BACKEND INTERVIEW · @CHALKESAHIL

Backend interview: the 50 questions that come up most, with answers

This guide is for students and early-career engineers preparing for backend developer interviews. It covers the 50 questions that come up most, from HTTP and databases to caching, queues, scaling and security. After working through it, you can explain each topic in plain words, give a short example, and handle the usual follow-up question.

How to use this PDF

1. Read the question and try to answer it aloud in about a minute before reading the answer.
2. Run each code example yourself in Node.js or Python and change one line to see what breaks.
3. Read the tip line last. It is usually the follow-up question an interviewer asks next.
4. After each section, answer the two practice questions without looking, then revisit the ones you missed.
5. Do one section a day, then repeat all eight sections as a mock interview with a friend.

INSIDE

01 HTTP and REST
7 questions

02 APIs and auth
6 questions

03 Databases and SQL
6 questions

04 Transactions and locks
6 questions

05 Caching
6 questions

06 Async and queues
6 questions

07 Scaling, reliability
6 questions

08 Security and testing
7 questions

SECTION 01 / 08

HTTP and REST

Every backend interview starts here. HTTP is the language clients and servers speak, and REST is the common style for designing APIs on top of it. Know the vocabulary cold and be ready to explain it with a small example.

01 What happens on typing a URL

Medium

What happens between typing a URL in the browser and seeing the page?

First the browser asks DNS (the internet's phone book) to turn the domain name into an IP address. Then it opens a TCP connection to that IP, and for https it runs a TLS handshake so the traffic is encrypted. Next it sends an HTTP request such as GET /home. The server may pass it through a load balancer, run your application code, read from a database or cache, and send back an HTTP response with a status code, headers and a body. The browser reads the HTML, fetches the CSS, JavaScript and images it points to, and paints the page. Caches can skip steps at almost every stage.

```
GET /home HTTP/1.1
Host: example.com
Accept: text/html

HTTP/1.1 200 OK
Content-Type: text/html
Cache-Control: max-age=60
```

Interview tip. Name the steps in order, then pick one (like DNS or TLS) and go deeper if asked.

02 HTTP methods and idempotency

Easy

What are the main HTTP methods, and which of them are safe or idempotent?

GET reads data, POST creates something or triggers an action, PUT replaces a whole resource, PATCH changes part of it, and DELETE removes it. A method is safe if it does not change server state, so only GET (and HEAD) are safe. A method is idempotent if calling it many times has the same effect as calling it once. GET, PUT and DELETE are idempotent. POST is not, because two identical POSTs can create two orders. This matters for retries: a client can retry a PUT after a timeout without fear, but retrying a POST can duplicate work unless you add an idempotency key.

```
PUT /users/7 {"name":"Asha"}   // same result if sent 5 times
POST /orders {"item":"book"}    // 5 times may create 5 orders
```

Interview tip. Interviewers love the follow-up: how do you make POST safe to retry? Answer: an idempotency key.

03 Common HTTP status codes

Easy

Which HTTP status codes should every backend developer know, and what do they mean?

Status codes are grouped by their first digit. 2xx means success: 200 OK, 201 Created (after a POST that made something), 204 No Content. 3xx means redirect: 301 permanent, 302 temporary, 304 Not Modified (use your cached copy). 4xx means the client made a mistake: 400 bad input, 401 not logged in, 403 logged in but not allowed, 404 not found, 409 conflict, 429 too many requests. 5xx means the server failed: 500 unexpected error, 502 bad gateway, 503 unavailable, 504 gateway timeout. Returning the right code lets clients, caches and monitoring tools react correctly without parsing your message text.

```
app.get('/users/:id', (req, res) => {
  const u = db.find(req.params.id);
  if (!u) return res.status(404).json({ error: 'not found' });
  res.status(200).json(u);
});
```

> **Interview tip.** Be ready to explain 401 vs 403. 401 means unauthenticated, 403 means authenticated but forbidden.

04 PUT vs PATCH

Easy

What is the difference between PUT and PATCH?

PUT replaces the entire resource with the body you send. If you leave out a field, it is treated as removed or reset. PATCH applies a partial update, so you send only the fields that change. PUT is idempotent by definition. PATCH can be idempotent (set email to x) or not (increment a counter), depending on how you design it. In practice many teams use PATCH for most updates because clients rarely hold the full object. Whichever you choose, document it clearly and validate input, so a missing field does not silently wipe data.

```
PUT   /users/7  {"name":"Asha","email":"a@x.com","city":"Pune"}
PATCH /users/7  {"city":"Mumbai"}
```

> **Interview tip.** Say that PUT sends the whole object and PATCH sends only the change. That one line usually satisfies the interviewer.

05 What makes an API RESTful

Medium

What is REST, and what rules make an API RESTful?

REST is an architectural style for APIs over HTTP. Its core ideas: resources are nouns identified by URLs (/users/7), and HTTP methods are the verbs on them. The server is stateless, which means each request carries everything needed to handle it, so no server-side session is needed between calls. Responses say whether they are cacheable. The client and server are separate and can change independently. Representations are usually JSON. Good REST design uses plural nouns, correct status codes, and consistent errors. Real-world APIs rarely follow every rule strictly, and that is fine as long as the design is consistent and well documented.

