APIs: a complete technical introduction — everything you need to know

Francesco Barbato · Full Stack · Cybersecurity · AI

Published 12 March 2026 · 20 min read

ABSTRACT

APIs are the connective tissue of modern software, but most explanations stop at “they let apps talk to each other”. What an API really is, how HTTP works, REST vs GraphQL vs gRPC vs WebSockets, OAuth 2.0 and JWT, API design that doesn't break consumers, and the vulnerabilities that matter in practice.

Keywords: API · REST · GraphQL · gRPC · WebSockets · Backend · Software architecture · Security

At some point in the last decade the word “API” escaped technical documentation and entered the everyday vocabulary of product managers, founders and analysts. It is now one of the most used, and most loosely used, terms in the industry.

That is a problem, because APIs are precise things. Understanding them precisely is useful whether you build software, evaluate vendors, manage a team, or just want to understand the systems you depend on. This guide starts from first principles and builds toward a complete picture: concepts, protocols, architectural styles, design patterns, security and production realities.

1 What an API actually is

API stands for **Application Programming Interface**, and the important word is *interface*: a defined point of interaction between two systems, a contract that says which inputs are accepted, which outputs are produced and which rules govern the exchange. The two sides need to know nothing about each other's internals, only the interface.

A light switch is a physical interface. You don't need to know the wiring behind the wall: up is on, down is off. The wiring can be replaced entirely, and as long as the switch behaves the same, nothing changes for you. An API is the same thing for software. It defines:

- **Endpoints**: the addresses through which the system can be reached
- **Operations**: what can be done (read data, create a record, delete a resource, trigger a process)

- **Input and output formats**: what must be sent and what comes back, and in which structure
- **Error conditions**: what happens when something goes wrong and how it is communicated
- **Authentication**: how the caller proves it is allowed to use the API

Any two pieces of software can have an API relationship. A library function exposes one (arguments in, return value out), an operating system exposes system calls, a web service exposes an API over HTTP without revealing which database sits behind it.

> **THE FUNDAMENTAL ABSTRACTION**
>
> An API gives a stable, well-defined interface that hides implementation complexity. This separation between what is promised and how it is done enables modularity, independent deployment, parallel development and the whole ecosystem of third-party integrations.

2 Why APIs matter

APIs are the infrastructure through which almost all modern software is composed. Think of what happens when you book a flight on an aggregator:

- it queries several airline APIs for availability and prices
- it calls a payment processor to take the money
- it calls hotel and car-rental APIs to offer bundles
- it calls a maps API to show the airport
- it calls email and SMS APIs for the confirmation and the boarding pass

The aggregator may have a few dozen engineers, yet what it offers is the combined work of thousands of engineers in dozens of companies, made composable by APIs. Stripe's business is an API for payments, Twilio's is an API for communications, and AWS is at its core a collection of APIs over computing infrastructure. The “API economy” is not a metaphor: it is how modern products are built and how value flows between them.

3 Anatomy of a web API request

Almost all modern APIs are built on HTTP, so understanding HTTP is the foundation for everything else. A request has a few parts.

3.1 The method

HTTP methods (verbs) express the intended action:

Table 1

Method	Intent	Idempotent?	Safe?
GET	Retrieve a resource	Yes	Yes
POST	Create a resource or trigger an action	No	No

PUT	Replace a resource entirely	Yes	No
PATCH	Partially update a resource	Not guaranteed	No
DELETE	Remove a resource	Yes	No
HEAD	Like GET, headers only	Yes	Yes
OPTIONS	Describe the available options	Yes	Yes

Idempotent means that repeating the same request has the same effect as making it once. *Safe* means the request does not change server state. These properties decide how clients retry, how caches behave and how servers can be safely replicated.

3.2 The URL and the path

The URL identifies the resource. In a well-designed REST API the structure itself carries meaning: `https://api.example.com/v1/users/42/orders` reads as "the orders of user 42, in version 1 of the API".

3.3 Headers

Headers carry metadata: credentials, content type, accepted formats, caching directives and more. The ones you meet most:

- `Authorization`: the credentials (API key, Bearer token, ...)
- `Content-Type`: the format of the request body, for example `application/json`
- `Accept`: the response formats the client can handle
- `X-Request-ID`: a client-generated identifier to trace a request across distributed systems

3.4 The body

For POST, PUT and PATCH the body carries the payload, usually JSON, sometimes XML, multipart form data or a binary format.

