).Append(', ');
}
string result = sb.ToString();
```

63. What is the difference between const and readonly in C#?

Level: junior C#

A const is a compile-time constant that must be assigned where it is declared, while a readonly field can be assigned at declaration or in the constructor, so its value can be decided at runtime.

const values are baked into the calling code at compile time and are implicitly static. They can only be primitive types, strings or null. If you change a const in a library, every project using it must be recompiled to see the new value.

readonly fields are normal fields that cannot be reassigned after the constructor finishes. They can hold any type, can differ per instance, and are read at runtime. Note that readonly makes the reference unchangeable, not the object it points to. For compile-time values that will never change (such as the number of days in a week) use const; otherwise prefer readonly.

```
public class Config
{
    public const int MaxRetries = 3; // compile-time
    public readonly DateTime CreatedAt; // runtime, per instance

    public Config() { CreatedAt = DateTime.UtcNow; }
}
```

64. What is the difference between ref and out parameters in C#?

Level: junior C#

Both pass an argument by reference, but ref requires the variable to be initialised before the call, while out requires the method to assign a value before it returns.

Use ref when the method both reads and modifies the caller's variable. Use out when the method only produces a value, such as the bool TryParse(string, out int result) pattern, which lets a method return a success flag and a value together. A third modifier, in, passes by reference as read-only, useful to avoid copying large structs.

In modern C# you can often return a tuple instead ((bool ok, int value)), which is clearer than out parameters.

```
if (int.TryParse(input, out int number))
{
    Console.WriteLine(number * 2);
}

static void AddTen(ref int x) => x += 10;
int n = 5;
AddTen(ref n); // n is now 15
```

65. What is the difference between an abstract class and an interface in C#?

Level: junior C#

An abstract class can hold state, constructors and partial implementation but a class can inherit only one, while an interface defines a contract that a class can implement many of.

Use an abstract class when related classes share fields, constructor logic or common behaviour, and you want to force subclasses to fill in the rest. A class can extend only one base class, so this is a strong coupling.

Use an interface to describe a capability (IDisposable, IComparable, IEmailSender) that unrelated types can implement, and to make code testable and replaceable through dependency injection. Since C# 8 interfaces may include default method implementations, but they still cannot hold instance fields. A common guideline: start with an interface for the contract, and add an abstract base class only if there is real shared code.

Key points

Abstract class: shared state and code, single inheritance

Interface: contract, multiple implementation

Interfaces make dependency injection and testing easier

```
public interface INotifier { void Send(string message); }

public abstract class NotifierBase : INotifier
{
    protected readonly ILogger Logger;
    protected NotifierBase(ILogger logger) => Logger = logger;
    public abstract void Send(string message);
}
```

66. What is boxing and unboxing in C# and why does it affect performance?

Level: junior C#

Boxing converts a value type to object by allocating it on the heap, and unboxing converts it back; both add allocations and casts, so avoid them in hot code paths.

Assigning an int to an object variable boxes it: the runtime allocates a new object on the heap and copies the value in. Casting back with (int)obj is unboxing, which checks the type and copies the value out. Boxing in a loop creates garbage that the collector has to clean.

Generics were introduced to avoid this: List<int> stores ints directly, whereas the old ArrayList boxed every item. Watch for hidden boxing when a struct is used as an interface type, or when you pass a value type to a method taking object, such as older string.Format overloads.

```
int i = 42;
object boxed = i; // boxing: heap allocation
int j = (int)boxed; // unboxing: type check + copy

var list = new List<int>(); // no boxing
list.Add(i);
```

67. What is LINQ in C# and what is deferred execution?

Level: junior C#

LINQ (Language Integrated Query) lets you query collections, databases and XML with a consistent syntax, and deferred execution means a query does not run when you define it but when you enumerate the results.

Methods such as Where, Select and OrderBy only build up a description of the query. The work happens when you iterate with foreach or call an operator that forces evaluation, such as ToList(), ToArray(), Count() or First().

This is efficient, but it has traps: if the source changes between defining and enumerating the query, you see the new data, and enumerating the same query twice runs it twice (a database query would hit the database twice). Call ToList() when you want a snapshot or need to reuse the results.

```
var numbers = new List<int> { 1, 2, 3 };
var evens = numbers.Where(n => n % 2 == 0); // nothing runs yet

numbers.Add(4);
Console.WriteLine(string.Join(", ", evens)); // 2,4 (runs now)

var snapshot = numbers.Where(n => n > 1).ToList(); // runs immediately
```

68. What is the difference between IEnumerable and IQueryable in C#?

Level: mid C#

IEnumerable runs the query in memory using delegates, while IQueryable builds an expression tree that a provider such as Entity Framework translates into SQL, so filtering happens in the database.

If you call `.Where(...)` on an `IQueryable<Customer>` from Entity Framework, the condition becomes part of the SQL WHERE clause and only matching rows come over the network. If you call it on an `IEnumerable<Customer>`, all rows are loaded first and then filtered in your application's memory.

A common bug is converting to `IEnumerable` or calling `ToList()` too early, which silently loads the full table. Keep the query as `IQueryable` until you have applied all filtering, sorting and paging, and only then materialise it. Remember that an `IQueryable` can only contain operations the provider can translate to SQL.

Key points

`IQueryable`: expression trees translated to SQL

`IEnumerable`: in-memory iteration with delegates

Materialise (`ToList`) as late as possible

```
// SQL: SELECT ... WHERE IsActive = 1 (filtered in the database)
var active = dbContext.Customers.Where(c => c.IsActive).ToList();

// loads ALL customers first, then filters in memory
IEnumerable<Customer> all = dbContext.Customers.ToList();
var slow = all.Where(c => c.IsActive).ToList();
```

69. What are transient, scoped and singleton service lifetimes in ASP.NET Core dependency injection?

Level: mid C#

Transient services are created every time they are requested, scoped services are created once per HTTP request, and singleton services are created once and shared for the life of the application.

Register services in `Program.cs` with `AddTransient`, `AddScoped` or `AddSingleton`. Use transient for lightweight, stateless services, scoped for things tied to a request such as an Entity Framework `DbContext`, and singleton for shared, thread-safe objects such as caches and configuration.

The classic bug is a captive dependency: a singleton that depends on a scoped service keeps that one instance alive forever, so a `DbContext` ends up shared across requests and threads. ASP.NET Core can detect this in development with scope validation. Singletons must also be thread-safe because many requests use them at once.

Key points

Transient: new every time. Scoped: per request. Singleton: one for the app

Never inject a scoped service into a singleton

Singletons must be thread-safe