> **Sample answer**
>
> GET /users list users
>
> GET /users/7 one user
>
> POST /users create
>
> DELETE /users/7 remove
>
> GET /users/7/orders orders of a user

Interview tip. A classic trap is URLs like /getUser or /deleteUser. Put the verb in the method, not the path.

06 Important HTTP headers

Easy

Which HTTP headers come up most often in backend work, and what do they do?

Content-Type says what format the body is in, like application/json. Accept tells the server which formats the client wants back. Authorization carries credentials, for example a Bearer token. Cache-Control sets caching rules such as max-age=60 or no-store. ETag is a version fingerprint of a response, and with If-None-Match it lets the server answer 304 Not Modified. Set-Cookie and Cookie handle cookies. Location is used with redirects and after creating a resource. User-Agent identifies the client, and X-Forwarded-For carries the original client IP behind a proxy. Headers are metadata, so they keep the body clean.

```
POST /login HTTP/1.1
Content-Type: application/json
Accept: application/json
Authorization: Bearer <token>
```

Interview tip. Mention ETag and 304 as a bandwidth saver. It shows you understand caching beyond the buzzword.

07 HTTP/1.1 vs HTTP/2 vs HTTPS

Medium

What is the difference between HTTP/1.1 and HTTP/2, and how is HTTPS different from both?

HTTP/1.1 handles roughly one request at a time per connection, so browsers open several connections and requests can block each other. HTTP/2 sends many requests at once over a single connection (multiplexing), compresses headers, and uses a binary format, which makes pages load faster. HTTP/3 goes further by running over QUIC instead of TCP. HTTPS is not a different version. It is HTTP wrapped in TLS encryption, so nobody on the network can read or change the traffic, and the server proves its identity with a certificate. HTTP/2 is used in practice over HTTPS, so the two usually appear together.

Interview tip. Do not say HTTPS is a newer HTTP. It is HTTP plus TLS, and it works with any HTTP version.

Test yourself

1. Walk through what happens when a POST request to create an order times out. Is it safe for the client to retry?
2. Design the URLs and status codes for a small blog API.

SECTION 02 / 08

APIs and auth

Login, tokens and permissions are asked in almost every backend round. Learn the difference between proving who you are and what you may do, then compare sessions, JWT and OAuth with clear trade-offs.

08 Authentication, authorization

Easy

What is the difference between authentication and authorization?

Authentication answers who are you. It is checking identity, for example with a password, an OTP or a fingerprint. Authorization answers what are you allowed to do. It is checking permissions after identity is known, for example whether this user may delete that post. Authentication always comes first. In an API, authentication usually yields a 401 when it fails, and authorization yields a 403. Common authorization models are role based (admin, editor, viewer) and attribute based (owner of the resource, same team). Always check authorization on the server for every request, never trust the client to hide a button.

```
function requireAdmin(req, res, next) {
  if (!req.user) return res.sendStatus(401);
  if (req.user.role !== 'admin') return res.sendStatus(403);
  next();
}
```

Interview tip. Use a real-life picture: showing your ID at the gate is authentication, the guest list decides authorization.

09 Sessions vs JWT

Medium

What is the difference between session based auth and JWT based auth, and when would you pick each?

With sessions, the server stores the login state (in memory, a database or Redis) and gives the browser a session id in a cookie. Each request is checked against the store. Logout is easy because you delete the session, but the server must keep state. With JWT, the server signs a token that contains the user's claims and hands it to the client. The server verifies the signature and keeps nothing, which scales easily across many servers, but you cannot cancel a token before it expires unless you add a block list. Sessions suit classic web apps, JWT suits APIs and service to service calls.

Interview tip. Do not call JWT strictly better. The honest answer is trade-offs: revocation versus statelessness.

10 How a JWT works

Medium

What is inside a JWT, and how does the server trust it?

A JWT has three parts joined by dots: header, payload and signature. The header names the algorithm. The payload holds claims like the user id (sub) and expiry (exp). The signature is created by signing header and payload with a secret (or a private key). Header and payload are only base64 encoded, not encrypted, so anyone can read them and you must never put secrets in them. When a request arrives, the server recomputes the signature and checks it matches and that the token has not expired. If the payload were edited, the signature would no longer match.

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ sub: 42 }, process.env.JWT_SECRET, { expiresIn: '15m' });
const data = jwt.verify(token, process.env.JWT_SECRET); // throws if invalid or expired
```

> **Interview tip.** Say it clearly: signed does not mean encrypted. Interviewers check for exactly this.

11 OAuth 2.0 in simple words

Hard

What is OAuth 2.0 and how does the authorization code flow work?

OAuth 2.0 lets a user give an app limited access to their data on another service without sharing their password. Example: letting a photo app read your Google contacts. In the authorization code flow the app redirects you to the provider's login page. You sign in and approve the requested scopes. The provider redirects back to the app with a short-lived code. The app's backend then exchanges that code, plus its client secret, for an access token, and sometimes a refresh token. The app uses the access token to call the provider's API. OAuth is about delegated access, and OpenID Connect adds identity (login) on top of it.

> **Interview tip.** Remember that OAuth is authorization, not authentication. Login with Google uses OpenID Connect on top.

12 Pagination: offset vs cursor

Medium

How do you paginate an API, and why is cursor pagination often better than offset?

Offset pagination uses page and limit, translated to LIMIT and OFFSET in SQL. It is simple but has two problems: the database still has to skip all earlier rows, so deep pages get slow, and if rows are added or deleted while the user pages, items repeat or go missing. Cursor pagination returns an opaque marker of the last item seen, such as its id or timestamp, and the next request asks for items after it. It stays fast with an index and gives stable results. The cost is that you cannot jump straight to page 50. Use offset for small admin tables and cursor for feeds and big lists.

