Le API: un'introduzione tecnica completa, tutto quello che serve sapere

Francesco Barbato · Full Stack · Cybersecurity · AI

Pubblicato il 12 marzo 2026 · 21 min di lettura

SOMMARIO

Le API sono il tessuto connettivo del software moderno, ma la maggior parte delle spiegazioni si ferma a «permettono alle app di parlarsi». Cos'è davvero un'API, come funziona HTTP, REST contro GraphQL, gRPC e WebSocket, OAuth 2.0 e JWT, come progettarle senza rompere chi le usa e quali vulnerabilità contano davvero.

Parole chiave: API · REST · GraphQL · gRPC · WebSocket · Backend · Architettura software · Sicurezza

A un certo punto dell'ultimo decennio la parola «API» è uscita dalla documentazione tecnica ed è entrata nel linguaggio quotidiano di product manager, fondatori e analisti. Oggi è uno dei termini più usati, e più usati a sproposito, del settore.

È un problema, perché le API sono cose precise. Capirle con precisione è utile sia che tu sviluppi software, sia che tu valuti fornitori, coordini un team o voglia semplicemente capire i sistemi da cui dipendi. Questa guida parte dai principi primi e costruisce un quadro completo: concetti, protocolli, stili architetturali, pattern di progettazione, sicurezza e realtà della produzione.

1 Cos'è davvero un'API

API sta per **Application Programming Interface**, e la parola importante è *interfaccia*: un punto di interazione definito tra due sistemi, un contratto che stabilisce quali input sono accettati, quali output vengono prodotti e quali regole governano lo scambio. Le due parti non devono sapere nulla dell'interno dell'altra, solo l'interfaccia.

L'interruttore della luce è un'interfaccia fisica. Non serve conoscere l'impianto dietro il muro: su è acceso, giù è spento. L'impianto può essere rifatto da capo e, finché l'interruttore si comporta allo stesso modo, per te non cambia nulla. Un'API è la stessa cosa per il software. Definisce:

- **Endpoint**: gli indirizzi attraverso cui si raggiunge il sistema
- **Operazioni**: cosa si può fare (leggere dati, creare un record, cancellare una risorsa, avviare un processo)

- **Formati di input e output**: cosa va inviato e cosa torna indietro, e con quale struttura
- **Condizioni di errore**: cosa succede quando qualcosa va storto e come viene comunicato
- **Autenticazione**: come il chiamante dimostra di avere il diritto di usare l'API

Due pezzi di software qualsiasi possono avere un rapporto di tipo API. Lo espone la funzione di una libreria (argomenti in ingresso, valore di ritorno in uscita), lo espone un sistema operativo con le chiamate di sistema, lo espone un servizio web su HTTP senza rivelare quale database ci sia dietro.

> **L'ASTRAZIONE FONDAMENTALE**
>
> Un'API offre un'interfaccia stabile e ben definita che nasconde la complessità dell'implementazione. Questa separazione tra ciò che si promette e come lo si realizza rende possibili modularità, rilasci indipendenti, sviluppo in parallelo e l'intero ecosistema di integrazioni di terze parti.

2 Perché le API contano

Le API sono l'infrastruttura con cui si compone quasi tutto il software moderno. Pensa a cosa succede quando prenoti un volo su un aggregatore:

- interroga le API di più compagnie aeree per disponibilità e prezzi
- chiama un processore di pagamenti per incassare
- chiama le API di hotel e autonoleggi per proporre pacchetti
- chiama un'API di mappe per mostrare l'aeroporto
- chiama API di email e SMS per conferma e carta d'imbarco

L'aggregatore può avere poche decine di ingegneri, ma ciò che offre è il lavoro combinato di migliaia di ingegneri in decine di aziende, reso componibile dalle API. Il business di Stripe è un'API per i pagamenti, quello di Twilio un'API per le comunicazioni, e AWS è in sostanza una raccolta di API sull'infrastruttura di calcolo. L'«economia delle API» non è una metafora: è il modo in cui si costruiscono i prodotti moderni e in cui il valore circola tra loro.

3 Anatomia di una richiesta a un'API web

Quasi tutte le API moderne poggiano su HTTP, quindi capire HTTP è la base per tutto il resto. Una richiesta ha alcune parti.