```
builder.Services.AddTransient<IEmailBuilder, EmailBuilder>();
builder.Services.AddScoped<AppDbContext>();
builder.Services.AddSingleton<IMemoryCache, MemoryCache>();

public class OrdersController(AppDbContext db, IEmailBuilder email) : ControllerBase { }
```

70. What is the difference between Task and Thread, and how does async/await work in C#?

Level: mid C#

A Thread is an operating-system execution unit, while a Task represents an asynchronous operation scheduled on the thread pool; await releases the current thread while the operation is pending and resumes the method when it completes.

Creating threads is expensive and you must manage them yourself. A Task is a higher-level abstraction: it can represent CPU work queued to the thread pool (Task.Run) or I/O that does not need a thread at all while waiting, such as an HTTP call or database query.

When the compiler sees await, it turns the method into a state machine. If the awaited task is not finished, the method returns control to its caller and the thread is free to serve other requests. When the task completes, the rest of the method continues, possibly on another thread. This is why async I/O lets a web server handle many more concurrent requests with the same number of threads. async/await is about scalability for I/O-bound work; for CPU-bound work it does not make anything faster by itself.

Key points

Task = promise of work, not a thread

await frees the thread during I/O

Use Task.WhenAll for independent operations

```
public async Task<string> GetPageAsync(HttpClient client, string url)
{
    // thread is released while waiting on the network
    string html = await client.GetStringAsync(url);
    return html.ToUpperInvariant();
}

// run independent calls concurrently
var results = await Task.WhenAll(GetPageAsync(c, a), GetPageAsync(c, b));
```

71. How does garbage collection work in C# and when should you use IDisposable?

Level: mid C#

The .NET garbage collector automatically frees managed objects that are no longer reachable, organising them into generations 0, 1 and 2 so short-lived objects are collected cheaply, and IDisposable is used to release unmanaged resources such as files and connections promptly.

New objects start in generation 0. Objects that survive a collection are promoted to generation 1 and then to generation 2. Collecting generation 0 is fast and frequent, while generation 2 collections are expensive. Objects of roughly 85,000 bytes or more go on the large object heap, which is collected with generation 2, so allocating many large temporary arrays can hurt performance.

The GC knows about memory, not about files, sockets or database connections. Types that hold those implement IDisposable, and you should wrap them in a using statement so the resource is released deterministically, instead of waiting for a finalizer. Common memory problems are not classic leaks but objects kept reachable by static fields, event handlers that are never unsubscribed, and caches without limits.

Key points

Generational GC: gen 0, 1, 2 and the large object heap
Dispose releases unmanaged resources; GC releases memory
Static references and event handlers keep objects alive

```
await using var connection = new SqlConnection(connectionString);
await connection.OpenAsync();
// connection is disposed (and returned to the pool) at the end of the scope
```

72. How do you avoid deadlocks with async/await in C#?

Level: senior C#

Use async all the way down instead of blocking on tasks with .Result or .Wait(), and in library code use ConfigureAwait(false) so continuations do not need to return to a captured context.

The classic deadlock happens in UI apps and legacy ASP.NET: the calling thread blocks on task.Result, while the awaited method is waiting to resume on that same captured synchronisation context. Neither can proceed. ASP.NET Core has no such context, so it avoids this particular deadlock, but blocking calls still waste threads and can cause thread-pool starvation under load.

Rules to follow: make the whole call chain async and await it; never call .Result, .Wait() or .GetAwaiter().GetResult() on a task that is not already complete; avoid async void except for event handlers; pass a CancellationToken through your calls; and use ConfigureAwait(false) in general-purpose libraries. Lock statements cannot contain await, so use SemaphoreSlim.WaitAsync() for async mutual exclusion.

Key points

Do not block on async code

Avoid async void

ConfigureAwait(false) in libraries, CancellationToken everywhere

```
// risky: blocks the calling thread
var data = GetDataAsync().Result;

// correct: async all the way
public async Task<IActionResult> Index(CancellationToken ct)
{
    var data = await GetDataAsync(ct);
    return Ok(data);
}
```

Java

73. What is the difference between JDK, JRE and JVM in Java?

Level: intern Java

The JVM runs Java bytecode, the JRE is the JVM plus the standard libraries needed to run Java programs, and the JDK is the JRE plus the development tools such as the compiler (javac).

You write source code in .java files, the compiler javac (part of the JDK) turns it into platform-independent bytecode in .class files, and the JVM on any operating system loads and executes that bytecode. That is how Java achieves "write once, run anywhere".

Since Java 11 Oracle and most vendors no longer ship a separate JRE download. You install a JDK, or build a trimmed runtime with jlink. The JVM itself also contains the class loader, the bytecode verifier, the JIT compiler that turns hot bytecode into native code, and the garbage collector.

Key points

JDK = JRE + development tools

JRE = JVM + class libraries

74. What is the difference between == and equals() in Java?

Level: intern Java

== compares primitives by value and objects by reference (whether they are the exact same object), while equals() compares the logical content of two objects when the class overrides it.

For String this surprises many people: two strings with the same text created with new are different objects, so == is false even though equals() is true. String literals may be equal under == only because of the string pool, which you should never rely on.

The default equals() inherited from Object behaves like ==. If you override equals() in your own class, you must also override hashCode() so that equal objects have equal hash codes, otherwise they will be lost in a HashMap or HashSet.

Key points

== compares references for objects

equals() compares content if overridden

Override hashCode() whenever you override equals()

```
String a = new String("java");
String b = new String("java");
System.out.println(a == b); // false - different objects
System.out.println(a.equals(b)); // true - same content

Integer x = 127, y = 127;
System.out.println(x == y); // true (cached small values) - do not rely on it
```

75. What is the difference between String, StringBuilder and StringBuffer in Java?

Level: intern Java

String is immutable, StringBuilder is a mutable and fast but not thread-safe buffer, and StringBuffer is a mutable buffer with synchronized methods that is thread-safe but slower.

Because String is immutable, every concatenation creates a new object, so building text in a loop with += is wasteful. Immutability also makes strings safe to share between threads, cache their hash code and use as HashMap keys, and it is what allows the string pool.

Use StringBuilder for building text in a single thread, which is almost always. StringBuffer is a legacy class from Java 1.0 and is rarely needed, since sharing a builder across threads is uncommon and you usually need your own higher-level synchronisation anyway.

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
    sb.append(i).append(', ');
}
String result = sb.toString(); // "0,1,2,3,4,"
```