3.5 The response

The server answers with a status code, headers and a body. Status codes are standardized and carry meaning:

Table 2

Range	Category	Common codes
2xx	Success	200 OK, 201 Created, 204 No Content
3xx	Redirection	301 Moved Permanently, 304 Not Modified
4xx	Client error	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Too Many Requests
5xx	Server error	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

4 Architectural styles

A style defines the constraints and conventions of an API, trading off simplicity, flexibility, performance and tooling. Each exists for a reason.

4.1 REST: Representational State Transfer

REST is not a protocol or a standard but an architectural style, described by Roy Fielding in his 2000 doctoral dissertation. An API that follows its constraints is called "RESTful":

- **Client-server**: client and server are separate
- **Stateless**: each request contains everything needed to understand it, and the server keeps no session state between requests
- **Cacheable**: responses say whether they can be cached
- **Uniform interface**: resources are identified in requests and manipulated through representations, messages are self-descriptive, and hypermedia drives application state (HATEOAS)
- **Layered system**: the client can't tell whether it talks to the server or to an intermediary
- **Code on demand** (optional): the server can extend the client by sending executable code

In practice most "REST APIs" implement a subset (statelessness, the uniform interface, resource-based URLs) and ignore the rest, HATEOAS above all. What most developers call REST is more accurately a *resource-oriented HTTP API*, but the term is so widespread that arguing about it has little practical value.

REST is the dominant style for public web APIs. Its strengths are simplicity, excellent tooling, readable URLs and full compatibility with HTTP infrastructure (caches, load balancers, CDNs). Its weaknesses are over-fetching (receiving more data than needed), under-fetching (needing several requests to assemble a view) and the lack of a standard schema language, a gap that OpenAPI (formerly Swagger) has largely filled.

> **EXAMPLE: GITHUB'S API**
>
> GitHub's API is a canonical example of REST at scale. `GET /repos/{owner}/{repo}` returns a repository, `GET /repos/{owner}/{repo}/issues` its issues, `POST /repos/{owner}/{repo}/issues` creates one and `PATCH /repos/{owner}/{repo}/issues/{issue_number}` updates it. The hierarchy is clear, the methods map to intuitive operations and responses are consistent across endpoints: an experienced developer can often guess the right endpoint without reading the docs.

4.2 GraphQL

GraphQL is a query language for APIs and a runtime to execute those queries, created at Facebook (today Meta) and open-sourced in 2015. It targets the limits of REST that hurt at scale. If a mobile client needs only a user's name and avatar but `/users/{id}` returns 40 fields, it receives 40 fields (over-fetching). If a profile screen needs the user, their recent posts and the comments on them, it

may need three calls (under-fetching). GraphQL lets the client ask for exactly what it needs in one query:

```graphql
query {
user(id: "42") {
name
avatar
recentPosts(limit: 5) {
title
publishedAt
comments(limit: 3) {
author { name }
body
}
}
}
}
```

The query walks a graph of related data and returns exactly what was requested. The server exposes a typed schema, and clients can introspect it, which enables powerful tooling.

Its strengths are flexibility for nested data, strong typing, a great developer experience and efficiency for data-hungry clients. Its weaknesses are server-side complexity (the N+1 query problem, authorization), harder caching (everything goes to a single endpoint) and the learning curve for people used to REST.

4.3 gRPC

gRPC is a high-performance remote procedure call framework from Google, built on HTTP/2 and Protocol Buffers. Where REST and GraphQL typically use human-readable JSON, gRPC uses binary serialization, which is faster to process and more compact on the wire. You describe the API in a `.proto` file:

```protobuf
service UserService {
rpc GetUser (GetUserRequest) returns (User);
rpc ListUsers (ListUsersRequest) returns (stream User);
rpc CreateUser (CreateUserRequest) returns (User);
}

message User {
string id = 1;
string name = 2;
string email = 3;
int64 created_at = 4;
}
```

From this definition the toolchain generates client and server code in many languages: a Go client can call a Python server as if it were a local function, with serialization and transport handled by the generated code. gRPC supports four patterns:

- **Unary**: one request, one response, like a function call

- **Server streaming**: one request, a stream of responses
- **Client streaming**: a stream of requests, one response
- **Bidirectional streaming**: two simultaneous streams

Its strengths are performance (binary format, HTTP/2 multiplexing, connection reuse), strong typing, multi-language support and native streaming. Its weaknesses are limited browser support (web clients need a proxy layer), lower readability than JSON and a heavier setup. It is a common choice for internal microservice-to-microservice traffic, where performance and type safety across many services and languages matter.

