Backend & System Design ��� Interview Prep

Backend & System Design — Interview Prep

A one-night revision sheet. Each topic has: **the concept**, **the answer you say out loud**, and **the follow-up they will ask**.

---

1. Client–Server Architecture ⭐⭐⭐⭐⭐

Client — anything that *initiates* a request. Browser, mobile app, Postman, another backend service. It owns the UI and holds no trusted logic (anything on the client can be tampered with).

Server — a program that *listens* on a port, waits for requests, does work, returns a response. It owns the trusted logic and the data.

Request–response cycle

```
1. Client resolves domain via DNS      → gets IP
2. Opens TCP connection (+ TLS if HTTPS)
3. Sends HTTP request: method + path + headers + body
4. Server routes → runs handler → queries DB
5. Server sends HTTP response: status + headers + body
6. Client renders / uses the data
```

It is **stateless**: the server does not remember you between requests. That is exactly why tokens/sessions exist (topic 5).

Frontend → Backend → Database flow

```
React  --HTTP/JSON-->  FastAPI  --SQL-->  PostgreSQL
       <--JSON-------           <--rows--
```

Example answer: “What happens when you log into a website?”

> User types email + password → frontend POSTs them over **HTTPS** to `/auth/login` → backend looks up the user by email → hashes the incoming password with the stored salt (bcrypt/argon2) and compares it to the stored hash — we never store plaintext → if it matches, backend issues a JWT or creates a session → returns it to the client (httpOnly cookie, or body if a token) → the client attaches it to every future request → protected endpoints verify it before doing anything → if it’s invalid we return **401**, if it’s valid but you lack permission, **403**.

Follow-ups: Why HTTPS? (password is plaintext on the wire otherwise). Why hash and not encrypt? (hashing is one-way; a DB leak shouldn’t reveal passwords). Why is comparing hashes slow on purpose? (bcrypt work factor makes brute force expensive).

---

2. Three-Tier Architecture ⭐⭐⭐⭐⭐

```
Presentation tier  →  React        (what the user sees)
Application tier   →  FastAPI/Node (business rules, auth, validation)
Data tier          →  PostgreSQL   (durable storage, integrity)
```

Why each layer exists — this is the actual question:

Layer	Exists because
Frontend	Rendering + UX are a different problem from data correctness. Ships independently.
Backend	The client can't be trusted. Auth, validation, and business rules must run somewhere the user can't edit. Also the only place that holds DB credentials.
Database	Durability, ACID transactions, concurrent access, integrity constraints. Not something you rebuild in app code.

The one-line justification: *separation of concerns* — each tier can be changed, tested, and scaled independently. You can put 3 backend instances behind a load balancer without touching React or Postgres.

Follow-up: "Why not let React talk to Postgres directly?" → You'd ship DB credentials to every browser, and anyone could run `DELETE FROM users`. No auth layer, no validation, no rate limiting.

---

3. REST APIs ☆☆☆☆☆

REST = resources are nouns (`/users`), actions are HTTP verbs.

Method	Meaning	Idempotent?	Safe?
GET	Read	Yes	Yes (no side effects)
POST	Create / non-idempotent action	**No**	No
PUT	Replace whole resource	Yes	No
PATCH	Partial update	Usually	No
DELETE	Remove	Yes	No

```
GET    /users          → list users
GET    /users/1        → one user
POST   /users          → create (body: {name, email}) → 201 + Location
PUT    /users/1        → replace entire user
PATCH  /users/1        → update just {email}
DELETE /users/1        → delete → 204
```

Idempotent = calling it 5 times leaves the system in the same state as calling it once. `DELETE /users/1` twice → still deleted, fine. `POST /users` twice → two users. That's why retries after a timeout are safe for GET/PUT/DELETE but dangerous for POST.

Status codes you must know

Code	Meaning
200 OK / 201 Created / 204 No Content	Success
400 Bad Request	Malformed / failed validation
401 Unauthorized	Not authenticated (who are you?)
403 Forbidden	Authenticated but not allowed
404 Not Found	Resource doesn't exist
409 Conflict	e.g. duplicate email
422 Unprocessable	Semantically invalid (FastAPI's default)
429 Too Many Requests	Rate limited
500 / 502 / 503	Server broke / bad gateway / overloaded

REST rules of thumb: nouns not verbs (`POST /users`, not `/createUser`), plural collections, nest for ownership (`/users/1/posts`), version it (`/api/v1/...`), paginate lists (`?page=2&limit=20`).

Follow-up: "PUT vs PATCH?" → PUT replaces the whole resource (omitted fields get wiped); PATCH sends only what changes.

