day_1

Generated: 2025-12-08 19:10:50

Cos'è un'API? (definizione moderna)

Cos'è un'API?

Un'API è un **contratto** che definisce come un software può interagire con un altro software.

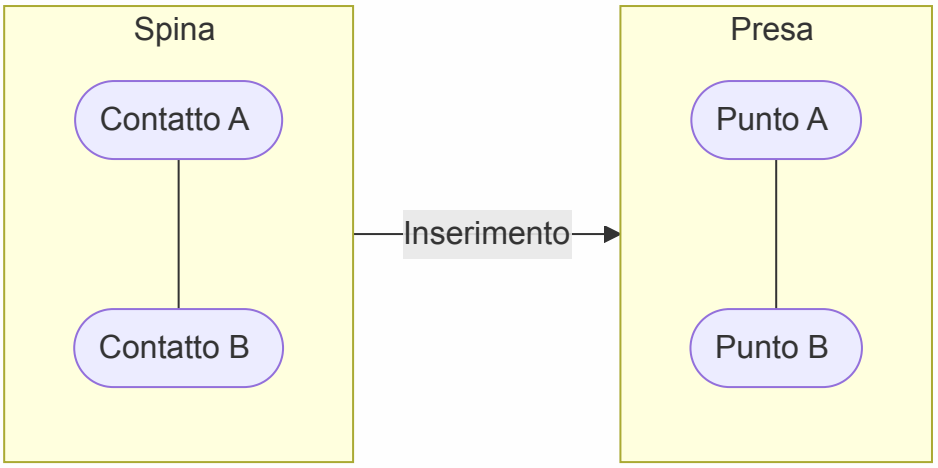
Caratteristiche fondamentali:

- Espone **funzionalità** in modo controllato
- Ha **regole chiare**: input, output, errori
- Permette **integrazione** tra sistemi diversi
- Nasconde i dettagli interni dell'implementazione

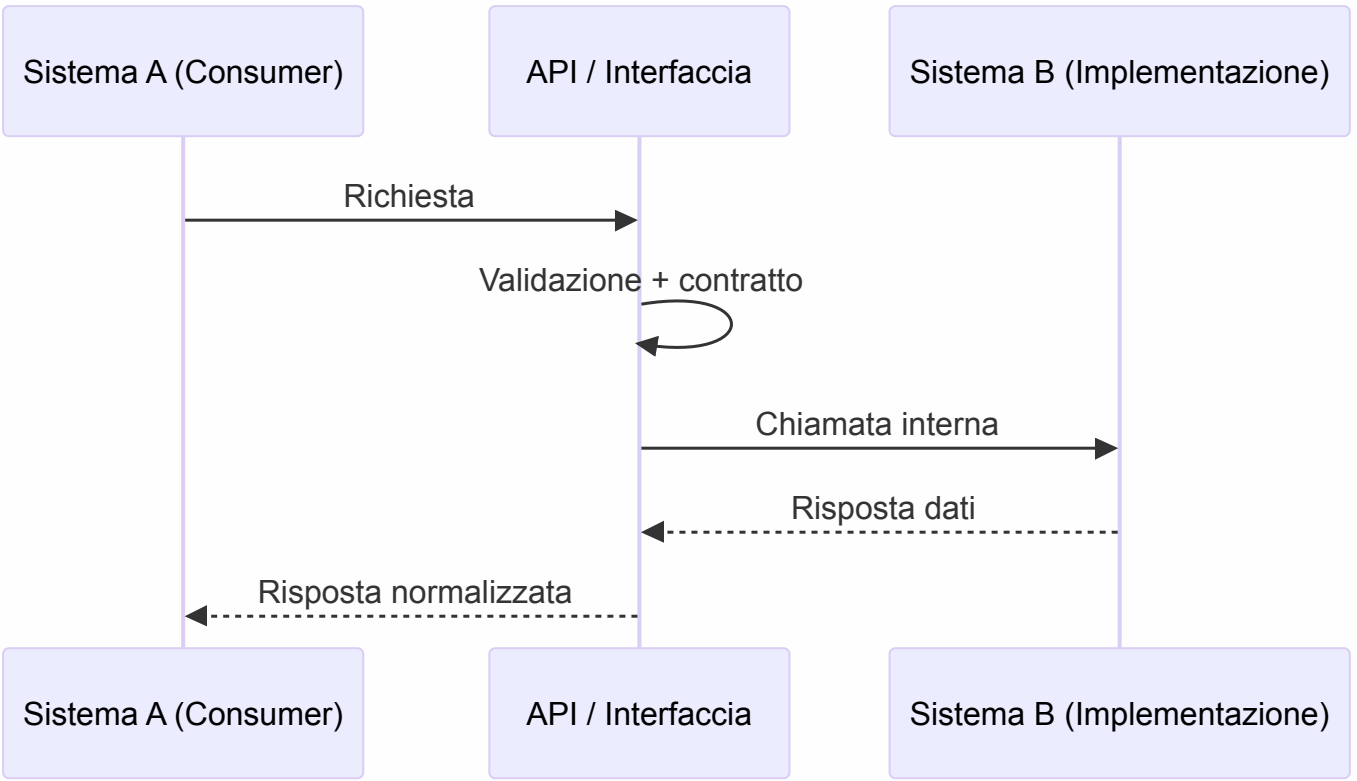
📘 *Example: Analogia ristorante*

```
Sistema = Ristorante
API = Menu
Consumer = Cliente
Implementazione = Cucina
Protocollo = Cameriere / Ordine
```

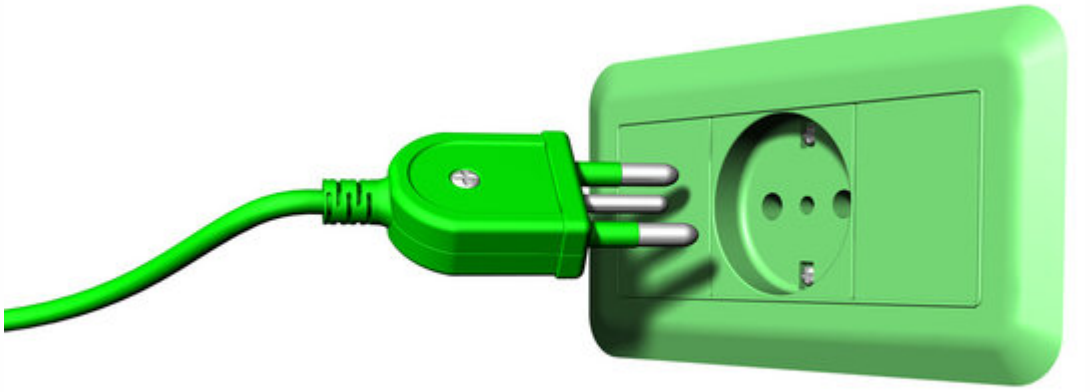
📘 Example: Metafora spina/presa (flowchart)



📘 Example: Interfaccia logica tra sistemi (sequence)



📘 *Example: Metafora visiva (immagine)*



Perché esistono le API?

Perché esistono le API?

- Per **condividere funzionalità** senza condividere codice
- Per **automatizzare** flussi tra sistemi
- Per rendere un software **estensibile**
- Per permettere ad altri di costruire sopra la nostra piattaforma
- Per garantire **controllo, stabilità e evoluzione** nel tempo

📘 *Example: Esempio API GitHub – Recupero repository*

Recuperare informazioni su un repository GitHub

```
curl -s https://api.github.com/repos/spring-projects/spring-boot
```

Risposta (estratto):

```
{
  "full_name": "spring-projects/spring-boot",
  "stargazers_count": 73000,
  "open_issues": 600,
  "language": "Java"
}
```

📌 **Messaggio chiave:** senza accedere al codice di GitHub, puoi integrare informazioni reali del progetto.

📘 *Example: Esempio API NASA – Astronomy Picture of the Day*

Recuperare la foto astronomica del giorno (APOD)

```
curl -s "https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY"
```

Risposta (estratto):

```
{
  "date": "2025-01-14",
  "title": "The Heart Nebula",
  "url": "https://apod.nasa.gov/apod/image/...jpg",
  "explanation": "Una nuova immagine della Nebulosa Cuore..."
}
```

📌 **Messaggio chiave:** un'API permette di ottenere contenuti aggiornati e integrabili in qualsiasi app senza scrivere logiche complesse.