3.1 Il metodo

I metodi HTTP (verbi) esprimono l'azione voluta:

Tabella 1

Metodo	Intento	Idempotente?	Sicuro?
GET	Leggere una risorsa	Sì	Sì
POST	Creare una risorsa o avviare un'azione	No	No

PUT	Sostituire interamente una risorsa	Sì	No
PATCH	Aggiornare parzialmente una risorsa	Non garantito	No
DELETE	Rimuovere una risorsa	Sì	No
HEAD	Come GET, solo intestazioni	Sì	Sì
OPTIONS	Descrivere le opzioni disponibili	Sì	Sì

Idempotente significa che ripetere la stessa richiesta ha lo stesso effetto che farla una volta sola. *Sicuro* significa che la richiesta non modifica lo stato del server. Queste proprietà decidono come i client ritentano, come si comportano le cache e come si possono replicare i server in sicurezza.

3.2 L'URL e il percorso

L'URL identifica la risorsa. In una REST API ben progettata la struttura stessa ha un significato: `https://api.example.com/v1/users/42/orders` si legge «gli ordini dell'utente 42, nella versione 1 dell'API».

3.3 Le intestazioni (header)

Gli header portano metadati: credenziali, tipo di contenuto, formati accettati, direttive di cache e altro. I più frequenti:

- `Authorization`: le credenziali (chiave API, token Bearer, ...)
- `Content-Type`: il formato del corpo della richiesta, per esempio `application/json`
- `Accept`: i formati di risposta che il client sa gestire
- `X-Request-ID`: un identificativo generato dal client per tracciare la richiesta nei sistemi distribuiti

3.4 Il corpo

Per POST, PUT e PATCH il corpo porta i dati, di solito in JSON, a volte XML, form multipart o un formato binario.

3.5 La risposta

Il server risponde con un codice di stato, header e un corpo. I codici sono standardizzati e hanno un significato preciso:

Tabella 2

Intervallo	Categoria	Codici comuni
2xx	Successo	200 OK, 201 Created, 204 No Content
3xx	Reindirizzamento	301 Moved Permanently, 304 Not Modified
4xx	Errore del client	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Too Many Requests
5xx	Errore del server	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

4 Gli stili architetturali

Uno stile definisce vincoli e convenzioni di un'API, bilanciando semplicità, flessibilità, prestazioni e strumenti disponibili. Ognuno esiste per una ragione.

4.1 REST: Representational State Transfer

REST non è un protocollo né uno standard ma uno stile architetturale, descritto da Roy Fielding nella sua tesi di dottorato del 2000. Un'API che ne rispetta i vincoli si dice «RESTful»:

- **Client-server**: client e server sono separati
- **Stateless**: ogni richiesta contiene tutto ciò che serve per capirla e il server non conserva lo stato di sessione tra una richiesta e l'altra
- **Cacheable**: le risposte dichiarano se possono essere messe in cache
- **Interfaccia uniforme**: le risorse sono identificate nelle richieste e manipolate tramite rappresentazioni, i messaggi sono autodescrittivi e l'ipermedia guida lo stato dell'applicazione (HATEOAS)
- **Sistema a livelli**: il client non può sapere se parla col server o con un intermediario
- **Code on demand** (opzionale): il server può estendere il client inviando codice eseguibile

Nella pratica la maggior parte delle «REST API» ne implementa un sottoinsieme (assenza di stato, interfaccia uniforme, URL basati sulle risorse) e ignora il resto, HATEOAS in testa. Quello che molti sviluppatori chiamano REST è più correttamente una *API HTTP orientata alle risorse*, ma il termine è così diffuso che discuterne ha poco valore pratico.

REST è lo stile dominante per le API web pubbliche. I suoi punti di forza sono semplicità, ottimi strumenti, URL leggibili e piena compatibilità con l'infrastruttura HTTP (cache, load balancer, CDN). I punti deboli sono l'over-fetching (ricevere più dati del necessario), l'under-fetching (servono più richieste per comporre una vista) e l'assenza di un linguaggio di schema standard, lacuna che OpenAPI (già Swagger) ha in gran parte colmato.

