API Authentication

01

What is Authentication?

The foundation of every secure API

The Core Idea

Authentication is the process of verifying the identity of a client (user, app, or service) that is trying to access an API. It answers one fundamental question:

> **"Who are you?"**

Simple Explanation 🟣

Think of it like this:

Imagine you go to a bank. Before they let you access your account, the teller asks for your ID and checks it against their records. Only after verifying who you are will they let you proceed. Authentication in APIs works the same way — prove your identity before you get access.

No ID = No access. Wrong ID = No access. Right ID = You're in.

Technical Explanation 💻

For the developer:

Authentication is the mechanism by which an API verifies the identity of the requester before processing a request. The client provides credentials (API key, token, username/password, or signed request), and the server validates them against a trusted store. Upon successful verification, the server either processes the request or issues a session/token for subsequent requests. It always precedes authorization.

The Authentication Flow

1. **Client makes a request**
   Your app sends a request to the API endpoint with credentials attached.

2. **Server extracts credentials**
   The server reads the Authorization header, API key, or token from the request.

3. **Server validates identity**
   Credentials are checked against the database, secret key, or token signature.

4. **Access granted or denied**
   200 OK if valid. 401 Unauthorized if identity check fails.

Interview Tip 💡

What interviewers want to hear:

Authentication (AuthN) verifies WHO you are.
Authorization (AuthZ) verifies WHAT you can do.
These are two separate steps: authn always comes first.
401 Unauthorized = authentication failed.
403 Forbidden = authenticated but not authorized.

02

Why Use Authentication?

Protecting your API from the chaos of the internet

Simple Explanation 🟣

Imagine you build a notes app and deploy the API without any authentication. Anyone on the internet can:

- Read every user's private notes
- Delete data they don't own
- Spam your server with 10 million requests
- Pretend to be you and perform actions on your behalf

The real-world analogy:

An API without authentication is like a bank vault with no door. Anyone can walk in and take whatever they want. Authentication is the door — and the lock, and the guard.

Technical Explanation 💻

Why it matters architecturally:

Without authentication, APIs are vulnerable to:
• Unauthorized data access — anyone can query any endpoint
• Privilege escalation — users can access resources they don't own
• Denial of service — no rate-limiting without identity
• Audit trail gaps — no way to log WHO did WHAT
• Regulatory non-compliance — GDPR, HIPAA, PCI-DSS all require it

Key Reasons to Use Authentication

🔒 **Data Security** Only verified users can access sensitive data. Prevents leaks and breaches.	👤 **Identity Tracking** Know exactly who is calling your API and log all actions for audit trails.
⚡ **Rate Limiting** Tie usage quotas to an identity. Prevent abuse and protect server resources.	📋 **Compliance** GDPR, HIPAA, PCI-DSS regulations require proving who accessed what data and when.

03
Types of Authentication
Four essential methods every developer must know

3A — HTTP Basic Authentication

SIMPLEST METHOD | LEGACY SUPPORT | REQUIRES HTTPS

Simple Explanation 🟣

The real-world analogy:

Every time you enter a building, the security guard checks your ID. In Basic Auth, your app hands over its username and password EVERY SINGLE REQUEST. The server checks it every time. It is the most straightforward method — but like shouting your password across the room, it's only safe if no one is listening (HTTPS encrypts the channel).

Technical Explanation 💻

How it works under the hood:

Credentials are formatted as username:password, then Base64-encoded (NOT encrypted).
The encoded string is sent in every request header:
Authorization: Basic dXNlcjpwYXNzd29yZA==
The server decodes, splits on ':', and validates against its credential store.
Base64 is trivially reversible — this MUST be used over HTTPS only.

HTTP REQUEST WITH BASIC AUTH

```
GET /api/data HTTP/1.1
Host: api.example.com
Authorization: Basic dXNlcjpwYXNzd29yZA==

# Decoded value: user:password
# Base64 is NOT encryption — always use HTTPS!
```

Flow: Step by Step

1. **Combine credentials**
   Format as "username:password"

2. **Base64 encode**
   Encode to: dXNlcjpwYXNzd29yZA==

3. **Send with request**
   Add Authorization: Basic <encoded> header to every request

4. **Server decodes and checks**
   Server decodes → validates → 200 OK or 401 Unauthorized

Pros

- ✓ Extremely simple to implement
- ✓ Universally supported
- ✓ Zero token management

