ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Progettazione
UNITÀ: Progettazione 2

Ing. Delisio Roberto

API REST

Fondamenti delle Applicazioni Web

Le applicazioni distribuite

Le applicazioni distribuite sono applicazioni in cui i vari componenti che compongono il sistema si trovano su macchine distinte.

- I vari processi cooperano scambiandosi dati e messaggi

Un'applicazione distribuita è strutturata solitamente con un'architettura multi layer

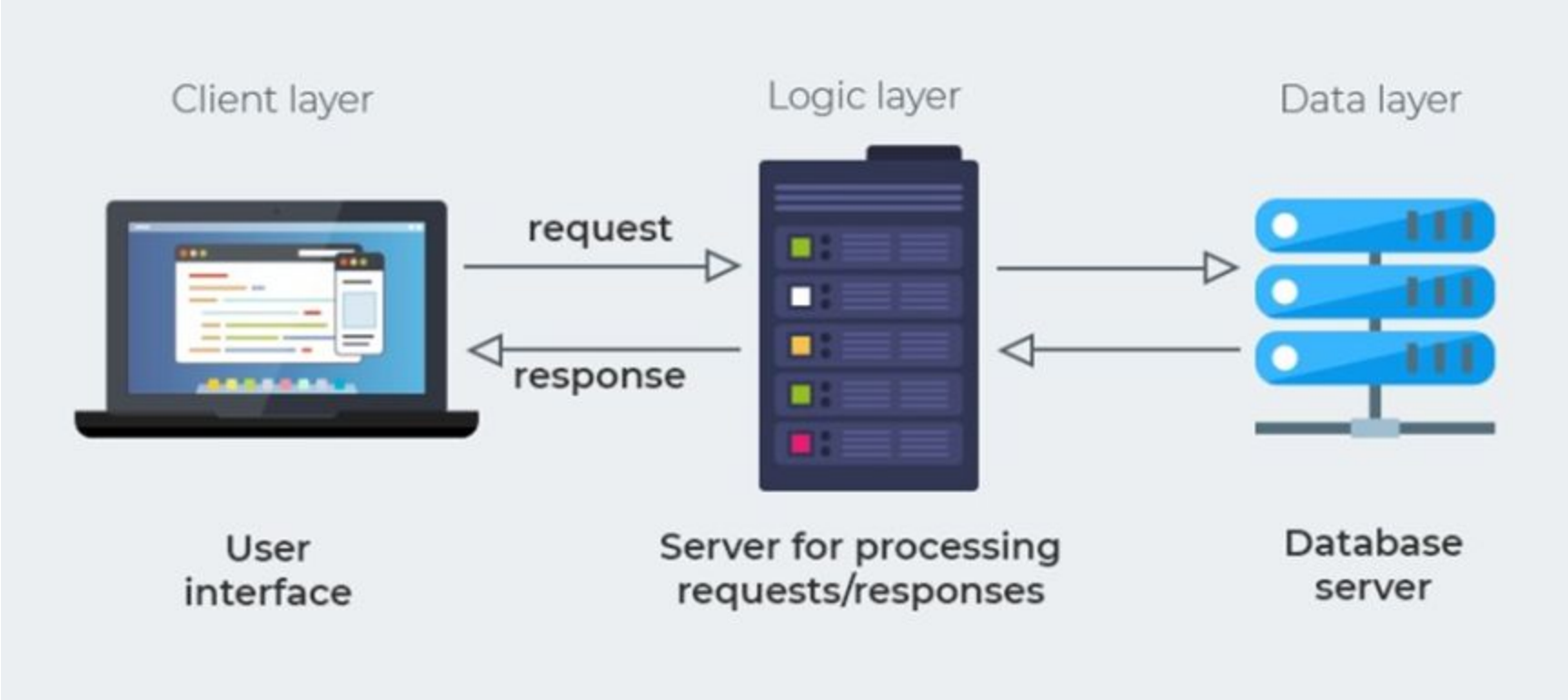
- Ogni layer è costituito da un set di componenti
- Ogni set di componenti realizza un set di funzionalità

Architettura multilivello

L’architettura multilivello più diffusa è l’architettura three-tier.

Essa prevede:

- Client Layer
- Logic Layer
- Data Layer



Client tier -> livello presentazione

Scopo principale è visualizzare le informazioni e raccogliere dati dall'utente e dall'interazione con l'utente

Questo tier di livello superiore, ad esempio, può essere eseguito su un browser web, come applicazione desktop o come GUI (finestra grafica).

Generalmente, i tier presentazione web sono sviluppati utilizzando HTML, CSS e JavaScript.

Logic tier -> livello applicativo

Scopo principale è recuperare informazioni dal client tier, elaborarle e metterle in relazione con il data tier.

Questo tier di livello intermedio **contiene la logica di business**, un insieme specifico di regole e comportamenti che il sistema fornisce.

Il tier applicativo può anche **aggiungere, eliminare o modificare i dati nel data tier** .

Il tier applicativo è tipicamente sviluppato utilizzando Python, Java, Perl, PHP o Ruby e comunica con il data tier tramite librerie(API)

Data tier -> livello persistenza

Scopo principale è archiviare e gestire i dati persistenti dell'applicazione, solitamente attraverso un database

Questo tier è il livello più basso e interagisce solo con il livello applicativo.

Questo livello si realizza con database relazionali come PostgreSQL, MySQL, MariaDB, Oracle, DB2, Informix o Microsoft SQL Server o database NoSQL come Cassandra, CouchDB o MongoDB.

Applicazione client - server

L'architettura Client-Server è il modello fondamentale su cui si basano quasi tutte le applicazioni web. Descrive una relazione in cui un programma, il Client, richiede servizi o risorse a un altro programma, il Server.

Pensa a un ristorante:

- **Il Cliente (Client)**: Sei tu, seduto al tavolo. Hai bisogno di qualcosa (cibo, bevande). Per ottenerlo, fai una richiesta. Nel mondo web, il client è tipicamente il browser sul tuo computer o smartphone.
- **Il Server (Server)**: È la cucina del ristorante. È sempre attiva, in attesa di ricevere ordini. Quando arriva un ordine (una richiesta), la cucina lo elabora (prepara il piatto) e restituisce il risultato (il piatto pronto). Nel mondo web, il server è un potente computer sempre connesso a Internet, in attesa di ricevere richieste dai client.

Il flusso di comunicazione è sempre lo stesso:

- **Richiesta (Request)**: Il client invia una richiesta al server (es. "dammi la pagina principale di questo sito").
- **Elaborazione (Processing)**: Il server riceve la richiesta, la elabora (cerca i dati in un database, esegue un calcolo, ecc.).
- **Risposta (Response)**: Il server invia una risposta al client (es. il codice HTML della pagina richiesta, dei dati, o un messaggio di errore).

Questa architettura permette di centralizzare la logica complessa e i dati sensibili sul server, mantenendo il client leggero e focalizzato sulla presentazione delle informazioni.

Applicazione client - server

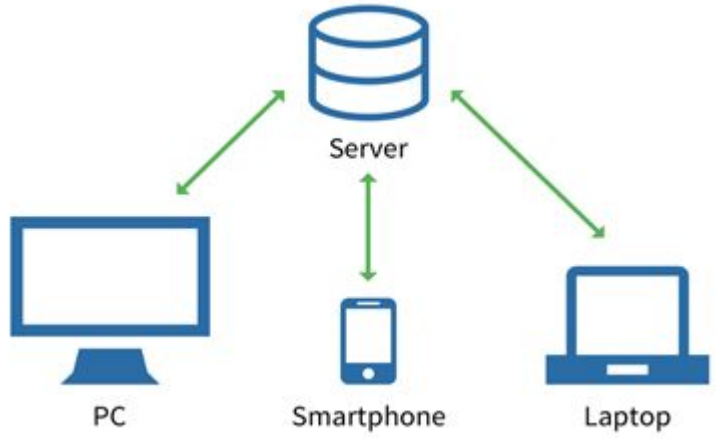
Un sistema client-server quindi è un contesto applicativo di rete dove un programma detto CLIENT richiede servizi ad un altro programma detto SERVER

Il programma cliente e il programma server girano tipicamente su macchine distinte collegate in rete.

Il Client potrebbe usare differenti dispositivi per collegarsi al Server

Regole generali:

1. Il server eroga servizi (su richiesta) per i client
2. I client si collegano ai server e richiedono servizi
3. Client e server devono aver concordato un protocollo di comunicazione e scambio dati



Architettura Client-Server

Il modello ha due attori, il Client (chi chiede) e il Server (chi risponde).

Come il Client Trova il Server: Domini e DNS

- **Il Problema**: I server su Internet sono identificati da indirizzi numerici chiamati indirizzi IP (es. 142.250.184.142), che sono difficili da ricordare per gli esseri umani.
- **La Soluzione**: Usiamo i nomi di dominio (es. www.google.com), che sono etichette facili da memorizzare.
- **Il Traduttore (DNS)**: Il DNS (Domain Name System) agisce come una "rubrica telefonica di Internet". Quando un utente digita un nome di dominio nel browser, il sistema DNS ha il compito di tradurre quel nome nell'indirizzo IP numerico corrispondente, permettendo al client di localizzare e contattare il server corretto.

Architettura Client-Server

Flusso di una Richiesta (con DNS):

1. L'utente scrive www.google.com nel browser (Client).
2. Il browser chiede al DNS: "Qual è l'IP di www.google.com?"
3. Il DNS risponde: "L'IP è 142.250.184.142".
4. Il browser invia la sua richiesta HTTPS all'indirizzo 142.250.184.142 (Server).
5. Il server risponde.

Indirizzo IP

Un indirizzo IP (Internet Protocol) è un'etichetta numerica che identifica in modo univoco un dispositivo (come un computer o uno smartphone) all'interno di una rete. La sua struttura dipende dalla versione del protocollo: IPv4 o IPv6.

IPv4 (Internet Protocol version 4)

Questo è il formato più comune e conosciuto. Un indirizzo IPv4 è un numero a 32 bit, rappresentato per comodità come quattro blocchi di numeri decimali separati da un punto (notazione "dotted-quad").

- Formato: X.X.X.X (es. 192.168.1.10)
- Struttura di ogni blocco: Ogni blocco (chiamato ottetto) rappresenta 8 bit e può avere un valore compreso tra 0 e 255.

Un indirizzo IPv4 è diviso in due parti logiche:

1. Network ID (Identificativo di Rete): La prima parte dell'indirizzo che identifica la rete specifica a cui il dispositivo è collegato. Tutti i dispositivi sulla stessa rete locale condividono lo stesso Network ID.
2. Host ID (Identificativo di Host): La seconda parte dell'indirizzo che identifica in modo univoco il dispositivo specifico all'interno di quella rete.

La separazione tra queste due parti è definita dalla Subnet Mask. Ad esempio, con l'indirizzo 192.168.1.10 e la subnet mask 255.255.255.0, significa che:

- 192.168.1 è il Network ID.
- 10 è l'Host ID.

Indirizzo IP

IPv6 (Internet Protocol version 6)

Introdotto per superare l'esaurimento degli indirizzi IPv4, IPv6 usa un numero a 128 bit, offrendo uno spazio di indirizzamento quasi illimitato.

- Formato: È rappresentato come otto gruppi di quattro cifre esadecimali, separati da due punti.
    1. Esempio: 2001:0db8:85a3:08d3:1319:8a2e:0370:7344
- Regole di Compressione: Per semplificarne la lettura, si possono abbreviare:
    1. I zeri iniziali di ogni blocco possono essere omessi (es. 0db8 diventa db8).
    2. Una sequenza contigua di blocchi composti solo da zeri può essere sostituita una sola volta da due punti (::).
        - Esempio: 2001:0db8:0000:0000:0000:8a2e:0370:7344 diventa 2001:0db8::8a2e:0370:7344.

La sua struttura è più complessa e generalmente divisa in un prefisso di routing globale (per la rete), un ID di sottorete e un ID di interfaccia (per il dispositivo).

IP Pubblici vs. IP Privati

Indipendentemente dalla versione, gli indirizzi IP si dividono in due categorie:

- IP Pubblico: È l'indirizzo univoco che il tuo Internet Service Provider (ISP) ti assegna per essere identificato su Internet.
- IP Privato: È un indirizzo usato all'interno di una rete locale (come la rete Wi-Fi di casa tua). Esistono intervalli specifici riservati per questo uso (es. 192.168.x.x, 10.x.x.x). Questi indirizzi non sono visibili su Internet e permettono a più dispositivi di condividere un unico IP pubblico tramite un processo chiamato NAT (Network Address Translation) gestito dal router.

Domini e DNS

Un nome di dominio è un'etichetta facile da ricordare che identifica una risorsa su Internet, come un sito web. Il DNS (Domain Name System) è il sistema che traduce questi nomi leggibili in indirizzi IP numerici, necessari ai computer per comunicare tra loro.

Com'è Fatto un Dominio: Un'Architettura Gerarchica

Un nome di dominio ha una struttura gerarchica che si legge da destra verso sinistra. Ogni "punto" separa un livello diverso della gerarchia.

Prendiamo come esempio l'indirizzo mail.google.com:

1. **Top-Level Domain (TLD) o Dominio di Primo Livello**: È la parte più a destra (.com). Definisce la categoria più ampia del dominio.
    - TLD generici (gTLD): .com (commerciale), .org (organizzazione), .net (network).
    - TLD nazionali (ccTLD): .it (Italia), .de (Germania), .uk (Regno Unito).
2. **Second-Level Domain (SLD) o Dominio di Secondo Livello**: È la parte centrale (google). Questo è il nome che viene registrato ed è il cuore dell'identità del brand. È univoco all'interno del suo TLD (esiste una sola google.com).
3. **Subdomain (Sottodominio) o Dominio di Terzo Livello**: È la parte più a sinistra (mail). I sottodomini vengono creati dal proprietario del dominio di secondo livello per organizzare e separare diverse sezioni o servizi del proprio sito.
    - www: Convenzionalmente usato per il sito web principale.
    - api: Spesso usato per esporre le API.
    - blog: Per una sezione dedicata al blog.

La Gestione del DNS: Nameserver e Record

Se il dominio è l'indirizzo, il DNS è la "rubrica telefonica" che lo rende utilizzabile. Questa gestione si basa su due componenti chiave: i Nameserver e i Record DNS.

Nameserver (NS)

Un Nameserver è un server specializzato che contiene tutti i record DNS per un determinato dominio. Quando acquisti un dominio (es. miosito.it), devi specificare quali sono i suoi "Nameserver autoritativi". Solitamente sono forniti dal tuo provider di hosting (es. ns1.tuohosting.com e ns2.tuohost-ing.com).

Questi server diventano la fonte ufficiale di verità per il tuo dominio. Qualsiasi computer nel mondo che voglia sapere l'indirizzo IP di miosito.it finirà per interrogare uno di questi nameserver.

I Record DNS

I **Record DNS** sono le singole "voci" all'interno della rubrica gestita dal nameserver. Ogni record ha un tipo specifico che definisce il tipo di informazione che fornisce. I più importanti sono:

- **Record A (Address):** È il record più comune. Associa un nome di dominio a un **indirizzo IPv4**.
  - miosito.it. A 88.99.100.123
  - (Significa: "chi cerca miosito.it deve andare all'indirizzo IP 88.99.100.123").
- **Record AAAA (Quad A):** Simile al record A, ma associa un dominio a un **indirizzo IPv6**.
  - miosito.it. AAAA 2001:0db8:85a3:0000:0000:8a2e:0370:7334
- **Record CNAME (Canonical Name):** È un alias. Permette a un dominio di puntare a un altro dominio, invece che a un indirizzo IP.
  - www.miosito.it. CNAME miosito.it.
  - (Significa: "www.miosito.it è solo un altro nome per miosito.it. Trovate l'IP di quest'ultimo"). È utile per avere più nomi che puntano allo stesso server.
- **Record MX (Mail Exchange):** Specifica quali sono i server di posta elettronica responsabili di ricevere le email per quel dominio.
  - miosito.it. MX 10 mail.provider.com.
  - (Significa: "tutte le email indirizzate a @miosito.it devono essere gestite dal server mail.provider.com". Il numero 10 indica la priorità).
- **Record TXT (Text):** Permette di associare al dominio una stringa di testo arbitrario. È comunemente usato per scopi di verifica (es. per dimostrare a Google o Microsoft di essere il proprietario del dominio) o per configurazioni di sicurezza email come SPF e DKIM.
- **Record NS (Name Server):** Indica quali sono i nameserver autoritativi per un dominio (o sottodominio), delegando di fatto la gestione.