> **ESEMPIO: L'API DI GITHUB**
>
> L'API di GitHub è un esempio canonico di REST su larga scala. `GET /repos/{owner}/{repo}` restituisce un repository, `GET /repos/{owner}/{repo}/issues` le sue issue, `POST /repos/{owner}/{repo}/issues` ne crea una e `PATCH /repos/{owner}/{repo}/issues/{issue_number}` la aggiorna. La gerarchia è chiara, i metodi corrispondono a operazioni intuitive e le risposte sono coerenti tra gli endpoint: uno sviluppatore esperto spesso indovina l'endpoint giusto senza aprire la documentazione.

4.2 GraphQL

GraphQL è un linguaggio di query per le API e un runtime per eseguirle, nato in Facebook (oggi Meta) e rilasciato open source nel 2015. Affronta i limiti di REST che fanno male su larga scala. Se un client mobile ha bisogno solo di nome e avatar di un utente ma `/users/{id}` restituisce 40 campi, riceve 40 campi (over-fetching). Se una schermata di profilo richiede l'utente, i suoi ultimi post e

i commenti ai post, possono servire tre chiamate (under-fetching). GraphQL permette al client di chiedere esattamente ciò che serve in una sola query:

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

La query attraversa un grafo di dati collegati e restituisce esattamente ciò che è stato richiesto. Il server espone uno schema tipizzato, che i client possono interrogare (introspezione), e questo abilita strumenti molto potenti.

I punti di forza sono la flessibilità sui dati annidati, la tipizzazione forte, l'ottima esperienza di sviluppo e l'efficienza per client che hanno bisogno di molti dati. I punti deboli sono la complessità lato server (il problema delle query N+1, l'autorizzazione), la cache più difficile (tutto passa da un unico endpoint) e la curva di apprendimento per chi viene da REST.

4.3 gRPC

gRPC è un framework per chiamate di procedura remota ad alte prestazioni, creato da Google e basato su HTTP/2 e Protocol Buffers. Dove REST e GraphQL usano di solito JSON leggibile, gRPC usa una serializzazione binaria, più veloce da elaborare e più compatta sulla rete. L'API si descrive in un file `.proto`:

```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;
}
```

Da questa definizione gli strumenti generano codice client e server in molti linguaggi: un client Go può chiamare un server Python come se fosse una funzione locale, con serializzazione e trasporto gestiti dal codice generato. gRPC supporta quattro modelli di comunicazione:

- **Unario**: una richiesta, una risposta, come una chiamata di funzione
- **Streaming dal server**: una richiesta, un flusso di risposte
- **Streaming dal client**: un flusso di richieste, una risposta
- **Streaming bidirezionale**: due flussi simultanei

I punti di forza sono le prestazioni (formato binario, multiplexing HTTP/2, riuso delle connessioni), la tipizzazione forte, il supporto multi-linguaggio e lo streaming nativo. I punti deboli sono il supporto limitato nei browser (i client web richiedono uno strato di proxy), la minore leggibilità rispetto a JSON e una configurazione più impegnativa. È una scelta comune per il traffico interno tra microservizi, dove contano prestazioni e sicurezza dei tipi tra molti servizi e linguaggi.

4.4 WebSocket

HTTP è richiesta-risposta: il client chiede, il server risponde, lo scambio finisce. Il modello non regge quando serve comunicazione in tempo reale e bidirezionale: chat, dashboard dal vivo, editing collaborativo, risultati sportivi in diretta.

I WebSocket aprono una connessione persistente e full-duplex. Dopo un handshake HTTP iniziale che la promuove, entrambe le parti possono inviare messaggi in qualsiasi momento, senza il costo di una nuova connessione per ogni messaggio, finché una delle due la chiude. Non ci sono più «richieste» e «risposte», solo messaggi che viaggiano nei due sensi.

4.5 Webhook

Un webhook è una callback HTTP: un modo per un server di avvisare un client che è successo qualcosa chiamando un URL che il client ha registrato. Una normale chiamata API parte dal client («dimmi cos'è successo»); un webhook parte dal server («è successo qualcosa, ecco i dati»).