📘 *Example: Esempio API SOAP – Servizio meteo (SOAP 1.1)*

Chiamata SOAP a un servizio meteo

```
curl -X POST https://www.example.com/weatherService \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCityWeather xmlns="http://weather.example.com/">
      <CityName>Rome</CityName>
    </GetCityWeather>
  </soap:Body>
</soap:Envelope>'
```

Risposta SOAP (estratto):

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCityWeatherResponse>
      <GetCityWeatherResult>
        <Temperature>18</Temperature>
        <Condition>Sunny</Condition>
      </GetCityWeatherResult>
    </GetCityWeatherResponse>
  </soap:Body>
</soap:Envelope>
```

📌 **Messaggio chiave:** SOAP è un *protocollo contrattuale* molto rigido e formale: perfetto per contesti enterprise.

API ≠ Endpoint

API ≠ Endpoint

L'API *non è il suo trasporto*, né i suoi URL.

- Un'API può essere:
  - una **Web API REST**
  - una **libreria Java**
  - un **SDK**

    - una **API SOAP**
    - una **API gRPC**
- L'API è il **contratto**; gli endpoint sono **una delle sue incarnazioni**.

📘 *Example: Mini contratto SOAP (WSDL)*

```xml
<definitions name="WeatherService"
             targetNamespace="http://weather.example.com/wsdl"
             xmlns:tns="http://weather.example.com/wsdl"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <message name="GetCityWeatherRequest">
    <part name="cityName" type="xsd:string"/>
  </message>

  <message name="GetCityWeatherResponse">
    <part name="temperature" type="xsd:int"/>
    <part name="condition" type="xsd:string"/>
  </message>

  <portType name="WeatherPortType">
    <operation name="GetCityWeather">
      <input message="tns:GetCityWeatherRequest"/>
      <output message="tns:GetCityWeatherResponse"/>
    </operation>
  </portType>

  <binding name="WeatherBinding" type="tns:WeatherPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  </binding>
</definitions>
```

📘 *Example: Mini contratto REST*

```yaml
paths:
  /orders/{id}:
    get:
      summary: Get order details
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Order found
```

```
        '404':
          description: Order not found
```

Esempio di API nella vita quotidiana

Un esempio di API quotidiana (pubblica)

Pensiamo a un telecomando:

- Ha **tasti** = operazioni disponibili
- Ha **regole** = cosa fa ogni tasto
- Non conosciamo il firmware interno della TV
- L'interazione è **stabile**, anche cambiando televisore

API = telecomando per i software.

Storia delle API: le origini

Storia delle API (1/3): le origini

- Anni '70–'80: API = **funzioni esportate** da librerie di sistema
- Linguaggi come C definiscono le API tramite **header file (.h)**
- Il concetto chiave era: **contratto binario / contrattualizzazione dell'accesso**
- Software ≠ black box → API come "porta d'ingresso controllata"

📘 Example: Confronto SOAP vs REST

```
<!-- SOAP Request semplificata -->
<soap:Envelope>
  <soap:Body>
    <GetCustomer>
      <customerId>123</customerId>
    </GetCustomer>
  </soap:Body>
</soap:Envelope>
```

```
GET /customers/123 HTTP/1.1
Host: api.example.com
Accept: application/json
```

Storia delle API: distribuzione

Storia delle API (2/3): API distribuite

- Anni ’90: crescita dei sistemi distribuiti
- Nascita di RPC: **Remote Procedure Call**
- Segue SOAP: **Simple Object Access Protocol**
- I contratti diventano **WSDL**, XML Schema
- Integrazione tra aziende → API come ponti tra piattaforme

Storia delle API: il web

Storia delle API (3/3): l’era del Web

- Anni 2000: l’esplosione del **Web 2.0**
- Nascono le Web API HTTP
- JSON sostituisce XML

- La tesi di Roy Fielding → **REST** diventa il modello dominante
- API come elemento centrale della trasformazione cloud

Tipi di API

Tipi di API

- **Web API**
  - HTTP/HTTPS, JSON, REST/gRPC/GraphQL
- **API di sistema**
  - Interfacce del sistema operativo (POSIX, Win32)
- **Librerie / SDK**
  - Funzioni e classi esposte come strumenti di sviluppo
- **API interne vs esterne**
  - Internal → team-to-team
  - External → partner/clienti

✏️ *Exercise: Mappa dominio → API ad alto livello*

```
🧪 **Esercizio breve (10 min)**
Scegliete un dominio (blog, e-commerce, banking, IoT) e provate ad elencare:
- 3 risorse principali (es. orders, products, customers)
- 2 operazioni per ciascuna (es. create order, get order)

Non serve dettaglio tecnico, basta la lista.
```

▶ 💡 Mostra una possibile soluzione

Possibile soluzione – Dominio E-commerce

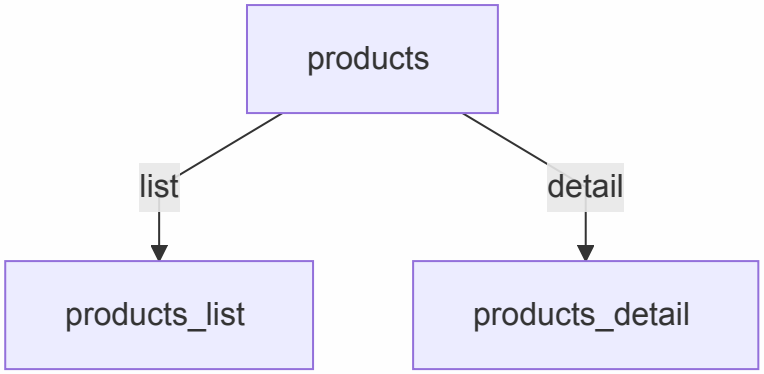
Risorse

- products
- orders
- customers

Operazioni

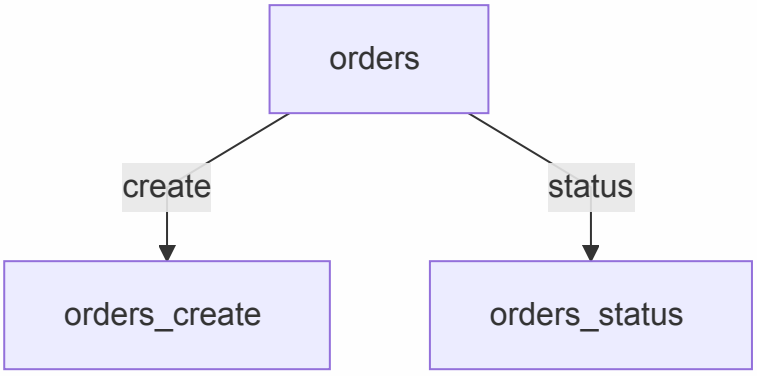
products

- GET /products — lista prodotti
- GET /products/{id} — dettaglio prodotto



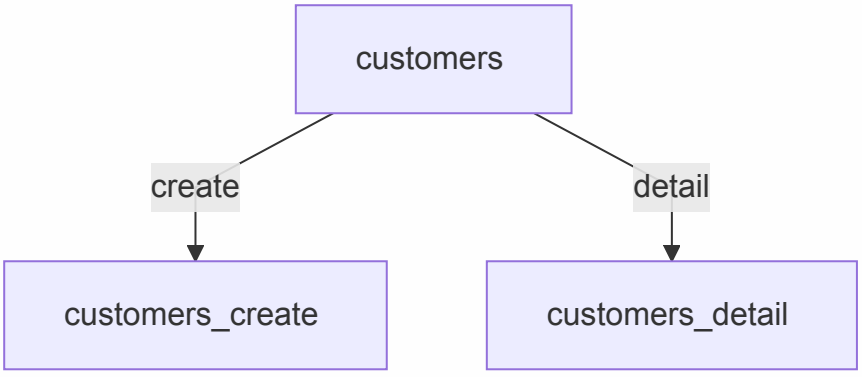
orders

- POST /orders — crea un ordine
- GET /orders/{id} — stato dell'ordine



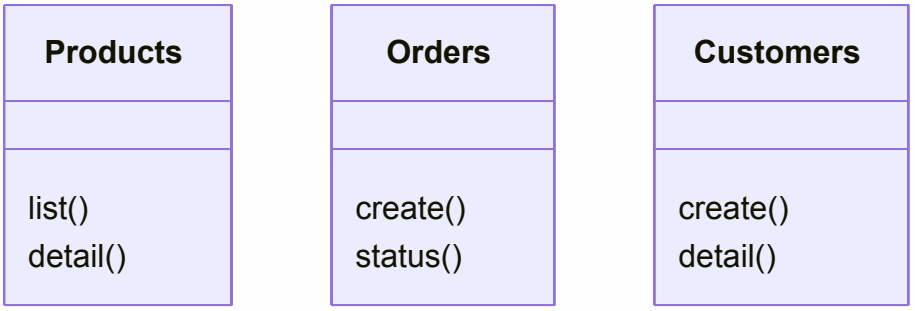
customers

- POST /customers — crea nuovo cliente
- GET /customers/{id} — profilo cliente

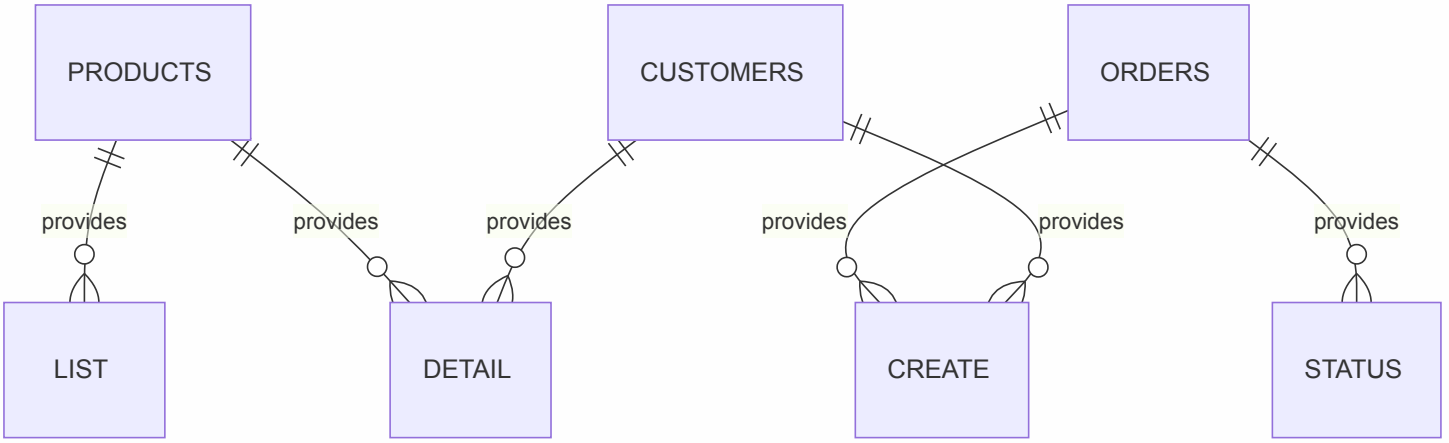


---

class diagram (molto base)



ERD (non perfetto ma utile)