---

4. Database Design ⭐⭐⭐⭐☆

```
Users                    Posts
-----                    -----
id      PK               id       PK
name                     title
email   UNIQUE           content
created_at               user_id  FK → Users.id
                         created_at
```

- **Primary key (PK)** — unique identity for a row.
- **Foreign key (FK)** — a column pointing at another table's PK. `Posts.user_id` is an FK.

Why is `user_id` a foreign key?

> It enforces **referential integrity** at the database level. You cannot insert a post for a user who doesn't exist, and you can't delete a user while leaving orphan posts behind — the DB either blocks it (`ON DELETE RESTRICT`) or cascades (`ON DELETE CASCADE`). It also documents the relationship and gives the query planner what it needs for joins.

Relationships

- **One-to-many** — one user has many posts. FK lives on the *many* side (`posts.user_id`).
- **Many-to-many** — a post has many tags, a tag has many posts. Needs a **join table**: `post_tags(post_id, tag_id)`.
- **One-to-one** — `users` ↔ `user_profiles`, FK + UNIQUE.

Normalization (short version)

Don't duplicate data. Store `user_id`, not a copy of the user's name in every post — otherwise a name change means updating 10,000 rows and you get inconsistency. **Denormalize deliberately** only when reads are too slow.

Indexes

An index is a B-tree that turns a full table scan into a lookup. Index columns you filter/join on (`posts.user_id`, `users.email`). Cost: slower writes, more disk. Don't index everything.

Follow-up: "How would you design likes?" → `likes(user_id, post_id, created_at)` with a **composite unique constraint** on `(user_id, post_id)` so one user can't like a post twice.

---

5. Authentication ⭐⭐⭐⭐☆

```
User → Login (email + password)
          ↓
Backend fetches user, compares bcrypt hash
          ↓
Issues JWT (or creates a session row/entry in Redis)
          ↓
Client stores it and sends it on every request
          ↓
Middleware verifies signature/session → attaches req.user
```

JWT vs Session

	Session	JWT
State	Stored server-side (DB/Redis)	Stateless — all data inside the token
Client holds	A random session ID (cookie)	A signed token (header.payload.signature)
Verify	Look it up in the store	Verify signature — no DB hit
Revoke	Easy — delete the row	**Hard** — valid until it expires
Scaling	Needs a shared store across servers	Any server can verify independently
Best for	Classic web apps, when instant logout matters	APIs, microservices, mobile

JWT = header.payload.signature, base64url encoded. Payload is **signed, not encrypted** — anyone can read it. Never put secrets in it.

The revocation problem + the standard fix: short-lived **access token** (~15 min) + long-lived **refresh token** stored server-side. Logout deletes the refresh token; the access token dies on its own within minutes.

Authentication vs Authorization: authN = *who are you* (401). authZ = *are you allowed* (403, roles/permissions).

Follow-up: "Where do you store the token?" → httpOnly + Secure + SameSite cookie is safest (JS can't read it → XSS can't steal it, SameSite blunts CSRF). localStorage is convenient but readable by any injected script.

---

6. Caching ⭐⭐⭐☆☆

Why cache? Reduce latency and take load off the database. Most workloads are read-heavy and read the same few things over and over.

What to cache: expensive query results, hot rows (user profiles), session data, rate-limit counters, computed feeds, API responses. **Don't cache** data that must be perfectly fresh (account balances) or is unique per request.

Why is Redis faster than a database?

> Redis keeps everything **in RAM** — no disk seek. Memory access is ~100 ns vs milliseconds for disk. It's a simple **key–value** store, so a `GET` is a hash lookup — no query parsing, no planner, no joins, no ACID transaction overhead, no locking. It's single-threaded on an event loop, so no lock contention either. Postgres pays all that cost because it guarantees durability and integrity; Redis trades those away for speed.

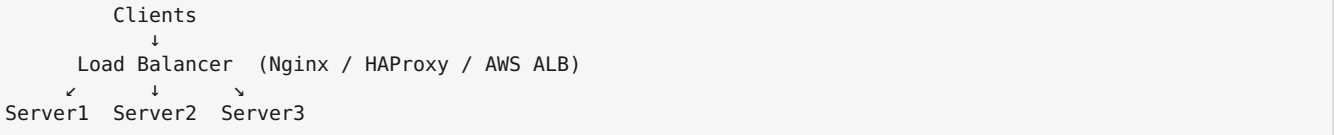
Cache-aside (the pattern to name):

```
read: check Redis → HIT? return it
                  → MISS? query DB → write to Redis with TTL → return
write: update DB → invalidate/delete the cache key
```

Key concepts: TTL (expiry, your safety net against stale data), **cache hit ratio**, **eviction policy** (LRU), **invalidation** (the hard part — "there are only two hard problems: cache invalidation and naming things"), **stampede** (many misses at once hammering the DB).

---

7. Load Balancer ⭐⭐⭐☆☆



Purpose: distributes traffic, improves **availability** (one server dies → LB routes around it), prevents overload, enables **zero-downtime deploys** (drain one server, update, add back), and terminates TLS in one place.