Il client registra un URL e gli eventi che gli interessano. Quando uno si verifica, il servizio invia una POST con la descrizione dell'evento e il client conferma con un 2xx. I processori di pagamento avvisano il tuo server quando un pagamento riesce o fallisce, il version control avvisa le pipeline CI/CD quando arriva codice, le piattaforme e-commerce avvisano la logistica quando c'è un ordine. Sono semplici e molto supportati, ma pongono problemi di affidabilità (cosa succede se l'URL del client è giù per un po'?) che vanno gestiti con cura.

> **WEBHOOK CONTRO POLLING**
>
> L'alternativa è il polling: il client chiede in continuazione «è cambiato qualcosa?». È semplice ma dispendioso (la maggior parte delle richieste non restituisce nulla di nuovo) e il ritardo è almeno pari all'intervallo di polling. I webhook eliminano entrambi i problemi ma richiedono che il client sia raggiungibile dall'esterno, cosa non sempre possibile.

4.6 SOAP

SOAP è un protocollo di messaggistica basato su XML, precedente a REST, che sopravvive soprattutto in ambito enterprise e finanziario. Un'API SOAP è descritta da un file WSDL con operazioni, input, output e indirizzo, da cui i client generano codice proxy tipizzato. I messaggi sono buste XML con un Header opzionale (autenticazione, informazioni transazionali) e un Body.

Le estensioni WS-* aggiungono sicurezza a livello di messaggio (WS-Security), consegna garantita (WS-ReliableMessaging) e transazioni distribuite (WS-AtomicTransaction), cose difficili da replicare in modo pulito in REST.

Verbosità, complessità e costo dell'elaborazione XML lo rendono poco adatto a usi ad alto volume e bassa latenza, ma dove servono consegna garantita, transazioni distribuite o sicurezza a livello di messaggio (e non solo di trasporto) resta attuale.

5 Autenticazione e autorizzazione

L'autenticazione dimostra chi sei; l'autorizzazione stabilisce cosa puoi fare. Un'API protetta male è una superficie d'attacco per furto di dati, operazioni non autorizzate e abuso del servizio.

5.1 Chiavi API

Una chiave API è una lunga stringa casuale che il client invia a ogni richiesta. Il server associa le chiavi a identità e permessi. Sono semplici e diffuse, ma sono credenziali di lunga durata: se una trapela, concede accesso persistente finché non viene revocata. Vanno inviate in un header, mai nell'URL, dove finiscono nei log dei server.

5.2 OAuth 2.0

OAuth 2.0 è il protocollo standard per l'autorizzazione delegata: permette a un utente di concedere a un'applicazione di terze parti l'accesso alle sue risorse su un altro servizio senza condividere le proprie credenziali. Quando una finestra chiede «Consentire a questa app di accedere al tuo Google Drive?», è OAuth 2.0. Definisce diversi **tipi di concessione** (grant type):

Tabella 3

Grant type	Caso d'uso	Come funziona
Authorization Code	App web che agiscono per conto di un utente	L'utente viene reindirizzato al server di autorizzazione, approva l'accesso e l'app scambia il codice ricevuto con i token
Authorization Code + PKCE	App mobile e single-page	Come Authorization Code, più una prova crittografica che impedisce l'intercettazione del codice
Client Credentials	Da server a server, senza utente	L'applicazione si autentica con client ID e segreto e riceve un token di accesso
Device Code	Dispositivi senza browser (smart TV, CLI)	Il dispositivo mostra un codice e l'utente autorizza da un altro dispositivo

5.3 JWT: JSON Web Token

Un JWT è un token compatto e autocontenuto per trasportare dichiarazioni (claim) tra le parti come oggetto JSON firmato (e opzionalmente cifrato). Ha tre parti codificate in base64url separate da punti: un header (algoritmo e tipo), un payload (i claim: ID utente, ruoli, scadenza, emittente) e una firma.

La proprietà chiave è che il server non deve conservare i token: li valida verificando la firma, senza consultare un database. Questo rende l'autenticazione JWT molto scalabile e adatta alle architettu-

re stateless. Il prezzo è che un JWT non si può revocare prima della scadenza senza infrastruttura aggiuntiva (una lista di revoca), il che in parte vanifica lo scopo.