```
-- offset (slow when offset is large)
SELECT * FROM posts ORDER BY id LIMIT 20 OFFSET 10000;
-- cursor
SELECT * FROM posts WHERE id > 10020 ORDER BY id LIMIT 20;
```

> **Interview tip.** Mention that the cursor column must be indexed and unique, or ties will break your ordering.

13 Rate limiting an API

Medium

How would you rate limit an API, and what algorithms exist?

Rate limiting caps how many requests a client can make in a time window, protecting your service from abuse and overload. You identify the client by API key, user id or IP, keep a counter in a fast shared store like Redis, and return 429 Too Many Requests with a Retry-After header when the limit is hit. A fixed window counter is simplest but allows bursts at window edges. A sliding window is smoother. A token bucket adds tokens at a steady rate and lets clients spend them, which allows short bursts while keeping the average under control. In production this is often done at the gateway.

```
// fixed window with Redis (node-redis)
const key = `rl:${userId}:${Math.floor(Date.now() / 60000)}`;
const n = await redis.incr(key);
if (n === 1) await redis.expire(key, 60);
if (n > 100) return res.sendStatus(429);
```

> **Interview tip.** The counter must live in a shared store, not in app memory, or each server will count separately.

Test yourself

1. A user logs out but their JWT is still valid. How would you handle that?
2. Why is returning 403 different from returning 401?

SECTION 03 / 08

Databases and SQL

Most backend work is reading and writing data, so this section carries a lot of weight. Focus on why indexes help, how joins behave, and the mistakes that make queries slow.

14 SQL vs NoSQL

Easy

When would you choose a SQL database over a NoSQL database, and the other way round?

SQL databases like PostgreSQL and MySQL store data in tables with a fixed schema, support joins, and give strong transaction guarantees. They fit data with clear relationships, such as orders, payments and users. NoSQL is a broad label: document stores like MongoDB, key value stores like Redis, wide column stores like Cassandra, and graph databases. They often allow flexible schemas and scale out across many machines more easily, trading away some joins or consistency. Pick SQL by default when correctness and relationships matter. Pick NoSQL when the access pattern is simple and huge, the shape of data varies, or you need very high write volume.

> **Interview tip.** Avoid the myth that NoSQL is always faster. The right answer starts from the access pattern.

15 What is an index

Medium

What is a database index, how does it work, and what does it cost?

An index is an extra data structure, usually a B-tree, that lets the database find rows by a column without scanning the whole table. Think of the index at the back of a book. Searching a table of a million rows by an indexed email takes a few steps instead of a million comparisons. The cost is that every insert, update and delete must also update the index, and indexes use disk space. So index columns you filter, join or sort on often, not every column. A composite index on (a, b) helps queries on a, or on a and b, but generally not on b alone.

```
CREATE INDEX idx_users_email ON users (email);
EXPLAIN SELECT * FROM users WHERE email = 'a@x.com';
```

> **Interview tip.** Mention EXPLAIN. Reading a query plan to spot a full table scan is what real engineers do.

16 SQL joins explained

Easy

What are INNER, LEFT, RIGHT and FULL joins?

A join combines rows from two tables using a matching condition. INNER JOIN keeps only rows that match on both sides. LEFT JOIN keeps every row from the left table and fills the right side with NULL when there is no match. RIGHT JOIN is the mirror image. FULL JOIN keeps all rows from both sides, with NULLs where there is no match. Use LEFT JOIN to find things like customers who have no orders: join, then filter where the right side's key IS NULL. Always join on indexed columns, and be careful with one to many joins, since they multiply rows.

```
SELECT c.name
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.id IS NULL;   -- customers with no orders
```

> **Interview tip.** Draw two overlapping circles when you explain it. It makes the answer easy to follow.

17 Normalization basics

Medium

What is normalization, and when would you denormalize?

Normalization is organizing tables so each fact is stored once. In first normal form each cell holds a single value. In second and third normal form, columns depend only on the key of their own table, so a customer's city lives in the customers table, not repeated in every order row. This avoids update anomalies, where changing one fact means editing many rows and missing one. The downside is more joins at read time. Denormalizing means adding controlled duplication, for example storing a product name in an order item, to speed up reads or to freeze historical values. Do it on purpose, with measured need.

> **Interview tip.** Give a concrete reason to denormalize, like keeping the price paid at order time.

18 The N+1 query problem

Medium

What is the N+1 query problem and how do you fix it?

N+1 happens when code runs one query to fetch a list, then one extra query per item to fetch related data. Fetching 100 posts and then querying the author for each post is 101 queries. It is often hidden by ORMs that lazy load relations, and it makes endpoints slow as data grows. Fix it by fetching related data in one go, using a JOIN, or eager loading, or a second query with WHERE id IN (...) that loads all authors together. Spot it by logging queries per request or using an APM tool, and watch for query counts that grow with the result size.