Il Processo di Risoluzione DNS in Breve

Quando digiti www.miosito.it nel browser:

1. Il tuo computer chiede al suo risolutore DNS (solitamente fornito dal tuo provider Internet).
2. Se il risolutore non conosce la risposta, inizia un processo a catena: interroga i Root Server mondiali, che lo indirizzano ai server del TLD .it.
3. I server del .it dicono al risolutore: "Per miosito.it, devi chiedere ai suoi nameserver autoritativi, ns1.tuohosting.com".
4. Il risolutore interroga ns1.tuohosting.com, che finalmente legge i suoi record.
5. Trova prima il record CNAME per www.miosito.it che punta a miosito.it, e poi il record A per miosito.it che punta a 88.99.100.123.
6. Restituisce l'IP 88.99.100.123 al tuo computer.
7. Il tuo browser contatta il server a quell'indirizzo IP.

Cos'è un'applicazione web?

Un'applicazione web (web app) è un programma software a cui si accede tramite un browser web (come Chrome, Firefox, Safari) attraverso una rete come Internet o una intranet.

A differenza di un'applicazione desktop tradizionale che viene installata ed eseguita interamente sul computer dell'utente, un'applicazione web ha una parte del suo codice (il backend) che risiede su un computer remoto, il server, e una parte (il frontend) che viene eseguita nel browser dell'utente.

La caratteristica distintiva è l'interattività. Mentre un sito web può essere una semplice pagina di sola lettura, un'applicazione web permette all'utente di compiere azioni e manipolare dati.

Esempi di applicazioni web: Gmail, Google Docs, Facebook, Amazon, Trello.

Caratteristiche principali:

- Nessuna installazione richiesta da parte dell'utente.
- Accessibilità da qualsiasi dispositivo con un browser e una connessione a Internet.
- Centralizzazione degli aggiornamenti (si aggiorna l'applicazione sul server e tutti gli utenti vedono subito la nuova versione).

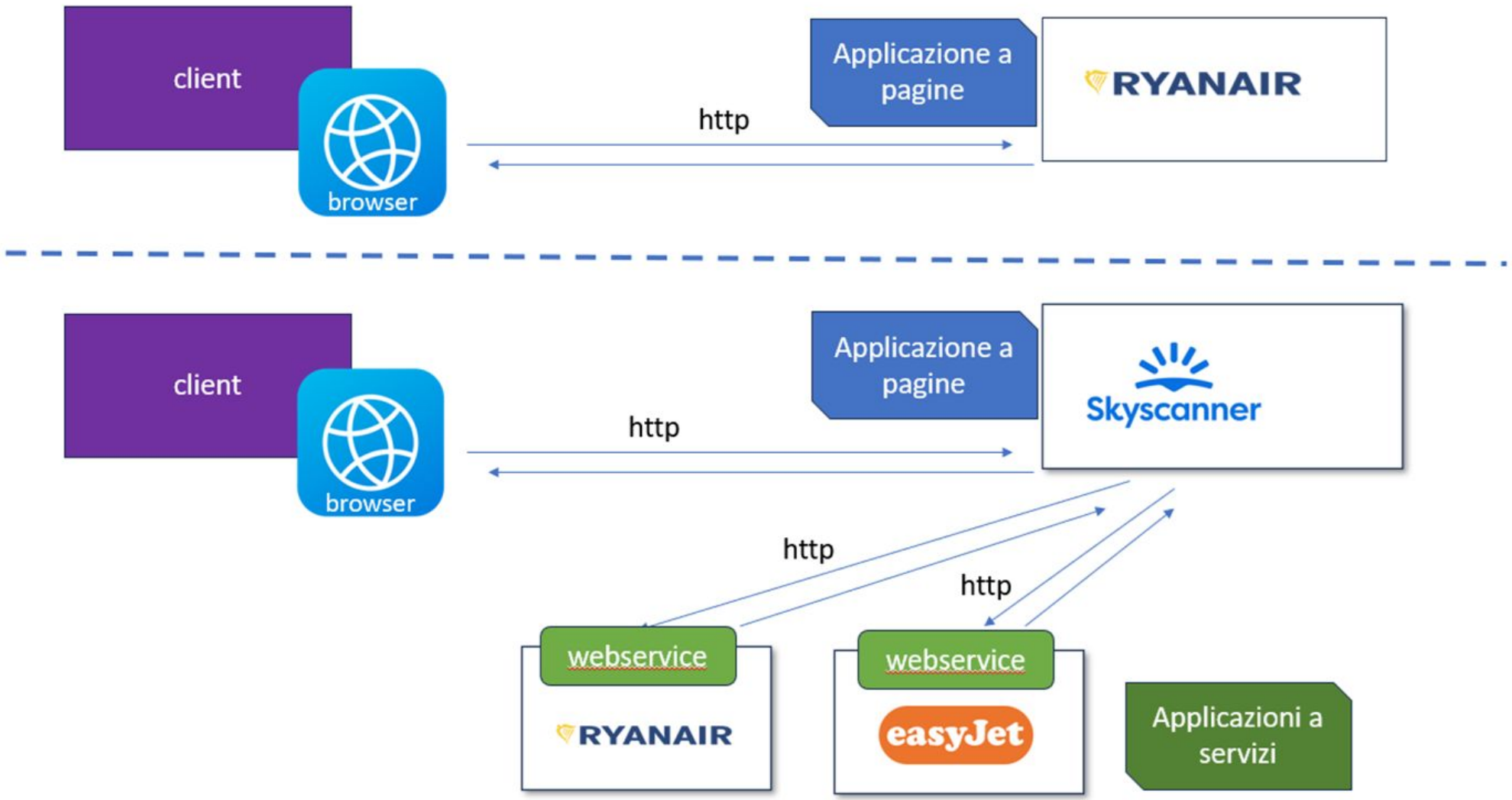
Le applicazioni web

Le applicazioni web quindi sono particolari applicazioni distribuite che usano l'architettura client-server

Caratteristiche

- Utilizzano il protocollo TCP/IP
- Si dividono in 2 famiglie principali:
  - Applicazioni a pagine: il client è l'utente che, attraverso il browser, chiede un servizio al server e riceve in risposta una pagina web
  - Applicazioni a servizi: il client è un programma che chiede un servizio al server e riceve in risposta dati

Applicazioni a confronto



Caratteristiche di un’applicazione a servizi

Il client è un programma (come il server)

- Applicazione Android
- Applicazione IOS
- Browser che esegue codice Javascript in una pagina web

Client e server si scambiano SOLO dati

Il sistema client-server è interoperabile

- Client e Server possono essere scritti con infrastruttura tecnologica e linguaggi indipendenti

La realizzazione segue uno di questi 2 stili principali:

- SOAP (stile classico)
- REST (nuovo stile)

Frontend vs Backend: Ruoli e Responsabilità

Se l'architettura è Client-Server, Frontend e Backend sono i due lati dello sviluppo che costruiscono le due parti.

Frontend (Lato Client)

Il Frontend è tutto ciò che l'utente vede e con cui interagisce nel browser. È la parte "visibile" dell'iceberg.

- **Ruolo**: Creare l'interfaccia utente (UI) e garantire una buona esperienza utente (UX).
- **Responsabilità**:
  - Struttura: Definire la struttura dei contenuti di una pagina.
  - Stile: Curare l'aspetto grafico, i colori, i layout e l'adattabilità ai diversi schermi (responsive design).
  - Interattività: Gestire click, animazioni, compilazione di form e aggiornamenti dinamici della pagina senza doverla ricaricare completamente.
- **Tecnologie principali:**
  - HTML (HyperText Markup Language): Per la struttura dei contenuti.
  - CSS (Cascading Style Sheets): Per lo stile e l'aspetto grafico.
  - JavaScript: Per l'interattività. Framework moderni come React, Angular e Vue.js sono basati su JavaScript e permettono di creare interfacce complesse e reattive.

Frontend vs Backend: Ruoli e Responsabilità

Backend (Lato Server)

Il Backend è il "cervello" dell'applicazione, la parte che l'utente non vede. È la parte "sommersa" dell'iceberg.

- **Ruolo**: Gestire la logica di business, i dati e la sicurezza.
- **Responsabilità**:
    - Logica applicativa: Eseguire le operazioni principali dell'applicazione (es. elaborare un pagamento, registrare un nuovo utente).
    - Gestione del database: Salvare, recuperare, modificare e cancellare dati da un database (es. lista di prodotti, post di un blog, informazioni degli utenti).
    - Autenticazione e autorizzazione: Verificare chi è l'utente (login) e cosa ha il permesso di fare.
    - **Esporre le API**: Creare i punti di contatto (endpoint) che il frontend può interrogare per ottenere o inviare dati.
- **Tecnologie principali:**
    - Linguaggi di programmazione: Python (con framework come Django, Flask), JavaScript (con Node.js), Java (Spring), PHP (Laravel), C# (.NET).
    - Database: SQL (PostgreSQL, MySQL) e NoSQL (MongoDB, Redis).
    - Web Server: Apache, Nginx.

Il ponte tra Frontend e Backend è l'API (Application Programming Interface). Il frontend non accede mai direttamente al database; chiede i dati al backend tramite una chiamata API.

Il ruolo del Browser e del Server Web

Questi sono i due attori software principali nell'architettura Client-Server.

Browser Web (il Client Software):

- Il suo compito principale è inviare richieste HTTP al server quando l'utente digita un indirizzo o clicca un link.
- Una volta ricevuta la risposta (tipicamente codice HTML, CSS, JavaScript), il suo secondo compito è interpretare (renderizzare) questo codice per mostrare all'utente una pagina web visivamente completa e interattiva.
- Esegue il codice JavaScript del frontend per gestire l'interattività.

Server Web (il Backend Software):

- È un software specifico (es. Apache, Nginx) in esecuzione sulla macchina server.
- Il suo compito è ascoltare le richieste HTTP in arrivo dai browser.
- Se la richiesta è per un file statico (un'immagine, un file CSS), il server web lo trova e lo restituisce direttamente.
- Se la richiesta è per una risorsa dinamica (es. api/products/123), il server web la passa all'applicazione backend (scritta in Python, Java, etc.). L'applicazione elabora la richiesta, genera una risposta (spesso dati in formato JSON), e la restituisce al server web, che a sua volta la inoltra al browser del client.

Differenza tra Siti Web Statici e Applicazioni Dinamiche

Questa distinzione è cruciale per capire perché abbiamo bisogno di un backend complesso e di API.

Sito Web Statico

Un sito web statico è composto da un insieme di file (HTML, CSS, JS) pre-costruiti.

- Quando un utente richiede una pagina, il server web si limita a trovare il file HTML corrispondente e a inviarlo al browser così com'è.
- Il contenuto non cambia in base all'utente o alle sue interazioni. Ogni visitatore vede esattamente la stessa cosa.
- Analogia: Una brochure o un volantino digitale.
- Uso ideale: Siti di presentazione, portfolio, documentazione, landing page. Non richiedono un backend complesso o un database.

Applicazione Web Dinamica

Un'applicazione web dinamica genera le sue pagine (o i suoi dati) "al volo" in risposta a una richiesta.

- Quando un utente richiede una pagina, la richiesta viene elaborata dalla logica del backend. Il backend può interrogare un database, personalizzare il contenuto in base all'utente (es. "Ciao, Marco!"), e costruire la risposta (HTML o dati JSON) al momento.
- Il contenuto è personalizzato e interattivo.
- Analogia: Una conversazione. La risposta dipende da chi sei e da cosa chiedi.
- Uso ideale: Social network, e-commerce, piattaforme di online banking, dashboard.

Nel contesto del tuo corso, le API REST sono il meccanismo che permette alle applicazioni dinamiche moderne di funzionare, consentendo al frontend di richiedere e manipolare dati in modo strutturato e prevedibile dal backend.

Il protocollo HTTP

Protocollo TCP/IP

Un protocollo è un insieme di regole per dialogare.

Il protocollo TCP/IP è il protocollo utilizzato sul web

Le figure base sono Client e Server

Principio base: Il client invoca il server (mai il contrario)

Il protocollo TCP/IP è una famiglia di protocolli composta da 4 livelli

Ogni livello si occupa di un aspetto e prevede un set di regole

I livelli sono: applicazione, trasporto, rete, fisico

Livelli del protocollo

Il dialogo avviene correttamente solo se per ciascuno dei 4 livelli viene utilizzato lo stesso sotto protocollo.

Significa che affinché la comunicazione tra due sistemi (mittente e destinatario) avvenga con successo, il protocollo utilizzato da un determinato livello sul sistema mittente deve essere compreso e utilizzato anche dal livello corrispondente sul sistema destinatario. Sono come due persone che parlano la stessa lingua per capirsi.

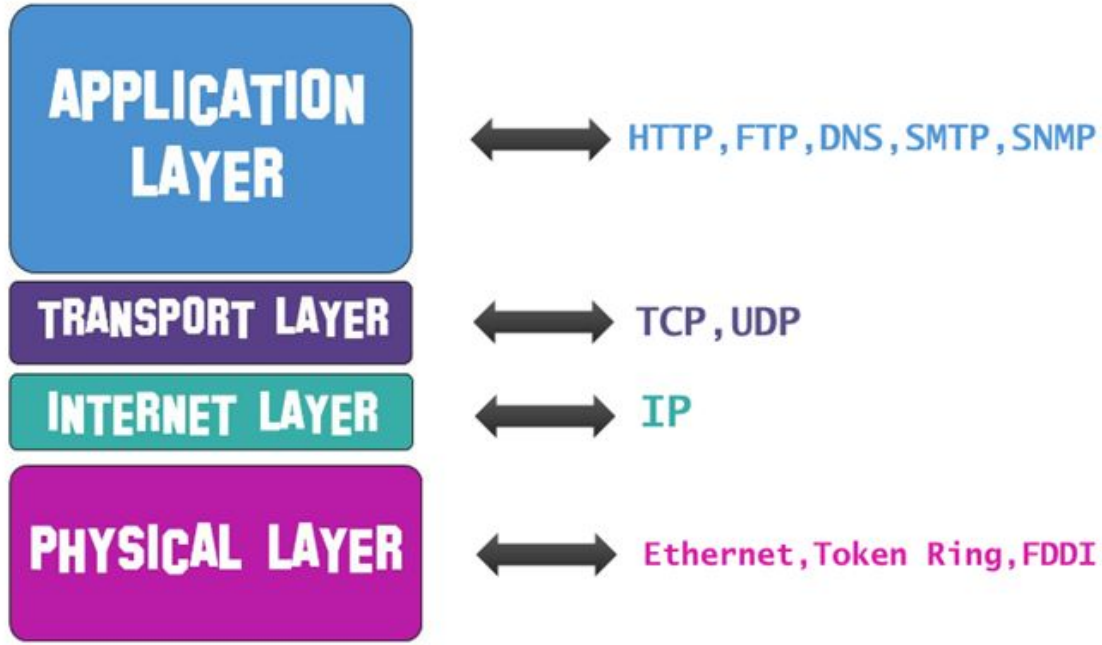
Livello fisico -> protocollo nativo

Livello rete -> IP

Livello trasporto -> TCP e UDP

Livello applicazione -> vari protocolli specifici per varie esigenze

- HTTP -> hyper text
- FTP -> file di grandi dimensioni
- SMTP -> mail
- ecc



Livello di Applicazione (Application Layer)

Il livello di Applicazione è il più vicino all'utente finale. È responsabile di fornire i servizi di rete direttamente alle applicazioni software, come i browser web, i client di posta elettronica e i software per il trasferimento di file. In pratica, questo livello gestisce la rappresentazione dei dati, la codifica e il controllo del dialogo.