76. What is the difference between final, finally and finalize in Java?

Level: intern Java

final is a keyword that prevents reassigning a variable, overriding a method or extending a class; finally is a block that always runs after try/catch for cleanup; finalize() was a method called by the garbage collector and is now deprecated.

final on a variable means it can be assigned once (the object it refers to can still change). On a method it cannot

be overridden, and on a class it cannot be subclassed (String is final). finally runs whether or not an exception was thrown, which is why it was used to close resources, but try-with-resources is now the preferred way. finalize() was unreliable because nobody knows when, or whether, the garbage collector will call it. It has been deprecated for removal since Java 18. Use try-with-resources and AutoCloseable, or Cleaner, instead.

```
try (BufferedReader reader = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(reader.readLine());
} catch (IOException e) {
    log.error("read failed", e);
} // reader is closed automatically
```

77. What is the difference between an abstract class and an interface in Java?

Level: junior Java

An abstract class can have instance fields, constructors and a mix of implemented and abstract methods but supports only single inheritance, while an interface defines a contract that a class can implement multiple times and, since Java 8, can include default methods.

Choose an abstract class when subclasses share state and base behaviour (for example a BaseRepository with a connection field). Choose an interface when you want to define a capability that unrelated classes can have, such as Comparable or Runnable, and when you want callers to depend on an abstraction.

Interface fields are implicitly public static final. Since Java 8 an interface can have default and static methods, and since Java 9 private methods, which lets library authors add methods without breaking existing implementations. A class can implement many interfaces but extend only one class.

```
interface Payment {
    void pay(double amount);
    default String currency() { return "USD"; }
}

abstract class BasePayment implements Payment {
    protected final String accountId;
    BasePayment(String accountId) { this.accountId = accountId; }
}
```

78. What is the difference between ArrayList and LinkedList in Java?

Level: junior Java

ArrayList is backed by a resizable array with O(1) index access and O(n) insertion in the middle, while LinkedList is a doubly linked list with O(n) index access and O(1) insertion once you hold the node; in practice ArrayList is faster for most workloads.

ArrayList stores elements contiguously, which is cache-friendly, so iteration and get(i) are very fast. Adding at the end is amortised O(1); inserting or removing in the middle shifts the following elements.

LinkedList stores each element in a node with previous and next pointers, which costs extra memory and scatters nodes around the heap. Finding the element at position i requires walking the list. It only wins when you repeatedly add or remove at the ends or through an iterator. For queue and deque behaviour, ArrayDeque is usually a better choice than LinkedList.

Key points

ArrayList: O(1) get, amortised O(1) add at end

LinkedList: O(n) get, extra memory per element

Default to ArrayList; use ArrayDeque for queues

79. What is the difference between checked and unchecked exceptions in Java?

Level: junior Java

Checked exceptions (subclasses of `Exception` other than `RuntimeException`) must be caught or declared with `throws`, while unchecked exceptions (`RuntimeException` and `Error`) do not have to be handled and usually signal programming bugs.

Checked exceptions such as `IOException` and `SQLException` represent failures the caller can reasonably be expected to handle or recover from, so the compiler forces you to deal with them. Unchecked exceptions such as `NullPointerException`, `IllegalArgumentException` and `IndexOutOfBoundsException` indicate bugs, and fixing the code is better than catching them. `Error` (for example `OutOfMemoryError`) signals serious JVM problems you should not catch.

Good practice: never swallow an exception silently, catch the most specific type, add context when you wrap it, and keep the original as the cause (new `ServiceException("msg", e)`). Many modern frameworks, including Spring, prefer unchecked exceptions because checked ones clutter method signatures.

80. What are the main features of Java 8: lambdas, streams and Optional?

Level: junior Java

Java 8 introduced lambda expressions, functional interfaces, the Stream API for declarative collection processing, `Optional` to represent missing values, default methods in interfaces, method references and the `java.time` date and time API.

A lambda is a short way to write an instance of a functional interface (an interface with one abstract method), such as `Comparator` or `Runnable`. The Stream API lets you describe a pipeline: a source, zero or more intermediate operations such as `filter` and `map` that are lazy, and a terminal operation such as `collect` that triggers the work.

Use `Optional` as a return type to make absence explicit instead of returning `null`, but not for fields or method parameters. Streams do not modify the source collection, and parallel streams should be used with care: they only help for large, CPU-bound, stateless operations.

Key points

Lambdas implement functional interfaces

Stream intermediate operations are lazy until a terminal operation runs

`Optional` is for return values, not fields or parameters

```
List<String> names = users.stream()
    .filter(u -> u.isActive())
    .map(User::getName)
    .sorted()
    .collect(Collectors.toList());

Optional<User> user = repository.findById(id);
String email = user.map(User::getEmail).orElse("unknown");
```

81. How does HashMap work internally in Java?

Level: mid Java

A `HashMap` stores entries in an array of buckets; it computes the key's hash code to choose a bucket, resolves collisions with a linked list that becomes a red-black tree when a bucket grows large, and resizes when the load factor of 0.75 is exceeded.

On `put(key, value)`, `HashMap` calls `key.hashCode()`, spreads the bits, and uses `hash & (capacity - 1)` to pick a bucket index. If the bucket is empty, the entry is stored there. If not, it walks the bucket comparing keys with `equals()`: a matching key has its value replaced, otherwise the new entry is added to the end. That is why correct `hashCode()` and `equals()` implementations are essential for keys.

The default capacity is 16 and the load factor is 0.75, so when more than 12 entries are stored, the table doubles and entries are redistributed. Since Java 8, a bucket with 8 or more entries (when the table has at least 64 buckets) is converted from a linked list to a red-black tree, which keeps worst-case lookup at $O(\log n)$ instead of $O(n)$. Average get and put are $O(1)$. HashMap allows one null key, does not guarantee ordering and is not thread-safe.