Algorithms: round robin, least connections, IP hash (sticky sessions), weighted.

Health checks: the LB pings `/health` every few seconds; failing instances are removed from the pool.

The key constraint it forces: your servers must be **stateless**. If request 1 hits Server1 and request 2 hits Server2, in-memory sessions break. → Push state to Redis/DB. This is *why* JWTs and Redis sessions matter (ties back to topics 5 and 6).

Follow-up: "Isn't the LB itself a single point of failure?" → Yes, so you run it in a redundant pair with a floating IP, or use a managed LB (ALB) that's already replicated.

---

8. Database Scaling ⭑⭑☆☆☆

	Vertical (scale up)	**Horizontal (scale out)**
How	Bigger machine — more CPU/RAM/SSD	More machines
Complexity	Trivial, no code change	High — routing, consistency
Limit	Hard ceiling; you can't buy an infinite box	Effectively unlimited
Failure	Single point of failure	Survives node loss
Cost	Superlinear at the top end	Linear-ish, commodity hardware

Start vertical. It's cheaper than you think and 90% of apps never outgrow it.

Horizontal options for databases, in the order you'd actually reach for them:

1. **Read replicas** — writes go to the primary, reads fan out to replicas. Fixes read-heavy load. Cost: **replication lag** (eventual consistency — you might not read your own write).
2. **Sharding / partitioning** — split rows across DBs by a shard key (e.g. `user_id % 4`). Fixes write-heavy load and data size. Cost: cross-shard joins are painful, rebalancing is painful.

Say this: stateless *app* servers scale horizontally trivially; *databases* are the hard part because they hold state.

---

9. SQL vs NoSQL ⭑⭑⭑⭑☆

	PostgreSQL (SQL)	**MongoDB (NoSQL, document)**
Model	Tables, rows, fixed schema	Collections of JSON-ish documents
Schema	Enforced up front	Flexible per document
Joins	First-class	Awkward; you embed or denormalize
Transactions	Strong ACID across tables	Improving, but weaker guarantees
Scale	Vertical + replicas; sharding is work	Sharding built in
Best at	Relationships, integrity, complex queries	High write volume, unstructured/varying data

Choose PostgreSQL when: data is relational (users → posts → comments), you need transactions (payments, orders), correctness matters more than raw write throughput, and you'll want ad-hoc queries. **This is the correct default.** Postgres also does JSONB, so you get schema flexibility where you need it.

Choose MongoDB when: documents vary in shape (logs, events, product catalogs with wildly different attributes), you read whole documents at a time and rarely join, or you need built-in horizontal sharding from day one.

Say this to sound senior: *“I’d default to Postgres. Most ‘we need NoSQL’ situations are really ‘we need an index’. I’d pick Mongo when the access pattern is genuinely document-shaped, not because of scale fears.”*

---

10. File Storage ⭐⭐⭐☆☆

Should you store images in the database? Usually no.

Why not: - Bloats the DB → backups get huge and slow, restores take hours. - Binary blobs pollute the buffer cache, evicting rows you actually query. - Serving a file means a round trip through your backend — you’re paying app-server CPU/bandwidth to do what a CDN does for free. - You can’t put a CDN in front of a Postgres row.

Do this instead:

```
Client
   ↓ upload
Backend  (validate type/size, authenticate)
   ↓
S3 / Cloudinary  →  CDN
   ↓
Save the URL (a short string) in the DB
```

The DB stores `avatar_url VARCHAR`. Object storage is cheap, infinitely scalable, and CDN-frontable.

Better still — presigned URLs: the backend issues a short-lived signed S3 URL, the client uploads **directly** to S3, then tells the backend the key. Your server never touches the bytes → no memory/bandwidth cost on your app tier.

Exception: tiny blobs (< a few KB) where transactional consistency with the row genuinely matters.

---

11. Message Queues ⭐⭐☆☆☆

Why queues exist: to decouple slow/unreliable work from the request–response cycle. The user shouldn’t wait 3 seconds for an SMTP server to respond, and their signup shouldn’t *fail* because the email provider is down.

```
User registers
      ↓
Create account, return 201 immediately   ← fast path, user is done
      ↓
Push "send_welcome_email" job → Queue
      ↓
Worker picks it up → sends email (retries if it fails)
```

What you get: async processing, **decoupling** (producer doesn’t know about the consumer), **buffering** against traffic spikes (queue absorbs the burst, workers drain at their own pace), **retries** with backoff, and independent scaling of workers.

Typical jobs: emails/SMS, image/video processing, report generation, webhooks, analytics events.