Funzioni principali:

- **Interfaccia con l'utente**: Fornisce i protocolli che le applicazioni utilizzano per scambiarsi dati sulla rete.
- **Gestione della sessione e presentazione**: A differenza del modello OSI, il TCP/IP raggruppa in questo unico livello anche le funzioni di gestione della sessione (apertura, mantenimento e chiusura di una connessione tra due host) e di presentazione (formattazione e traduzione dei dati, come la crittografia e la compressione).

Protocolli comuni:

- HTTP (HyperText Transfer Protocol): Utilizzato per la navigazione web.
- HTTPS (HTTP Secure): La versione sicura di HTTP, che utilizza la crittografia.
- FTP (File Transfer Protocol): Per il trasferimento di file.
- SMTP (Simple Mail Transfer Protocol): Per l'invio di posta elettronica.
- POP3 (Post Office Protocol 3) e IMAP (Internet Message Access Protocol): Per la ricezione della posta elettronica.
- DNS (Domain Name System): Per tradurre i nomi di dominio (es. www.google.com) in indirizzi IP.

Livello di Trasporto (Transport Layer)

Il livello di Trasporto ha il compito fondamentale di gestire la comunicazione tra due processi (applicazioni) su host diversi. Fornisce un canale logico per il flusso dei dati, garantendo che arrivino a destinazione in modo affidabile e ordinato, se richiesto.

Funzioni principali:

- **Segmentazione dei dati**: Suddivide i dati provenienti dal livello di Applicazione in segmenti più piccoli e gestibili.
- **Controllo di flusso**: Regola la quantità di dati che possono essere inviati per evitare di sovraccaricare il destinatario.
- **Controllo di errore e affidabilità**: Si assicura che tutti i segmenti arrivino a destinazione senza errori, in ordine e senza duplicati.

Protocolli principali:

- **TCP (Transmission Control Protocol)**: È un protocollo orientato alla connessione e affidabile. Stabilisce una connessione (tramite un processo chiamato "three-way handshake") prima di inviare i dati e garantisce che ogni pacchetto inviato venga ricevuto correttamente, richiedendo la ritrasmissione in caso di perdita. È utilizzato dalla maggior parte delle applicazioni comuni come web e email.
- **UDP (User Datagram Protocol)**: È un protocollo "connectionless" (senza connessione) e non affidabile. Invia i dati (datagrammi) senza stabilire una connessione preliminare e senza garantire la consegna. È più veloce del TCP ed è utilizzato per applicazioni dove la velocità è critica e la perdita di qualche pacchetto è tollerabile, come lo streaming video, i giochi online e le chiamate VoIP.

Livello di Rete o Internet (Network/Internet Layer)

Questo livello è il cuore del modello TCP/IP. La sua responsabilità principale è l'instradamento dei pacchetti di dati attraverso la rete, dall'host di origine a quello di destinazione. Si occupa dell'indirizzamento logico e della determinazione del percorso migliore per i dati.

Funzioni principali:

- **Indirizzamento logico**: Assegna un indirizzo univoco (indirizzo IP) a ogni dispositivo sulla rete per identificarlo.
- **Instradamento (Routing)**: Determina il percorso migliore che i pacchetti devono seguire per raggiungere la loro destinazione, anche attraverso reti diverse.
- **Frammentazione e riassemblaggio dei pacchetti:** Se un pacchetto è troppo grande per essere gestito da una rete intermedia, viene suddiviso in frammenti più piccoli, che verranno poi riassemblati a destinazione.

Protocolli principali:

- **IP (Internet Protocol)**: È il protocollo fondamentale di questo livello. Si occupa di incapsulare i segmenti del livello di trasporto in pacchetti (o datagrammi) e di aggiungere le informazioni di indirizzamento (IP di origine e destinazione). Esistono due versioni principali: IPv4 e IPv6.
- **ICMP (Internet Control Message Protocol)**: Utilizzato per inviare messaggi di errore e di controllo, come quelli generati dal comando ping per verificare la raggiungibilità di un host.
- **ARP (Address Resolution Protocol)**: Utilizzato per risolvere (tradurre) un indirizzo IP in un indirizzo fisico (MAC address) all'interno di una rete locale.

Livello di Accesso alla Rete (Network Access Layer)

Questo è il livello più basso del modello TCP/IP e si occupa di tutti gli aspetti legati alla trasmissione fisica dei dati sulla rete locale. Corrisponde ai livelli Fisico e di Collegamento Dati (Data Link) del modello OSI.

Funzioni principali:

- **Interfacciamento con l'hardware**: Gestisce la comunicazione diretta con la scheda di rete (NIC) e il mezzo fisico (cavo Ethernet, onde radio per il Wi-Fi, ecc.).
- **Indirizzamento fisico (MAC)**: Utilizza gli indirizzi MAC (Media Access Control) per identificare i dispositivi all'interno della stessa rete locale.
- **Framing**: Organizza i bit provenienti dal livello fisico in unità logiche chiamate "frame".
- **Trasmissione fisica**: Converte i frame in segnali elettrici, luminosi o radio per la trasmissione sul mezzo fisico.

Protocolli e standard comuni:

- **Ethernet**: Lo standard più diffuso per le reti locali cablate (LAN).
- **Wi-Fi (IEEE 802.11)**: Lo standard per le reti locali senza fili (WLAN).
- **PPP (Point-to-Point Protocol)**: Utilizzato per le connessioni dirette tra due nodi.

Cos'è il Modello ISO/OSI?

Oltre al modello TCP/IP, che è il modello operativo su cui si basa Internet, esiste un altro importante framework concettuale per comprendere le reti: il modello ISO/OSI.

Cos'è il Modello ISO/OSI?

Il modello OSI (Open Systems Interconnection) è stato sviluppato dall'ISO (International Organization for Standardization). A differenza del TCP/IP, che è nato da esigenze pratiche ed è diventato uno standard de facto, il modello OSI è un modello di riferimento teorico. Il suo scopo principale era quello di creare uno standard universale per la progettazione di protocolli di rete, garantendo che sistemi di produttori diversi potessero comunicare tra loro (interoperabilità).

La sua caratteristica fondamentale è la suddivisione delle funzioni di rete in sette livelli (layer) astratti. Ogni livello ha un compito specifico e comunica solo con il livello immediatamente superiore e inferiore.

I 7 Livelli del Modello OSI

I livelli sono numerati dal basso verso l'alto. Un modo comune per ricordarli è la frase in inglese: "Please Do Not Throw Sausage Pizza Away".

7 - Livello di Applicazione (Application Layer)

- Scopo: Fornisce l'interfaccia diretta per le applicazioni utente. È il livello con cui l'utente finale interagisce.
- Funzioni: Protocolli per servizi specifici come la navigazione web, l'invio di email o il trasferimento di file. Esempi: HTTP, FTP, SMTP, DNS.

6 - Livello di Presentazione (Presentation Layer)

- Scopo: Assicurare che i dati inviati dal livello Applicazione di un sistema siano comprensibili dal livello Applicazione di un altro sistema.
- Funzioni:
  - Traduzione dati: Converte i dati in un formato intermedio standard (es. da ASCII a EBCDIC).
  - Crittografia/Decrittografia: Gestisce la cifratura dei dati per garantire la privacy.
  - Compressione: Comprime i dati per ridurre la quantità di bit da trasmettere.

5 - Livello di Sessione (Session Layer)

- Scopo: Stabilire, gestire e terminare le connessioni (chiamate "sessioni") tra applicazioni.
- Funzioni:
  - Controllo del dialogo: Decide chi può trasmettere e per quanto tempo.
  - Sincronizzazione: Inserisce dei "checkpoint" nel flusso dati, in modo che in caso di errore la trasmissione possa riprendere dall'ultimo checkpoint, invece che dall'inizio.

I 7 Livelli del Modello OSI

4 - Livello di Trasporto (Transport Layer)

- Scopo: Fornisce un trasporto dati affidabile e trasparente end-to-end, da un processo su un host a un processo su un altro host.
- Funzioni: Segmentazione dei dati, controllo di flusso, controllo degli errori. Protocolli principali:
  TCP (affidabile, orientato alla connessione) e UDP (veloce, non affidabile).

3 - Livello di Rete (Network Layer)

- Scopo: Gestire l'indirizzamento logico dei dispositivi e determinare il percorso migliore (routing) per i dati attraverso la rete.
- Funzioni: Indirizzamento logico (indirizzi IP), instradamento dei pacchetti tra reti diverse. Protocollo principale:
  IP (Internet Protocol).

2 - Livello di Collegamento Dati (Data Link Layer)

- Scopo: Fornire un transito affidabile dei dati attraverso un singolo collegamento fisico (locale).
- Funzioni:
  - Framing: Organizza i bit in unità logiche chiamate "frame".
  - Indirizzamento fisico: Utilizza gli indirizzi MAC per identificare i dispositivi sulla rete locale.
  - Controllo degli errori: Rileva (e a volte corregge) gli errori avvenuti nel livello Fisico.

1 - Livello Fisico (Physical Layer)

- Scopo: Gestire la trasmissione e la ricezione dei bit grezzi non strutturati su un mezzo fisico.
- Funzioni: Definisce le specifiche elettriche, meccaniche e funzionali del mezzo trasmissivo (cavi, tensioni, frequenze radio, connettori)

Comunicazione tra livelli

Nel modello ISO/OSI, ogni livello comunica principalmente con i livelli immediatamente adiacenti a sé (quello superiore e quello inferiore) sullo stesso sistema, fornendo servizi al livello superiore e utilizzando i servizi del livello inferiore.

Inoltre, un concetto fondamentale del modello OSI è la **comunicazione logica "peer-to-peer" (tra pari)**. Questo significa che ogni livello su un sistema (mittente) comunica concettualmente con il livello corrispondente sull'altro sistema (destinatario).

Comunicazione verticale (tra livelli adiacenti)

Quando i dati vengono inviati, passano attraverso i livelli del mittente dall'alto verso il basso. Ogni **livello aggiunge le proprie informazioni** di controllo (intestazioni e/o trailer) ai dati ricevuti dal livello superiore, un **processo chiamato incapsulamento**. Poi passa il pacchetto risultante al livello sottostante.

Quando i dati vengono ricevuti, il processo è inverso: i dati risalgono i livelli del destinatario dal basso verso l'alto. Ogni livello rimuove le informazioni di controllo aggiunte dal livello corrispondente sul mittente, decapsulando i dati, e poi li passa al livello superiore.

- Il livello N fornisce servizi al livello N+1. Ad esempio, il livello di trasporto fornisce servizi al livello di sessione.
- Il livello N utilizza i servizi del livello N-1. Ad esempio, il livello di trasporto utilizza i servizi del livello di rete

Comunicazione tra livelli

Quando i dati vengono inviati dal mittente, il flusso è dall'alto verso il basso nei livelli del modello ISO/OSI (e anche nel TCP/IP).

Riprendiamo il processo corretto:

- **Livello Applicazione (7):** L'applicazione genera i dati (es. un'email).
- **Livello Presentazione (6)**: I dati vengono formattati e, se necessario, compressi o crittografati.
- **Livello Sessione (5)**: Viene stabilita, gestita e terminata la sessione di comunicazione.
- **Livello Trasporto (4)**: I dati vengono segmentati (divisi in parti più piccole) e vengono aggiunte le informazioni per il controllo di flusso e degli errori (es. numeri di sequenza). Questo è il primo livello che aggiunge un'intestazione significativa al "payload" (i dati dei livelli superiori).
- **Livello Rete (3)**: Ogni segmento viene incapsulato in un pacchetto (o datagramma) e vengono aggiunte le informazioni di indirizzamento logico (es. indirizzi IP) per l'instradamento attraverso la rete.
- **Livello Collegamento Dati (2)**: Il pacchetto viene incapsulato in un frame e vengono aggiunte le informazioni per l'indirizzamento fisico (es. indirizzi MAC) e il controllo degli errori a livello locale. Qui possono essere aggiunti sia un'intestazione che un trailer.
- **Livello Fisico (1)**: Il frame viene convertito in un flusso di bit e trasmesso sul mezzo fisico (cavi, onde radio, ecc.).

Questo processo di aggiungere informazioni di controllo a ogni livello mentre i dati scendono, abbiamo detto, è chiamato incapsulamento.
Quando i dati arrivano al destinatario, il processo è inverso: i dati risalgono i livelli dal basso verso l'alto, e ogni livello rimuove le intestazioni aggiunte dal livello corrispondente del mittente, un processo chiamato **decapsulamento**, fino a quando i dati originali non raggiungono l'applicazione

Comunicazione orizzontale (logica "peer-to-peer")

Sebbene la comunicazione fisica avvenga solo verticalmente attraverso lo stack di protocolli, ogni livello opera come se stesse comunicando direttamente con il suo "pari" sul sistema remoto. Questo avviene attraverso l'uso di protocolli. Ogni protocollo definisce le regole e il formato dei dati per la comunicazione tra due entità dello stesso livello su sistemi diversi.

Ad esempio:

- Il livello di rete su un computer comunica logicamente con il livello di rete sull'altro computer per instradare i pacchetti.
- Il livello di trasporto su un computer comunica logicamente con il livello di trasporto sull'altro computer per garantire l'affidabilità della consegna dei dati.

Questo approccio a strati rende il modello OSI molto utile per la comprensione e la risoluzione dei problemi delle reti, poiché le responsabilità sono chiaramente separate e definite per ogni livello.

Confronto tra il Modello OSI e il Modello TCP/IP

Sebbene il mondo operi sul modello TCP/IP, il modello OSI è estremamente utile per capire le reti in modo più granulare. La mappatura tra i due modelli può essere descritta come segue:

- **Livello Applicazione (TCP/IP)**: Questo livello raggruppa le funzioni dei tre livelli superiori del modello OSI:
  - Livello 7 - Applicazione (OSI): Fornisce l'interfaccia e i servizi di rete per le applicazioni utente (es. HTTP, SMTP).
  - Livello 6 - Presentazione (OSI): Gestisce la formattazione, la crittografia e la compressione dei dati.
  - Livello 5 - Sessione (OSI): Stabilisce, gestisce e termina il dialogo tra gli host.
- **Livello Trasporto (TCP/IP)**: Corrisponde direttamente al Livello 4 - Trasporto (OSI). Entrambi si occupano della comunicazione end-to-end affidabile e del controllo di flusso, utilizzando protocolli come TCP e UDP.
- **Livello Internet (TCP/IP)**: Coincide con il Livello 3 - Rete (OSI). Entrambi gestiscono l'indirizzamento logico (IP), l'instradamento e la determinazione del percorso dei pacchetti attraverso le reti.
- **Livello di Accesso alla Rete (TCP/IP)**: Questo livello unisce le responsabilità dei due livelli inferiori del modello OSI:
  - Livello 2 - Collegamento Dati (OSI): Gestisce l'indirizzamento fisico sulla rete locale (MAC address) e il framing.
  - Livello 1 - Fisico (OSI): Si occupa della trasmissione fisica dei bit attraverso il mezzo trasmissivo (cavi, Wi-Fi).

Differenze principali:

- Modello Teorico vs. Pratico: OSI è un modello di riferimento generico, mentre TCP/IP è il modello su cui Internet è stato effettivamente costruito.
- Numero di Livelli: OSI ha 7 livelli, offrendo una separazione più netta delle funzioni, mentre TCP/IP ne raggruppa diverse in 4 livelli.