Key points

Index = hash of key mapped onto the bucket array

Collisions: linked list, treeified at 8 entries

Resizes at load factor 0.75; not thread-safe

82. What is the difference between HashMap, Hashtable and ConcurrentHashMap in Java?

Level: mid Java

HashMap is not thread-safe, Hashtable is thread-safe by locking the whole map on every method, and ConcurrentHashMap is thread-safe with much finer-grained locking and lock-free reads, so it scales far better.

Using a plain HashMap from several threads can lose updates or, in older versions, even loop forever during a resize. Hashtable and Collections.synchronizedMap() fix this by synchronising every operation on a single lock, so threads queue up behind each other.

ConcurrentHashMap locks only the bucket being modified (using CAS and synchronized on the bucket head since Java 8) and allows reads without locking, so many threads can work in parallel. It does not allow null keys or values. Its compound atomic methods such as computeIfAbsent, merge and putIfAbsent should be used instead of a separate get followed by put, which is a race condition.

```
ConcurrentHashMap<String, Integer> hits = new ConcurrentHashMap<>();
hits.merge("home", 1, Integer::sum); // atomic increment

// racy even on a ConcurrentHashMap:
// if (!map.containsKey(k)) map.put(k, v);
```

83. How does JVM memory management and garbage collection work in Java?

Level: mid Java

The JVM divides memory into a per-thread stack, a shared heap (split into young and old generations) and Metaspace; the garbage collector frees heap objects that are no longer reachable, collecting short-lived young objects frequently and old objects less often.

Each thread has its own stack holding method frames with local variables and references. All objects live on the heap. New objects are allocated in the young generation (Eden and two survivor spaces); objects that survive several minor collections are promoted to the old generation. Class metadata lives in Metaspace, which replaced PermGen in Java 8.

An object is garbage when it can no longer be reached from a GC root such as a thread stack, static field or JNI reference. G1 is the default collector since Java 9 and balances throughput with pause times; ZGC and Shenandoah target very low pauses on large heaps. Typical production problems are OutOfMemoryError from leaks (static collections, listeners, ThreadLocals not cleared) and long pauses from undersized or oversized heaps. Diagnose them with GC logs, heap dumps and tools such as VisualVM or Eclipse MAT.

Key points

Stack per thread, heap shared, Metaspace for class data

Generational collection: young then old

Leaks are objects still reachable by mistake

84. What is the difference between synchronized and volatile in Java, and how do you avoid deadlocks?

Level: senior Java

synchronized gives mutual exclusion and memory visibility for a block of code, while volatile only guarantees that reads and writes of one variable are visible to all threads and does not make compound operations such as count++ atomic.

Without synchronisation a thread may keep a cached copy of a variable and never see another thread's update. volatile forces reads and writes to go through main memory (and establishes a happens-before relationship), so it is suitable for simple flags such as volatile boolean running. It does not protect count++, which is really read, add, write, and can interleave with another thread.

For compound actions use synchronized, a ReentrantLock, or atomic classes like AtomicInteger and LongAdder, which use compare-and-swap. A deadlock needs four conditions: mutual exclusion, hold and wait, no preemption and circular wait. You prevent it by always acquiring locks in a consistent global order, holding locks for as short a time as possible, using tryLock with a timeout, or avoiding shared mutable state altogether with immutable objects, message passing and higher-level utilities from java.util.concurrent. A thread dump (jstack) reports detected deadlocks.

Key points

volatile: visibility, not atomicity

synchronized / locks: mutual exclusion and visibility

Prevent deadlocks with consistent lock ordering and timeouts

```
private volatile boolean running = true; // visibility only
private final AtomicInteger counter = new AtomicInteger();

void work() {
    while (running) {
        counter.incrementAndGet(); // atomic, no lock
    }
}
```

Python

85. What is the difference between a list and a tuple in Python?

Level: intern Python

A list is mutable and can be changed after creation, while a tuple is immutable; tuples are slightly faster, use less memory and can be used as dictionary keys when their items are hashable.

Use a list for a collection that grows, shrinks or is reordered, such as the items in a shopping cart. Use a tuple for a fixed group of values that belong together, such as a coordinate (x, y) or a database row. Tuples also signal to the reader that the data should not change.

Immutability applies to the tuple itself: if a tuple contains a list, that list can still be modified. A one-item tuple needs a trailing comma, (5,), because (5) is just the number 5 in parentheses.

```
items = [1, 2, 3]
items.append(4) # fine

point = (10, 20)
point[0] = 5 # TypeError: 'tuple' object does not support item assignment

locations = {(10, 20): "home"} # tuples can be dict keys
```

86. What is the difference between is and == in Python?

Level: intern Python

== checks whether two objects have equal values, while is checks whether they are the exact same object in memory.

== calls the object's `__eq__` method. is compares identity (the same as comparing `id(a) == id(b)`). Two separate lists with the same content are == but not is.

Use is for singletons, mainly None (if x is None), True and False. Do not use is to compare numbers or strings: CPython caches small integers and some strings, so the result can look right in a quick test and fail in real code.

```
a = [1, 2]
b = [1, 2]
a == b # True (same value)
a is b # False (different objects)
c = a
c is a # True

if result is None:
    ...
```

87. What are *args and **kwargs in Python?

Level: junior Python

***args collects extra positional arguments into a tuple and **kwargs collects extra keyword arguments into a dictionary, which lets a function accept a variable number of arguments.**

The names args and kwargs are conventions; the stars are what matter. They are useful for wrappers and decorators that must pass any arguments through to another function, and for APIs that accept optional settings.

The same stars work at the call site to unpack: `print(*numbers)` spreads a list into separate arguments and `func(**options)` spreads a dictionary into keyword arguments. The order in a signature is: normal parameters, *args, keyword-only parameters, **kwargs.

```
def report(title, *args, **kwargs):
    print(title, args, kwargs)

report("Sales", 1, 2, region="EU", year=2026)
# Sales (1, 2) {'region': 'EU', 'year': 2026}

nums = [3, 4]
opts = {"sep": "-"}
print(*nums, **opts) # 3-4
```

88. Why should you not use a mutable default argument in Python?

Level: junior Python

Default argument values are evaluated once when the function is defined, so a default list or dictionary is shared between every call that does not pass its own, causing data to leak between calls.