```
</details>


---

## Web API

# Web API

Una Web API espone funzionalità via **HTTP**, tipicamente con:

- URL per identificare risorse o azioni
- Metodi HTTP (GET, POST, …)
- Payload JSON come formato più recente e più usato, XML ancora presente in contesti enterprise
- Codici di stato
- Sicurezza (API Key, OAuth2, ecc.)

---

## API di sistema

# API di sistema
```

````
Esempi:

- API POSIX
- API Win32
- Java NIO

Caratteristiche:
- alta stabilità
- forte accoppiamento al sistema operativo
- performance elevate
- non sono “API Web”: sono **API locali**

### ✏️ Exercise: Individuare incoerenze in una API

🧪 **Esercizio – individua incoerenze**

Osserva il seguente esempio di API per la gestione ordini e annota almeno **3 incoerenze** (naming, status code, struttura dei dati):

```http
GET /getOrders
200 OK
[
  {
    "order_id": "123",
    "CustomerName": "Mario",
    "total_amount": 99.9
  }
]

POST /orders/create
200 OK
{
  "id": 124,
  "customer_name": "Luigi",
  "totalAmount": 149.90,
  "status": "CREATED_OK"
}
````

Prova a proporre una versione più coerente (URL, metodi, campi, status code).

▶ 💡 Possibile soluzione

Una possibile versione più coerente potrebbe essere:

```http
GET /orders
200 OK
[
  {
    "id": "123",
    "customerName": "Mario",
    "totalAmount": 99.90,
    "status": "CREATED"
  }
]
```
```

```
POST /orders
201 Created
Location: /orders/124
{
  "id": "124",
  "customerName": "Luigi",
  "totalAmount": 149.90,
  "status": "CREATED"
}
```

Cambiamenti principali:

- URL risorsa al plurale (`/orders`) e niente verbi nei path (`/getOrders`, `/orders/create` → `/orders`)
- campi JSON in `camelCase` e nomi coerenti tra richiesta e risposta (`customerName`, `totalAmount`, `status`)
- uso di `201 Created` per la creazione e header `Location` con il nuovo resource URI
- status applicativo normalizzato (`CREATED` invece di `CREATED_OK`)

SDK e librerie

SDK e librerie

Un SDK è un set di API + strumenti:

- librerie client (Java, TS, Python)
- strumenti di test
- script di automazione
- documentazione

Le API sono l'**interfaccia**; l'SDK è l'**esperienza completa**.

Il ruolo delle API nel software moderno

Il ruolo delle API oggi

- Collegano microservizi
- Permettono integrazioni SaaS (Software as a service)
- Aprono ecosistemi (API economy)
- Supportano mobile, IoT, automazione
- Abilitano modelli di business basati su consumatori esterni

---

Casi d'uso tipici

Casi d’uso delle API

- Pagamenti (Stripe, PayPal)
- Mappe (Google Maps API)
- Notifiche (FCM, APNs)
- Analisi (Mixpanel, Segment)
- Integrazione sistemi enterprise
- Microservizi interni

Cosa rende un'API di successo?

- **Semplice da capire**
- **Predictable**
- **Ben documentata**
- **Coerente**
- **Stabile nel tempo** (backward compatibility prima di tutto)
- **Sicura**

Principi fondamentali di progettazione API

- Contratto chiaro
- Superfici piccole ma espressive
- Coerenza interna
- Evolvibilità
- Bassa sorpresa
- Errori chiari e documentati
- Sicurezza integrata

Consistenza: perché è fondamentale

- Riduce il carico cognitivo
- Permette agli sviluppatori di “indovinare” l’uso
- Rende l’API più robusta a cambiamenti
- Facilita il versioning
- Aumenta l’adozione

Idempotenza nelle API

Un'operazione è **idempotente** quando ripeterla più volte produce **lo stesso effetto** della singola esecuzione.

Definizione formale

> f(f(f(x))) = f(x)

Cioè: applicare più volte la stessa operazione **non cambia ulteriormente lo stato**.

Nei metodi HTTP

- **GET** → idempotente (lettura, non modifica lo stato)
- **PUT** → idempotente (sovrascrive la risorsa → risultato identico)
- **DELETE** → idempotente (la risorsa scompare, e continua a non esistere)
- **POST** → non idempotente (crea effetti cumulativi)

Perché importa?

Permette di progettare API più robuste in caso di:

- retry automatici del client
- timeout
- rete instabile
- concorrenza

📘 *Example: Esempi HTTP*

```
# PUT è idempotente
PUT /users/10
{ "active": true }

# Ripetuto N volte → lo stato finale è uguale
PUT /users/10
{ "active": true }
```

```
# DELETE è idempotente
DELETE /orders/123
# La risorsa scompare

DELETE /orders/123
# Rimane scomparsa (anche se 404)
```

📘 *Example: Parallelo SQL*

```
-- Idempotente
UPDATE users SET active = TRUE WHERE id = 1;
-- Ripetuto 10 volte → stesso risultato

-- NON idempotente
UPDATE users SET balance = balance + 10 WHERE id = 1;
-- Ogni esecuzione modifica lo stato