Comunicazione tra livelli TCP/IP

I principio di comunicazione tra i livelli nel modello TCP/IP è molto simile a quello del modello OSI, sebbene ci siano alcune differenze nella suddivisione e nel numero dei livelli.

Nel modello TCP/IP, come nell'OSI:

1. Comunicazione verticale (tra livelli adiacenti): I dati, quando vengono inviati, passano attraverso i livelli del mittente dall'alto verso il basso. Ogni livello aggiunge le proprie informazioni di controllo (intestazioni) ai dati ricevuti dal livello superiore (processo di incapsulamento) e poi li passa al livello sottostante. Al contrario, quando i dati vengono ricevuti, risalgono i livelli del destinatario dal basso verso l'alto, e ogni livello rimuove le intestazioni pertinenti (processo di decapsulamento) prima di passare i dati al livello superiore.
2. Comunicazione orizzontale (logica "peer-to-peer"): Concettualmente, ogni livello su un sistema comunica con il livello corrispondente sull'altro sistema. Questa comunicazione "tra pari" è definita dai protocolli specifici di quel livello. Ad esempio, il protocollo TCP (Transmission Control Protocol) opera a livello di trasporto e gestisce la comunicazione affidabile tra i processi di trasporto sui due sistemi.

Differenze principali tra TCP/IP e OSI:

- Numero di livelli: Il modello TCP/IP è generalmente rappresentato con 4 o 5 livelli, mentre il modello OSI ne ha 7.
  - TCP/IP (4 livelli): Applicazione, Trasporto, Internet, Accesso alla Rete.
  - TCP/IP (5 livelli): Applicazione, Trasporto, Rete (Internet), Collegamento Dati, Fisico. (Questa è una suddivisione più dettagliata che separa il livello di accesso alla rete in collegamento dati e fisico, rendendolo più simile all'OSI per i livelli inferiori).
  - OSI (7 livelli): Applicazione, Presentazione, Sessione, Trasporto, Rete, Collegamento Dati, Fisico.
- Combinazione di livelli: Il modello TCP/IP combina alcune funzionalità che nell'OSI sono separate:
  - Il livello di Applicazione del TCP/IP include le funzionalità dei livelli di Applicazione, Presentazione e Sessione dell'OSI.
  - Il livello di Accesso alla Rete (o Collegamento Dati + Fisico) del TCP/IP include le funzionalità dei livelli di Collegamento Dati e Fisico dell'OSI.

In sintesi, anche se i nomi e il numero dei livelli cambiano, il principio fondamentale di una comunicazione stratificata, con interazioni verticali tra livelli adiacenti e comunicazioni logiche orizzontali tra livelli corrispondenti (peer-to-peer), rimane valido in entrambi i modelli.

Livello application

La realizzazione di un'applicazione web coinvolge solamente il livello più alto del protocollo (application)

Per le applicazioni a pagine è previsto il protocollo HTTP(S)

Per le applicazioni a servizi non esisteva un protocollo specifico e si è deciso di utilizzare ancora HTTP(S)

Cos'è HTTPS?

HTTPS sta per HyperText Transfer Protocol Secure. Non è un protocollo separato, ma è il risultato della sovrapposizione del protocollo HTTP a un livello di sicurezza aggiuntivo chiamato SSL/TLS. In pratica, è lo stesso protocollo HTTP, ma con tutte le comunicazioni tra client e server protette da crittografia.

Visualmente, un sito che usa HTTPS è immediatamente riconoscibile dall'icona a forma di lucchetto 🔒 nella barra degli indirizzi del browser e dal prefisso https:// nell'URL.

Il livello di sicurezza è garantito da due protocolli storicamente successivi:

- SSL (Secure Sockets Layer): La versione originale, oggi considerata obsoleta e insicura.
- TLS (Transport Layer Security): Il successore di SSL. È lo standard attuale, più robusto e sicuro. Sebbene oggi si usi quasi esclusivamente TLS, è comune usare ancora il termine "SSL" per indicare il certificato di sicurezza.

Le Colonne Portanti della Sicurezza: Crittografia, Autenticazione e Integrità

HTTPS fornisce tre garanzie di sicurezza fondamentali:

1. **Crittografia (Encryption)**: Rende i dati illeggibili a chiunque cerchi di intercettarli. Se un malintenzionato "ascolta" la comunicazione tra il tuo browser e il server, vedrà solo una sequenza di caratteri senza senso. Questo si ottiene tramite due tipi di crittografia che lavorano insieme:
    - **Crittografia Asimmetrica (a chiave pubblica/privata)**: Usata all'inizio della comunicazione per scambiare in modo sicuro la chiave di sessione. Il server ha una chiave pubblica (che tutti possono vedere) e una chiave privata (segreta). I dati crittografati con la chiave pubblica possono essere decifrati solo con la chiave privata corrispondente.
    - **Crittografia Simmetrica (a chiave condivisa)**: Molto più veloce di quella asimmetrica. Una volta che client e server hanno stabilito una chiave segreta condivisa (usando la crittografia asimmetrica), la usano per crittografare tutti i dati scambiati per il resto della sessione.
2. **Autenticazione (Authentication):** Garantisce che stai comunicando esattamente con il server a cui intendi connetterti (es. google.com) e non con un impostore. Questo processo si basa sul Certificato SSL/TLS.
3. **Integrità (Integrity) :** Assicura che i dati inviati non siano stati alterati o manomessi durante il transito. Ogni messaggio viene accompagnato da un "Message Authentication Code" (MAC), una sorta di firma digitale che permette al destinatario di verificare che il messaggio sia arrivato intatto.

Come Funziona: Il Certificato SSL/TLS e l'Handshake

Il processo che stabilisce una connessione sicura è chiamato TLS Handshake. Ecco una versione semplificata dei passaggi:

1. **Client Hello**: Il tuo browser (client) contatta il server e dice: "Ciao, vorrei stabilire una connessione sicura. Queste sono le versioni di TLS e gli algoritmi di crittografia che supporto".
2. **Server Hello**: Il server risponde: "Ciao. D'accordo, usiamo questa versione di TLS e questo algoritmo. Ecco il mio Certificato SSL/TLS e la mia chiave pubblica".
3. **Verifica del Certificato**: Il browser controlla il certificato del server. Un certificato SSL/TLS è come una carta d'identità digitale. Contiene informazioni sul proprietario del dominio ed è stato firmato digitalmente da un'entità fidata chiamata Certificate Authority (CA) (es. Let's Encrypt, DigiCert). Il browser verifica che:
    - Il certificato non sia scaduto.
    - Sia stato emesso per il dominio corretto.
    - La firma della CA sia valida (il browser ha un elenco preinstallato di CA attendibili).
4. **Scambio della Chiave di Sessione**: Se il certificato è valido, il browser si fida del server. A questo punto, il browser genera una chiave per la crittografia simmetrica (la "chiave di sessione"), la crittografa usando la chiave pubblica del server e la invia.
5. **Inizio della Sessione Sicura**: Solo il server, con la sua chiave privata, può decifrare il messaggio e ottenere la chiave di sessione. Ora, sia il client che il server possiedono la stessa chiave segreta. Da questo momento in poi, tutta la comunicazione avviene in modo rapido e sicuro tramite crittografia simmetrica.

Let's Encrypt e la Differenza con i Certificati Tradizionali

Ottenere un certificato TLS/SSL è il passo fondamentale per abilitare HTTPS. Storicamente, questo processo era manuale e costoso. L'arrivo di Let's Encrypt ha rivoluzionato questo panorama.

Cos'è Let's Encrypt?

Let's Encrypt è una Certificate Authority (CA) globale, non-profit, la cui missione è rendere il web un luogo più sicuro per tutti. Per raggiungere questo obiettivo, fornisce certificati TLS in modo gratuito e automatizzato. Il progetto è sostenuto da importanti organizzazioni come Electronic Frontier Foundation (EFF), Mozilla, Cisco, Google Chrome e altre.

Le caratteristiche chiave di Let's Encrypt sono:

- **Gratuito**: Chiunque possieda un nome di dominio può ottenere un certificato affidabile senza alcun costo.
- **Automatizzato**: L'intero processo di richiesta, convalida e rinnovo del certificato è pensato per essere gestito da software, senza intervento umano. Questo avviene tramite il protocollo ACME (Automated Certificate Management Environment).
- **Sicuro**: I certificati Let's Encrypt usano gli stessi standard crittografici dei certificati a pagamento, offrendo un livello di sicurezza identico per la cifratura dei dati.
- **Durata Breve**: I certificati hanno una validità di soli 90 giorni. Questo non è un difetto, ma una scelta di sicurezza: incoraggia l'automazione del rinnovo e riduce la finestra di danno in caso di compromissione di una chiave.

I Certificati Tradizionali a Pagamento

I certificati tradizionali sono emessi da **Certificate Authority (CA)** commerciali come DigiCert, Sectigo (ex Comodo), GlobalSign, ecc. Questi servizi offrono una gamma più ampia di prodotti e funzionalità rispetto a Let's Encrypt, solitamente dietro il pagamento di un canone annuale.

È fondamentale chiarire che, contrariamente a quanto si potrebbe pensare, i certificati SSL/TLS sono legati quasi esclusivamente ai nomi di dominio, non agli indirizzi IP. Lo scopo di un certificato è autenticare l'identità di un sito, e sul web l'identità è rappresentata dal nome di dominio (es. www.miosito.it). Il browser verifica che il nome nel certificato corrisponda a quello del sito visitato; in caso contrario, segnala un errore di sicurezza.

Legare i certificati agli IP sarebbe impraticabile a causa dell'hosting condiviso, dove un singolo IP ospita centinaia di siti. La tecnologia che rende possibile l'HTTPS in questi scenari è la Server Name Indication (SNI), un'estensione di TLS che permette al browser di comunicare al server il dominio a cui si vuole connettere, consentendo al server di presentare il certificato corretto.

I Livelli di Validazione dei Certificati: DV, OV ed EV

Quando una Certificate Authority (CA) emette un certificato, esegue un processo di verifica per assicurarsi che il richiedente sia chi dice di essere. Il livello di approfondimento di questa verifica determina il tipo di certificato.

1. Domain Validation (DV)

È il livello di validazione più basilare e comune. Il processo è completamente automatizzato e verifica solo che il richiedente abbia il controllo amministrativo del nome di dominio (ad esempio, dimostrando di poter ricevere email a admin@miodominio.it o di poter modificare i record DNS del dominio).

Perché si usa: È estremamente rapido (richiede pochi minuti), facile da ottenere e ideale per proteggere la comunicazione con la crittografia. Non fornisce alcuna informazione sull'identità del proprietario del sito.

Chi lo usa: È il tipo di certificato emesso da Let's Encrypt. È perfetto per blog, siti personali, piccole imprese e qualsiasi sito in cui non sia critica la verifica dell'identità legale dell'organizzazione.

2. Organization Validation (OV)

Questo livello include la verifica del dominio (DV), ma aggiunge un passo di verifica manuale da parte della CA. Un operatore umano controlla l'esistenza e la legittimità dell'organizzazione che richiede il certificato, consultando database aziendali e registri pubblici.

Cosa mostra: Il certificato OV contiene il nome verificato dell'organizzazione e la sua località (città, paese). Gli utenti possono visualizzare queste informazioni cliccando sul lucchetto nella barra degli indirizzi del browser.

Perché si usa: Per aumentare la fiducia degli utenti. Dimostra che il sito non è gestito da un'entità anonima, ma da un'organizzazione legale e registrata.

Chi lo usa: Aziende, siti di e-commerce e organizzazioni che vogliono un livello di fiducia superiore rispetto al semplice DV.

3. Extended Validation (EV)

È il livello di validazione più rigoroso e standardizzato. La CA esegue un processo di verifica molto approfondito, definito da linee guida severe, che include il controllo dello stato legale, operativo e fisico dell'organizzazione, nonché la verifica che la richiesta del certificato sia stata autorizzata dall'azienda stessa.

Cosa mostra: In passato, i browser premiavano i siti con certificati EV mostrando una vistosa "barra verde" con il nome dell'azienda direttamente nella barra degli indirizzi. Oggi, questa interfaccia è stata per lo più rimossa, ma le informazioni sull'identità legale dell'azienda rimangono le più dettagliate e facilmente accessibili cliccando sul lucchetto.

Perché si usa: Per offrire il massimo livello di fiducia e garanzia sull'identità del proprietario del sito. È un potente strumento contro il phishing, poiché un malintenzionato non potrebbe mai superare un processo di verifica così stringente.

Chi lo usa: Banche, istituzioni finanziarie, grandi piattaforme di e-commerce e agenzie governative, dove la fiducia dell'utente è assolutamente critica.

Confronto Punto per Punto

Costo

- Let's Encrypt: Totalmente gratuito.
- Tradizionali: A pagamento, con costi variabili in base al tipo di certificato e al fornitore.

Processo di Ottenimento e Automazione

- Let's Encrypt: Progettato per essere completamente automatico tramite il protocollo ACME. Un client software gestisce richiesta, convalida e rinnovo.
- Tradizionali: Processo spesso manuale o semi-manuale che richiede la generazione di un CSR, l'invio alla CA e l'installazione manuale.

Periodo di Validità

- Let's Encrypt: 90 giorni, una scelta che incentiva l'automazione e aumenta la sicurezza.
- Tradizionali: Tipicamente 1 anno, riducendo la frequenza dei rinnovi manuali.

Livelli di Validazione

- Let's Encrypt: Offre solo la Domain Validation (DV), una verifica automatica che attesta unicamente il controllo del dominio.
- Tradizionali: Offrono anche la Organization Validation (OV), che verifica l'identità legale dell'azienda, e la Extended Validation (EV), un processo di verifica ancora più rigoroso.

Garanzia e Supporto Tecnico

- Let's Encrypt: Non offre garanzie finanziarie e il supporto è gestito dalla comunità tramite forum e documentazione.
- Tradizionali: Offrono garanzie monetarie in caso di emissione errata e forniscono supporto tecnico dedicato (telefono, chat, email).

Quando Scegliere Quale?

Scegli Let's Encrypt se:

- Gestisci un sito personale, un blog, un portfolio o il sito di una piccola impresa.
- Devi proteggere API, servizi backend o ambienti di sviluppo/staging.
- Apprezzi l'automazione e vuoi eliminare la gestione manuale dei certificati.
- Il tuo budget è una considerazione prioritaria.

Considera un Certificato Tradizionale (OV/EV) se:

- Gestisci un grande sito di e-commerce, una piattaforma di pagamenti, un sito bancario o di un'istituzione governativa.
- La fiducia dell'utente, basata sulla verifica dell'identità legale della tua azienda, è un fattore critico di business.
- Hai bisogno di una garanzia finanziaria che protegga te e i tuoi clienti.
- Necessiti di accesso a un supporto tecnico dedicato.

Perché HTTPS è Fondamentale per le API REST?

Nel contesto delle API REST, usare HTTP semplice è un errore di sicurezza gravissimo. L'uso di HTTPS non è opzionale, è un requisito indispensabile.

- **Protezione dei Dati Sensibili:** Le API trasportano dati. Che si tratti di dati personali degli utenti, credenziali di accesso, informazioni di pagamento o segreti aziendali, questi viaggiano nel body o nei parametri della richiesta/risposta. Senza HTTPS, sarebbero in chiaro, facilmente intercettabili.
- **Sicurezza dei Token di Autenticazione**: Molte API REST usano token (es. Bearer Token, JWT) inviati nell'header Authorization per autenticare le richieste. Se un token venisse intercettato su una connessione HTTP, un malintenzionato potrebbe usarlo per impersonare l'utente e accedere senza restrizioni alle sue risorse.
- **Prevenzione di Attacchi "Man-in-the-Middle" (MITM)**: Con HTTP, un attaccante potrebbe posizionarsi tra il client e il server, non solo per leggere ma anche per modificare le richieste e le risposte. Potrebbe, ad esempio, cambiare l'importo di una transazione finanziaria o iniettare dati malevoli nella risposta del server. HTTPS rende questo tipo di attacco quasi impossibile.
- **Affidabilità e Professionalità**: Esporre un'API su HTTP denota una mancanza di attenzione alla sicurezza. Molti client (specialmente le applicazioni mobile e i browser moderni) bloccano di default le richieste a risorse non sicure (chiamate mixed content), rendendo di fatto l'API inutilizzabile.