This is one of the most common Python gotchas. The fix is to use None as the default and create a new object inside the function.

```
def add_item(item, items=[]): # bug: one shared list
    items.append(item)
    return items

add_item("a") # ['a']
add_item("b") # ['a', 'b'] <- surprise

def add_item(item, items=None): # correct
    if items is None:
        items = []
    items.append(item)
    return items
```

89. What is the difference between a shallow copy and a deep copy in Python?

Level: junior Python

A shallow copy creates a new container but reuses the same nested objects, while a deep copy recursively copies everything so the new object is fully independent.

With a shallow copy (`copy.copy()`, `list.copy()`, slicing `items[:]`), changing a nested list in the copy also changes it in the original, because both point to the same inner object. `copy.deepcopy()` duplicates the nested objects as well, handling circular references, but it is slower and uses more memory.

Plain assignment `b = a` does not copy at all: both names refer to the same object.

```
import copy

original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0].append(99)
print(shallow[0]) # [1, 2, 99] - shared inner list
print(deep[0]) # [1, 2] - independent
```

90. What is the difference between `@staticmethod` and `@classmethod` in Python?

Level: junior Python

A classmethod receives the class as its first argument (`cls`) and can create or configure instances, while a staticmethod receives no automatic argument and is just a function placed inside the class namespace.

Use `@classmethod` for alternative constructors such as `Date.from_string(...)`, and because it uses `cls`, it works correctly for subclasses. Use `@staticmethod` for a utility function that is related to the class but needs neither the instance nor the class. An ordinary method receives `self`, the instance.

```
class User:
    def __init__(self, name):
        self.name = name

    @classmethod
    def from_email(cls, email):
        return cls(email.split("@")[0])

    @staticmethod
    def is_valid_email(email):
        return "@" in email
```

91. What are decorators in Python and how do they work?

Level: mid Python

A decorator is a function that takes another function and returns a modified version of it, letting you add behaviour such as logging, caching or access control without changing the original code.

The `@decorator` syntax is shorthand for `func = decorator(func)`. A typical decorator defines an inner wrapper function that runs code before and after calling the original function, and returns that wrapper. Use `functools.wraps` so the wrapped function keeps its name and docstring.

Decorators rely on functions being first-class objects and on closures. Common built-in ones include `@property`, `@staticmethod`, `@classmethod` and `functools.lru_cache`. Frameworks use them heavily, for example `@app.route("/")` in Flask.

```
import functools, time

def timed(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        print(f'{func.__name__} took {time.perf_counter() - start:.3f}s')
        return result
    return wrapper

@timed
def load_data():
    ...
```

92. What are generators and the yield keyword in Python?

Level: mid Python

A generator is a function that uses `yield` to produce values one at a time and remembers its state between calls, so it can process large or infinite sequences without holding everything in memory.

Calling a generator function does not run it; it returns an iterator. Each `next()` call resumes the function until the next `yield`. A list of ten million numbers takes hundreds of megabytes, while a generator producing them one by one uses almost none.

A generator expression, `(x * x for x in range(10**7))`, is the lazy counterpart of a list comprehension. Generators can only be consumed once. They are ideal for reading big files line by line, streaming API pages and building data pipelines.

Key points

`yield` pauses the function and keeps local state

Lazy evaluation saves memory

A generator can be iterated only once

```
def read_large_file(path):
    with open(path) as f:
        for line in f:
            yield line.strip()

for row in read_large_file("big.csv"):
    process(row) # one line in memory at a time

total = sum(x * x for x in range(1_000_000)) # no list created
```

93. How does Python manage memory and garbage collection?

Level: mid Python

CPython frees objects mainly through reference counting, deleting an object as soon as nothing refers to it, and uses a cyclic garbage collector to clean up groups of objects that reference each other.

Every object keeps a count of references to it. When the count reaches zero its memory is released immediately. This makes cleanup predictable, but reference counting cannot free cycles, such as two objects that point at each other. The gc module's generational collector periodically finds and frees those.

Memory leaks in Python are usually objects kept alive by long-lived references: global lists or dictionaries, caches without limits, or closures. Use weakref for caches that should not keep objects alive, __slots__ to shrink many small objects, generators to stream data, and tools such as tracemalloc to see what is allocating memory.

94. What is the GIL in Python and how does it affect multithreading?

Level: senior Python

The Global Interpreter Lock (GIL) in CPython lets only one thread execute Python bytecode at a time, so threads do not speed up CPU-bound code but still help with I/O-bound work.

The GIL exists to keep CPython's memory management (reference counting) simple and safe. A thread releases it while waiting on I/O such as network calls and file reads, so many I/O-bound threads can overlap their waiting. For pure-Python CPU-bound loops, threads take turns and the program is no faster, sometimes slower.

To use multiple cores, use multiprocessing or concurrent.futures.ProcessPoolExecutor, which run separate interpreters, or push the heavy work into C extensions such as NumPy that release the GIL. Recent Python versions have an experimental free-threaded build that removes the GIL, but the default interpreter still has it.

Key points

One thread runs Python bytecode at a time in CPython

Threads are fine for I/O-bound work

Use processes for CPU-bound work

95. What is the difference between threading, multiprocessing and asyncio in Python?

Level: senior Python

Threading runs concurrent threads that share memory and suit blocking I/O, multiprocessing runs separate processes to use multiple CPU cores for CPU-bound work, and asyncio runs many non-blocking tasks cooperatively on one thread for high-volume I/O.

Choose by the type of bottleneck. For CPU-bound work such as image processing or number crunching, use multiprocessing so each worker has its own GIL, and accept the cost of starting processes and passing data between them. For I/O-bound work with libraries that block, such as requests, threads are simple and effective.

For thousands of concurrent network connections, asyncio with async/await has the lowest overhead because tasks yield control at each await, but every library in the path must be async-aware, and one blocking call stalls the whole event loop. Many production systems combine them, for example an async web server that sends CPU-heavy jobs to a process pool or task queue.

Key points

CPU-bound: multiprocessing

Blocking I/O: threading

Massive concurrent I/O: asyncio

```
import asyncio, httpx

async def fetch(client, url):
    r = await client.get(url)
    return r.status_code

async def main(urls):
    async with httpx.AsyncClient() as client:
        return await asyncio.gather(*(fetch(client, u) for u in urls))

asyncio.run(main(["https://example.com"] * 5))
```

Node.js

96. What is Node.js and why is it called single-threaded?

Level: intern Node.js

Node.js is a JavaScript runtime built on Chrome's V8 engine that runs your JavaScript on a single main thread and uses non-blocking I/O, so one thread can handle many concurrent connections.