Cons

- ✗ Credentials sent every request
- ✗ Base64 is NOT encryption
- ✗ Must use HTTPS — always
- ✗ No token expiry

Used by:
Legacy enterprise systems, internal tooling, quick API prototypes

3B — API Key Authentication

MOST COMMON | MACHINE-TO-MACHINE | EASY TO REVOKE

Simple Explanation 🟣

The real-world analogy:

Imagine a gym membership card. The provider (gym) gives you a unique card number when you sign up. You swipe that card every visit — the system looks up your number and lets you in. Your card IS your identity. If someone steals it, they can pretend to be you. But the gym can deactivate that card instantly and issue you a new one.

Technical Explanation 💻

How it works under the hood:

The API provider generates a unique, random string (the key) and associates it with your account in their database. You include this key in every API request, typically in a header. The server hashes/looks up the key to identify you. Keys are long random strings, making them computationally infeasible to guess. They can be scoped, rate-limited, and revoked.

API KEY — THREE COMMON PLACEMENTS

```
# Option 1: Authorization header (recommended)
GET /api/users
Authorization: Api-Key sk_live_a1b2c3d4e5f6g7h8i9j0

# Option 2: Custom header
X-API-Key: sk_live_a1b2c3d4e5f6g7h8i9j0

# Option 3: Query parameter (least secure)
GET /api/users?api_key=sk_live_a1b2c3d4e5f6g7h8i9j0
```

Flow: Step by Step

1 **Provider generates unique key**
A long random string is created: sk_live_a1b2c3...

2 **You store it securely**
Save in environment variables — NEVER in source code

3 **Attach key to every request**
Add it in the Authorization header or custom header

4 **Server looks up the key**
Server finds the key in DB → maps to your account → grants access

Pros

- ✓ Simple to implement & use
- ✓ Easy to revoke instantly
- ✓ Can be scoped to specific endpoints
- ✓ No user password exposed

Cons

- ✗ Key theft = full access
- ✗ No built-in expiry
- ✗ Not user-specific
- ✗ Risky if committed to git

Used by:
Stripe, OpenAI, Google Maps API, Twilio, SendGrid

3C — JWT (JSON Web Token)

STATELESS | SELF-CONTAINED | SCALABLE

Simple Explanation 🟣

The real-world analogy:

Think of a JWT like a passport. When you log in, the server gives you a passport (token) that contains your identity and is cryptographically stamped (signed) by the issuer.
At every border (endpoint), the guard checks the stamp — not a database — to verify it's real. The stamp itself IS the proof. You carry your identity with you.

Technical Explanation 💻

How it works under the hood:

A JWT has 3 Base64Url-encoded parts separated by dots: Header.Payload.Signature.
Header: algorithm used (e.g. HS256). Payload: claims (user_id, role, expiry).
Signature: HMAC(base64(header) + '.' + base64(payload), secret).
The server verifies the signature by recomputing it — no database call needed.
If someone tampers with the payload, the signature won't match → rejected.

JWT STRUCTURE — DECODED

```
# Encoded JWT (3 parts separated by dots):
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxMjMsInJvbGUiOiJhZG1pbiIsImV4cCI6MTcxMjM0NTY3OH
0.SflKxwRJSMeKKF2QT4fwpMeJf36POk

# Part 1 - HEADER (blue):
{ "alg": "HS256", "typ": "JWT" }

# Part 2 - PAYLOAD (green):
{ "user_id": 123, "role": "admin", "exp": 1712345678 }

# Part 3 - SIGNATURE (red):
HMACSHA256(base64(header) + '.' + base64(payload), YOUR_SECRET)
```

Flow: Step by Step

1	**User logs in once** Client sends username + password to /auth/login
2	**Server builds JWT** Creates header + payload + cryptographic signature
3	**Token sent to client** Client stores JWT (memory / HttpOnly cookie)
4	**Client sends JWT on every request** Authorization: Bearer eyJhbGci...
5	**Server verifies signature only** No DB lookup needed — math proves the token is genuine

Pros

- ✓ Stateless — no DB lookup per request
- ✓ Self-contained user info in payload
- ✓ Scales across microservices
- ✓ Built-in expiry (exp claim)

Cons

- ✗ Cannot invalidate before expiry
- ✗ Token theft risk if not secured
- ✗ Payload is base64 — not encrypted
- ✗ Larger than a session ID

Used by:
Microservices architectures, mobile apps, single-page apps, SSO systems

3D — OAuth 2.0

DELEGATED ACCESS | THIRD-PARTY | NO PASSWORD SHARING

Simple Explanation 🟣

The real-world analogy:

When you click "Sign in with Google" on Notion, Notion never sees your Google password. Instead, Google asks YOU directly: 'Do you allow Notion to see your name and email?' You say yes, and Google gives Notion a limited permission slip (access token). Notion uses that slip to act on your behalf — and you can revoke it anytime from Google settings.