Protocollo HTTP: descrizione

Il server offre servizi ed è in attesa del client

Il client chiede un servizio indicando:

- **URL** -> protocollo, ip, porta e path verso la risorsa da agganciare

  http://IP:port/path/resource

- **VERB** -> sono parole codificate dal protocollo che servono a indicare il tipo di azione che si vuole compiere sulla risorsa indicata dalla URL

  In ambito REST si considerano solo le seguenti:

  - GET -> lettura
  - POST -> inserimento
  - PUT -> modifica totale
  - PATCH -> modifica parziale
  - DELETE -> cancellazione

Trasporto dei dati

Request e Response hanno un header e un body

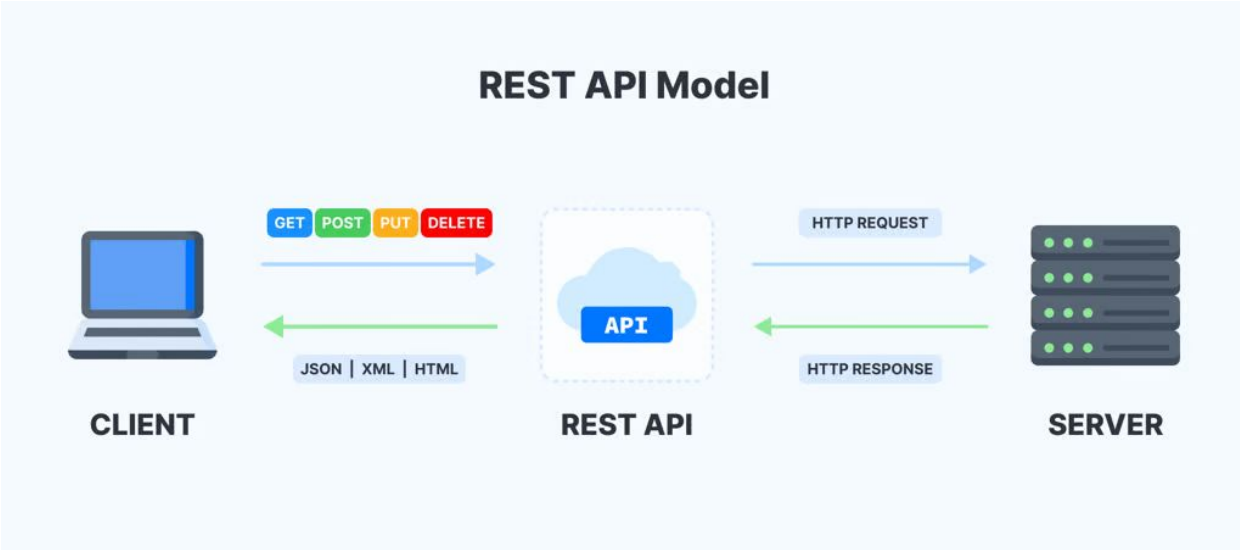
- Le informazioni dell'header sono meta dati e sono strutturate
- Il body invece non segue regole (in passato si usava XML, in ambito REST si usa il formato JSON)

Il client ha 3 modi di inviare dati al server:

- Utilizzando la query string che si accoda alla URL
  - http://ip:port/path/resource?var1=aaa&var2=bbb
- Inserendoli direttamente nella URL
  - http://ip:port/path/resource/aaa/bbb
- Inserendoli nel body della richiesta

Il server invece ha un solo modo di inviare dati:

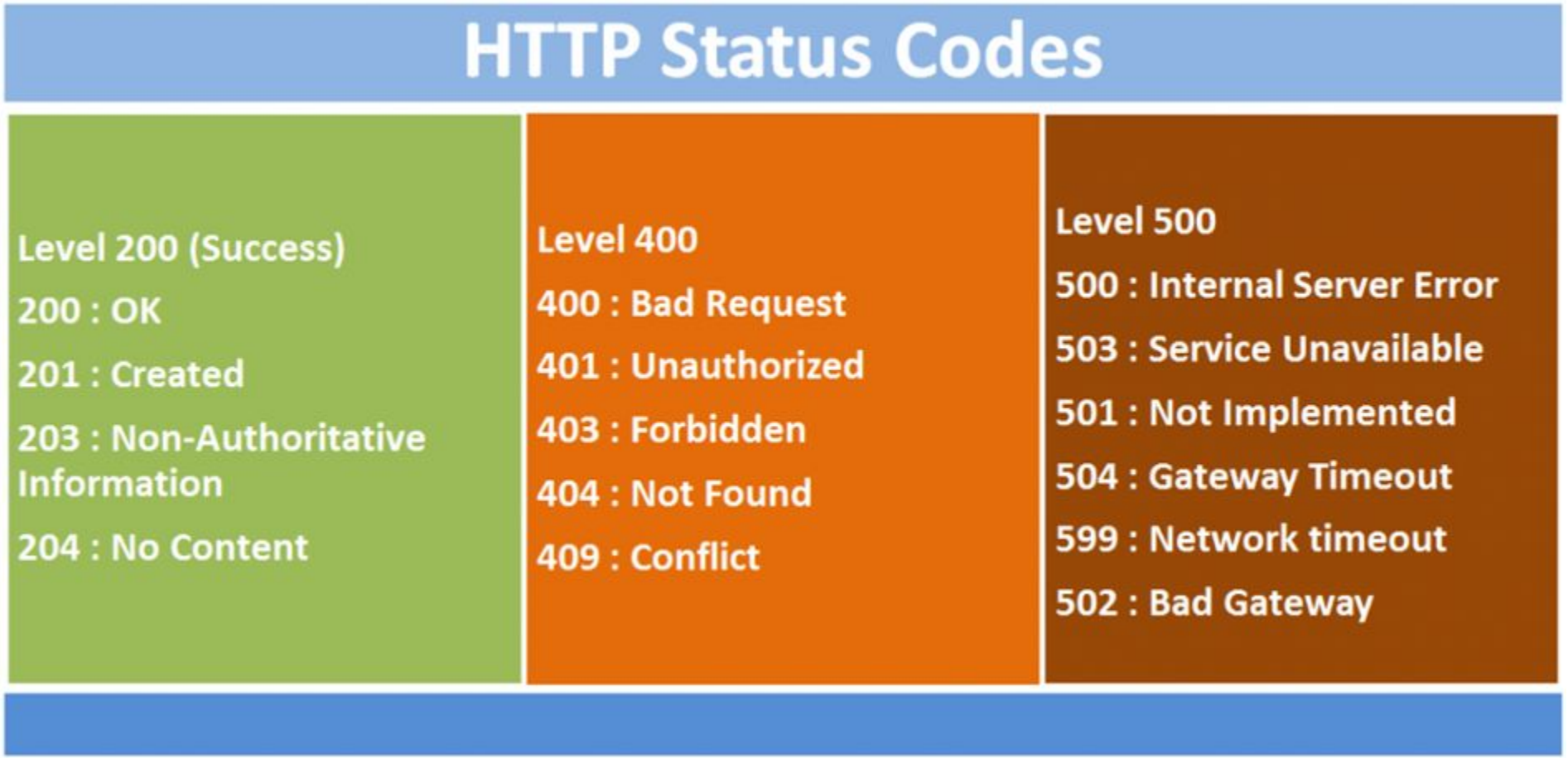
- Inserendoli nel body della risposta



Status code

Il server è tenuto a restituire uno status code per ogni richiesta col quale indicherà l'esisto dell'operazione.

I codici vanno per gruppi di 100



Il formato json 1/2

Il formato JSON è simile all'XML, ma molto più leggero

Questo è un oggetto rappresentato in formato JSON

Le proprietà sono racchiuse da {…} e separate da virgole.

Sono indicate con

nome (stringa) : valore (in base al tipo di variabile)

```
{
    "idUtente": 444,
    "nome": "nome4",
    "cognome": "cognome4",
    "mail": "mail4@xx.it",
    "telefono": "444.444"
}
```

Il valore può essere numerico, testo oppure un oggetto strutturato in json

- Sia client che server possono inviare dati in formato JSON e devono specificarlo nel content-type dell'header (della richiesta/risposta) in questo modo:

  content-type='application/JSON'

Il formato json 2/2

Se il client esegue una richiesta per la quale si aspetta dati in formato JSON dovrà dichiarare nell'header della richiesta:

accept = 'application/JSON'

Di conseguenza il server dovrà rispondere inviando dati in questo formato.

Se il client indica diverse opzioni di formato e il server le supporta, allora può sceglierne una tra quelle indicate

API - Application Programming Interface

API (Application Programming Interface)

Un'API (Application Programming Interface), o Interfaccia di Programmazione di un'Applicazione, è un **intermediario** che stabilisce un **insieme di regole precise** affinché due componenti software possano comunicare e interagire tra loro.

Pensa a un'API come a **un contratto o a un confine ben definito**. Questo contratto specifica quali richieste si possono fare, come formularle e quali risposte aspettarsi. Il suo scopo principale è permettere la collaborazione tra software diversi, nascondendo la complessità interna di ciascuno.

Questo principio si manifesta in due contesti principali, entrambi fondamentali nel mondo dello sviluppo.

1. L'API come Ponte tra Programmi Diversi

Questo è l'uso più comune del termine oggi e si riferisce alla comunicazione tra applicazioni separate, che spesso si trovano su computer diversi e comunicano attraverso una rete come Internet.

L'analogia più efficace per capire cos'è un'API è quella del ristorante:

- **Tu (il Cliente)**: Sei l'applicazione che ha bisogno di un servizio (ad esempio, l'app sul tuo telefono).
- **La Cucina (il Server)**: È il sistema che possiede i dati e le funzionalità (ad esempio, i server di Facebook o di un sito meteo).
- Il **Cameriere (l'API)**: È l'intermediario. Tu non vai direttamente in cucina a prendere il cibo. Chiami il cameriere, gli fai un ordine preciso seguendo il menu (le regole dell'API), e lui porta la tua richiesta in cucina. Poi, torna con il piatto pronto (la risposta del server).

L'API (il cameriere) si assicura che la comunicazione avvenga in modo standardizzato e controllato, senza che tu debba conoscere i dettagli complessi di come funziona la cucina.

1. L'API come Ponte tra Programmi Diversi

Non vai direttamente in cucina per prendere il cibo. Chiami il cameriere (l'API), consulti il menu (le regole e la documentazione dell'API) e fai un ordine. Il cameriere porta la tua richiesta alla cucina e ti riporta il piatto pronto (i dati che hai richiesto).

In questo ruolo, l'API agisce come un messaggero sicuro, permettendo all'app di usare funzionalità o dati di un server remoto.

Esempi di questo tipo di API:

- **Login con Google**: Un sito web usa l'API di Google per verificare la tua identità senza mai vedere la tua password.
- **App Meteo**: Usa l'API di un servizio meteorologico per recuperare e mostrarti le previsioni.

2. L'API come "Manuale d'Istruzioni" di un Software

Il concetto di API è più ampio e non si limita alla comunicazione di rete. Un'API definisce anche come uno sviluppatore può usare le funzionalità di una libreria, di un framework o del sistema operativo all'interno del proprio programma.

L'API è come la "cassetta degli attrezzi" che un software mette a disposizione. L'API è sia il manuale di istruzioni che il design degli attrezzi stessi: spiega quali strumenti si hanno, come si chiamano e come usarli correttamente per costruire qualcosa.

Esempio pratico: La libreria React

La libreria React offre allo sviluppatore due "interfacce" (due API) principali e distinte per scrivere i componenti:

- **Componenti a Classe (Class Components)**: L'API "classica". Richiede di creare una classe JavaScript che estende React.Component e di usare metodi specifici come render() per la visualizzazione e componentDidMount() per la logica del ciclo di vita. Lo stato è gestito tramite this.state e this.setState().
- **Componenti Funzionali con Hooks (Functional Components)**: L'API "moderna". Consente di scrivere componenti come semplici funzioni JavaScript. Per aggiungere stato o altre funzionalità si usano delle funzioni speciali chiamate Hooks, come useState() o useEffect().

Entrambe le modalità permettono di creare componenti, ma rappresentano due API diverse. Sono due "manuali" distinti per interagire con le funzionalità interne di React e comandarle, senza che lo sviluppatore debba conoscere la loro complessa implementazione.

Un altro esempio quotidiano è l'API del DOM, che i browser forniscono per permetterci di manipolare le pagine web con JavaScript, usando comandi come document.getElementById().

Il Principio Fondamentale che le Unisce: L'Astrazione

Che si tratti di un "cameriere" per un servizio web o di un "manuale" per una libreria, il ruolo chiave di un'**API è fornire un livello di astrazione**.

L'API nasconde la complessità.

Grazie all'API, non hai bisogno di conoscere la complicata architettura dei server di Google per usare le sue mappe, né devi studiare il codice sorgente di Vue per creare un'interfaccia reattiva. Devi solo conoscere e rispettare le regole del "contratto" che l'API ti offre.

In sintesi, un'API è una promessa fondamentale nel mondo del software: "Se interagisci con me seguendo queste regole, io ti garantirò un risultato specifico, senza che tu debba preoccuparti di come lo ottengo."

Vantaggi delle API

Le API non sono solo un meccanismo tecnico, ma un approccio strategico allo sviluppo del software che offre vantaggi significativi.

- **Riutilizzo (Reusability)**: Una volta che una funzionalità è stata sviluppata e resa accessibile tramite un'API (ad esempio, un sistema di pagamento o un servizio di geolocalizzazione), può essere riutilizzata da innumerevoli altre applicazioni. Questo elimina la necessità di "reinventare la ruota", riducendo drasticamente i tempi e i costi di sviluppo.
- **Modularità (Modularity)**: Le API incoraggiano a costruire sistemi software come un insieme di moduli indipendenti e intercambiabili (microservizi). Ogni modulo ha un compito specifico e comunica con gli altri tramite API ben definite. Questo rende i sistemi più facili da mantenere, aggiornare e scalare, poiché si può intervenire su un singolo modulo senza "rompere" l'intera applicazione.
- **Sviluppo Parallelo (Parallel Development)**: Poiché l'API funge da "contratto" tra diverse parti di un sistema, i team possono lavorare in parallelo. Ad esempio, il team che sviluppa l'interfaccia utente (frontend) e il team che si occupa della logica del server (backend) possono iniziare a lavorare contemporaneamente. Al team frontend basta sapere come sarà fatta l'API (il contratto) per poter procedere, senza dover attendere che il backend sia completamente finito.

Tipi di API

Le API possono essere classificate in base a diversi criteri, come la tecnologia che usano, il loro modello di comunicazione o chi può accedervi.

API Web (HTTP-based)

Questa è la categoria più comune e si riferisce a tutte le API che usano il protocollo HTTP per la comunicazione, proprio come fanno i browser per navigare sui siti web. Sfruttano gli standard di Internet per permettere l'interazione tra sistemi diversi. All'interno di questa famiglia troviamo diversi "stili" architettonici.