-- Idempotente
DELETE FROM orders WHERE id = 123;
-- Ripetuto = nessun cambiamento
```

✏️ *Exercise: Riconoscere operazioni idempotenti*

🧪 **Esercizio breve**
Indica quali delle seguenti operazioni sono idempotenti:

1. `PUT /devices/12 { "mode": "AUTO" }`
2. `POST /logs { "event": "login" }`
3. `DELETE /sessions/99`
4. `UPDATE accounts SET score = score + 1`

▶ 💡 Soluzione

1. ✔ Idempotente (PUT sovrascrive)
2. ❌ Non idempotente (POST crea nuovi log)
3. ✔ Idempotente (DELETE rende lo stato finale stabile)
4. ❌ Non idempotente (incremento cumulativo)

Prevedibilità (Predictability)

Prevedibilità

Un'API prevedibile:

- si usa "senza aprire la doc ogni 2 minuti"
- ha risorse e operazioni intuitive
- mantiene pattern riconoscibili
- evita eccezioni non intuitive

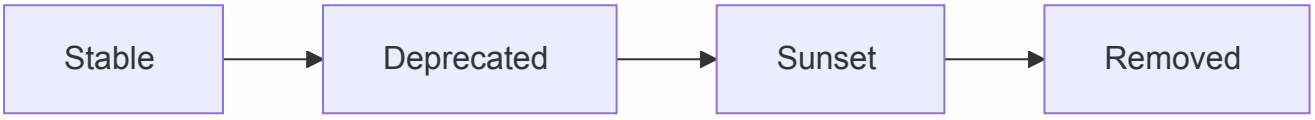
Semplicità vs Completezza

- Un'API semplice ≠ un'API povera
- Ridurre l'API surface senza sacrificare potenza
- Separare casi comuni da casi avanzati
- Offrire estensioni opzionali, non obbligatorie

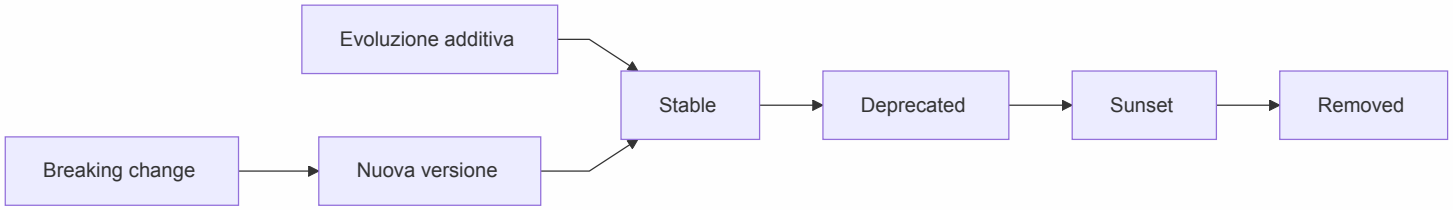
Evoluzione e versionamento

- Le API **devono evolvere** nel tempo
- Evitare breaking change il più possibile
- Preferire evoluzione **additiva**:
  - nuovi campi (opzionali)
  - nuovi endpoint
- Versioning come **ultima risorsa**
- Definire un ciclo di vita chiaro:
  stable → deprecated → sunset → removed
- Comunicare sempre i cambiamenti ai consumer

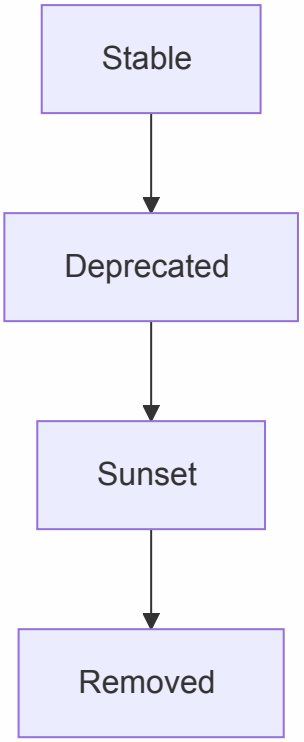
📘 *Example: Lifecycle API – versione semplice*



📘 *Example: Lifecycle API – versione completa (evoluzione + versioning)*



📘 *Example: Lifecycle API – versione slide-friendly*



Sicurezza (introduzione)

Sicurezza nelle API (introduzione)

La sicurezza non è un'aggiunta: è parte del design dell'API.

Elementi fondamentali:

- **Autenticazione** → chi sei?
- **Autorizzazione** → cosa puoi fare?
- **Confidenzialità** → proteggere i dati in transito (HTTPS)
- **Integrità** → evitare manomissioni dei dati
- **Rate limiting** → proteggere da abusi e sovraccarichi
- **Errori sicuri** → niente informazioni sensibili negli errori
- **Minimo privilegio** → dare solo ciò che serve

Analisi di API esistenti

Analisi di API esistenti (esercizio)

Osserviamo insieme alcune API pubbliche:

- **GitHub REST API**
- **Stripe Payments API**
- **Spotify Web API**

Cosa analizziamo?

- Coerenza del design
- Prevedibilità degli endpoint
- Qualità della documentazione
- Semplicità vs complessità
- Modellazione centrata sul *consumer*, non sul DB
- ❗ La sicurezza **non** si aggiunge nel payload: è un layer separato (header, token, HTTPS)

Obiettivo: imparare a riconoscere cosa rende un'API ben pensata.

📘 *Example: Esempio: endpoint semplice e consistente (GitHub)*

```
GET https://api.github.com/users/octocat

200 OK
{
  "login": "octocat",
  "id": 1,
  "public_repos": 8,
  "followers": 3933
}
```

Coerente, prevedibile, con payload chiaro.

📘 *Example: Esempio: payload consumer-centric (Stripe)*

```
POST /v1/payment_intents
{
  "amount": 2000,
  "currency": "eur"
}

200 OK
{
  "id": "pi_3ABC...",
  "amount": 2000,
  "status": "requires_payment_method"
}
```

Notare:

- payload piccolo
- niente dettagli interni
- naming semplice e stabile

✏️ *Exercise: Analizzare una API esistente*

🧪 **Esercizio**: scegli una delle API elencate (GitHub, Stripe, Spotify) e rispondi:

1. Quali aspetti del design ti sembrano ben fatti?
2. Dove vedi incoerenze o comportamenti poco prevedibili?
3. Il payload è pensato per il consumer o per il dominio interno?
4. Qual è una modifica semplice che migliorerebbe l'API?

❗ Nota: **non** proporre campi di sicurezza nel payload (es. `token`, `role`, `auth`): la sicurezza vive in un layer separato (header, protocolli, autorizzazione lato server).

▶ 💡 Possibile traccia di soluzione

GitHub (possibile risposta)

- **Coerenza:** naming uniforme e struttura stabile.
- **Prevedibilità:** molto alta; gli URL comunicano chiaramente l'intenzione.
- **Documentazione:** ottima, con esempi sempre aggiornati.
- **Criticità:** alcuni endpoint storici non sono del tutto uniformi.

Stripe (possibile risposta)

- **Eccellente DX:** esempi brevi e sempre funzionanti.
- **Payload minimal:** pensato per essere facile da usare da mobile/backend.
- **Prevedibilità:** mol to alta.
- **Criticità:** alcuni concetti avanzati richiedono lettura approfondita.

Spotify (possibile risposta)

- **Forte orientamento consumer:** ideale per app mobile.
- **Coerenza buona:** naming chiaro.
- **Criticità:** rate limiting molto aggressivo, alcuni endpoint sono molto annidati.

❌ Non si aggiungono campi di sicurezza nel payload: la sicurezza si gestisce tramite header (es. `Authorization`), HTTPS e controlli di autorizzazione lato server, non dentro l'oggetto JSON restituito dall'API.

Best Practices (overview)

Best Practices (overview)

- Naming uniforme
- Minimizzare gli endpoint
- Minimizzare gli errori 500
- Documentare sempre
- Non sorprendere l'utente dell'API
- Non rompere la compatibilità

Stili architetturali per API

Stili architetturali per API

- REST (architettura web)
- SOAP (contratto rigido XML)
- RPC (veloce ma fragile)
- GraphQL (query-based)
- gRPC (binary, veloce)

Nota: il naming nelle API dovrebbe riflettere il dominio reale → *Ubiquitous Language*

📘 *Example: 2 esempi*

```
# ❌ Sbagliato (DB-driven)
GET /order_headers
GET /order_rows

# ✔ Corretto (Ubiquitous Language)
GET /orders
GET /orders/{id}/items
```

✏️ *Exercise: Rinominare secondo Ubiquitous Language*

Esercizio: hai questi endpoint, modellati sul database interno:

```
GET /tbl_ord_hdr
GET /tbl_ord_row
```

Rinominali usando un linguaggio del dominio comprensibile ai consumer.

Non usare nomi tecnici o derivati dal database.

▶ Possibile soluzione