```
// bad: 1 + N queries
const posts = await db.query('SELECT * FROM posts');
for (const p of posts) p.author = await db.query('SELECT * FROM users WHERE id=$1',
[p.user_id]);
// good: 2 queries
const ids = posts.map(p => p.user_id);
const authors = await db.query('SELECT * FROM users WHERE id = ANY($1)', [ids]);
```

> **Interview tip.** Say how you would detect it, not just fix it: log the query count per request.

19 WHERE vs HAVING

Easy

What is the difference between WHERE and HAVING, and what does GROUP BY do?

GROUP BY collects rows that share a value into groups so you can run aggregate functions like COUNT, SUM and AVG on each group. WHERE filters individual rows before grouping happens, so it cannot use aggregates. HAVING filters groups after aggregation, so it can. The logical order is FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. Use WHERE whenever you can, because filtering early means fewer rows to group. Use HAVING only for conditions that depend on an aggregate, such as groups with more than five rows.

```
SELECT department, COUNT(*) AS people
FROM employees
WHERE active = true
GROUP BY department
HAVING COUNT(*) > 5;
```

> **Interview tip.** Remember the order: WHERE works on rows, HAVING works on groups.

Test yourself

1. A query on a 5 million row table is slow. List the steps you would take to investigate.
2. Write a query for customers who placed more than three orders.

SECTION 04 / 08

Transactions and locks

This is where interviews separate beginners from people who have debugged production data. Understand what a transaction guarantees, how isolation levels trade speed for safety, and why deadlocks happen.

20 ACID properties

Easy

What does ACID stand for, and why does it matter?

ACID describes the guarantees of a database transaction. Atomicity means all steps happen or none do, so a money transfer never debits one account without crediting the other. Consistency means a transaction moves the database from one valid state to another, respecting constraints. Isolation means concurrent transactions do not step on each other in surprising ways. Durability means once a commit is acknowledged, the data survives a crash, usually because it was written to a log on disk. These guarantees are why relational databases are trusted for payments and inventory.

```
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT;  -- or ROLLBACK on any error
```

Interview tip. Use the bank transfer example. Everyone understands it and it shows atomicity instantly.

21 Transaction isolation levels

Hard

What are the SQL isolation levels, and what does each prevent?

Isolation levels control how much concurrent transactions can see of each other. READ UNCOMMITTED allows reading other transactions' unfinished changes. READ COMMITTED only shows committed data, and is the default in PostgreSQL. REPEATABLE READ makes sure rows you already read do not change under you, and is the default in MySQL InnoDB. SERIALIZABLE behaves as if transactions ran one after another, and is the safest and slowest. Higher levels prevent more anomalies but cause more blocking or retries. Exact behaviour differs by database, so check the documentation of the one you use.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- queries here
COMMIT;
```

Interview tip. Say that stricter isolation costs throughput, and that the defaults differ between PostgreSQL and MySQL.

22 Dirty, non-repeatable, phantom

Hard

What are dirty reads, non-repeatable reads and phantom reads?

A dirty read happens when you read data another transaction has changed but not yet committed. If it rolls back, you acted on data that never existed. A non-repeatable read happens when you read the same row twice in one transaction and get different values, because someone committed an update in between. A phantom read happens when you run the same query twice and a new matching row appears, because someone inserted it in between. READ COMMITTED stops dirty reads. REPEATABLE READ stops non-repeatable reads, and in some databases phantoms too. SERIALIZABLE stops all three.

> **Interview tip.** Learn the table of which level stops which anomaly, but also mention that databases implement it a little differently.

23 Optimistic vs pessimistic lock

Medium

What is the difference between optimistic and pessimistic locking?

Pessimistic locking assumes conflicts are likely, so it locks the row when you read it and others wait until you finish. In SQL this is SELECT ... FOR UPDATE. It is safe but can reduce concurrency and cause deadlocks. Optimistic locking assumes conflicts are rare. You read without locking, keep a version number on the row, and at update time you check that the version has not changed. If it did, the update affects zero rows and you retry or tell the user. Use pessimistic locking for hot contested rows like stock counts, and optimistic locking for low contention data like profile edits.

```
UPDATE products
SET stock = stock - 1, version = version + 1
WHERE id = 9 AND version = 4;   -- 0 rows updated means someone else changed it
```

> **Interview tip.** Show the version column check. That small detail proves you have actually used it.

24 What is a deadlock

Hard

What is a database deadlock, how does it occur, and how do you prevent it?

A deadlock happens when two transactions each hold a lock the other needs, so both wait forever. Example: transaction A locks row 1 and wants row 2, while transaction B locks row 2 and wants row 1. The database detects the cycle, kills one transaction as the victim, and returns an error that your code should handle by retrying. To reduce deadlocks, always lock rows in the same order, keep transactions short, avoid user interaction inside a transaction, and use proper indexes so fewer rows get locked. Setting a lock timeout is a useful safety net.

```
-- Tx A                          -- Tx B
UPDATE t SET v=1 WHERE id=1;     UPDATE t SET v=1 WHERE id=2;
UPDATE t SET v=1 WHERE id=2;     UPDATE t SET v=1 WHERE id=1;  -- deadlock
```

> **Interview tip.** The two magic words are consistent lock ordering. Also say you retry the victim transaction.

25 Preventing lost updates

Hard

Two users buy the last item at the same time. How do you stop overselling?