- **RabbitMQ** — a classic message broker. Job queues, routing, per-message ack. Message is consumed and gone.
- **Kafka** — a distributed, append-only **log**. High throughput, messages are retained and replayable, many consumer groups read the same stream. Event streaming/analytics.
- **Redis + Celery / BullMQ** — the pragmatic choice for a normal app.

Follow-up: “What if the job fails?” → Retry with exponential backoff; after N attempts push it to a **dead letter queue** for inspection. Also: jobs should be **idempotent**, because at-least-once delivery means a job can run twice.

---

12. Rate Limiting ⭐⭐☆☆☆

“How do you stop someone calling your API 10,000 times?” → Rate limiting.

Cap requests per identity per time window. Over the limit → **429 Too Many Requests** + a `Retry-After` header.

Key by: API key / user ID (best — survives IP changes), or IP (for unauthenticated endpoints like login and signup).

Algorithms:

- **Fixed window** — “100 req/min”, counter resets on the minute. Simple; flaw is the burst at the boundary (100 at 0:59 + 100 at 1:00 = 200 in 2 seconds).
- **Sliding window** — smooths that out.
- **Token bucket** — bucket refills at a steady rate, each request takes a token. Allows short bursts but limits sustained rate. Most common in practice.
- **Leaky bucket** — enforces a strictly constant outflow.

Where: at the edge — API gateway / Nginx / Cloudflare, before it costs you app time. Counters live in **Redis** because they must be shared across all servers behind the load balancer (topics 6 + 7 again).

Why bother: stops abuse and scraping, blunts brute-force login attempts and DDoS, protects the DB, controls cost, and enforces pricing tiers.

Follow-up: “Login endpoint?” → Tighter limits, keyed on IP *and* account, with exponential backoff/lockout after repeated failures.

---

13. Deployment Flow ⭐⭐⭐☆☆

```
git push → GitHub
     ↓
CI/CD (GitHub Actions)
   ├─ lint
   ├─ run tests
   ├─ build Docker image
   └─ push image to registry
     ↓
Deploy → Backend server(s) behind the LB
     ↓
Run DB migrations → PostgreSQL
```

- **CI (Continuous Integration)** — every push is automatically built and tested. Catches breakage before it reaches main.
- **CD (Continuous Deployment/Delivery)** — passing builds ship automatically (or with one click).

Concepts to name:

- **Environments:** dev → staging → prod. Staging mirrors prod.
- **Docker:** “works on my machine” → the image *is* the machine. Same artifact runs everywhere.
- **Env vars / secrets:** config lives in the environment, never in the repo. `.env` in `.gitignore`.
- **Migrations:** schema changes are versioned code (Alembic/Prisma), run as a deploy step. Make them **backwards-compatible** — add a column before the code that uses it, drop it a release later.
- **Rollback:** deploy the previous image. This is why images are tagged.
- **Deploy strategies:** rolling (replace instances a few at a time), blue-green (two full environments, flip the LB), canary (5% of traffic first).
- **After deploy:** health checks, logs, metrics, error tracking (Sentry).

Rapid-fire recap

Ask	Say
Why a backend at all?	The client can't be trusted; auth, validation, and credentials live server-side.
GET vs POST?	GET reads, safe + idempotent, cacheable. POST creates, not idempotent.
401 vs 403?	401 = who are you. 403 = I know you, you can't.
Why an FK?	Referential integrity enforced by the DB, not by hope.
JWT vs session?	Stateless & scales vs revocable & simple. Access + refresh token gets you both.
Why Redis is fast?	RAM + key-value, no parser/planner/joins/ACID overhead.
Why a load balancer?	Distribute traffic, survive a dead server, deploy with zero downtime.
Vertical vs horizontal?	Bigger box (simple, ceiling) vs more boxes (complex, unlimited).
SQL or NoSQL?	Postgres by default; Mongo when data is genuinely document-shaped.
Images in the DB?	No — S3/Cloudinary + CDN, store the URL.
Why a queue?	Don't make the user wait for slow work; decouple, buffer, retry.
Stop 10k calls?	Rate limit at the edge, counters in Redis, return 429.
What is CI/CD?	Auto test on push, auto ship on green, roll back by redeploying the old image.

How to answer a system design question

1. **Ask clarifying questions first.** Scale? Read-heavy or write-heavy? Who are the users?
2. **Draw the boxes**: Client → LB → App servers → Cache → DB, plus a Queue + Workers and S3 off to the side.
3. **Define the API** — a few endpoints.
4. **Sketch the schema** — tables, PKs, FKs, the indexes you'd add.
5. **Then scale it**: add caching, read replicas, a queue for async work — and say *why* at each step.
6. **Name the trade-off out loud.** "This adds eventual consistency, which is fine for a feed, not for a balance." That single sentence is what separates a good answer from a memorized one.

If you don't know something: say what you'd reach for and why, and say what you'd verify. Reasoning beats recall.

Good luck tomorrow.