```
GET /orders
GET /orders/{id}/items
```

Motivazione:

- "orders" è il termine del dominio usato dal business
- "items" descrive cosa rappresentano le righe dell'ordine

Transizione: perché REST domina

Perché REST domina oggi?

- Usa principi del Web (Internet+HTTP)
- Scalabile e robusto
- Cache-friendly
- Stateless
- Ottimo per team distribuiti
- Supportato ovunque (browser, mobile, cloud)

Roy Fielding e la sua tesi

Roy Fielding e la sua tesi

Anno 2000: Roy Fielding pubblica la sua tesi sulla **architectural style of the Web**.

Da essa nasce:

- il termine “REST”
- i 6 vincoli dell’architettura REST
- il modello “resource”
- la separazione client/server
- l’uso uniforme di HTTP

REST: Non è un protocollo

REST: non è un protocollo

REST **non è**:

- un formato (JSON)
- un protocollo (HTTP)
- una libreria

REST è un **modello architetturale** basato su vincoli.

REST: i 6 vincoli

I 6 vincoli REST

1. Client–Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-on-Demand (opzionale)

REST: Client–Server

Vincolo 1: Client–Server

Il client e il server hanno **responsabilità diverse**.
L'API è l'interfaccia che separa ciò che il client *vuole fare* da ciò che il server *sa fare*.

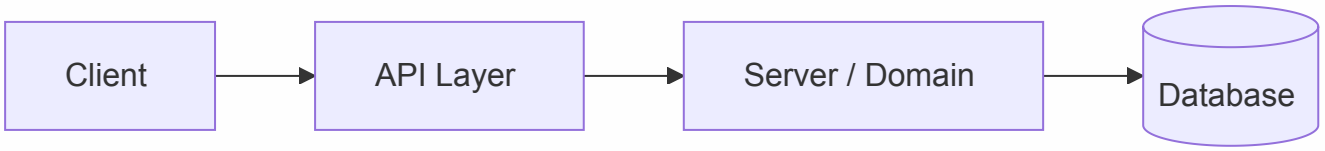
- **Client** → UI, interazione, esperienza utente, orchestrazione
- **Server** → logica di dominio, storage, regole di business, API

Vantaggi della separazione:

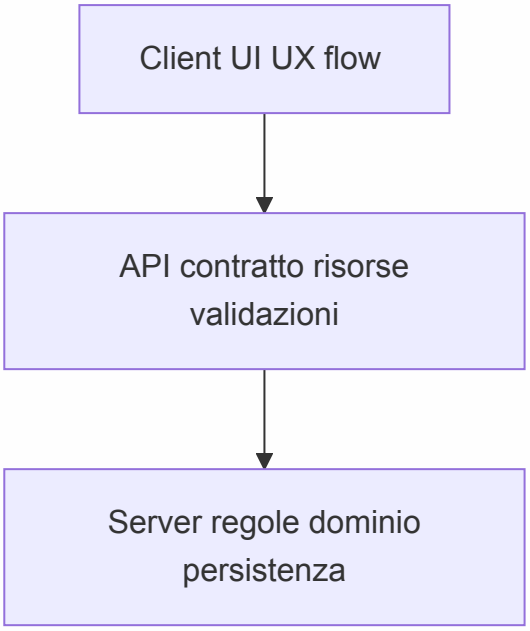
- scalabilità
- evolvibilità indipendente
- portabilità su più client (web, mobile, partner)

📌 Il client non accede mai al database: parla solo con l'API.

📘 *Example: Schema: separazione Client–Server*



📘 *Example: Schema: responsabilità separate*



✏️ *Exercise: Esercizio: capire il ruolo dell'API*

🧪 **Esercizio**
Hai un dominio complesso con molte tabelle e logiche interne.

👉 **Domanda**: quali informazioni *minimali* deve vedere un client per poter lavorare?

Scrivi:

1. cosa vede il client
2. cosa NON deve vedere
3. in che forma l'API dovrebbe presentare i dati

▶ 💡 Possibile soluzione

REST: Stateless

Vincolo 2: Stateless

Il server **non mantiene stato** tra una richiesta e l’altra.

Ogni richiesta deve essere completa di:

- autenticazione
- contesto
- parametri necessari

Vantaggi:

- scalabilità orizzontale
- caching server-side
- semplificazione del backend

REST: Cache

Vincolo 3: Cache

REST incoraggia:

- caching lato client
- caching lato proxy
- caching lato CDN

Vantaggi:

- riduzione carico backend
- maggiore velocità percepita
- minor costo infrastrutturale

🔹 **Nota didattica:** quando si parla di risorse REST, occorre distinguere tra:

- relazioni espresse come **attributo** (campo JSON)
- relazioni espresse come **navigazione** (endpoint dedicato)

Questo evita payload enormi e ricorsivi.

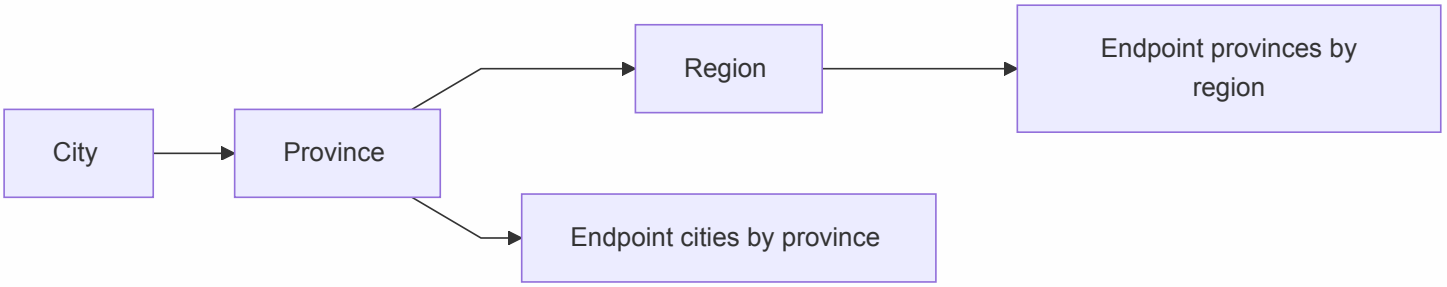
📘 *Example: Esempio — relazione di attributo*

```
{
  "id": "C123",
  "name": "Torino",
  "provinceId": "P001",
  "regionId": "R001"
}
```

📘 *Example: Esempio — relazione di navigazione*

```
GET /regions/R001/provinces
GET /provinces/P001/cities
```

📘 *Example: Diagramma — attributo vs navigazione*



✏️ *Exercise: Relazioni: attributo o navigazione?*

🧪 **Esercizio**
Considera il dominio Regione → Provincia → Comune.

Per ciascuna domanda, decidi se è una **relazione di attributo** o di **navigazione**.

1. "Dato un Comune, chi è la sua Provincia?"
2. "Dalla Provincia, mostra i suoi Comuni"

3. "Dato un Comune, elenca tutti i Comuni della stessa Regione"
4. "Dato un Comune, mostrami anche tutte le Province della Regione"

▶ 💡 Soluzione

1. **Attributo** → `provinceId`
2. **Navigazione** → `GET /provinces/{id}/cities`
3. **Navigazione** → prima trovi la regione (`regionId`), poi `GET /regions/{id}/cities`
4. ❌ Errore: non includere tutte le province nel payload del Comune.
   ✔ Navigazione corretta: `GET /regions/{id}/provinces`

REST: Uniform Interface

Vincolo 4: Uniform Interface

Il vincolo più importante di REST.

Principi chiave:

- Interfaccia **uniforme e consistente** tra tutte le risorse
- Uso sistematico dei metodi HTTP
- Risorse identificate tramite URI
- Rappresentazioni standard (es. JSON, XML)
- Messaggi autodescrittivi
- Hypermedia come motore dell'applicazione (**HATEOAS**)

Effetto: il client può evolvere senza conoscere implementazioni interne e scopre cosa può fare tramite i link forniti dal server.

📘 *Example: Esempio base di HATEOAS*

```
{
  "id": "123",
  "status": "shipped",
  "total": 89.50,
  "_links": {
    "self": { "href": "/orders/123" },
    "cancel": { "href": "/orders/123/cancel" },
    "customer": { "href": "/customers/77" }
  }
}
```

📘 *Example: Esempio HAL (Hypertext Application Language)*

```
{
  "_links": {
    "self": { "href": "/orders" },
    "next": { "href": "/orders?page=2" },
    "create": { "href": "/orders", "method": "POST" }
  },
  "_embedded": {
    "orders": [
      {
        "_links": { "self": { "href": "/orders/123" } },
        "status": "processing"
      }
    ]
  }
}
```

📘 *Example: Diagramma HATEOAS (compatibile Typora)*

✏️ *Exercise: HATEOAS: quali link aggiungeresti?*

🧪 **Esercizio**
Hai una risorsa semplice:

```
{
  "id": "A55",
  "status": "draft",
  "ownerId": "U99"
}
```

👉 **Domanda:** quali link aggiungeresti per renderla HATEOAS-friendly?
(Pensa alle azioni che un client potrebbe voler compiere.)

▶ 💡 Possibile soluzione