Your JavaScript code runs on one thread, so there is no shared-memory race condition between your own functions. But Node.js is not single-threaded overall: the libuv library handles network I/O through the operating system's asynchronous mechanisms and uses a thread pool (four threads by default) for file system access, DNS lookups, compression and some crypto functions.

When a request arrives and needs a database result, Node starts the operation, registers a callback and moves on to other work instead of waiting. This model is excellent for I/O-heavy applications such as APIs, chat and streaming, and poor for long CPU-bound computation on the main thread, which blocks every other request.

Key points

Your JavaScript runs on one thread

libuv uses the OS and a thread pool for I/O

Great for I/O-bound work, risky for CPU-bound work

97. What is the difference between package.json and package-lock.json?

Level: intern Node.js

package.json lists your project's dependencies with allowed version ranges, while package-lock.json records the exact versions that were installed so every machine gets an identical dependency tree.

In package.json, a range such as ^4.18.0 allows any compatible 4.x update. The lock file pins the exact resolved version and the integrity hash of every package, including nested dependencies. Commit both files, and use npm ci in CI and production builds: it installs exactly what the lock file says and fails if the two files disagree, while npm install may update the lock file.

Also know the difference between dependencies (needed at runtime) and devDependencies (build and test tools), and run npm audit regularly to check for known vulnerabilities.

98. What is the difference between require and import in Node.js?

Level: junior Node.js

require is the CommonJS module system, which loads modules synchronously at runtime, while import is the ES module standard, which is statically analysed, asynchronous and supports top-level await and tree shaking.

CommonJS uses require() and module.exports; it is the default in .js files unless your package.json says otherwise. ES modules use import and export, enabled by "type": "module" in package.json or by using .mjs files.

Imports are hoisted and resolved before the code runs, which allows tools to remove unused exports.

Differences to remember: ES modules require file extensions in relative imports, have no `__dirname` (use `import.meta.url`), and can only be loaded from CommonJS with `dynamic import()`. `Dynamic import()` works in both systems.

```
// CommonJS
const express = require("express");
module.exports = { createApp };

// ES modules
import express from "express";
export function createApp() {}
```

99. What is middleware in Express and how does it work?

Level: junior Node.js

Middleware are functions that receive the request, response and a next callback; they run in the order they are registered and can modify the request, end the response or pass control on with `next()`.

Each middleware can do work such as logging, parsing JSON bodies, checking authentication or setting headers, and then either sends a response or calls `next()` to continue to the next function. Order matters: a body parser must be registered before the routes that read `req.body`.

Error-handling middleware has four parameters (`err`, `req`, `res`, `next`) and is registered last. In Express 4, an error thrown inside an async handler is not caught automatically; you must catch it and call `next(err)` or use a small wrapper (Express 5 handles rejected promises for you).

```
app.use(express.json());

function requireAuth(req, res, next) {
  if (!req.headers.authorization) return res.status(401).json({ error: "Unauthorized" });
  next();
}

app.get("/profile", requireAuth, (req, res) => res.json({ ok: true }));

app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ error: "Something went wrong" });
});
```

100. What is the difference between `process.nextTick`, `setImmediate` and `setTimeout` in Node.js?

Level: mid Node.js

`process.nextTick` runs its callback right after the current operation, before the event loop continues; `setTimeout` runs in the timers phase after a minimum delay; and `setImmediate` runs in the check phase right after the poll (I/O) phase.

The Node.js event loop has phases: timers, pending callbacks, poll (I/O), check (`setImmediate`) and close callbacks. The `nextTick` queue and the promise microtask queue are drained between phases, and the `nextTick` queue goes first. Recursively calling `process.nextTick` can therefore starve I/O.

From the main module, the order of `setTimeout(fn, 0)` and `setImmediate(fn)` depends on process timing and is not guaranteed. Inside an I/O callback, `setImmediate` always runs first. Prefer `setImmediate` or `queueMicrotask` for deferring work, and reserve `nextTick` for cases such as emitting an event after a constructor returns.

```
setTimeout(() => console.log("timeout"), 0);
setImmediate(() => console.log("immediate"));
Promise.resolve().then(() => console.log("promise"));
process.nextTick(() => console.log("nextTick"));
// nextTick, promise, then timeout/immediate (order of last two varies)
```

101. What are streams in Node.js and why should you use them?

Level: mid Node.js

Streams process data piece by piece as it arrives instead of loading it all into memory, which lets you handle large files and network data efficiently; they come in Readable, Writable, Duplex and Transform types.

Reading a 2 GB file with `fs.readFile` loads all 2 GB into memory and can crash the process. A readable stream emits small chunks, so memory use stays flat. Streams can be chained: read, compress, then write.

Backpressure is the key concept: when the destination is slower than the source, the stream signals the source to pause. Use `stream.pipeline()` (or `pipeline` from `stream/promises`) instead of manual `.pipe()` chains, because it handles errors and cleans up all streams properly.

```
import { pipeline } from "node:stream/promises";
import { createReadStream, createWriteStream } from "node:fs";
import { createGzip } from "node:zlib";

await pipeline(
  createReadStream("access.log"),
  createGzip(),
  createWriteStream("access.log.gz")
);
```

102. What is the difference between JWT and session-based authentication?

Level: mid Node.js

Session authentication stores the login state on the server and gives the browser a session ID cookie, while JWT authentication gives the client a signed token that contains the user's claims so the server can verify it without storing state.

Sessions are simple and easy to revoke: delete the server-side record and the user is logged out immediately, but you need shared storage such as Redis once you run several servers. A JWT is self-contained and works well across services, but you cannot invalidate a single token before it expires unless you keep a deny list or use short lifetimes with refresh tokens.

Whichever you pick, store tokens in `HttpOnly`, `Secure`, `SameSite` cookies rather than `localStorage` to limit XSS risk, hash passwords with `bcrypt` or `Argon2`, use short access-token lifetimes, and verify the signature algorithm explicitly. Authentication confirms who you are; authorisation decides what you may do.