4.4 WebSockets

HTTP is request-response: the client asks, the server answers, the exchange ends. That breaks down when you need real-time, bidirectional communication: chat, live dashboards, collaborative editing, live scores.

WebSockets open a persistent, full-duplex connection. After an initial HTTP handshake that upgrades the connection, both sides can send messages at any time without the cost of a new connection per message, until one of them closes it. There are no "requests" and "responses" any more, just messages flowing both ways.

4.5 Webhooks

A webhook is an HTTP callback: a way for a server to tell a client that something happened by calling a URL the client registered. A normal API call is client-initiated ("tell me what happened"); a webhook is server-initiated ("something happened, here is the data").

The client registers a URL and the events it cares about. When one occurs, the service sends an HTTP POST with a description of the event, and the client acknowledges with a 2xx. Payment processors notify your server when a payment succeeds or fails, version control notifies CI/CD pipelines when code is pushed, e-commerce platforms notify fulfilment when an order is placed. They are simple and widely supported, but raise reliability questions (what if the client's URL is down for a while?) that need careful handling.

> **WEBHOOKS VS. POLLING**
>
> The alternative is polling: the client keeps asking "has anything changed?". It is simple but wasteful (most requests return nothing new) and the delay is at least one polling interval. Webhooks remove both problems but need the client to be reachable from outside, which is not always possible.

4.6 SOAP

SOAP is an XML-based messaging protocol that predates REST and survives mostly in enterprise and financial contexts. A SOAP API is described by a WSDL file listing operations, inputs, outputs and the endpoint, from which clients generate typed proxy code. Messages are XML envelopes with an optional Header (authentication, transaction information) and a Body. The WS-* extensions add message-level security (WS-Security), guaranteed delivery (WS-ReliableMessaging) and distributed transactions (WS-AtomicTransaction), things that are hard to replicate cleanly in REST.

Verbosity, complexity and XML processing make it a poor fit for high-volume, latency-sensitive uses, but where guaranteed delivery, distributed transactions or message-level (not just transport-level) security are needed, it is still relevant.

5 Authentication and authorization

Authentication proves who you are; authorization decides what you may do. A poorly protected API is an attack surface for data theft, unauthorized operations and abuse of the service.

5.1 API keys

An API key is a long random string the client sends with each request. The server maps keys to identities and permissions. Keys are simple and widely used, but they are long-lived credentials: if one leaks, it grants persistent access until revoked. Send them in a header, never in the URL, where they end up in server logs.

5.2 OAuth 2.0

OAuth 2.0 is the industry-standard protocol for delegated authorization: it lets a user grant a third-party application access to their resources on another service without sharing their credentials with it. When a dialog asks “Allow this app to access your Google Drive?”, that is OAuth 2.0. It defines several **grant types**:

Table 3

Grant type	Use case	How it works
Authorization Code	Web apps acting for a user	The user is redirected to the authorization server, approves access and the app exchanges the returned code for tokens
Authorization Code + PKCE	Mobile and single-page apps	Like Authorization Code, plus a cryptographic proof that prevents code interception
Client Credentials	Server-to-server, no user	The application authenticates with its client ID and secret and receives an access token
Device Code	Devices without a browser (smart TVs, CLIs)	The device shows a code and the user authorizes on another device

5.3 JWT: JSON Web Tokens

A JWT is a compact, self-contained token for carrying claims between parties as a signed (and optionally encrypted) JSON object. It has three base64url-encoded parts separated by dots: a header (algorithm and type), a payload (the claims: user ID, roles, expiry, issuer) and a signature.

The key property is that the server need not store tokens: it validates one by checking the signature, with no database lookup. That makes JWT authentication very scalable and suited to stateless architectures. The price is that a JWT can't be revoked before it expires without extra infrastructure (a revocation list), which partly defeats the point.

5.4 mTLS: mutual TLS

Standard TLS authenticates the server to the client. Mutual TLS also authenticates the client to the server with a client certificate, so both sides must present and verify certificates before the connection is established. It is common for service-to-service authentication in microservice and zero-trust environments, where every service must prove its identity, and it gives strong, cryptographically verified authentication without shared secrets.

6 API design principles

A well-designed API can be used correctly and efficiently with minimal friction. That is harder than it sounds: API design is a product discipline as much as a technical one, because the developers who use your API are your users.