5.4 mTLS: TLS mutuo

Il TLS standard autentica il server verso il client. Il TLS mutuo autentica anche il client verso il server con un certificato client: entrambe le parti devono presentare e verificare i certificati prima che la connessione si stabilisca. È comune per l'autenticazione tra servizi nei microservizi e negli ambienti zero-trust, dove ogni servizio deve provare la propria identità, e dà un'autenticazione forte e verificata crittograficamente senza segreti condivisi.

6 Principi di progettazione

Un'API ben progettata si usa correttamente ed efficacemente con il minimo attrito. È più difficile di quanto sembri: progettare un'API è una disciplina di prodotto oltre che tecnica, perché gli sviluppatori che la usano sono i tuoi utenti.

6.1 Coerenza

La coerenza è la qualità più importante. Nomi, schemi degli URL, formato degli errori, paginazione e filtri devono essere uniformi in tutta l'API. Se un endpoint usa `created_at` e un altro `createdAt`, o uno pagina con `page`/`per_page` e l'altro con `offset`/`limit`, ogni incoerenza è una fonte di bug e confusione.

6.2 Idempotenza

Le richieste di rete falliscono in modi ambigui: un timeout può scattare dopo che il server ha elaborato la richiesta ma prima che risponda, lasciando il client incerto sull'esito. Per le operazioni che cambiano stato (pagamenti, ordini, scritture) il client deve poter ritentare in sicurezza.

Il meccanismo standard è la **chiave di idempotenza**: un identificativo univoco generato dal client e inviato in un header. Se il server ha già elaborato una richiesta con quella chiave, restituisce la risposta memorizzata invece di rifare il lavoro. L'API di Stripe è l'esempio usuale di questo schema fatto bene.

6.3 Versionamento

Le API cambiano: si aggiungono campi, se ne deprecano altri, si modificano i comportamenti. Il versionamento gestisce il cambiamento senza rompere i consumatori esistenti. Gli approcci comuni:

- **Versione nell'URL**: `/v1/users`, `/v2/users`. Semplice, esplicita e facile da instradare, ma può portare a molte versioni parallele
- **Versione nell'header**: `Accept: application/vnd.api+json;version=2`. Mantiene puliti gli URL ma è meno scopribile
- **Versione come parametro di query**: `/users?version=2`. Semplice, ma sporca la query string

Il principio chiave è la **retrocompatibilità**: aggiungere campi opzionali alle risposte non rompe nessuno; rimuovere campi, cambiare tipi o cambiare comportamento sì. Il versionamento semantico (major.minor.patch) vale per le API come per le librerie.

6.4 Paginazione

Ogni endpoint che restituisce una collezione deve paginare: restituire tutto in una sola risposta è impraticabile su larga scala e potenzialmente pericoloso. Gli approcci principali sono due.

La paginazione a offset usa un numero di pagina o un offset e una dimensione: `GET /users?page=3&per_page=20`. È facile da implementare e permette di saltare a qualsiasi pagina, ma inserimenti e cancellazioni tra una richiesta e l'altra possono far comparire record due volte o saltarli.

La paginazione a cursore usa un cursore opaco, in genere un riferimento codificato all'ultimo elemento visto: `GET /users?cursor=eyJpZCI6NDJ9&limit=20`. È stabile in presenza di modifiche concorrenti e scala meglio sui grandi insiemi di dati, ma non permette di saltare a una pagina arbitraria. È la scelta preferita per feed e dati in tempo reale.

6.5 Rate limiting

Il rate limiting limita quante richieste un client può fare in una finestra di tempo, proteggendo il server dal sovraccarico e ripartendo le risorse in modo equo. Le API ben progettate comunicano lo stato del limite con degli header:

- `X-RateLimit-Limit`: il numero massimo di richieste nella finestra
- `X-RateLimit-Remaining`: le richieste rimaste nella finestra corrente
- `X-RateLimit-Reset`: quando la finestra si azzera
- `Retry-After`: con un 429, quanto aspettare prima di riprovare

I nomi `X-RateLimit-*` sono una convenzione molto diffusa più che uno standard unico, quindi verifica cosa invia davvero ciascuna API.