Key points

Session: state on the server, easy to revoke

JWT: stateless, hard to revoke before expiry

Use `HttpOnly` `Secure` cookies and short token lifetimes

103. How do you secure a Node.js Express API?

Level: mid Node.js

Validate all input, use parameterised queries, add security headers with `helmet`, rate-limit requests, restrict CORS, hash passwords, keep secrets in environment variables, and keep dependencies updated

and audited.

Input: validate and sanitise every body, query and parameter with a schema library such as Zod or Joi, and never build SQL or shell commands by string concatenation, which prevents injection. Output: do not send stack traces or internal error messages to clients.

Transport and headers: use HTTPS, set helmet() for secure headers, configure CORS to allow only your own origins, and set cookies as HttpOnly, Secure and SameSite. Abuse: add rate limiting (for example express-rate-limit), limit request body size and add stricter limits on login and password reset endpoints.

Operations: store secrets in environment variables or a secret manager, run npm audit, pin dependency versions with a lock file, run the process as a non-root user and log security events.

```
import helmet from "helmet";
import rateLimit from "express-rate-limit";
import cors from "cors";

app.use(helmet());
app.use(cors({ origin: ["https://www.example.com"] }));
app.use(express.json({ limit: "100kb" }));
app.use("/api/login", rateLimit({ windowMs: 15 * 60 * 1000, max: 10 }));
```

104. How do you scale a Node.js application and avoid blocking the event loop?

Level: senior Node.js

Keep the main thread free by moving CPU-heavy work to worker threads or a job queue, and scale across cores and machines by running multiple processes (cluster or PM2) behind a load balancer with shared, external state.

One Node.js process uses one core for your JavaScript. Run one process per core with the cluster module, PM2 or by running several containers behind a load balancer. Because the processes do not share memory, session data, caches and rate-limit counters must move to an external store such as Redis.

Blocking the event loop stalls every request in that process. Common causes are synchronous file or crypto calls (fs.readFileSync, bcrypt sync versions), parsing huge JSON documents, heavy loops, and regular expressions with catastrophic backtracking. Fix them with async APIs, worker_threads for CPU work, queues such as BullMQ for background jobs, streaming instead of buffering, and input size limits. Measure event loop delay with perf_hooks.monitorEventLoopDelay() and alert on it, and make the app stateless so instances can be added or removed freely.

Key points

One process per core, behind a load balancer

Move CPU work to workers or queues

Keep state in Redis or a database, not in process memory

```
import { Worker } from "node:worker_threads";

function runInWorker(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker("./hash-worker.js", { workerData: data });
    worker.once("message", resolve);
    worker.once("error", reject);
  });
}
```

System Design

105. What is the difference between horizontal and vertical scaling?

Level: junior System Design

Vertical scaling means giving one machine more CPU, memory or disk, while horizontal scaling means adding more machines and spreading the load across them.

Vertical scaling is the simplest option and needs no code changes, but it has a hard ceiling, usually costs more per unit of capacity at the top end, and leaves a single point of failure. Horizontal scaling has almost no ceiling and improves availability, but the application must be designed for it: stateless servers, shared sessions and caches, and a load balancer in front.

Databases are the hard part. Stateless web servers scale out easily; a relational database is usually scaled up first, then read replicas are added, and finally the data is partitioned (sharded) when one primary can no longer handle the writes.

Key points

Vertical: bigger machine. Horizontal: more machines

Horizontal scaling needs stateless services

Databases are scaled up, then replicated, then sharded

106. What is a load balancer and what are the common load balancing algorithms?

Level: junior System Design

A load balancer distributes incoming requests across several servers to improve capacity and availability, using algorithms such as round robin, least connections or consistent hashing.

Round robin sends each request to the next server in turn and works when requests cost about the same. Least connections sends the request to the server with the fewest active connections, which is better when request times vary. IP hash or consistent hashing sends the same client or key to the same server, which helps with caches and sticky sessions.

A load balancer also runs health checks and stops sending traffic to failed servers. Layer 4 balancers route by IP and port and are very fast, while layer 7 balancers understand HTTP and can route by path, host or header and terminate TLS. The load balancer itself must not be a single point of failure, so it is deployed redundantly.

107. When should you use a SQL database vs a NoSQL database?

Level: mid System Design

Choose SQL when your data is relational and you need transactions, joins and strong consistency; choose NoSQL when you need flexible schemas, very high write throughput or horizontal scaling for a specific access pattern.

Relational databases such as PostgreSQL and MySQL give you ACID transactions, constraints and ad-hoc queries, which suit orders, payments, users and most business applications. They are a safe default and scale a long way with indexes, replicas and caching.

NoSQL is a family: document stores (MongoDB) for flexible, nested records; key-value stores (Redis, DynamoDB) for very fast lookups by key; wide-column stores (Cassandra) for huge write-heavy workloads; graph databases for relationship queries. Pick by access pattern, not fashion. Many systems use both, for example PostgreSQL as the source of truth and Redis for caching and rate limits.

Key points

SQL: relations, transactions, ad-hoc queries

NoSQL: specific access patterns and horizontal scale

Polyglot persistence is common

108. What is caching and what are the common cache invalidation strategies?

Level: mid System Design

Caching stores frequently read data in fast storage such as memory to reduce latency and database

load; the main strategies are cache-aside, write-through, write-back and time-based expiry (TTL).

In cache-aside, the application checks the cache first, loads from the database on a miss and then writes the result to the cache. It is the most common pattern. Write-through writes to the cache and the database together, keeping them in sync at the cost of write latency. Write-back writes to the cache first and flushes to the database later, which is fast but risks data loss if the cache fails.

Invalidation is the hard part. A TTL accepts a bounded amount of stale data; explicit invalidation deletes or updates the key whenever the source changes; versioned keys avoid the problem by creating a new key. When the cache is full, an eviction policy such as LRU decides what to remove. Watch for a cache stampede, where many requests rebuild an expired popular key at once, and mitigate it with request coalescing, locks or slightly randomised TTLs.

Cache-aside with a five-minute TTL.

```
async function getProduct(id) {
  const cached = await redis.get("product:" + id);
  if (cached) return JSON.parse(cached);

  const product = await db.products.findById(id);
  await redis.set("product:" + id, JSON.stringify(product), "EX", 300);
  return product;
}
```

109. What is the difference between REST, GraphQL and gRPC?

Level: mid System Design

REST exposes resources over HTTP URLs and is simple and cache-friendly, GraphQL lets clients ask for exactly the fields they need from a single endpoint, and gRPC uses binary Protocol Buffers over HTTP/2 for fast service-to-service calls.