6.1 Consistency

Consistency is the most important quality. Naming, URL patterns, error formats, pagination and filtering should be uniform across the API. If one endpoint uses `created_at` and another `createdAt`, or one paginates with `page`/`per_page` and another with `offset`/`limit`, every inconsistency is a source of bugs and confusion.

6.2 Idempotency

Network requests fail in ambiguous ways: a timeout can happen after the server processed the request but before it answered, leaving the client unsure whether the operation succeeded. For operations that change state (payments, orders, writes) the client must be able to retry safely.

The standard mechanism is an **idempotency key**: a unique, client-generated identifier sent in a header. If the server has already processed a request with that key, it returns the stored response instead of doing the work again. Stripe's API is the usual example of the pattern done well.

6.3 Versioning

APIs change: fields are added, others deprecated, behaviours modified. Versioning manages change without breaking existing consumers. The common approaches:

- **URL versioning**: `/v1/users`, `/v2/users`. Simple, explicit and easy to route, but it can lead to many parallel versions
- **Header versioning**: `Accept: application/vnd.api+json;version=2`. Keeps URLs clean, but is less discoverable
- **Query parameter versioning**: `/users?version=2`. Simple, but it pollutes the query string

The key principle is **backward compatibility**: adding new optional fields to responses doesn't break anyone; removing fields, changing types or changing behaviour does. Semantic versioning (major.minor.patch) applies to APIs as to libraries.

6.4 Pagination

Any endpoint that returns a collection must paginate: returning everything in one response is impractical at scale and potentially dangerous. There are two main approaches.

Offset-based pagination uses a page number or offset and a page size: `GET /users?page=3&per_page=20`. It is easy to implement and allows jumping to any page, but inserts and deletes between requests can make records appear twice or be skipped.

Cursor-based pagination uses an opaque cursor, typically an encoded reference to the last item seen: `GET /users?cursor=eyJpZCI6NDJ9&limit=20`. It is stable under concurrent changes and scales better on large datasets, but you can't jump to an arbitrary page. It is the preferred choice for feeds and real-time data.

6.5 Rate limiting

Rate limiting caps how many requests a client can make in a time window, protecting the server from overload and sharing resources fairly. Well-designed APIs communicate the state of the limit with headers:

- `X-RateLimit-Limit`: the maximum number of requests in the window
- `X-RateLimit-Remaining`: the requests left in the current window
- `X-RateLimit-Reset`: when the window resets
- `Retry-After`: with a 429, how long to wait before retrying

The `X-RateLimit-*` names are a widespread convention rather than a single standard, so check what each API actually sends.

6.6 Error design

Error responses matter as much as successes. A good one says what went wrong, why, and what to do. At a minimum: a machine-readable error code (not just the HTTP status), a human-readable message and, ideally, a link to the documentation.

```JSON
{
"error": {
"code": "VALIDATION_ERROR",
"message": "The request body contains invalid fields.",
"details": [
{
"field": "email",
"issue": "INVALID_FORMAT",
"description": "Must be a valid email address."
}
],
"request_id": "req_8f3k2m9x",
"documentation_url": "https://docs.api.example.com/errors#VALIDATION_ERROR"
}
}
```

7 Documentation

An API is only as good as its documentation. A technically excellent API with poor documentation gets used incorrectly, adopted slowly and abandoned for alternatives. Documentation is a product deliverable, not an afterthought.

The industry standard for REST APIs is the **OpenAPI Specification**, a machine-readable YAML or JSON description of endpoints, parameters, bodies, responses, authentication and schemas. From an OpenAPI document you can generate documentation portals, client SDKs in many languages, mock servers and test suites. For GraphQL, the introspectable schema is the documentation, and tools such as GraphiQL and Apollo Studio let developers explore and test the API in the browser.

> **STRIPE: THE USUAL BENCHMARK FOR API DOCS**
>
> Stripe's documentation is widely cited as the best in the industry, not because the API can do more than others but because it makes the API easy to understand: every endpoint fully described, parameters with types and constraints, runnable code examples in several languages, real example payloads and every error code explained. It is a good reminder that documentation is a competitive differentiator, not just a way to cut support costs.

8 Security: the attack surface

APIs are among the most targeted surfaces of modern applications. The OWASP API Security Top 10 lists the most critical API-specific risks, several of them different from generic web application concerns.

8.1 Broken Object Level Authorization (BOLA)

BOLA, also called IDOR (Insecure Direct Object Reference), is the most common API vulnerability. It happens when an endpoint takes a resource identifier from the client and returns the resource without checking that the requesting user may see it. `GET /api/orders/12345` must verify that the authenticated user owns order 12345, not merely that they are authenticated.