Technical Explanation 💻

How it works under the hood:

OAuth 2.0 is an authorization FRAMEWORK that enables delegated access. Key roles:
- Resource Owner: the user who owns the data
- Client: the third-party app requesting access
- Authorization Server: issues tokens (Google, GitHub, etc.)
- Resource Server: the API being accessed

The Authorization Code flow exchanges a short-lived code for an access token.

OAUTH 2.0 — AUTHORIZATION CODE FLOW

```
Step 1: Client redirects user to Auth Server
GET https://accounts.google.com/o/oauth2/auth
  ?client_id=YOUR_CLIENT_ID
  &redirect_uri=https://yourapp.com/callback
  &scope=email profile
  &response_type=code

Step 2: User logs in & approves → Auth Server redirects back
GET https://yourapp.com/callback?code=AUTH_CODE_HERE

Step 3: Client exchanges code for access token (server-to-server)
POST https://oauth2.googleapis.com/token
  client_id, client_secret, code, redirect_uri

Step 4: Use access token to call APIs
GET https://api.example.com/userinfo
Authorization: Bearer ya29.a0AX9GBdX...
```

Flow: Step by Step

1. **User clicks "Sign in with Google."**
   App redirects user to Google's login page — never sees the password

2. **User authenticates & approves**
   User logs in on Google's page and grants specific permissions

3. **Auth code returned to app**
   Google sends a short-lived authorization code to the app's callback URL

4. **App exchanges code for token**
   App sends code + secret to Google → receives access token

5. **Access token used for API calls**
   App calls APIs on user's behalf. User can revoke anytime.

Auth Code	Client Credentials	Implicit (deprecated)	Password
Web apps (most secure)	Server-to-server	Legacy SPAs only	First-party apps only

Pros

- ✓ User password never shared
- ✓ Granular scope control
- ✓ Revocable access tokens
- ✓ Industry standard

Cons

- ✗ Complex to implement
- ✗ Multiple redirect steps
- ✗ Requires HTTPS everywhere
- ✗ Token management overhead

Used by:
Google, GitHub, Facebook, Spotify — any "Sign in with..." button

3E — Session-Based Authentication

CLASSIC WEB | STATEFUL | SERVER-SIDE STORAGE

Simple Explanation 🟣

The real-world analogy:

Think of a coat check at a restaurant. You hand over your coat (credentials), they give you a ticket number (session ID). Every time you need something, you show your ticket. The coat check (server) still holds your coat — they do the lookup each time. When you leave (logout), they destroy the ticket and return your coat.

Technical Explanation 💻

How it works under the hood:

After successful login, the server creates a session object and stores it in memory, a DB, or Redis. A session_id is sent to the client as an HttpOnly cookie. On every subsequent request, the browser automatically sends the cookie. The server looks up the session_id in its store to retrieve the user. Logout = delete the session server-side instantly.

SESSION-BASED AUTH FLOW

```
# Step 1: Login — server creates session
POST /login  { username: 'john', password: 'secret' }
Response: Set-Cookie: session_id=abc123xyz; HttpOnly; Secure

# Step 2: Subsequent requests — browser auto-sends cookie
GET /api/profile
Cookie: session_id=abc123xyz

# Step 3: Server looks up session
Redis/DB: sessions['abc123xyz'] → { user_id: 42, role: 'admin' }

# Step 4: Logout — server deletes session
DELETE sessions['abc123xyz']  ← token instantly invalid
```

Pros

- ✓ Instant logout / revocation
- ✓ Simple to implement
- ✓ Session data stays server-side
- ✓ Works great for web apps

Cons

- ✗ Stateful — hard to scale horizontally
- ✗ DB/Redis lookup every request
- ✗ Cookie-based — CSRF risk
- ✗ Not ideal for mobile / APIs

Used by:

Traditional web apps — Django, Rails, Laravel default session handling

3F — HMAC Authentication

TAMPER-PROOF | CRYPTOGRAPHIC SIGNATURE | HIGH INTEGRITY

Simple Explanation 🟣

The real-world analogy:

Imagine sealing a letter with a wax stamp unique to you. When the recipient gets it, they can verify the stamp is yours AND that the letter hasn't been opened or tampered with. HMAC works the same — it proves the request came from you AND that nothing changed in transit. Even one changed character breaks the signature.