6.6 Progettare gli errori

Le risposte d'errore contano quanto quelle di successo. Una buona dice cosa è andato storto, perché e cosa fare. Come minimo: un codice d'errore leggibile da una macchina (non solo lo status HTTP), un messaggio leggibile da una persona e, idealmente, un rimando alla documentazione.

```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 Documentazione

Un'API vale quanto la sua documentazione. Un'API tecnicamente eccellente con documentazione scadente viene usata male, adottata lentamente e abbandonata per le alternative. La documentazione è un prodotto da consegnare, non un ripensamento.

Lo standard del settore per le REST API è la **OpenAPI Specification**, una descrizione leggibile da macchina, in YAML o JSON, di endpoint, parametri, corpi, risposte, autenticazione e schemi. Da un documento OpenAPI si possono generare portali di documentazione, SDK client in più linguaggi, server mock e suite di test. Per GraphQL lo schema interrogabile è la documentazione, e strumenti come GraphiQL e Apollo Studio permettono di esplorare e provare l'API direttamente nel browser.

> **STRIPE: IL RIFERIMENTO ABITUALE PER LA DOCUMENTAZIONE**
>
> La documentazione di Stripe è citata da molti come la migliore del settore, non perché l'API faccia più delle altre ma perché la rende facile da capire: ogni endpoint descritto per intero, parametri con tipi e vincoli, esempi di codice eseguibili in più linguaggi, payload d'esempio reali e ogni codice d'errore spiegato. Ricorda che la documentazione è un fattore competitivo, non solo un modo per ridurre il carico del supporto.

8 Sicurezza: la superficie d'attacco

Le API sono tra le superfici più bersagliate delle applicazioni moderne. L'OWASP API Security Top 10 elenca i rischi più critici specifici delle API, diversi dei quali non coincidono con quelli delle web application in generale.

8.1 Broken Object Level Authorization (BOLA)

BOLA, detta anche IDOR (Insecure Direct Object Reference), è la vulnerabilità più comune delle API. Si verifica quando un endpoint riceve dal client l'identificativo di una risorsa e la restituisce senza controllare che l'utente possa vederla. `GET /api/orders/12345` deve verificare che l'utente autenticato sia il proprietario dell'ordine 12345, non solo che sia autenticato.

8.2 Broken Function Level Authorization

Si verifica quando un'API espone funzioni amministrative o privilegiate senza limitarle a dovere. `DELETE /api/users/{id}` dovrebbe essere raggiungibile solo dagli amministratori, ma se il controllo manca o si basa su uno schema prevedibile (per esempio «la richiesta arriva dall'interfaccia di admin») invece che sui claim di ruolo del token, qualsiasi utente autenticato potrebbe chiamarla.

8.3 Mass assignment

Molti framework mappano in automatico le proprietà della richiesta sugli attributi del modello. Se l'API assegna ogni proprietà inviata senza una lista consentita, un attaccante può mandarne di inattese, come `is_admin: true` o `account_balance: 999999`, e vedersele applicare al proprio account.

8.4 Esposizione eccessiva di dati

Le API spesso restituiscono più di quanto richiesto e contano sul client per filtrare ciò che mostra. Così trapelano dati che non dovrebbero mai lasciare il server: identificativi interni, dati personali, campi sensibili per il business. L'API dovrebbe restituire solo ciò che serve al client, applicandolo lato server.

> **IL DIVARIO DELL'AUTORIZZAZIONE**
>
> L'autenticazione conferma l'identità; l'autorizzazione stabilisce cosa quell'identità può fare. La maggior parte dei fallimenti di sicurezza delle API è un fallimento di autorizzazione, non di autenticazione. Un attaccante con un token valido che può leggere i dati di altri utenti, chiamare endpoint privilegiati o modificare campi che non dovrebbe toccare ha sfruttato un buco di autorizzazione, e in produzione sono molto più frequenti dell'autenticazione rotta.

9 API gateway e gestione

Su larga scala, gestire molte API su molti servizi richiede infrastruttura oltre le singole implementazioni. Un **API gateway** si pone tra client e servizi di backend come punto d'ingresso di tutto il traffico API e si occupa di aspetti trasversali che altrimenti andrebbero reimplementati in ogni servizio:

- **Autenticazione e autorizzazione**: valida i token prima di inoltrare le richieste
- **Rate limiting**: applica le politiche d'uso in un unico punto
- **Instradamento**: manda ogni richiesta al servizio di backend giusto
- **Load balancing**: distribuisce il traffico tra le istanze
- **Terminazione TLS**: gestisce il TLS sul gateway e parla coi servizi internamente
- **Trasformazione**: traduce formati, aggiunge o rimuove header
- **Osservabilità**: log, metriche e tracce centralizzati per tutto il traffico API
- **Cache**: riduce il carico sui backend

Tra i prodotti comuni ci sono Kong, AWS API Gateway, Google Cloud Apigee, Azure API Management e soluzioni open source come Traefik e NGINX. In Kubernetes l'Ingress controller e il service mesh (Istio, Linkerd) svolgono funzioni affini per il traffico interno tra servizi.

10 Il ciclo di vita di un'API

Le API sono manufatti di lunga durata e il loro ciclo di vita va ben oltre il primo rilascio.

10.1 Design-first contro code-first

L'approccio **design-first** definisce la specifica (OpenAPI, protobuf) prima di qualsiasi implementazione. La specifica viene rivista e concordata da chi produce e chi consuma l'API (team frontend e mobile, partner esterni) prima che esista una riga di codice server. Intercetta i problemi

di progetto quando costa poco correggerli e permette di lavorare in parallelo, perché dalla specifica si possono generare server mock.

L'approccio **code-first** implementa l'API e genera la documentazione dal codice. È più veloce da avviare ma tende a produrre API che riflettono i dettagli implementativi invece dei bisogni dei consumatori, e rende più difficile accorgersi delle modifiche incompatibili perché specifica e implementazione sono legate.

10.2 Deprecazione e dismissione

Prima o poi le vecchie versioni vanno ritirate. La prassi è un **periodo di deprecazione** in cui la vecchia versione continua a funzionare mentre i consumatori vengono avvisati con documentazione, email e header di risposta: l'header `Deprecation` e l'header `Sunset` (RFC 8594), che riporta la data di spegnimento, per esempio `Sunset: Thu, 31 Dec 2026 23:59:59 GMT`.

La durata del periodo va commisurata all'uso dell'API e alla fatica della migrazione. Un'API pubblica usata da migliaia di sviluppatori esterni può richiedere 12–24 mesi di preavviso; quelle interne possono muoversi più in fretta.

> **IN SINTESI**
>
> Le API sono prodotti, contratti e infrastruttura. Le scelte di progetto si ripercuotono su ogni consumatore, ogni integrazione, ogni esperienza di sviluppo. Una buona API è coerente, sicura, ben documentata, versionata con criterio e costruita attorno a ciò che serve a chi la usa e non a ciò che è comodo da implementare. Niente di tutto questo è casuale: nasce dal trattare il design delle API come una disciplina a sé.

11 Conclusione

Le API sono il tessuto connettivo del software moderno: il modo in cui i sistemi si parlano, in cui le organizzazioni espongono ciò che sanno fare e in cui gli sviluppatori costruiscono prodotti che da soli non potrebbero costruire. Capirle a fondo, non solo in superficie, è una delle competenze tecniche più utili a chiunque lavori con il software.

Il panorama di stili, protocolli e pattern è ampio, ma i principi sottostanti sono coerenti: definire contratti chiari, farli rispettare in modo uniforme, progettare per chi consuma e non per chi implementa, proteggere gli accessi con rigore e cambiare le cose con cautela. Un'API che li incarna è un'infrastruttura su cui altri possono costruire con fiducia, e quella fiducia è in fondo lo scopo delle API.

Questa guida riflette lo stato della progettazione e delle tecnologie per le API a inizio 2026. Strumenti e versioni dei protocolli evolvono; i principi architetturali restano stabili.

COME CITARE QUESTO ARTICOLO

Francesco Barbato (2026). Le API: un'introduzione tecnica completa, tutto quello che serve sapere. localhost:3000.

Versione online: http://localhost:3000/it/blog/api-guida-tecnica-completa