8.2 Broken Function Level Authorization

It occurs when an API exposes administrative or privileged functions without restricting them properly. `DELETE /api/users/{id}` should be reachable only by administrators, but if the check is missing or rests on a predictable pattern (such as “the request comes from the admin UI”) instead of the role claims in the token, any authenticated user may be able to call it.

8.3 Mass assignment

Many frameworks map request properties to model attributes automatically. If the API assigns every submitted property without an allowlist, an attacker can send unexpected ones, like `is_admin: true` or `account_balance: 999999`, and have them applied to their account.

8.4 Excessive data exposure

APIs often return more than the client asked for and rely on the client to filter what it shows. That leaks data that should never leave the server: internal identifiers, personal information, business-sensitive fields. The API should return only what the client needs, enforced server-side.

> **THE AUTHORIZATION GAP**
>
> Authentication confirms identity; authorization decides what that identity may do. Most API security failures are authorization failures, not authentication ones. An attacker with a valid token who can read other users' data, call privileged endpoints or edit fields they shouldn't touch has exploited an authorization gap, and these are far more common in production than broken authentication.

9 API gateways and management

At scale, managing many APIs across many services needs infrastructure beyond each implementation. An **API gateway** sits between clients and backend services as the entry point for all API traffic, and handles cross-cutting concerns that would otherwise be reimplemented in every service:

- **Authentication and authorization**: validate tokens before forwarding requests
- **Rate limiting**: enforce usage policies in one place
- **Routing**: send each request to the right backend service
- **Load balancing**: spread traffic across instances
- **TLS termination**: handle TLS at the gateway and talk to services internally
- **Transformation**: translate formats, add or remove headers
- **Observability**: central logs, metrics and traces for all API traffic
- **Caching**: reduce load on the backends

Common products include Kong, AWS API Gateway, Google Cloud Apigee, Azure API Management and open-source options like Traefik and NGINX. In Kubernetes, the Ingress controller and the service mesh (Istio, Linkerd) play related roles for internal service-to-service traffic.

10 The API lifecycle

APIs are long-lived artefacts, and their lifecycle goes well beyond the first deployment.

10.1 Design-first vs. code-first

Design-first defines the specification (OpenAPI, protobuf) before any implementation. It is reviewed and agreed by producers and consumers (frontend and mobile teams, external partners) before a line of server code exists. It catches design problems when they are cheap to fix and enables parallel work, since mock servers can be generated from the spec.

Code-first implements the API and generates the documentation from the code. It is faster to start but tends to produce APIs that reflect implementation details rather than consumer needs, and makes breaking changes harder to spot because spec and implementation are tied together.

10.2 Deprecation and sunset

Old versions eventually have to be retired. The standard practice is a **deprecation period** in which the old version keeps working while consumers are warned through documentation, email and response headers: the `Deprecation` header and the `Sunset` header (RFC 8594), which carries the shutdown date, for example `Sunset: Thu, 31 Dec 2026 23:59:59 GMT`.

The length of the period should match the API's usage and the migration effort. A public API used by thousands of external developers may call for 12–24 months of notice; internal APIs can move faster.

> **KEY TAKEAWAY**
>
> APIs are products, contracts and infrastructure. Decisions in API design ripple out to every consumer, every integration, every developer experience. A good API is consistent, secure, well documented, thoughtfully versioned and built around what consumers need rather than what is convenient to implement. None of this is accidental: it comes from treating API design as a discipline of its own.

11 Conclusion

APIs are the connective tissue of modern software: how systems talk to each other, how organisations expose what they can do, and how developers build products that would be impossible to build alone. Understanding them deeply, not just superficially, is one of the most broadly useful technical skills for anyone who works with software.

The landscape of styles, protocols and patterns is wide, but the principles underneath are consistent: define clear contracts, enforce them consistently, design for the consumer rather than the implementer, protect access rigorously and change things carefully. An API that embodies them is infrastructure others can build on with confidence, and that confidence is ultimately what APIs are for.

This guide reflects the state of API design and technology as of early 2026. Tools and protocol versions evolve; the architectural principles are stable.

> **HOW TO CITE THIS ARTICLE**
>
> Francesco Barbato (2026). APIs: a complete technical introduction — everything you need to know. localhost:3000.
>
> Online version: http://localhost:3000/en/blog/api-complete-technical-guide