Technical Explanation 💻

How it works under the hood:

Client and server share a secret key. For each request, the client computes:
Signature = HMAC-SHA256(secret, method + path + timestamp + body)
The signature + timestamp are sent with the request. The server recomputes the exact same signature using the shared secret. If they match → request is authentic AND untampered. The timestamp prevents replay attacks (old requests can't be reused).

HMAC SIGNATURE EXAMPLE (AWS STYLE)

```
# Client computes signature:
string_to_sign = 'GET' + '/api/data' + '2024-04-18T10:00:00Z' + body_hash
signature = HMAC-SHA256(secret_key, string_to_sign)

# Request headers:
GET /api/data
X-Timestamp: 2024-04-18T10:00:00Z
Authorization: HMAC-SHA256 key_id=mykey, signature=a3f9b2c1...

# Server recomputes same signature and compares
# If timestamp > 5 mins old → reject (replay attack prevention)
```

Pros

- ✓ Request integrity guaranteed
- ✓ Replay attack prevention
- ✓ Secret never transmitted
- ✓ Used by AWS, Stripe webhooks

Cons

- ✗ Complex to implement correctly
- ✗ Both sides must share secret
- ✗ Clock sync required
- ✗ Harder to debug

Used by:

AWS Signature V4, Stripe webhook verification, payment gateways

3G — Magic Link / Passwordless

NO PASSWORD | EMAIL-BASED | ONE-TIME TOKEN

Simple Explanation 🟣

The real-world analogy:

Instead of remembering a key to your house, someone sends a one-time teleportation door directly to you. You click it, you're in — and the door disappears forever after. No key to remember, no key to lose. Next time you need in, you request a new door.

Technical Explanation 💻

How it works under the hood:

User submits their email. Server generates a short-lived, single-use token (UUID or JWT), stores it with an expiry (usually 15 mins), and emails a link containing it.
User clicks the link → server validates the token (exists + not expired + not used) → marks token as used → creates a session or issues a JWT. No password ever stored.

MAGIC LINK FLOW

```
# Step 1: User requests login
POST /auth/magic-link  { email: 'user@example.com' }

# Step 2: Server generates token and emails link
token = crypto.randomUUID()  // e.g. 'a1b2c3d4-e5f6...'
store: { token, email, expires_at: now + 15mins, used: false }
Email: 'Click to login: https://app.com/auth?token=a1b2c3d4-e5f6...'

# Step 3: User clicks link — server validates
GET /auth?token=a1b2c3d4-e5f6...
Check: token exists? not expired? not already used?
If valid → mark used → issue JWT or session → redirect to app
```

Pros

- ✓ No password to remember or store
- ✓ Phishing resistant
- ✓ Simple UX
- ✓ No credential breach risk

Cons

- ✗ Depends on email deliverability
- ✗ Token can be intercepted via email
- ✗ Not great for high-frequency use
- ✗ 15-min expiry can frustrate users

Used by:

Notion, Slack, Linear, Medium — common in modern SaaS products

3H — OTP / MFA (Multi-Factor Authentication)

SECOND LAYER | TIME-BASED | EXTRA SECURITY

Simple Explanation 🟣

The real-world analogy:

A bank vault has two locks — one only the bank manager can open, one only you can open. Both must be unlocked. MFA works the same: even if someone steals your password, they still can't get in without your second factor (your phone). Two locks, two keys, two owners.

Technical Explanation 💻

How it works under the hood:

MFA adds a second verification step after the primary credential (password/token).
TOTP (Time-based OTP): Client and server share a secret seed. An algorithm generates a 6-digit code from seed + current 30-second time window. Codes expire every 30 seconds.
SMS OTP: Server generates random 6-digit code, sends via SMS, stores hashed version.
User submits code → server validates hash + expiry. OTP is single-use.