This is a race condition. Both requests read stock = 1, both decide it is available, and both write stock = 0, so you sold two items. The cleanest fix is to make the check and the change one atomic statement, so the database serializes them. You can also lock the row with SELECT ... FOR UPDATE inside a transaction, or use optimistic locking with a version column. Application level checks alone do not work when you run several servers. A unique constraint or a CHECK (stock >= 0) adds a last line of defence.

```
UPDATE products
SET stock = stock - 1
WHERE id = 9 AND stock > 0;
-- if affected rows = 0, the item sold out
```

> **Interview tip.** Prefer the single atomic UPDATE. It is simple, fast and needs no explicit lock.

Test yourself

1. Explain a bank transfer going wrong under concurrency and how you would prevent it.
2. How do you handle a deadlock error in your application code?

SECTION 05 / 08

Caching

Caching is the cheapest way to make a backend faster, and the easiest way to serve wrong data. Be clear about where to cache, how entries expire, and what goes wrong at scale.

26 Why and where to cache

Easy

What is caching, why do we use it, and where can you cache in a backend system?

A cache keeps a copy of data that is expensive to compute or fetch in a faster place, so repeated requests are cheap. It cuts latency, reduces database load and saves money. You can cache at many layers: in the browser, on a CDN near the user, at a reverse proxy, inside the application process, or in a shared store like Redis. Good candidates are data that is read often and changes rarely, such as product details or configuration. Bad candidates are data that changes constantly or must always be exactly fresh, such as an account balance during a payment.

> **Interview tip.** Always add the trade-off: a cache makes data faster but possibly stale.

27 Cache-aside vs write-through

Medium

What are cache-aside, write-through and write-back strategies?

In cache-aside (lazy loading), the app checks the cache first. On a miss it reads the database, stores the result in the cache and returns it. On writes it updates the database and deletes or updates the cache entry. It is the most common pattern. In write-through, every write goes to the cache and the database together, so the cache is always warm but writes are slower. In write-back, writes go to the cache first and are saved to the database later in batches. It is very fast but risks losing data if the cache fails before saving. Pick based on how much staleness and risk you accept.

```
async function getUser(id) {
  const hit = await redis.get(`user:${id}`);
  if (hit) return JSON.parse(hit);
  const user = await db.findUser(id);
  await redis.set(`user:${id}`, JSON.stringify(user), { EX: 300 });
  return user;
}
```

> **Interview tip.** Cache-aside is the default answer. Mention write-back risk to show you understand durability.

28 Cache invalidation and TTL

Medium

How do you keep cached data from going stale? What is TTL?

Invalidation means removing or updating cache entries when the source data changes. The simplest tool is TTL (time to live), which expires an entry automatically after a set time, such as 5 minutes. TTL is easy but allows staleness until expiry. For fresher data, delete or update the key whenever you write to the database, or publish an event that tells other services to evict. Versioned keys such as product:42:v7 avoid the problem by making old entries unreachable. There is no perfect answer, so choose a TTL that matches how wrong the data is allowed to be for the business.

```
await db.updateProduct(id, data);
await redis.del(`product:${id}`);   // next read refills from the database
```

Interview tip. Say that invalidation is hard because of races between the write and the cache refill.

29 LRU and eviction policies

Medium

What is cache eviction, and how does an LRU cache work?

A cache has limited memory, so when it is full it must remove something to make room. That is eviction. LRU (least recently used) removes the entry that has not been used for the longest time, assuming recent items will be used again. LFU removes the least frequently used, and FIFO removes the oldest added. LRU is usually built with a hash map for quick lookup and a doubly linked list to track order, so both get and put take constant time. Redis offers policies such as allkeys-lru and volatile-ttl, set through the maxmemory-policy setting.

```
class LRU {
  constructor(n) { this.n = n; this.m = new Map(); }
  get(k) {
    if (!this.m.has(k)) return undefined;
    const v = this.m.get(k); this.m.delete(k); this.m.set(k, v);
    return v;
  }
  put(k, v) {
    this.m.delete(k); this.m.set(k, v);
    if (this.m.size > this.n) this.m.delete(this.m.keys().next().value);
  }
}
```

Interview tip. A JavaScript Map keeps insertion order, which makes this short LRU possible. Be ready to code it.

30 Cache stampede

Hard

What is a cache stampede (thundering herd) and how do you prevent it?

A cache stampede happens when a popular key expires and many requests miss at the same moment. All of them hit the database to rebuild the same value, which can overload it. Ways to prevent it: use a lock or single flight so only one request recomputes while others wait; refresh popular keys in the background before they expire; add random jitter to TTLs so keys do not all expire together; and serve slightly stale data while one request refreshes it. It is a good example of a problem that only appears at scale, not in testing.

```
const inflight = new Map();
function load(key, fetcher) {
  if (!inflight.has(key)) {
    inflight.set(key, fetcher().finally(() => inflight.delete(key)));
  }
  return inflight.get(key);
}
```

Interview tip. Mention TTL jitter. It is a one-line fix that interviewers like.

31 CDN and HTTP caching

Medium

What is a CDN, and how do Cache-Control and ETag help?