REST vs SOAP vs GraphQL:

- **SOAP (Simple Object Access Protocol)**: È uno standard più vecchio e rigido. Utilizza principalmente il formato XML per i messaggi e ha regole severe su come devono essere strutturati. È noto per la sua robustezza e per le funzionalità di sicurezza integrate, ma è anche più complesso e "verboso" (produce messaggi più pesanti).
- **REST (REpresentational State Transfer):** È uno stile architettonico, non uno standard rigido. È più flessibile di SOAP e usa i metodi standard di HTTP (GET, POST, DELETE, etc.) in modo intuitivo. Solitamente utilizza JSON per i dati, che è più leggero e facile da leggere dell'XML. È lo stile dominante per la maggior parte delle API web moderne.
- **GraphQL**: È un linguaggio di query per API sviluppato da Facebook. La sua caratteristica principale è che permette al client di richiedere esattamente i dati di cui ha bisogno, e nient'altro. A differenza di REST, che ha molti endpoint, GraphQL espone tipicamente un unico endpoint. Il client invia una "query" che descrive la struttura dei dati desiderati, evitando di ricevere dati inutili (under-fetching) o di dover fare più chiamate per ottenere tutto ciò che serve (over-fetching).

API Sincrone vs Asincrone

Questa distinzione riguarda il modo in cui il client attende una risposta.

- **API Sincrone**: Funzionano secondo un modello di richiesta-risposta bloccante. Il client invia una richiesta e attende attivamente che il server completi l'operazione e restituisca una risposta. La maggior parte delle API REST sono sincrone. L'applicazione si "ferma" finché non ha ricevuto l'esito.

- **API Asincrone**: Il client invia una richiesta e non rimane in attesa bloccante. Il server accetta subito la richiesta (rispondendo, ad esempio, "OK, ho preso in carico il tuo lavoro") e la elabora in background. Quando l'operazione è completata, il server notifica al client il risultato, spesso usando un meccanismo chiamato "webhook" o "callback". Questo modello è ideale per operazioni lunghe, come l'elaborazione di un video o la generazione di un report complesso.

API Pubbliche vs Private

Questa classificazione si basa sull'accessibilità.

API Pubbliche (o Esterne): Sono aperte a chiunque voglia utilizzarle. Sono pensate per permettere a sviluppatori esterni di integrare i dati o le funzionalità di un'azienda nelle proprie applicazioni (es. l'API di Google Maps o di Twitter).

API Private (o Interne): Sono utilizzate solo all'interno di una specifica organizzazione. Servono a far comunicare tra loro i diversi sistemi o microservizi interni di un'azienda in modo efficiente e standardizzato. Anche se non sono esposte all'esterno, seguono gli stessi principi di progettazione delle API pubbliche.

API Pubbliche vs Private

Questa classificazione si basa sull'accessibilità.

API Pubbliche (o Esterne): Sono aperte a chiunque voglia utilizzarle. Sono pensate per permettere a sviluppatori esterni di integrare i dati o le funzionalità di un'azienda nelle proprie applicazioni (es. l'API di Google Maps o di Twitter).

API Private (o Interne): Sono utilizzate solo all'interno di una specifica organizzazione. Servono a far comunicare tra loro i diversi sistemi o microservizi interni di un'azienda in modo efficiente e standardizzato. Anche se non sono esposte all'esterno, seguono gli stessi principi di progettazione delle API pubbliche.

Endpoint

Indipendentemente dal tipo, tutte le API web condividono alcuni concetti di base.

Endpoint

L'endpoint è il punto di accesso specifico di un'API. È un URL che identifica in modo univoco una risorsa o una collezione di risorse. Funziona esattamente come l'indirizzo di una casa: indica al client dove deve inviare la sua richiesta per ottenere una specifica informazione.

Esempi:

https://api.example.com/users -> L'indirizzo per accedere alla lista di tutti gli utenti.

https://api.example.com/products/123 -> L'indirizzo per accedere al prodotto specifico con ID 123.

Payload

Il payload (o "carico utile") rappresenta i dati effettivi che vengono trasportati in una richiesta o in una risposta API. È il contenuto della "lettera" che viene spedita all'indirizzo indicato dall'endpoint.

Request Payload: Sono i dati che il client invia al server, tipicamente nel "corpo" (body) di una richiesta POST o PUT. Ad esempio, quando si crea un nuovo utente, il payload della richiesta conterrà i dati del nuovo utente (nome, email, etc.).

Response Payload: Sono i dati che il server restituisce al client. Ad esempio, dopo una richiesta GET a /users, il payload della risposta sarà la lista degli utenti.

Formati comuni del payload:

- JSON (JavaScript Object Notation): Il formato più comune per le API REST grazie alla sua leggerezza e leggibilità.
- XML (eXtensible Markup Language): Usato principalmente dalle API SOAP.
- Form-data: Il formato usato dai browser per inviare i dati dei moduli HTML.

Sistemi REST

Sistemi REST

REST è uno stile architetturale formato da vincoli, linee guida e best practice che si utilizza per la realizzazione di sistemi distribuiti.

Punto cruciale: REST non è un'architettura

- Non è una tecnologia specifica
- Non è un protocollo come HTTP o SOAP
- Non è un framework o una libreria
- **È un insieme di principi di design che guidano la progettazione**

L'obiettivo è quello di regolamentare lo sviluppo di sistemi complessi che altrimenti evolverebbero in maniera "caotica". Senza questi principi, ogni sviluppatore implementerebbe API in modo diverso, creando inconsistenze e difficoltà di integrazione.

Un sistema che si attiene a tali vincoli prende il nome di sistema RESTful.

Le Origini: Roy Fielding e la Rivoluzione del 2000

L'acronimo REST - REpresentational State Transfer ("trasferimento dello stato di rappresentazione") - deriva dalla tesi di dottorato di Roy Fielding intitolata "Architectural Styles and the Design of Network-based Software Architectures" (Stili architetturali e progettazione di architetture software basate sul networking).

La tesi risale all'anno 2000 - un momento cruciale nella storia del web:

- Internet stava esplodendo commercialmente
- I sistemi distribuiti diventavano sempre più complessi
- Le tecnologie esistenti (come CORBA e RPC) erano pesanti e difficili da usare
- Il web stesso stava dimostrando il potere della semplicità

Le Origini: Roy Fielding e la Rivoluzione del 2000

Chi è Roy Fielding?

Roy Fielding non era un teorico isolato, ma uno degli architetti fondamentali del web moderno:

- Co-fondatore del progetto Apache HTTP Server (il web server più usato al mondo)
- Principale autore delle specifiche HTTP/1.0 e HTTP/1.1
- Co-fondatore della Apache Software Foundation
- Membro del W3C (World Wide Web Consortium)

La sua posizione unica gli permetteva di vedere sia i problemi pratici dello sviluppo web sia le necessità teoriche per sistemi scalabili.

Il Contesto Storico: Perché REST Era Necessario

Prima del 2000: Il Caos delle Architetture Distribuite

- CORBA (Common Object Request Broker Architecture): Complesso, pesante, difficile da debuggare
- RPC (Remote Procedure Call): Tentava di nascondere la natura distribuita, causando problemi di affidabilità
- SOAP (Simple Object Access Protocol): Ironicamente non così "semplice", con envelope XML complessi
- Soluzioni proprietarie: Ogni azienda inventava il proprio protocollo

I Problemi che REST Doveva Risolvere

- Complessità eccessiva: I sistemi esistenti richiedevano conoscenze specialistiche
- Accoppiamento stretto: Client e server erano troppo dipendenti l'uno dall'altro
- Scarsa scalabilità: I protocolli non erano progettati per il volume del web
- Mancanza di interoperabilità: Sistemi diversi non riuscivano a comunicare facilmente

Le Motivazioni Fondamentali di REST

1. Semplicità

Il principio guida di REST è eliminare la complessità superflua. Invece di inventare un nuovo protocollo di comunicazione, REST sfrutta l'HTTP, il protocollo che già fa funzionare il web. Questo ha reso tutto più semplice per diversi motivi:

- **Curva di apprendimento ridotta**: Gli sviluppatori web non dovevano studiare nuovi standard complessi, perché conoscevano già i verbi HTTP (GET, POST, etc.) e i codici di stato (200, 404, etc.).
- **Niente "buste" complicate**: A differenza di protocolli come SOAP, che avvolgono ogni messaggio in una struttura XML complessa (l'envelope), una richiesta REST è diretta e leggibile.
- **Debugging facile**: Le richieste REST sono semplici messaggi di testo che possono essere facilmente ispezionati con gli strumenti di sviluppo di qualsiasi browser, rendendo più rapida l'individuazione di errori.

Per esempio REST usa una semplice richiesta GET a un URL intuitivo, mentre SOAP richiede la costruzione di un intero documento XML.

Le Motivazioni Fondamentali di REST

2. Scalabilità

All'epoca della sua concezione, il web stava crescendo in modo esponenziale. Serviva un'architettura in grado di gestire un numero enorme di utenti senza collassare. REST è progettato per questo grazie a principi chiave:

- **Stateless (senza stato)**: Ogni richiesta contiene tutte le informazioni necessarie per essere elaborata. Il server non ha bisogno di "ricordarsi" nulla del client tra una richiesta e l'altra. Questo permette di distribuire il carico su molti server diversi (scalabilità orizzontale), perché qualsiasi server può gestire qualsiasi richiesta.
- **Caching**: REST sfrutta i meccanismi di cache del protocollo HTTP. Le risposte possono essere salvate temporaneamente (in cache) dal client o da nodi intermedi, riducendo il numero di richieste dirette al server e migliorando drasticamente le prestazioni.
- **Sistema a livelli (Layered system)**: È possibile inserire tra client e server vari livelli intermedi (come proxy o load balancer) per gestire sicurezza, bilanciamento del carico o caching, senza che il client o il server se ne accorgano.

Le Motivazioni Fondamentali di REST

3. Indipendenza

REST promuove l'interoperabilità e il disaccoppiamento tra i componenti di un sistema.

- **Indipendenza di piattaforma e linguaggio**: Poiché REST si basa sullo standard universale HTTP, un client scritto in JavaScript su un browser può comunicare senza problemi con un server scritto in Python, Java o qualsiasi altro linguaggio, e viceversa. La tecnologia sottostante diventa irrilevante.
- **Evoluzione indipendente**: Il team che sviluppa il client e quello che sviluppa il server possono lavorare e rilasciare aggiornamenti in modo indipendente. Finché il "contratto" dell'API non viene rotto, il client può evolvere senza temere di rompere il server, e il server può essere aggiornato senza costringere tutti i client a modificarsi.

Vincoli: stateless e risorse

Questi "vincoli" non sono regole arbitrarie, ma principi di progettazione che, se seguiti, garantiscono che un'architettura sia scalabile, semplice e affidabile, proprio come il web stesso.

1. Il Vincolo "Stateless" (Senza Stato)

Questo è uno dei vincoli più importanti e significa che ogni richiesta inviata dal client al server deve contenere tutte le informazioni necessarie affinché il server possa comprenderla ed elaborarla.

Nessuna Sessione sul Server: **Il server non memorizza alcuna informazione sullo stato del client tra una richiesta e l'altra**. Non sa chi sei o cosa hai fatto prima. Ogni richiesta è un evento isolato e indipendente. Se hai bisogno di autenticazione, ad esempio, il client deve inviare le credenziali (come un token di autenticazione) in ogni singola richiesta.

Perché è fondamentale?

- **Scalabilità**: Poiché il server non deve gestire sessioni, qualsiasi richiesta può essere inviata a qualsiasi server. Questo rende molto più semplice distribuire il carico su più macchine e aggiungere nuovi server man mano che il traffico aumenta.
- **Affidabilità**: Se una richiesta fallisce, il client può semplicemente inviarla di nuovo, magari a un altro server, senza che ci siano problemi di "sessioni interrotte".
- **Semplicità**: La logica del server è più semplice perché non deve preoccuparsi di creare, gestire e distruggere le sessioni degli utenti.

Vincoli: stateless e risorse

2. L'Interazione basata su Risorse

Questo punto è strettamente legato al **vincolo di Interfaccia Uniforme**, che impone che la comunicazione avvenga in modo standardizzato. La parte cruciale di questo vincolo è che tutte le operazioni ruotano attorno al concetto di **risorsa**.

Tutto è una Risorsa: In REST, **ogni "cosa" con cui si vuole interagire è una risorsa**: un utente, un prodotto, un articolo, una foto. Ogni risorsa è identificata da un indirizzo univoco (URL), come /utenti/123 o /prodotti/45.

Manipolare Risorse, non Produrle Sempre: Il punto chiave non è tanto che ogni richiesta debba produrre una risorsa, ma che ogni richiesta opera su una risorsa usando i verbi standard del protocollo HTTP.

- GET /utenti/123: **Recupera** la rappresentazione della risorsa "utente 123".
- POST /utenti: **Crea** una nuova risorsa "utente". (Qui sì che si produce una risorsa).
- PUT /utenti/123: **Aggiorna** o sostituisce completamente la risorsa "utente 123".
- DELETE /utenti/123: **Elimina** la risorsa "utente 123".

L'idea fondamentale è che l'interfaccia è prevedibile e coerente perché si basa sulla manipolazione di "oggetti" (le risorse) tramite un set di operazioni standard (i verbi HTTP).

Architettura Stateless (Senza Stato)

In questa architettura, ogni richiesta inviata dal client al server deve contenere tutte le informazioni necessarie per essere compresa ed elaborata. Il server non conserva alcuna memoria delle interazioni precedenti.

Analogia: Pizzeria con camerieri intercambiabili

Immagina una pizzeria dove tutti i camerieri usano un sistema centralizzato (come un tablet) per gestire gli ordini.

- **Vantaggio**: Qualunque cameriere può servire qualunque tavolo in qualsiasi momento, perché lo stato dell'ordine (cosa è stato ordinato, cosa è stato servito) è nel sistema centrale e non nella memoria di un singolo cameriere. Se ci sono molti clienti, il lavoro si distribuisce facilmente tra tutti i camerieri disponibili.
- **Svantaggio**: Nessun cameriere conosce personalmente i clienti o le loro preferenze, perché ogni interazione è un evento a sé stante.

Vantaggi Chiave:

- **Scalabilità**: È il vantaggio più grande. Se il numero di richieste aumenta, si possono aggiungere nuovi server senza problemi. Poiché nessuna informazione è legata a un server specifico, qualsiasi server può gestire qualsiasi richiesta.
- **Affidabilità**: Se un server si guasta, la richiesta può essere semplicemente inviata a un altro server funzionante senza che l'utente se ne accorga.
- **Visibilità**: Ogni richiesta è autonoma e comprensibile senza contesto, il che semplifica il monitoraggio e il debugging.

Svantaggi Chiave:

- Prestazioni: Il client deve inviare informazioni aggiuntive (come token di autenticazione) in ogni richiesta, aumentando leggermente la quantità di dati trasferiti.

Architettura Stateful (Con Stato)

In questa architettura, il server crea e mantiene una sessione per ogni client. Il server "si ricorda" chi è il client e cosa ha fatto nelle richieste precedenti.

Analogia: Pizzeria con camerieri dedicati

Immagina una pizzeria di lusso dove ogni cliente ha un proprio cameriere dedicato per tutta la durata della cena.

- **Vantaggio**: Il cameriere conosce i dettagli, le preferenze del cliente ("il solito?") e lo stato esatto del suo ordine, offrendo un servizio personalizzato e veloce.
- **Svantaggio**: Le modifiche all'ordine diventano onerose se non si passa dal cameriere assegnato. Se arrivano molti clienti contemporaneamente, il carico di lavoro non si distribuisce facilmente, perché ogni cameriere è legato ai propri tavoli.

Vantaggi Chiave:

- **Performance**: Le richieste possono essere più leggere perché il server conosce già il contesto e lo stato della sessione.
- **Esperienza Utente**: Permette un'interazione più fluida e contestualizzata, simile a una conversazione continua.

Svantaggi Chiave:

- **Scalabilità Limitata**: È molto più difficile scalare. Se un client è legato a un server specifico, non si può semplicemente reindirizzare la sua prossima richiesta a un altro server, perché quest'ultimo non conoscerebbe lo stato della sessione.
- **Complessità e Costi**: La gestione delle sessioni aumenta la complessità del server. Aggiungere nuovi server è più costoso e complicato.
- **Fragilità**: Se il server che mantiene la sessione si guasta, tutte le informazioni di stato del client vengono perse, costringendolo a ricominciare da capo.