```
{
  "id": "A55",
  "status": "draft",
  "ownerId": "U99",
  "_links": {
    "self": { "href": "/articles/A55" },
    "publish": { "href": "/articles/A55/publish" },
    "owner": { "href": "/users/U99" },
    "delete": { "href": "/articles/A55" }
  }
}
```

Spiegazione:

- self → navigazione interna
- publish → azione di business
- owner → relazione verso un'altra risorsa
- delete → azione comune

Vincolo 5: Layered System

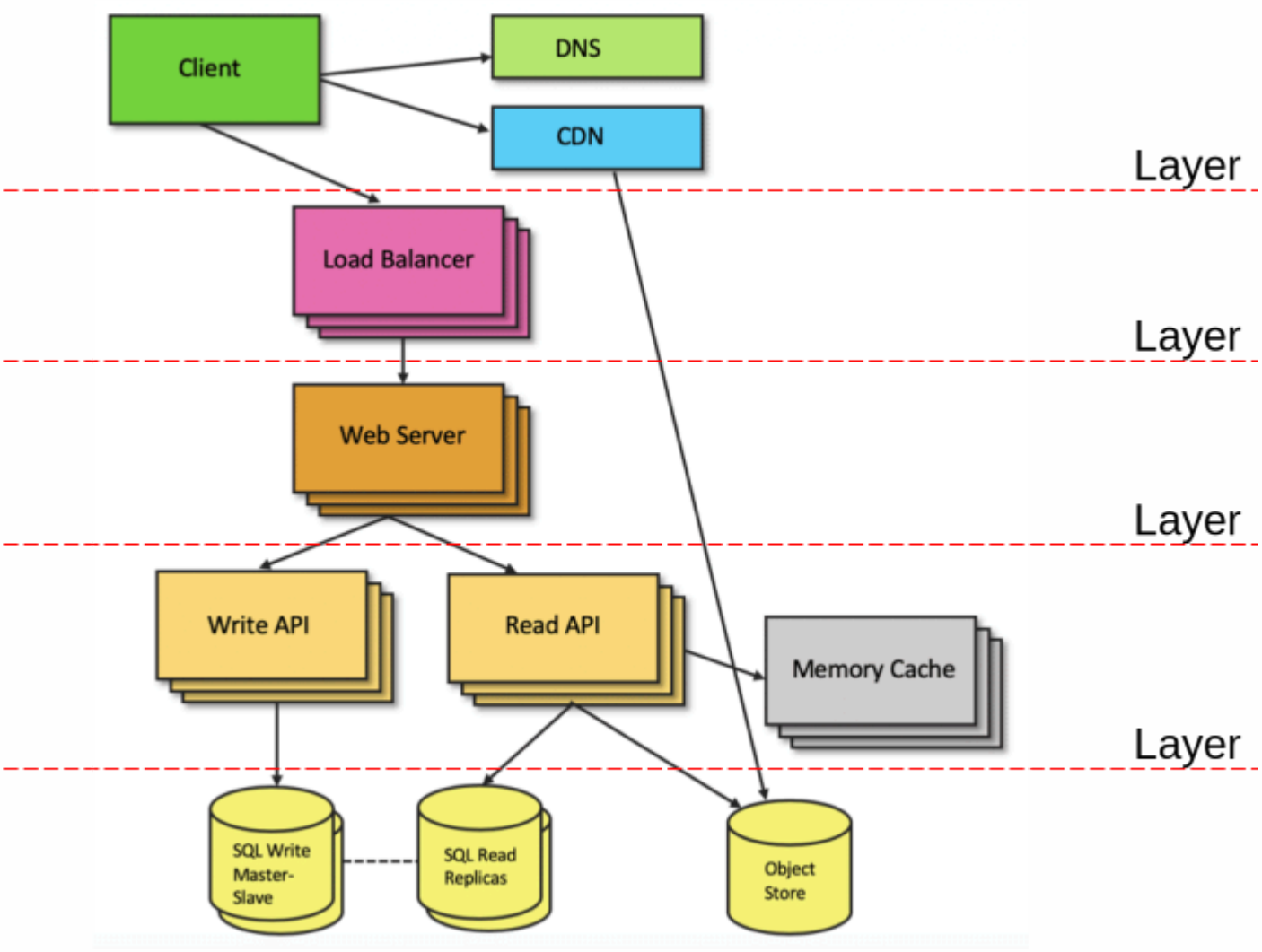
In un sistema REST le componenti sono organizzate a layer che si interpongono tra client e server.

Esempi di livelli:

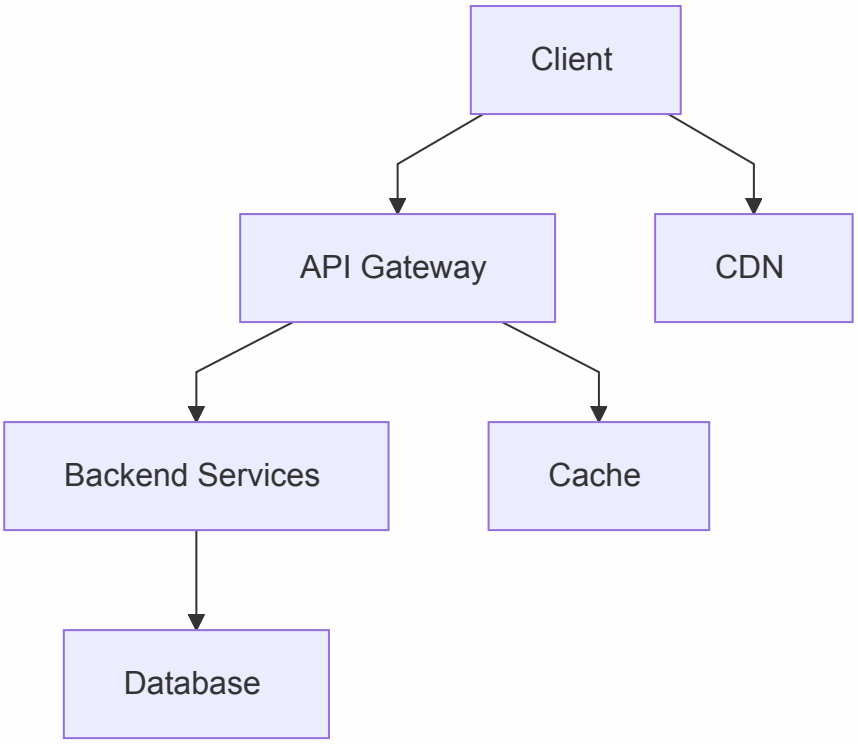
- Client (web/mobile)
- API Gateway / Reverse Proxy
- Backend Services (microservizi)
- Database / Storage
- Cache intermedie
- CDN (Content Delivery Network)

Vantaggi:

- maggiore sicurezza
- separazione delle responsabilità
- evoluzione indipendente dei layer
- possibilità di caching distribuito


Client
DNS
CDN
Layer
Load Balancer
Layer
Web Server
Layer
Write API
Read API
Memory Cache
Layer
SQL Write Master-Slave
SQL Read Replicas
Object Store

📘 *Example: Schema Layered System (Mermaid)*



📘 *Example: URI stabili vs URI che espongono dettagli interni*

```
# Sbagliato: URI accoppiato all'implementazione
GET /v1_2023_beta/users_mysql_2