A CDN (content delivery network) is a set of servers around the world that keep copies of your static files and sometimes API responses, so users get them from a nearby location. This lowers latency and protects your origin server. You control caching through headers. Cache-Control: max-age=3600 lets caches reuse a response for an hour. no-store forbids caching, and private keeps it to the user's browser. ETag lets a client ask if its copy is still valid with If-None-Match, and the server replies 304 with no body if nothing changed. Versioned file names like app.3f2a.js allow very long caching safely.

```
res.set('Cache-Control', 'public, max-age=3600');
res.set('ETag', hashOfBody);
```

Interview tip. Never cache personalized or private responses on a shared CDN. Say that explicitly.

Test yourself

1. Your cache hit rate is dropping after a deploy. What would you check?
2. How would you cache a product page that has a per-user price?

SECTION 06 / 08

Async and queues

Not everything should happen inside the request. Learn how async code, queues and retries keep the API fast and the system reliable when parts of it fail.

32 Sync vs async, event loop

Medium

What is the difference between synchronous and asynchronous code, and how does the Node.js event loop work?

Synchronous code runs one step at a time and blocks until each step finishes. Asynchronous code starts a slow operation, like a network call or file read, and carries on while it waits, then handles the result later. Node.js runs your JavaScript on a single thread with an event loop. When you await a database call, the loop is free to handle other requests, and the result comes back through a callback or promise. This handles many concurrent connections cheaply for I/O heavy work. A long CPU task, like hashing a huge file in a loop, blocks the loop and stalls every request.

```
app.get('/users/:id', async (req, res) => {
  const user = await db.findUser(req.params.id); // thread is free meanwhile
  res.json(user);
});
```

Interview tip. Say that async helps with waiting (I/O), not with heavy computation. Use worker threads for CPU work.

33 Concurrency vs parallelism

Medium

What is the difference between concurrency and parallelism?

Concurrency means dealing with many tasks at once by switching between them. It is about structure. Parallelism means actually running many tasks at the same instant on multiple CPU cores. It is about execution. A single threaded Node.js server is concurrent, since it interleaves thousands of requests, but not parallel for its own JavaScript. Running several worker processes on a multi core machine gives you parallelism. Python threads are limited by the GIL for CPU work, so Python uses multiple processes for parallel CPU tasks, while threads or asyncio work well for I/O waiting.

```
import asyncio
async def fetch(n):
    await asyncio.sleep(1)   # waiting, so others can run
    return n
async def main():
    print(await asyncio.gather(fetch(1), fetch(2), fetch(3)))  # about 1 second total
asyncio.run(main())
```

Interview tip. A good one-liner: concurrency is juggling, parallelism is several people each juggling.

34 Why use a message queue

Medium

What is a message queue and why would a backend use one?

A message queue sits between a producer, which adds messages, and a consumer, which processes them later. It lets slow work move out of the request. For example, after signup the API saves the user and puts a send welcome email message on the queue, then replies at once. A worker sends the email in the background. Queues smooth traffic spikes, since work waits instead of crashing the service. They decouple services, since the producer does not care whether the consumer is up. They also enable retries. Common tools are RabbitMQ, Amazon SQS and Kafka.

```
// producer
await queue.add('send-email', { userId: 42 });
res.status(202).json({ status: 'queued' });
```

Interview tip. Use 202 Accepted to show the work is accepted but not finished.

35 Queue vs pub/sub

Medium

What is the difference between a work queue and publish/subscribe?

In a work queue, each message goes to exactly one consumer. Several workers share the load, so it is used to spread jobs such as resizing images. In publish/subscribe, a message is published to a topic and every subscriber gets its own copy. It is used when many parts of the system react to one event, for example an OrderPlaced event that triggers email, inventory and analytics separately. Many systems combine both: a topic fans out to several queues, each with a pool of workers. Kafka and SNS plus SQS are common examples of this setup.

Interview tip. Short version: queue means one consumer per message, pub/sub means every subscriber gets a copy.

36 At-least-once and idempotency

Hard

Why can a consumer receive the same message twice, and how do you handle it?

Most queues promise at-least-once delivery. If a consumer processes a message but crashes before acknowledging it, the queue sends it again. So duplicates are normal, and exactly-once delivery is very hard in practice. The fix is to make consumers idempotent, meaning processing a message twice has the same effect as once. Store a unique message id or an idempotency key and skip ids you have already handled, or use operations that are naturally repeatable, like set status to paid instead of add 100 to balance. A unique database constraint on the key makes this reliable even with several workers.

```
async function handle(msg) {
  try {
    await db.query('INSERT INTO processed(id) VALUES($1)', [msg.id]);
  } catch (e) {
    if (e.code === '23505') return; // already processed (PostgreSQL unique violation)
    throw e;
  }
  await sendEmail(msg.userId);
}
```

Interview tip. Say the phrase idempotent consumer. It is the answer the interviewer is waiting for.

37 Retries and dead letter queue

Medium

How should failed jobs be retried, and what is a dead letter queue?

Some failures are temporary, like a network blip, so retrying helps. But retrying instantly in a loop can overload a struggling service. Use exponential backoff, which waits longer after each failure (1s, 2s, 4s, 8s), plus random jitter so many workers do not retry together. Set a maximum number of attempts. A message that keeps failing, perhaps because the data is bad, goes to a dead letter queue (DLQ) instead of blocking the main queue forever. Engineers then inspect the DLQ, fix the cause and replay the messages. Alert on DLQ growth.

```
async function retry(fn, tries = 5) {
  for (let i = 0; i < tries; i++) {
    try { return await fn(); }
    catch (e) {
      if (i === tries - 1) throw e;
      const wait = 2 ** i * 1000 + Math.random() * 250;
      await new Promise(r => setTimeout(r, wait));
    }
  }
}
```

Interview tip. Only retry operations that are safe to repeat. That links back to idempotency.