REST is the default for public APIs: it uses standard HTTP methods and status codes, works with browser and CDN caching and is easy to debug. Its weaknesses are over-fetching and under-fetching, which can force clients to make several round trips.

GraphQL solves that for complex front ends, such as mobile apps needing different shapes of data, but caching is harder and you must guard against expensive queries with depth and cost limits. gRPC gives strongly typed contracts, streaming and low latency, which suits internal microservices, but browsers need a proxy to call it directly.

110. What is idempotency in REST APIs and why does it matter?

Level: mid System Design

An operation is idempotent if repeating it any number of times has the same effect as doing it once; it matters because network retries can otherwise create duplicate orders or payments.

In HTTP, GET, PUT and DELETE are defined as idempotent, while POST is not. Clients and gateways retry requests after timeouts, so a non-idempotent payment endpoint could charge a customer twice.

The standard fix is an idempotency key: the client sends a unique key in a header, the server stores the key with the result of the first request, and any repeat with the same key returns the stored result instead of running again. Store the key and the business change in the same transaction, and expire old keys after a sensible period.

```
POST /payments
Idempotency-Key: 7f3c1a2e-9d41-4b6e-8a53-0c1f6d2b9e11
Content-Type: application/json

{ "orderId": 1042, "amount": 4999 }
```

111. How do you design a rate limiter?

Level: mid System Design

Choose an algorithm such as token bucket or sliding window, store a counter per client (by user ID, API key or IP) in a fast shared store like Redis, and return HTTP 429 with a Retry-After header when the limit is exceeded.

A token bucket gives each client a bucket that refills at a steady rate; each request takes one token, and the bucket size allows short bursts. A fixed window counter is simplest but allows double the limit across a window boundary; a sliding window avoids that at the cost of more storage.

In a distributed system the counters must be shared, so use Redis with atomic operations (INCR with EXPIRE, or a Lua script) so that two servers cannot both let the last request through. Decide what happens when Redis is down (fail open to keep serving, or fail closed to protect the system), apply stricter limits to expensive or sensitive endpoints such as login, and return the limit headers so well-behaved clients can slow down.

112. How do you design a URL shortener like bit.ly?

Level: senior System Design

Generate a short unique code for each long URL (for example by base62-encoding a unique ID), store the mapping in a key-value or relational database, serve redirects through a cache because reads far outnumber writes, and record analytics asynchronously.

Clarify requirements first: how many URLs per day, how long links live, whether custom aliases are required and what analytics are needed. A service with 100 million new links per year and a 100:1 read-to-write ratio is read-heavy, which drives the design toward caching and replicas.

API: POST /links creates a link, and GET /{code} responds with a 301 or 302 redirect. Use 302 if you need to count every click, because browsers cache a 301. Code generation: a counter encoded in base62 gives short, collision-free codes (7 characters give about 3.5 trillion combinations); to avoid a single counter bottleneck, hand each app server a range of IDs. Hashing the URL is an alternative but needs collision handling.

Storage: one table keyed by code holding the long URL, owner and expiry. Put Redis in front for hot links, and add read replicas or partition by code as data grows. Send click events to a queue and aggregate them offline so analytics never slow the redirect. Add rate limiting and URL validation to prevent abuse.

Key points

Clarify scale and read/write ratio first

Base62 of a unique ID gives short collision-free codes

Cache the redirect path, make analytics asynchronous

```
const ALPHABET = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

function toBase62(num) {
  let out = "";
  do {
    out = ALPHABET[num % 62] + out;
    num = Math.floor(num / 62);
  } while (num > 0);
  return out;
}

toBase62(125); // "21"
```

113. What is the CAP theorem and what does it mean in practice?

Level: senior System Design

The CAP theorem says that during a network partition a distributed system must choose between consistency (every read sees the latest write) and availability (every request gets a response); you

cannot have both while the partition lasts.

Network partitions are a fact of life in distributed systems, so the real choice is what to give up when one happens. A CP system, such as a strongly consistent store using consensus, may refuse requests on the minority side of a partition to avoid returning stale or conflicting data. An AP system, such as Dynamo-style stores, keeps answering and reconciles differences later, giving eventual consistency.

In practice the choice is per operation, not per company. A bank balance transfer needs consistency, while a social media like count can be stale for a few seconds. Mention also that when there is no partition, systems still trade latency against consistency (the PACELC extension).

114. What is the difference between database sharding and replication?

Level: senior System Design

Replication copies the same data to several servers to improve read capacity and availability, while sharding splits the data across servers so each holds only a part, which increases write capacity and total storage.

With replication, one primary accepts writes and replicas follow it. Reads can be spread across replicas and a replica can be promoted if the primary fails, but replicas can lag, so reads from them may be slightly stale, and write throughput is still limited by the single primary.

Sharding partitions rows by a shard key, such as `user_id`, so each shard handles a fraction of the traffic. Choosing the key is the critical decision: a bad key creates hot shards, and cross-shard queries, joins and transactions become expensive or impossible. Range sharding supports range queries but can be uneven; hash sharding spreads load evenly; consistent hashing makes adding shards cheaper. Shard only when replication, caching, indexing and a bigger primary no longer suffice, because it adds permanent operational complexity.

Key points

Replication: copies, for reads and failover

Sharding: partitions, for writes and storage

Shard key choice decides hot spots and query cost

115. Monolith vs microservices: which should you choose?

Level: senior System Design

Start with a well-structured monolith unless you have several teams and clear service boundaries, because microservices add distributed-system complexity such as network failures, data consistency and operational overhead.

A monolith is one deployable unit: easy to develop, test, debug and deploy, with in-process calls and straightforward transactions. Its problems appear at scale: long build and release cycles, teams blocking each other, and the inability to scale one hot component independently.

Microservices split the system by business capability, so teams can deploy and scale independently and choose different technologies. The costs are real: network latency and partial failure, distributed transactions and eventual consistency, service discovery, monitoring and tracing across services, and a lot more infrastructure. A pragmatic path is a modular monolith with clear internal boundaries, extracting a service only when there is a concrete reason, such as independent scaling or a separate team owning it.

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>