Perché servizi stateless?

Perché c'è un vantaggio economico!

Nei servizi statefull le varie chiamate usavano la sessione sul server per salvare dati parziali e la sessione è molto costosa!

Nei servizi REST la sessione non si usa (nel senso che non è necessaria)

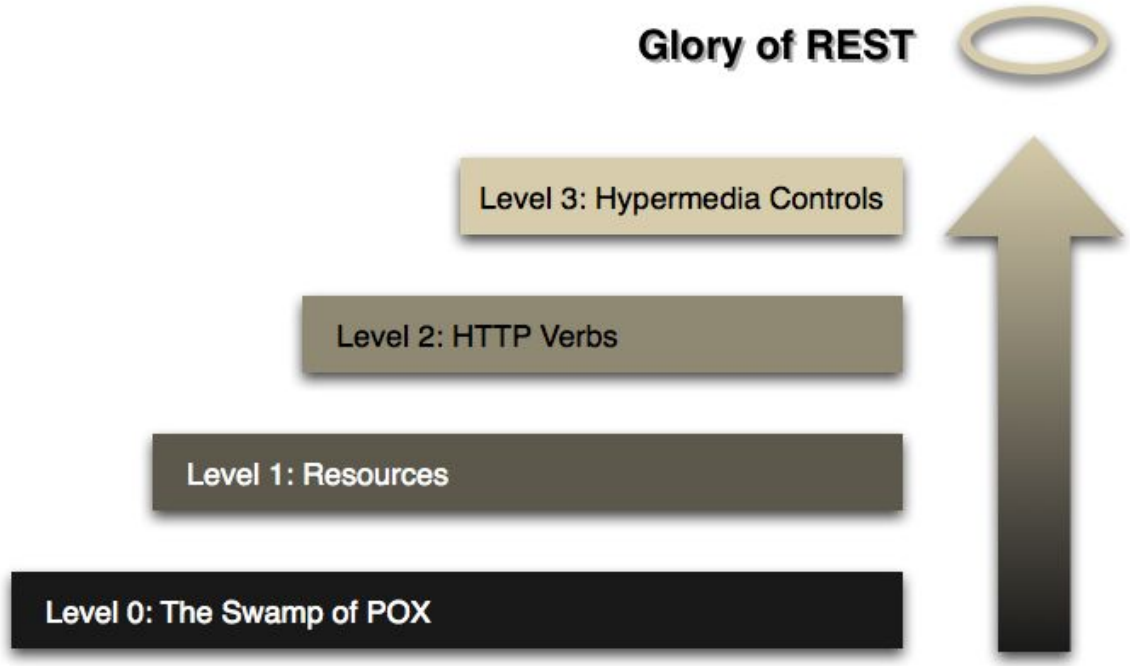
I servizi sono stateless e la gestione della conversazione viene spostata sul client (che è un programma su una macchina)

- Le azioni 'su più step' usano la memoria del client e, solo alla fine, parte la chiamata al server (come chiamata stateless)
- Chi scrive il servizio (sul server) è totalmente disaccoppiato dal client

Vantaggio economico + disaccoppiamento Client/Server

La scala di livelli

- Fielding teorizzò una scala di livelli di compatibilità allo stile REST.
- Livello zero(il "Far West "): sistemi a basso grado di maturità in cui tutte le risorse tecnologiche sono disponibili, e tutti gli stili sono ammessi
- Livello tre: stato di totale maturità. Sistemi che rispettano i vincoli e dunque possono chiamarsi RESTful



I livelli 1 e 2 si realizzano sempre, mentre per il livello 3 si può valutare se/come realizzarlo

Quanto visto è una rappresentazione della Scala di Maturità di Richardson (Richardson Maturity Model), un modello che classifica le **API web** in base a quanto aderiscono ai principi dello stile REST.

Non si tratta di una "legge", ma di un modo utile per capire i diversi gradi di maturità di un'API.

Approfondiamo ogni livello.

Livello 0: La Palude del POX (The Swamp of POX)

Questo è il punto di partenza, il livello più basso di maturità. "POX" sta per Plain Old XML (ma oggi si potrebbe dire anche Plain Old JSON).

- Come funziona: A questo livello, si usa l'HTTP come un semplice "tunnel" per eseguire chiamate a procedure remote (RPC). Tipicamente, esiste un unico endpoint (un solo URL) e tutte le operazioni, che siano di lettura, scrittura o cancellazione, vengono inviate tramite il metodo POST.
- Significato: L'API non sfrutta quasi nessuna delle caratteristiche del web. È solo un modo per far parlare due sistemi, ma manca di struttura e chiarezza. È definita una "palude" perché è un approccio poco strutturato e potenzialmente confusionario.

I sistemi POX:

- Scambiano messaggi in modo sincrono attraverso il protocollo HTTP
- Non possiedono il concetto di risorsa (ma di funzione)
- Non sfruttano i verbi HTTP (tipicamente agiscono solo con GET e POST)

Esempio: Gestione Utenti a Livello 0

Immaginiamo di dover gestire degli utenti. Un'API di Livello 0 avrebbe un unico endpoint per tutte le operazioni, ad esempio: https://api.example.com/userService.

1. Richiedere i Dati di un Utente

Anche se stiamo solo leggendo dati, usiamo POST. L'azione specifica ("getUser") è scritta nel **payload**.

```
POST /userService
Host: api.example.com
Content-Type: application/json

{
  "action": "getUser",
  "userId": 123
}
```

2. Creare un Nuovo Utente

La richiesta è identica alla precedente nella forma, cambia solo il contenuto del payload.

```
POST /userService
Host: api.example.com
Content-Type: application/json

{
  "action": "createUser",
  "userData": {
    "name": "Mario Rossi",
    "email": "mario.rossi@example.com"
  }
}
```

3. Eliminare un Utente

Ancora una volta, usiamo POST allo stesso URL, specificando l'azione nel corpo.

```
POST /userService
Host: api.example.com
Content-Type: application/json

{
  "action": "deleteUser",
  "userId": 456
}
```

Livello 1: Risorse (Resources)

Questo è il primo, fondamentale passo verso un'architettura REST per superare la "palude" del Livello 0 e organizzare l'intero servizio attorno al concetto di risorsa.

- **Come funziona**: Si smette di usare un singolo URL per tutto e si inizia a modellare il sistema attorno al concetto di risorsa. Ogni **risorsa** (un utente, un prodotto, un ordine) ottiene un proprio identificatore univoco (URL). Anziché fare una richiesta POST a /servizio con dentro le istruzioni, si interagisce direttamente con gli indirizzi delle risorse: /utenti/123 o /prodotti/45.

- **Significato**: Si introduce una struttura chiara e gerarchica. L'API inizia a essere più intuitiva e leggibile, perché gli indirizzi stessi descrivono su cosa si sta operando.

Livello 1: Risorse (Resources)

Una **risorsa** è qualunque "cosa" o "oggetto" del nostro dominio che vogliamo esporre e rendere accessibile sul web: un libro, un utente, le previsioni meteo, un corso universitario.

È l'elemento centrale di un'architettura REST.

Esempi

- un libro venduto on-line,
- le previsioni meteo di Londra,
- un item oggetto di un’asta eBay,
- i dati identificativi di un volo aereo,
- un valore di cambio valuta,
- i prezzi di un prodotto finanziario e le informazioni relative allo stesso,
- la spiegazione di un termine tratta da Wikipedia

Livello 1: Risorse (Resources)

L'Identificativo Univoco: l'URI

La direttiva principale del Livello 1 è assegnare a ogni singola risorsa un identificativo univoco, ovvero un URI (Uniform Resource Identifier). **L'URI è uno standard globale che funziona come un indirizzo specifico per ogni risorsa.**

La sua struttura è composta da due parti:

- Parte fissa: Indica il "luogo" dove si trova il servizio (es. https://www.amazon.it).
- Parte variabile: Identifica in modo univoco la risorsa specifica (es. /guida-pratica-microservizi-REST/dp/B0CR6Q6KBX).

Livello 1: Risorse (Resources)

Dal punto di vista del server, le risorse possono essere viste come oggetti del dominio “esposte” nel modello REST

URI	risposta
http://myUniversity.it/professors	Elenco di tutti i professori dell’università
http://myUniversity.it/professors/123	Dati del professore con id=123
http://myUniversity.it/professors/123/lectures	Lezioni tenute dal professore con id=123
http://myUniversity.it/lectures	Elenco di tutte le lezioni tenute all’università
http://myUniversity.it/lectures/ing_sw	Dati del corso di Ingegneria del software (ing_sw)
http://myUniversity.it/lectures/ing_sw/examsessions	Elenco delle sessioni di esame relative al corso di ing_sw
http://myUniversity.it/examsessions	Elenco di tutte le sessioni di esame (di tutti i corsi)
http://myUniversity.it/examsessions/110	Dati della sessione di esame con id=110

Regole e Convenzioni per gli URI

Per mantenere l'interfaccia chiara e prevedibile, si seguono delle regole stilistiche comuni.

- **Nomi al plurale per le collezioni:** Per accedere a una lista di risorse, si usa il nome plurale della risorsa.
  - Esempio: .../professors
- **ID per risorse specifiche:** Per un'azione su una singola risorsa, si aggiunge il suo ID all'URL della collezione.
  - Esempio: .../professors/123
- **Query string per parametri:** Per filtrare o inviare parametri opzionali, si usano le query string (i valori dopo il ?).
  - Esempio: .../professors/123?newMail=Giordano@uni.na.it

Body per dati complessi: Se si devono inviare dati strutturati (come un oggetto JSON per creare una nuova risorsa), questi vengono inseriti nel corpo (body) della richiesta.

URI Progressivi per Risorse Correlate

Spesso le risorse sono in relazione tra loro (es. un professore HA delle materie). Per accedere a dati correlati, si usa una struttura di URI "progressiva" che segue la relazione.

- Regola: http://domain/risorsaPrimaria/idPrimaria/risorsaCorrelata
- Esempio: Per ottenere tutte le materie insegnate dal professore con id 123, si invoca:
  - http://universita.napoli.it/professori/123/materie

La Potenza dei Link (URI)

L'uso di URI standardizzati offre vantaggi enormi:

- **Comprensibili agli umani**: Un URI ben fatto è leggibile e fa intuire quale risorsa si sta richiedendo.
- **Leggibili dalle macchine**: Un browser sa che un link http://... è un indirizzo web da visitare.
- Standard globale: Permettono a un servizio di richiamare una risorsa presente su qualsiasi server nel mondo.

Negoziazione del contenuto: L'HTTP permette di specificare nell'header il tipo di contenuto che ci si aspetta, consentendo al server di rispondere nel formato più appropriato (es. JSON, HTML, un video, etc.).

Sono associati all' HTTP che possiede il Content Type Negotiation.

- ad esempio, se il link punta ad un video, verrà interpretato con un lettore video

E' possibile assegnare informazioni aggiuntive nell'header HTTP

Content-Type Negotiation: "Parlare la Stessa Lingua"

Questo meccanismo permette al client e al server di negoziare il formato della rappresentazione di una risorsa. Non tutte le applicazioni capiscono gli stessi formati (alcune preferiscono JSON, altre XML, un browser capisce l'HTML, etc.).

La negoziazione funziona così:

- Il Client Chiede: Il client, nella sua richiesta, invia un header chiamato Accept, dove elenca i formati che è in grado di capire.
- Esempio: Accept: application/json (dice al server: "Per favore, se puoi, rispondimi in formato JSON").
- Il Server Risponde: Il server legge l'header Accept. Se può fornire la risorsa in uno dei formati richiesti, lo fa. Nella sua risposta, include un header chiamato Content-Type per specificare in quale formato sta effettivamente rispondendo.
- Esempio: Content-Type: application/json (dice al client: "Ok, ecco i dati che hai chiesto, te li ho preparati in formato JSON").

Il tuo esempio del video è perfetto: quando un browser richiede un URL e il server risponde con Content-Type: video/mp4, il browser sa che non deve mostrare del testo, ma avviare il suo lettore video integrato.

Content-Type Negotiation: "Parlare la Stessa Lingua"

Informazioni Aggiuntive negli Header HTTP

L'header HTTP è una sezione di metadati che accompagna ogni richiesta e risposta. È un contenitore di informazioni aggiuntive che forniscono contesto all'interazione.

Oltre a Accept e Content-Type per la negoziazione, si possono usare innumerevoli altri header per molti scopi, ad esempio:

- Authorization: Per inviare credenziali di accesso (come un token) e autenticare la richiesta.
- Cache-Control: Per dare istruzioni su come e per quanto tempo la risposta può essere salvata in cache.
- User-Agent: Per descrivere quale tipo di client sta effettuando la richiesta (es. Chrome, Firefox, un'app mobile).

In pratica, mentre l'URI dice cosa si vuole, gli header HTTP specificano come, chi e in quale formato lo si vuole, rendendo la comunicazione potente e flessibile.

Esempio: Gestione Utenti a Livello 1

A differenza del Livello 0, ora non abbiamo più un solo endpoint, ma indirizzi specifici per le nostre risorse.

Collezione di utenti: https://api.example.com/users

Singolo utente: https://api.example.com/users/{userID}

Tuttavia, continuiamo a usare il metodo POST per tutte le operazioni.

1. Ottenere gli Album di un Artista

In un servizio musicale, si vuole vedere la lista degli album di un artista specifico.

- Risorsa: Album dell'artista con ID queen.
- URL (Livello 1): https://api.musica.com/artists/queen/albums
- Interazione (Livello 1): Si usa POST invece di GET per richiedere i dati.

```
POST /artists/queen/albums
Host: api.musica.com
```

Esempio: Gestione Utenti a Livello 1

2. Aggiungere un Prodotto al Carrello

In un sito e-commerce, un utente aggiunge un prodotto al proprio carrello.

- Risorsa: Il carrello dell'utente con ID 123.
- URL (Livello 1): https://api.commerce.com/carts/123/items
- Interazione (Livello 1): Si inviano i dati del prodotto da aggiungere. In questo caso, l'uso di POST è casualmente corretto, ma altre azioni (come vedere il carrello) potrebbero usare POST invece di GET.

```
POST /carts/123/items
Host: api.commerce.com
Content-Type: application/json

{
  "productId": "xyz-456",
  "quantity": 2
}
```

Esempio: Gestione Utenti a Livello 1

3. Cancellare un Commento da un Post

In un blog, un moderatore vuole eliminare un commento specifico.

- Risorsa: Il commento con ID 987 all'interno del post 45.
- URL (Livello 1): https://api.blog.com/posts/45/comments/987
- Interazione (Livello 1): Si usa POST per un'operazione di cancellazione, invece del verbo corretto DELETE.

```
POST /posts/45/comments/987
Host: api.blog.com
Content-Type: application/json

{
  "action": "delete"
}
```

Livello 2: Verbi HTTP (HTTP Verbs)

A questo livello, non solo si usano indirizzi distinti per le risorse, ma si sfruttano anche i metodi (o verbi) del protocollo HTTP per specificare l'intenzione dell'operazione.

Come funziona: Invece di usare POST per ogni azione, si usa il verbo HTTP appropriato:

- GET per leggere una risorsa.
- POST per creare una nuova risorsa.
- PUT o PATCH per aggiornare una risorsa esistente.
- DELETE per eliminare una risorsa.

Significato: L'API diventa ancora più espressiva e standard. Sfrutta a pieno le semantiche del protocollo HTTP, rendendo le interazioni prevedibili. La maggior parte delle API REST "commerciali" oggi si ferma a questo livello, perché offre un ottimo equilibrio tra pragmatismo e aderenza ai principi REST.

Livello 2: Verbi HTTP GET

Il metodo GET serve per leggere o recuperare dati dal server. È l'operazione di sola lettura per eccellenza.