Test yourself

1. How do you make sure a payment message processed twice only charges once?
2. When would you pick a queue over just calling the other service directly?

SECTION 07 / 08

Scaling, reliability

Once an app has real users, one server and one database stop being enough. These questions test whether you can grow a system and keep it up when parts fail. Explain each idea with its trade-off.

38 Vertical vs horizontal scaling

Easy

What is the difference between vertical and horizontal scaling?

Vertical scaling means giving one machine more power: more CPU, RAM or faster disks. It is simple, needs no code change, but has a ceiling, costs rise sharply, and the machine remains a single point of failure. Horizontal scaling means adding more machines and spreading the work across them. It can grow almost without limit and survives the loss of one machine, but it needs a load balancer, and the app must be stateless, or share state through a database or cache. Most teams scale vertically first because it is easy, then move to horizontal when they hit limits.

> **Interview tip.** Mention stateless servers. Horizontal scaling only works if any server can handle any request.

39 Load balancing

Medium

What does a load balancer do, and what algorithms does it use?

A load balancer sits in front of several servers and spreads incoming requests among them. It improves capacity and availability. It also runs health checks and stops sending traffic to servers that fail. Common algorithms: round robin sends requests in turn, least connections picks the server with the fewest active requests, and IP hash or consistent hashing sends the same client to the same server. Layer 4 balancers route on IP and port, and layer 7 balancers can route on URL or headers. Sticky sessions pin a user to one server, but they make scaling harder, so prefer stateless servers with shared session storage.

```
upstream api {
  least_conn;
  server 10.0.0.1:3000;
  server 10.0.0.2:3000;
}
```

> **Interview tip.** Remember that the load balancer itself must not be a single point of failure, so run more than one.

40 Replication and read replicas

Medium

What is database replication and when do you use read replicas?

Replication copies data from a primary database to one or more replicas. All writes go to the primary and are streamed to replicas. Replicas serve read queries, which spreads read load and lets you keep working on a copy if the primary fails, through failover. Most apps read far more than they write, so read replicas help a lot. The catch is replication lag. A replica can be slightly behind, so a user who just saved a post may not see it when reading from a replica. Fixes include reading from the primary right after a write, or tracking the user's last write.

> **Interview tip.** Name replication lag and how you handle read-your-own-writes. That is the follow-up.

41 Database sharding

Hard

What is sharding, and what problems does it bring?

Sharding splits a large dataset across several databases, called shards, so no single machine holds everything. You choose a shard key, such as user id, and a rule maps each key to a shard, for example hash(user_id) mod N or a range. It lets writes and storage scale beyond one machine. The costs are real: queries across shards and joins become hard, transactions across shards are complex, a bad key creates hot shards that get most of the traffic, and adding shards means moving data. Because of this, try indexing, caching, replicas and partitioning first, and shard only when needed.

```
const shardFor = (userId, n) => userId % n;   // simple modulo routing
```

Interview tip. Always warn that sharding is a last resort. Choosing a good shard key is the heart of the answer.

42 CAP theorem

Hard

What is the CAP theorem?

CAP says that a distributed data store cannot guarantee all three of consistency, availability and partition tolerance at once. Consistency means every read sees the latest write. Availability means every request gets a response. Partition tolerance means the system keeps working when the network between nodes breaks. Network partitions will happen in any distributed system, so the real choice during a partition is between consistency and availability. A CP system may refuse some requests to stay correct. An AP system keeps answering but may return stale data and sort it out later. Many systems let you tune this per operation.

Interview tip. Do not say pick any two of three. The choice only bites when a partition happens.

43 Timeouts and circuit breaker

Hard

How do you stop one slow dependency from taking down your whole service?

Never wait forever. Set timeouts on every network call, so a hung dependency fails fast instead of holding threads or connections. Retry only safe operations, with backoff. Then use a circuit breaker. It watches failures, and after too many it opens, so calls fail immediately without hitting the struggling service. After a cool down it lets a few trial calls through, and closes again if they succeed. Pair this with a fallback, such as cached data or a friendly error, and bulkheads that limit how many resources one dependency can use. Together they stop a small failure from spreading everywhere.

```
const controller = new AbortController();
const timer = setTimeout(() => controller.abort(), 2000);
const r = await fetch(url, { signal: controller.signal });
clearTimeout(timer);
```

Interview tip. Use the electrical analogy. A circuit breaker trips to protect the rest of the house.

Test yourself

1. Your database CPU is at 90 percent from reads. What options do you have, in order?
2. Explain what happens to your service when a payment provider becomes very slow.

SECTION 08 / 08

Security and testing

Security bugs and untested code are what get backends into trouble. Know the most common web attacks and how to block them, then explain how you would test your own code with confidence.

44 SQL injection

Medium

What is SQL injection and how do you prevent it?