# Corretto: URI stabile e astratto
GET /users
```

✏️ *Exercise: Identificare i layer in un'architettura reale*

🧪 **Esercizio breve**
Pensa a un sistema reale su cui lavori (o che conosci) e prova a individuare i layer.

Scrivi:

1. Chi è il client? (web, mobile, altro)
2. Qual è l'entrypoint pubblico? (API Gateway, reverse proxy, ingress)
3. Quali servizi backend ci sono dietro?
4. Dove risiedono i dati? (DB, file, storage)
5. Ci sono cache o CDN intermedie?

▶ 💡 Possibile traccia di soluzione

REST: Code-on-Demand (opzionale)

Vincolo 6: Code-on-Demand *(opzionale)*

Il server può fornire **frammenti di codice eseguibili dal client**
(es: JavaScript).

Oggi poco usato nelle API, ma fondamentale per comprendere la visione originale di Fielding.

Serve per:

- estendere dinamicamente le capacità del client
- ridurre accoppiamento

REST non è CRUD

REST non è CRUD

REST ≠ “4 endpoint: GET/POST/PUT/DELETE”.

REST è:

- un **modello architetturale**
- basato su 6 vincoli
- orientato alle **rappresentazioni delle risorse**
- non agli oggetti di dominio

CRUD è solo un pattern comune.

REST: Highlights pratici

REST: Punti chiave da ricordare

- Risorse, non metodi
- URI stabili, semantici
- Interfaccia uniforme
- Stateless
- Caching integrato
- Evoluzione additiva

SOAP: introduzione

SOAP — Introduzione

SOAP (Simple Object Access Protocol) è un protocollo basato su:

- XML
- Contratti rigidi
- Standard enterprise (WS-*)
- WSDL (Web Services Description Language) per descrizione del servizio

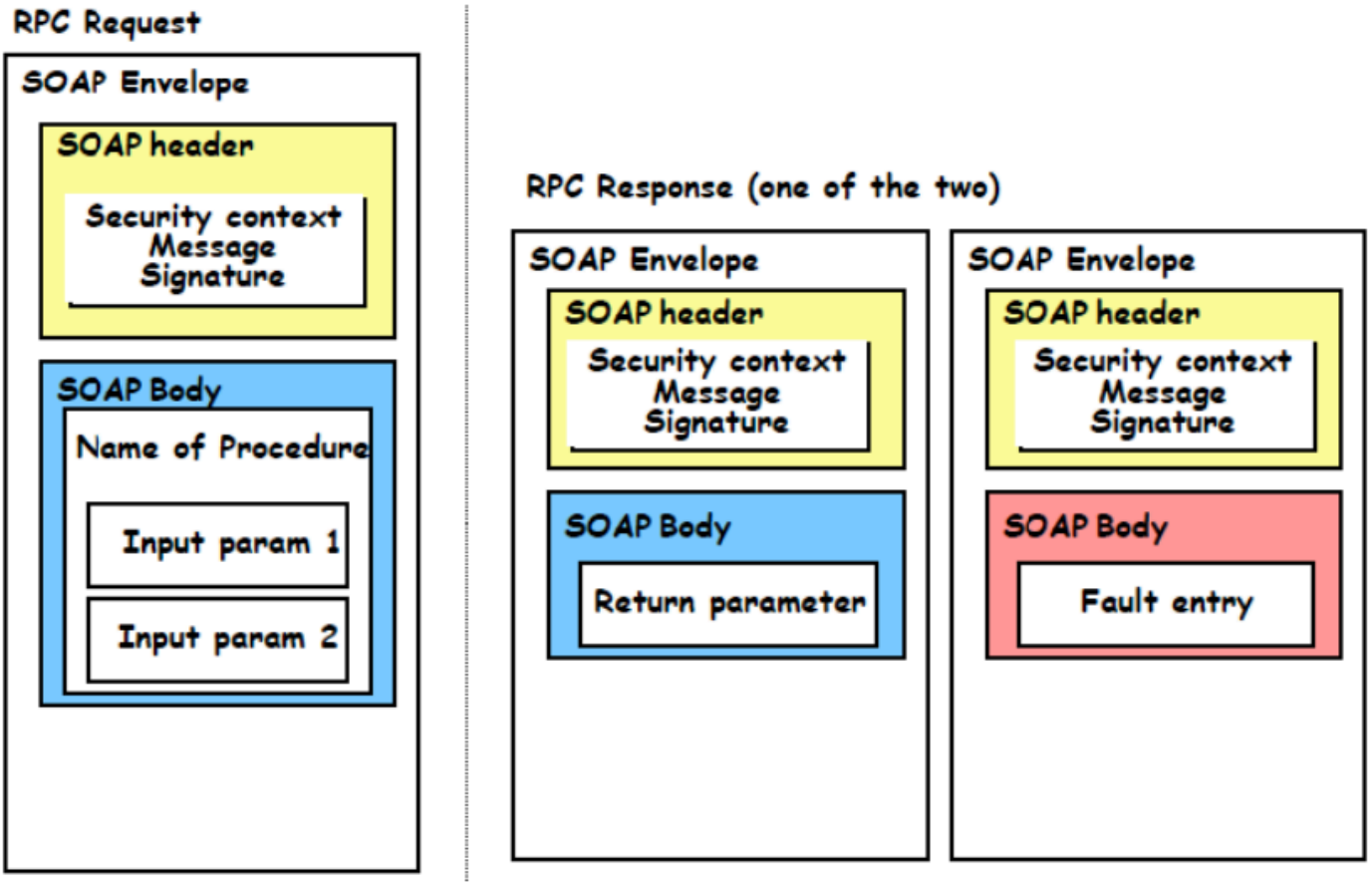
Utilizzato tipicamente in contesti enterprise, banking, legacy.

SOAP — Come funziona

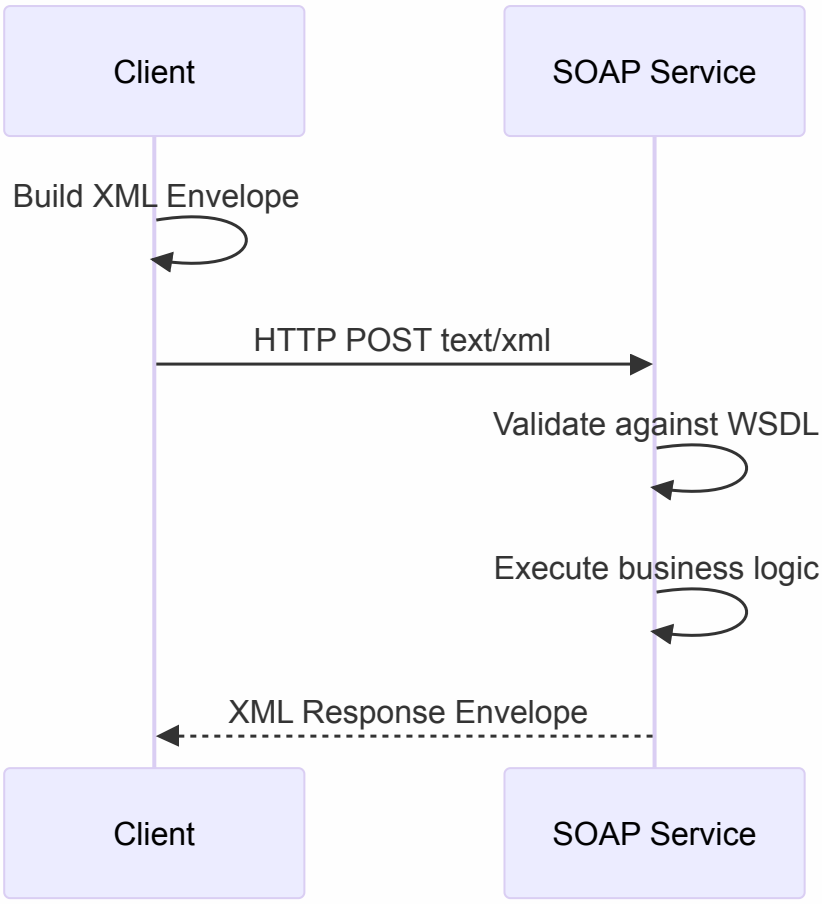
Una API SOAP tipica si basa su:

- **Envelope XML**
- **Header** (sicurezza, routing, metadati)
- **Body** (messaggi applicativi)
- **WSDL** che definisce:
    - Operazioni (portType)
    - Input/output (message)
    - Binding (SOAP/HTTP)
    - Endpoint fisici (service)

Schema visivo



📘 *Example: ⚠️ 3 examples (showing first, use JSON editor for full edit)*



✏️ *Exercise: Identificare Header e Body in SOAP*

🧪 **Esercizio**
Dato questo frammento SOAP:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <Timestamp>2025-01-01T12:00:00Z</Timestamp>
  </soap:Header>
  <soap:Body>
    <SearchRequest>
      <Query>Roma</Query>
    </SearchRequest>
  </soap:Body>
</soap:Envelope>
```

Indica:

1. cosa va interpretato come **metadato**
2. cosa è il **vero contenuto applicativo**

▶ 💡 Soluzione

SOAP vs REST

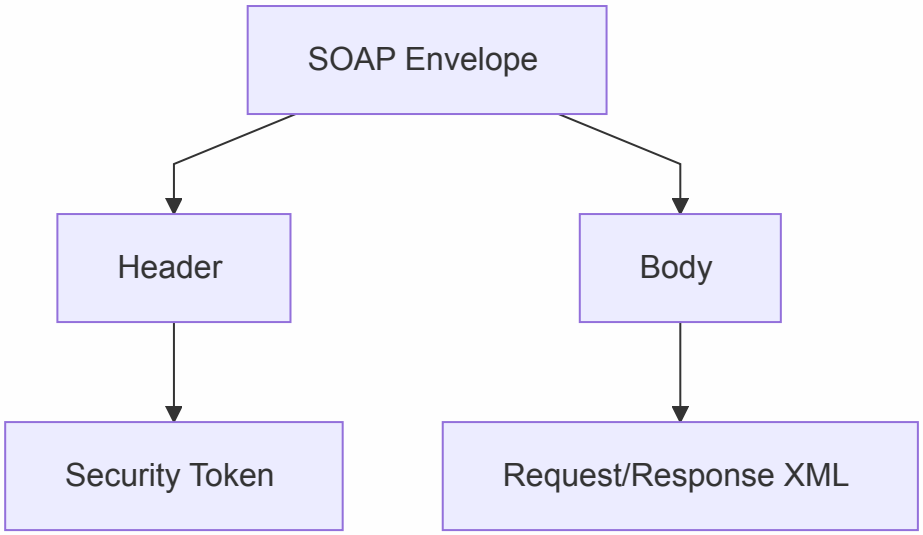
SOAP vs REST (alto livello)

Aspetto	SOAP	REST
Trasporto	Può usare diversi protocolli (HTTP, SMTP, JMS…)	Indipendente dal protocollo, ma usato quasi sempre su HTTP
Formato	XML	JSON / XML
Contratto	WSDL	OpenAPI (non obbligatorio)
Pattern	RPC-oriented (spesso)	Resource-based
Complessità	Alta	Media / Bassa

📌 Nota: REST **non è legato a HTTP**, ma HTTP realizza molto bene i vincoli REST ed è diventato lo standard di fatto per le Web API.

