

Mid-Level Developer Interview Questions and Answers

Mid-level (2-5 years) | All technologies

Interview questions and answers for mid-level software engineers with 2-5 years of experience: performance, concurrency, database design, testing and trade-offs.

33 questions Read online: <https://www.freedevtemplates.com/interview-questions/level/mid-level>

JavaScript

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

Level: mid JavaScript

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

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

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

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

Key points

One call stack, many queues

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

Blocking the stack blocks rendering and input

Synchronous code first, then microtasks, then timers.

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

2. What is prototypal inheritance in JavaScript?

Level: mid JavaScript

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

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

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

Object.getPrototypeOf(obj).

Key points

Lookups follow the prototype chain

class is syntactic sugar over prototypes

Shared methods live on the prototype, not on each instance

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

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

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

Level: mid JavaScript

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

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

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

Key points

Debounce: run after the burst ends

Throttle: run at a steady maximum rate

Both rely on closures to keep the timer between calls

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

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

React

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

Level: mid React

useMemo caches the result of a calculation, while useCallback caches a function definition; both only

recompute when their dependencies change.

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

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

Key points

useMemo returns a value, useCallback returns a function

Both depend on a dependency array

Measure before optimising; memoisation has a cost

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

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

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

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

Level: mid React

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

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

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

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

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

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

Level: mid React

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

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

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

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

Key points

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

SSR and SSG improve first paint and SEO

Hydration makes server-rendered HTML interactive

HTML & CSS

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

Level: mid HTML & CSS

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

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

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

8. How does the browser render a web page?

Level: mid HTML & CSS

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

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

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

Key points

DOM + CSSOM render tree layout paint composite

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

Animate transform and opacity to avoid layout

TypeScript

9. What is type narrowing in TypeScript?

Level: mid TypeScript

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

Control-flow analysis follows your if and switch. `typeof x === "string"` narrows `string | number` to `string` in that

branch. `x instanceof Date` works for classes. "width" in value distinguishes object shapes. A user-defined guard is a function whose return type is value is Fish.

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

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

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

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

Level: mid TypeScript

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

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

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

11. How does TypeScript type inference work?

Level: mid TypeScript

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

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

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

SQL

12. What are ACID properties in a database?

Level: mid SQL

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

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

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

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

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

Level: mid SQL

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

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

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

Key points

Faster reads, slower writes, more storage

Column order in composite indexes matters

Functions on the column and leading wildcards prevent index use

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

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

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

Level: mid SQL

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

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

A non-clustered index stores the indexed columns plus a pointer (the clustered key or a row locator). A lookup finds the entry and then does a second lookup to fetch the remaining columns, unless the index covers the query by including all needed columns. Choose a narrow, ever-increasing clustered key, because wide or random keys (such as random GUIDs) are copied into every non-clustered index and cause page splits.

C#

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

Level: mid C#

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

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

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

Key points

`IQueryable`: expression trees translated to SQL

`IEnumerable`: in-memory iteration with delegates

Materialise (`ToList`) as late as possible

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

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

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

Level: mid C#

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

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

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

Key points

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

Never inject a scoped service into a singleton

Singletons must be thread-safe

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

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

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

Level: mid C#

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

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

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

Key points

Task = promise of work, not a thread

await frees the thread during I/O

Use Task.WhenAll for independent operations

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

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

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

Level: mid C#

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

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

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

Key points

Generational GC: gen 0, 1, 2 and the large object heap

Dispose releases unmanaged resources; GC releases memory

Static references and event handlers keep objects alive

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

Java

19. How does HashMap work internally in Java?

Level: mid Java

A HashMap stores entries in an array of buckets; it computes the key's hash code to choose a bucket,

resolves collisions with a linked list that becomes a red-black tree when a bucket grows large, and resizes when the load factor of 0.75 is exceeded.

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

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

Key points

Index = hash of key mapped onto the bucket array

Collisions: linked list, treeified at 8 entries

Resizes at load factor 0.75; not thread-safe

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

Level: mid Java

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

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

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

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

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

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

Level: mid Java

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

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

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

heaps. Diagnose them with GC logs, heap dumps and tools such as VisualVM or Eclipse MAT.

Key points

Stack per thread, heap shared, Metaspace for class data

Generational collection: young then old

Leaks are objects still reachable by mistake

Python

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

Level: mid Python

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

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

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

```
import functools, time

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

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

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

Level: mid Python

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

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

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

Key points

`yield` pauses the function and keeps local state

Lazy evaluation saves memory

A generator can be iterated only once

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

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

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

24. How does Python manage memory and garbage collection?

Level: mid Python

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

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

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

Node.js

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

Level: mid Node.js

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

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

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

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

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

Level: mid Node.js

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

Reading a 2 GB file with `fs.readFile` loads all 2 GB into memory and can crash the process. A readable stream

emits small chunks, so memory use stays flat. Streams can be chained: read, compress, then write.

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

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

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

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

Level: mid Node.js

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

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

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

Key points

Session: state on the server, easy to revoke

JWT: stateless, hard to revoke before expiry

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

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

Level: mid Node.js

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

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

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

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

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

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

System Design

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

Level: mid System Design

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

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

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

Key points

SQL: relations, transactions, ad-hoc queries

NoSQL: specific access patterns and horizontal scale

Polyglot persistence is common

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

Level: mid System Design

Caching stores frequently read data in fast storage such as memory to reduce latency and database load; the main strategies are cache-aside, write-through, write-back and time-based expiry (TTL).

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

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

Cache-aside with a five-minute TTL.

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

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

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

Level: mid System Design

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

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

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

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

Level: mid System Design

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

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

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

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

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

33. How do you design a rate limiter?

Level: mid System Design

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

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

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

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