SQL injection happens when user input is glued into a SQL string, so an attacker can change the query. If you build a query by joining text, input like ' OR '1'='1 can log someone in without a password or dump a whole table. The fix is parameterized queries (prepared statements), where the query and the values are sent separately and the database never treats input as code. ORMs do this for you, if you avoid raw string building. Also give the database user the least privileges it needs, and validate input. Escaping by hand is fragile, so use parameters.

```
# vulnerable
cur.execute(f"SELECT * FROM users WHERE email = '{email}'")
# safe
cur.execute("SELECT * FROM users WHERE email = %s", (email,))
```

> **Interview tip.** Say parameterized queries in the first sentence. Everything else is extra credit.

45 Cross-site scripting (XSS)

Medium

What is XSS and how do you defend against it?

XSS (cross-site scripting) happens when an attacker gets their JavaScript to run in another user's browser, usually because the site shows untrusted input as HTML. The script can steal session data, act as the user, or change the page. Stored XSS saves the payload in your database, for example in a comment. Reflected XSS bounces it from a URL. The main defence is output encoding: escape user data for the place it is shown, and use frameworks that escape by default. Add a Content-Security-Policy header to limit which scripts run, and mark session cookies HttpOnly so scripts cannot read them.

```
// risky
el.innerHTML = comment;
// safe
el.textContent = comment;
```

> **Interview tip.** Name the three tools: output encoding, CSP header, HttpOnly cookies.

46 CSRF attack

Medium

What is CSRF and how is it prevented?

CSRF (cross-site request forgery) tricks a logged-in user's browser into sending an unwanted request to your site. Because browsers attach cookies automatically, a malicious page can submit a hidden form to your transfer endpoint and your server sees a valid session. Defences: set session cookies with SameSite=Lax or Strict so they are not sent on cross-site requests; use anti-CSRF tokens, a secret value in each form that an attacker cannot guess; and check the Origin header on state changing requests. APIs that use an Authorization header instead of cookies are mostly not exposed, since browsers do not attach that header on their own.

```
res.cookie('sid', token, { httpOnly: true, secure: true, sameSite: 'lax' });
```

Interview tip. Do not mix CSRF and XSS up. CSRF abuses the browser's trust in cookies, XSS runs attacker code.

47 CORS explained

Medium

What is CORS and why does the browser block my request?

Browsers follow the same-origin policy: a page from one origin (scheme, host and port) cannot read responses from another origin by default. CORS (cross-origin resource sharing) is how a server opts in. The server sends headers such as Access-Control-Allow-Origin naming the origins it trusts. For non-simple requests the browser first sends an OPTIONS preflight to ask permission. CORS is enforced by the browser, not the server, and it protects users, not your API from curl. Never set the allowed origin to * for APIs that use cookies, and list trusted origins explicitly.

```
const cors = require('cors');
app.use(cors({ origin: 'https://myapp.com', credentials: true }));
```

Interview tip. Say clearly that CORS is a browser rule. Postman and curl ignore it.

48 Hashing passwords safely

Medium

How should passwords be stored?

Never store passwords in plain text or with fast hashes like MD5 or plain SHA-256, and never encrypt them in a way you can reverse. Use a slow, purpose built password hashing algorithm such as bcrypt, scrypt or Argon2. Each password gets a unique random salt, which stops attackers from using precomputed tables and makes identical passwords hash differently. The algorithm has a cost setting, so you can make it slower as hardware improves. On login you hash the attempt with the stored salt and compare. Also rate limit login attempts and consider multi-factor authentication.

```
const bcrypt = require('bcrypt');
const hash = await bcrypt.hash(password, 12);
const ok = await bcrypt.compare(attempt, hash);
```

Interview tip. Hashing is one way, encryption is two way. Passwords need hashing.

49 Unit vs integration tests

Easy

What is the difference between unit tests and integration tests?

A unit test checks one small piece of code, like a function, in isolation, with outside parts such as the database replaced by fakes. It is fast and points straight at the broken line. An integration test checks that several parts work together, for example an API endpoint with a real test database. It is slower but catches wiring problems, wrong queries and bad configuration that unit tests miss. End-to-end tests go further and drive the whole system like a user. A healthy suite has many unit tests, fewer integration tests and a few end-to-end tests.

```
def apply_discount(price, pct):
    return round(price * (1 - pct / 100), 2)

def test_apply_discount():
    assert apply_discount(200, 10) == 180.0
```

Interview tip. Mention the test pyramid: many fast unit tests at the bottom, few slow end-to-end tests on top.

50 Mocking and test pyramid

Medium

What is mocking, and when should you avoid it?

A mock or stub is a fake stand-in for a dependency, such as an email service or payment API, so a test is fast, repeatable and does not hit the real thing. You can also check that your code called it correctly. Mock slow or external things you do not control. But too much mocking makes tests that only prove your mocks, and they keep passing while the real code breaks. Do not mock the code under test, and prefer a real database or in-memory fake for data logic. Tests should check behaviour, not implementation details, so refactoring does not break them.

```
from unittest.mock import Mock

def test_sends_welcome_email():
    mailer = Mock()
    signup('a@x.com', mailer)
    mailer.send.assert_called_once_with('a@x.com', 'Welcome')
```

Interview tip. Show balance: mock what is external, keep real what you own.

Test yourself

1. Write the safe version of a login query and explain what makes it safe.
2. What would you test with a unit test and what with an integration test for a signup endpoint?