TOTP FLOW (Google Authenticator style)

```
# Setup: Server generates shared secret
secret = base32_encode(random_bytes(20))  // e.g. 'JBSWY3DPEHPK3PXP'
QR code shown to user → scanned into Google Authenticator

# Every 30 seconds, app generates new code:
time_step = floor(unix_timestamp / 30)
otp = HMAC-SHA1(secret, time_step) → truncate to 6 digits
// e.g. → 482 916

# Login flow:
Step 1: User enters password → verified
Step 2: User enters 6-digit OTP from app → server computes same OTP
Step 3: Codes match + within 30s window → access granted
```

SMS OTP	TOTP (Authenticator App)	Hardware Key (FIDO2)
Easy but SMS interception risk	Secure, no internet needed	Most secure, phishing-proof

Pros

- ✓ Huge security boost over password alone
- ✓ Stops stolen credential attacks
- ✓ Industry standard for sensitive apps
- ✓ Multiple implementation options

Cons

- ✗ Extra friction for users
- ✗ SMS vulnerable to SIM swapping
- ✗ App loss = locked out risk
- ✗ Extra implementation complexity

Used by:
Every major platform — Google, GitHub, banks, AWS console, enterprise apps

Authentication Types — Tier Overview

Must Know	HTTP Basic, API Key, JWT, OAuth 2.0
Good to Know	Session-Based, HMAC, Magic Link / Passwordless, OTP / MFA
Advanced	mTLS (Certificate-Based), SAML, Biometric / FIDO2

Quick Comparison: All 8 Methods

Method	Stateless?	Expiry	Best For
HTTP Basic	Yes	No	Legacy systems, internal tools
API Key	Yes	No (manual)	Machine-to-machine, public APIs
JWT	Yes	Yes (exp)	Microservices, mobile, SPAs
OAuth 2.0	Yes	Yes	Third-party integrations, SSO
Session-Based	No	Yes	Traditional web apps
HMAC	Yes	Yes (timestamp)	Payment APIs, webhooks, AWS
Magic Link	Yes	Yes (15 mins)	Modern SaaS, passwordless UX
OTP / MFA	Yes	Yes (30 secs)	Second factor — any sensitive app

04
Authentication vs Authorization

The most common interview confusion — cleared up forever

The One-Line Answer

Authentication	Authorization
"Who are you?"	"What can you do?"
Verifying identity	Checking permissions

Simple Explanation 🟣

Hotel analogy (you'll never forget this):

AUTHENTICATION: You arrive at a hotel and show your passport at reception. The staff checks your ID and confirms you are who you claim to be. This is authentication.

AUTHORIZATION: Your room keycard only opens YOUR room — not the penthouse suite, not the staff office. The card proves you belong here AND limits what you can access.

Both happen: Authentication proves identity. Authorization controls access.

Technical Explanation 💻

For the developer:

Authentication (AuthN): Validates credentials and establishes identity. Uses mechanisms like API keys, JWTs, or OAuth tokens. If it fails → 401 Unauthorized.

Authorization (AuthZ): Once identity is established, determines what resources/actions the identity is permitted to access. Typically implemented via RBAC (Role-Based Access Control) or ABAC (Attribute-Based Access Control). If it fails → 403 Forbidden.

They are separate concerns. A user can be authenticated (logged in) but unauthorized (trying to delete another user's data).

The HTTP Status Code Rule

Code	Name	Meaning
401	**Unauthorized**	Authentication failed. Who are you? I don't know you.
403	**Forbidden**	Authorization failed. I know who you are, but you can't do this.

Side-by-Side Comparison

Aspect	Authentication (AuthN)	Authorization (AuthZ)
Question	Who are you?	What can you do?
Purpose	Verify identity	Check permissions
Comes first?	Yes — always first	Only after AuthN
HTTP failure	401 Unauthorized	403 Forbidden
Examples	JWT, API Key, OAuth token	RBAC roles, ACL rules, scopes
Think of it as...	Passport check	Visa / ticket check

Interview Cheat Sheet 🎯

Say exactly this in an interview:

"Authentication verifies WHO you are. Authorization determines WHAT you are allowed to do. They are separate concerns — authentication always happens first."

"A 401 means the request lacks valid credentials — authentication failed."
"A 403 means credentials are valid but permissions are insufficient — authorization failed."

Common implementations:
AuthN: JWT tokens, API keys, OAuth access tokens, Basic Auth
AuthZ: RBAC (roles), OAuth scopes, resource-level permissions, ACLs