SOAP Envelope: diagramma

SOAP Envelope — Diagramma



WSDL: struttura

WSDL — Struttura

Un file **WSDL** (Web Services Description Language) descrive completamente un servizio SOAP.

Contiene:

- **types** → definizione dei tipi (XSD)
- **message** → input/output dei messaggi
- **portType** → operazioni disponibili
- **binding** → protocollo e stile
- **service** → endpoint reali

Serve come contratto formale per generare client e server.

📘 *Example: WSDL minimale (esempio didattico)*

```
<definitions xmlns='http://schemas.xmlsoap.org/wsdl/'
             xmlns:xsd='http://www.w3.org/2001/XMLSchema'
             xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
             name='HelloService'>

  <types>
    <xsd:schema>
```

```
      <xsd:element name='HelloRequest' type='xsd:string'/>
      <xsd:element name='HelloResponse' type='xsd:string'/>
    </xsd:schema>
  </types>

  <message name='HelloRequestMessage'>
    <part name='request' element='HelloRequest'/>
  </message>

  <message name='HelloResponseMessage'>
    <part name='response' element='HelloResponse'/>
  </message>

  <portType name='HelloPortType'>
    <operation name='sayHello'>
      <input message='HelloRequestMessage'/>
      <output message='HelloResponseMessage'/>
    </operation>
  </portType>

  <binding name='HelloBinding' type='HelloPortType'>
    <soap:binding style='document' transport='http://schemas.xmlsoap.org/soap/http'/>
  </binding>

  <service name='HelloService'>
    <port name='HelloPort' binding='HelloBinding'>
      <soap:address location='http://example.com/hello'/>
    </port>
  </service>

</definitions>
```

✏️ *Exercise: Individuare le sezioni di un WSDL*

🧪 **Esercizio**
Osserva l'esempio di WSDL minimale nella slide.
Individua dove sono:

- la definizione dei tipi (types)
- i messaggi di richiesta e risposta (message)
- l'operazione esposta (portType)
- il binding SOAP/HTTP (binding)
- l'endpoint reale (service)

▶ 💡 Soluzione

SOAP: esempi di codice

SOAP — Esempi di codice

Esempio 1: richiesta SOAP inviata via HTTP.

```
POST /hello HTTP/1.1
Host: example.com
Content-Type: text/xml; charset=utf-8
Content-Length: ...
SOAPAction: "sayHello"

<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'>
  <soap:Body>
    <HelloRequest>World</HelloRequest>
  </soap:Body>
</soap:Envelope>
```

Esempio 2: client Java (JAX-WS) generato dal WSDL.

```
HelloService service = new HelloService();
HelloPortType port = service.getHelloPort();
String response = port.sayHello("World");
System.out.println(response);
```

Quando scegliere SOAP?

Quando scegliere SOAP?

- Interoperabilità enterprise
- Transazioni complesse
- Contratti rigidi
- Sistemi legacy
- Compliance/regolamentazione forte (es: Banking)

📘 *Example: Pattern uniforme cartelle/risorse*

```
/customer
/customer/{id}
/customer/{id}/orders
/customer/{id}/orders/{orderId}/items
```

SOAP e sicurezza

SOAP e sicurezza

Supporta:

- WS-Security
- Firma e cifratura dei messaggi
- Token e assunzioni di identità
- Protezione end-to-end

Complessità elevata, ma potenza enterprise.

Perché è ancora rilevante?

Perché SOAP è ancora rilevante?

- Esistono sistemi critici che non possono essere migrati
- Standardizzazione forte
- Strumenti legacy ancora molto utilizzati
- Aziende che operano con normative stringenti
- Molte API B2B sono SOAP

✏️ *Exercise: Validare semantica HTTP su un dominio scelto*

```
🧪 **Esercizio**
Scegli una collezione (products, invoices, tickets) e definisci correttamente:
- cosa fa GET
- cosa fa POST
- quali status code devono essere usati
- cosa *non* deve essere permesso
```

REST prende il sopravvento

Perché REST ha superato SOAP?

- Minor complessità
- JSON molto più semplice di XML
- Adozione nativa nel Web
- Performance migliori per casi generici
- Ecosistema tool migliore (OpenAPI, Swagger, Postman)

Il ciclo di vita di un'API moderna

Ciclo di vita di una API moderna

1. Analisi dei consumer, ovvero analisi dei **client dell'API** — cioè quali software useranno l'API e quali esigenze hanno/avranno.
2. Design (design-first)
3. Documentazione
4. Mocking
5. Implementazione
6. Deploy
7. Monitoraggio
8. Evoluzione

“Il design-first riduce la necessità di versioni e velocizza l’evoluzione.”

📘 *Example: Trasformare un'operazione RPC in REST*

```
❌ POST /approveOrder?orderId=123
✔️ POST /orders/123/approval
```

Design-first

Design-first (Regioni / Province / Comuni)

Invece di "scrivere codice e vedere cosa succede", il **design-first** parte da una domanda semplice:

> "Che cosa devono poter fare gli utilizzatori della nostra API?"

Usiamo l'esempio reale dell'app **Regioni / Province / Comuni**.

1️⃣ *Partiamo dai bisogni (prima ancora della tecnologia)*

Esempio di domande tipiche:

- "Voglio vedere l'elenco delle regioni italiane"
- "Una volta scelta la regione, voglio le sue province"
- "Voglio i comuni di una certa provincia"
- "Vorrei filtrare per nome (es. tutti i comuni che contengono 'San')"

Qui non parliamo di codice: parliamo di **casi d'uso**.

2️⃣ *Disegniamo la mappa dell'API (le risorse)*

Dal ragionamento precedente nascono le **"cose" principali** dell'API:

- `Region` (Regione)
- `Province` (Provincia)
- `City` (Comune)

E da queste cose derivano naturalmente le operazioni:

- "Dammi tutte le regioni" → `/regions`
- "Dammi le province di una regione" → `/provinces`
- "Dammi i comuni" → `/cities`

Ancora: nessun codice, solo una **mappa concettuale** di cosa esporrà l'API.

3️⃣ *Pensiamo alle versioni (V1, V2, V3…) PRIMA di implementare*

- **V1** → API molto semplice, solo liste complete e dettagli
  (es. tutte le regioni, tutte le province, tutti i comuni)
- **V2** → aggiungiamo i **filtri** (senza rompere i client):
  es. `?regionId=`, `?provinceId=`, `?name=`
- **V3** → aggiungiamo **paginazione e link di navigazione**
  (utile quando i comuni sono tanti)

Design-first significa: questa roadmap la decidiamo **a tavolino**, non a metà progetto quando ormai è tardi.

4️⃣ *Validiamo il design con esempi, non con il codice*

Prima di implementare, immaginiamo delle richieste/risposte tipo:

```
GET /regions
→ elenco delle regioni con id e nome

GET /cities?provinceId=101&name=San
→ elenco di comuni di quella provincia che contengono "San" nel nome
```

Se questi esempi funzionano **sulla carta**, il codice sarà molto più lineare da scrivere.

> Il design-first non è per programmatori: è per chiunque debba decidere **che cosa fa** l'API.

Esercizio — Dai bisogni alle funzionalità dell’API

Obiettivo: progettare insieme la **prima versione dell’API** (V1), partendo dai bisogni reali degli utilizzatori.

🔹 Traccia dell’esercizio

Si considerino i seguenti bisogni informativi:

- vedere l’elenco delle regioni italiane
- vedere le province appartenenti a una regione
- vedere i comuni di una provincia
- cercare un comune per nome

Compito:

1. Raggruppare i bisogni in "oggetti" o "entità" del dominio.
2. Derivare da essi le operazioni fondamentali dell’API (senza pensare al codice).
3. Proporre una prima bozza di V1 e una prima estensione naturale per V2.

▶ **Mostra una possibile soluzione**

✅ *Possibile soluzione*

1. Entità (risorse principali)

- Region
- Province
- City

2. Operazioni principali (V1)

- `GET /regions` — tutte le regioni
- `GET /provinces` — tutte le province
- `GET /cities` — tutti i comuni
- (opzionale) endpoint di dettaglio via ID

3. Estensioni naturali (V2 — evoluzione additiva)

- `GET /provinces?regionId=...`
- `GET /cities?provinceId=...`
- `GET /cities?regionId=...`
- `GET /cities?name=...` (ricerca per nome)

Questa soluzione mette in evidenza che:

- le risorse nascono dai bisogni
- V1 rappresenta la base minima
- V2 introduce filtri senza rompere nulla