- Scopo: Recuperare la rappresentazione di una risorsa identificata dall'URI.
- Esempi:
  - Leggere i dettagli di un prodotto in vendita.
  - Ottenere la lista di tutti gli esami sostenuti da uno studente.
- Importante: Se l'URI si riferisce a un processo che esegue un calcolo, la risposta GET deve contenere i risultati di tale processo

Livello 2: Verbi HTTP POST

Il metodo POST serve per inviare dati al server per creare una nuova risorsa o per eseguire azioni che non rientrano negli altri verbi.

- Scopo: **Creare una nuova entità** come "figlia" della risorsa specificata nell'URI (es. creare un commento per un post). Può essere usato anche per aggiornamenti parziali se PATCH non è supportato.
- **Body**: A differenza di GET, una richiesta POST può includere un "corpo" (body) per trasportare dati complessi, superando i limiti di lunghezza di un URL.
- Esempi Tipici:
  - Invio dei dati di un form di registrazione.
  - Pubblicazione di un commento su un blog.
  - Caricamento di un file (upload).

Livello 2: Verbi HTTP PUT e PATCH

Entrambi servono per modificare una risorsa esistente, ma con una differenza fondamentale.

- PUT: **Esegue una modifica totale**. Il client invia nel corpo della richiesta la rappresentazione completa e aggiornata della risorsa. Se la risorsa specificata nell'URI esiste, viene interamente sostituita. Se non esiste, il server dovrebbe crearla.
  - Risposte comuni: 200 OK o 204 No Content se l'aggiornamento ha successo; 201 Created se la risorsa viene creata.
- PATCH: **Esegue una modifica parziale**. Il client invia solo i campi che intende modificare, senza dover fornire l'intera rappresentazione della risorsa. Ha solo la logica di aggiornamento.

Livello 2: Verbi HTTP DELETE

Il metodo DELETE serve per eliminare una risorsa specifica dal server.

- Scopo: **Rimuovere permanentemente la risorsa** identificata dall'URI.
- Risposte comuni:
  - 200 OK: La risorsa è stata eliminata (la risposta può contenere un messaggio di conferma).
  - 202 Accepted: La richiesta è stata accettata, ma l'eliminazione non è ancora stata completata (utile per operazioni lunghe).
  - 204 No Content: La risorsa è stata eliminata con successo e la risposta non include alcun corpo.

Nota: Anche se il server risponde con successo, non c'è una garanzia assoluta che la risorsa sia stata fisicamente cancellata dal disco in quell'istante, ma solo che non è più accessibile

Livello 2: Sicurezza e Idempotenza

A Livello 2 diventano cruciali due criteri che caratterizzano i verbi HTTP.

- Sicurezza (Safety): Un metodo è sicuro se la sua esecuzione non modifica lo stato della risorsa sul server.
  - L'unico metodo sicuro per definizione è GET. Si possono fare mille richieste GET a una risorsa senza alterarla.
- Idempotenza (Idempotence): **Un metodo è idempotente se eseguire la stessa richiesta più volte produce lo stesso risultato di eseguirla una sola volta**.
  - Sono idempotenti: GET, PUT, DELETE. Se si invia 3 volte la stessa richiesta DELETE per un utente, l'effetto finale è lo stesso della prima volta: l'utente viene eliminato.
  - NON è idempotente: POST. Se si invia 3 volte la stessa richiesta POST per creare un commento, verranno creati 3 commenti identici, alterando lo stato del server a ogni chiamata

Livello 2: Differenze tra POST e PUT: la questione dell'Idempotenza

Anche se formalmente sia POST che PUT possono essere usati per inserire o modificare dati, la loro differenza fondamentale risiede nel concetto di idempotenza, che definisce il loro comportamento e il loro utilizzo corretto.

- **PUT è una funzione IDEMPOTENTE** → si usa per azioni con comportamento prevedibile. L'operazione PUT è associata a un'azione di update completo. Il client invia l'intera rappresentazione di una risorsa a un URL specifico.
  - Se la risorsa esiste, viene sostituita con quella inviata.
  - Se la risorsa non esiste, viene creata. La chiave è che inviare la stessa richiesta PUT 1, 10 o 100 volte produrrà sempre lo stesso identico stato finale sul server.
- **POST è una funzione NON IDEMPOTENTE** → si usa per azioni con comportamento non idempotente. L'operazione POST è associata a un'azione di creazione (o a processi generici). Ogni volta che si invia una richiesta POST, ci si aspetta che venga creata una nuova risorsa o che avvenga una nuova azione.
  - Se invii la stessa richiesta POST per pubblicare un commento 3 volte, creerai 3 commenti identici. Lo stato del server cambia a ogni richiesta.

Livello 2: Riepilogo delle Proprietà dei Metodi

Vediamo un riassunto delle proprietà di sicurezza e idempotenza per i verbi HTTP principali:

- GET → è per natura Idempotente e Sicuro. Invocare una richiesta GET non cambia mai lo stato delle risorse sul server. È un'operazione di sola lettura.
- PUT, PATCH, DELETE → devono essere Idempotenti.
  - PUT: Se si invoca PUT una seconda volta con gli stessi dati, la risorsa viene semplicemente sovrascritta con lo stesso contenuto. Lo stato finale non cambia.
  - DELETE: Se si invoca DELETE una seconda volta sulla stessa risorsa, la risorsa è già stata cancellata. Il server risponderà probabilmente con un errore (es. 404 Not Found) o 204 No Content, ma lo stato finale del sistema (la risorsa non esiste) rimane invariato.
- POST → è per natura NON Idempotente. Come visto, ogni esecuzione di una richiesta POST è pensata per generare un nuovo cambiamento sul server.

Livello 2: Vantaggi della Proprietà di Idempotenza

L'idempotenza non è un concetto puramente accademico; ha un vantaggio pratico enorme, soprattutto nella gestione degli errori di rete.

- SAFE (Sicurezza nelle ripetizioni): La proprietà di idempotenza rende le integrazioni tra sistemi molto più robuste e sicure.
- INTEGRATION (Semplificazione della logica): Semplifica la semantica dei connettori, perché il client sa come comportarsi in caso di incertezza.

Esempio pratico: Un client esegue un'operazione di aggiornamento (PUT) ma, a causa di un problema di rete, non riceve una risposta in tempo (timeout). Il client si trova in un limbo:

- La richiesta è arrivata e il server l'ha processata?
- La richiesta non è mai arrivata?

Senza l'idempotenza, il client non saprebbe cosa fare. Riprovare potrebbe causare una duplicazione o un errore.

Grazie all'idempotenza, il client può tranquillamente inviare di nuovo la stessa identica richiesta PUT senza alcun rischio. Se la prima era andata a buon fine, la seconda semplicemente sovrascriverà i dati con gli stessi valori. Se la prima non era arrivata, la seconda completerà l'operazione. In entrambi i casi, lo stato finale del server sarà quello corretto.

Livello 3: Controlli Ipermediali (Hypermedia Controls - HATEOAS)

Questo è il livello finale, la "gloria di REST" (Glory of REST). È ciò che, secondo i puristi, rende un'API veramente RESTful.

- Come funziona: Il concetto chiave è HATEOAS (Hypermedia as the Engine of Application State). **Significa che la risposta del server non contiene solo i dati richiesti, ma anche i link (controlli ipermediali) alle azioni successive che si possono compiere su quella risorsa.**
- Analogia: È come navigare un sito web. Quando visiti una pagina, non hai bisogno di conoscere in anticipo gli URL delle altre pagine; la pagina stessa ti fornisce i link su cui cliccare.
- Esempio: Se richiedi i dati di un ordine (GET /ordini/789), la risposta potrebbe contenere non solo i dettagli dell'ordine, ma anche i link per annullarlo, modificarlo o tracciarne la spedizione.
- Significato: Il client è completamente disaccoppiato dal server. Non ha bisogno di avere gli URL "hardcoded" nel suo codice. Può evolvere semplicemente seguendo i link che il server gli fornisce. Questo rende il sistema incredibilmente flessibile e resiliente ai cambiamenti.

```
{
  "id": 789,
  "stato": "in elaborazione",
  "totale": 99.50,
  "_links": {
    "self": { "href": "/ordini/789" },
    "modifica": { "href": "/ordini/789/modifica" },
    "annulla": { "href": "/ordini/789/annulla" }
  }
}
```

Livello 3: Come Funziona HATEOAS

Torniamo all'esempio della segreteria universitaria. Supponiamo che un client chieda tutte le sessioni d'esame per una materia specifica.

Un server che implementa HATEOAS potrebbe non restituire subito tutti i dettagli di ogni sessione, ma una lista di link per eseguire azioni su ciascuna di esse. Questo riduce il carico iniziale sulla rete. La risposta conterrebbe un elenco di sessioni e, per ognuna, i link per:

- Ottenere i dettagli completi della sessione.
- Ottenere i dettagli dell'aula abbinata.
- Prenotare la sessione di esame.

L'Oggetto JSON Restituito

Se il client segue uno di questi link per ottenere i dettagli di una sessione, la risposta del server sarà strutturata per essere "auto-descrittiva", come mostrato nell'esempio:

```
{
  "id": 123,
  "sessions": [
    {
      "self": "http://myUniversity.ac.uk/examsessions/110", // Link alla risorsa sessione stessa
      "room": { "self": "http://myUniversity.ac.uk/rooms/222" }, // Link alla risorsa aula
      "link": {
        "rel": "http://myUniversity.ac.uk/examSessions/reserve", // Descrive la relazione (prenotazione)
        "uri": "/examsessions/110" // Link all'azione di prenotazione
      }
    },
    {
      "self": "http://myUniversity.ac.uk/examsessions/211",
      "room": { "self": "http://myUniversity.ac.uk/rooms/115" },
      "link": {
        "rel": "http://myUniversity.ac.uk/examSessions/reserve",
        "uri": "/examsessions/211"
      }
    }
  ]
}
```

Il client non ha bisogno di sapere in anticipo come si prenota un esame; scopre l'URL per farlo direttamente dalla risposta che riceve.

Livello 3: Vantaggi di HATEOAS

Implementare il Livello 3 offre vantaggi strategici significativi.

1. Disaccoppiamento tra Client e Server: Questo è il vantaggio più grande. Il client non ha bisogno di avere URL "hardcoded" nel proprio codice. Se il team del server decide di cambiare la struttura degli URL (ad esempio, da /reserve a /booking/new), non romperà i client, perché questi seguiranno semplicemente il nuovo link fornito dinamicamente dal server. Questo permette al server di evolvere autonomamente.
2. Riduzione del Carico di Rete: Invece di restituire oggetti complessi e pesanti, il server può fornire una risorsa più leggera con i link alle informazioni di dettaglio. Sarà poi il client, in base alle reali esigenze, a decidere se e quali dati aggiuntivi recuperare seguendo i link.

Conclusioni

In definitiva, lo stile REST, culminando nel Livello 3 con HATEOAS, promuove un'architettura che rispecchia quella del World Wide Web stesso, basata su principi potenti:

- Separazione delle Responsabilità: Il client si occupa dell'interfaccia utente, il server della logica di business. Questo semplifica l'implementazione e la manutenzione di entrambi.
- Elaborazione Intermedia: Forza i messaggi a essere auto-descrittivi (stateless, con metodi e media type standard), permettendo a componenti intermedi (come proxy e cache) di operare in modo efficiente.
- Proprietà Fondamentali: Le architetture RESTful ereditano le migliori proprietà dell'infrastruttura web: semplicità, scalabilità (capacità di gestire un carico crescente), portabilità (indipendenza dalla tecnologia) e performance elevate.

Esempi di chiamate curl

curl è uno strumento a riga di comando che permette di trasferire dati con URL. In parole povere, è un modo per comunicare con i server web direttamente dal tuo terminale. Immaginalo come un browser web senza interfaccia grafica, perfetto per automatizzare compiti e testare API.

La sua sintassi è piuttosto semplice: curl [opzioni] [URL]. Le opzioni modificano il comportamento di curl, mentre l'URL indica la risorsa a cui vuoi accedere.

Ad esempio, per scaricare una pagina web, useresti curl seguito dall'indirizzo della pagina. Oppure, per inviare dati a un server, useresti l'opzione -X per specificare il metodo HTTP (come POST o PUT) e l'opzione -d per includere i dati da inviare.

Esempi di chiamate curl

Ottenere tutti gli utenti (GET)

curl -X GET "https://jsonplaceholder.typicode.com/users"

Ottenere un singolo utente (GET)

curl -X GET "https://jsonplaceholder.typicode.com/users/1"

Esempi di chiamate curl

Creare un nuovo utente (POST)

Questo comando invia i dati di un nuovo utente al server. Il server risponderà con i dati dell'utente creato e un id assegnato.

```
curl -X POST "http://localhost:3000/users" \
    -H "Content-type: application/json; charset=UTF-8" \
     -d '{
          "name": "Mario Rossi",
          "username": "mariorossi",
          "email": "mario.rossi@example.com",
          "address": {
            "street": "Via Roma",
            "suite": "App. 1",
            "city": "Milano",
            "zipcode": "20121",
            "geo": {
              "lat": "45.4642",
              "lng": "9.1900"
            }
          },
          "phone": "333-1234567",
          "website": "mariorossi.it",
          "company": {
            "name": "Rossi SpA",
            "catchPhrase": "Costruzioni di qualità",
            "bs": "realizzazione progetti"
          }
        }'
```

Esempi di chiamate curl

Aggiornare un utente (PUT)

Il metodo PUT viene utilizzato per sostituire completamente una risorsa esistente. È necessario fornire l'intero oggetto dell'utente nel corpo della richiesta.

```
curl -X PUT "https://jsonplaceholder.typicode.com/users/1" \
    -H "Content-type: application/json; charset=UTF-8" \
     -d '{
          "id": 1,
          "name": "Mario Rossi",
          "username": "mariorossi",
          "email": "mario.rossi@example.com",
          "address": {
            "street": "Via Roma",
            "suite": "App. 1",
            "city": "Milano",
            "zipcode": "20121",
            "geo": {
              "lat": "45.4642",
              "lng": "9.1900"
            }
          },
          "phone": "333-1234567",
          "website": "mariorossi.it",
          "company": {
            "name": "Rossi SpA",
            "catchPhrase": "Costruzioni di qualità",
            "bs": "realizzazione progetti"
          }
        }'
```

Aggiornare parzialmente un utente (PATCH)

Il metodo PATCH è utile quando si desidera aggiornare solo alcuni campi di una risorsa esistente, senza dover inviare l'intero oggetto.

```
curl -X PATCH "https://jsonplaceholder.typicode.com/users/1" \
    -H "Content-type: application/json; charset=UTF-8" \
     -d '{
          "username": "mariorossi",
          "email": "mario.rossi@example.com",
          }’
```

Cos'è una Richiesta OPTIONS

OPTIONS è uno dei metodi HTTP, come GET, POST, PUT, DELETE.

La sua funzione è chiedere informazioni sul server senza modificare dati.

Il Browser Invia OPTIONS Automaticamente

Cosa Fa OPTIONS

Quando invii una richiesta OPTIONS a un server, stai chiedendo:

- "Quali metodi HTTP accetti per questo endpoint?"
- "Quali headers posso inviarti?"
- "Da quali domini accetti richieste?"

Richieste OPTIONS

Esempio Pratico

- OPTIONS /api/users HTTP/1.1 Host: api.example.com

Risposta del server:

HTTP/1.1 200 OK
Allow: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization

Il server dice: "Su /api/users puoi usare GET, POST, PUT, DELETE. Accetto richieste da qualsiasi dominio e permetto headers Content-Type e Authorization."

OPTIONS MANUALI?

Raramente hai necessità di inviare OPTIONS direttamente. Potresti farlo per:

- Debug: Verificare cosa permette un'API
- Sviluppo: Capire le capacità di un endpoint
- Tool di test: Ispezionare configurazioni server

```
fetch('/api/users', { method: 'OPTIONS' })
  .then(response => {
    console.log('Metodi permessi:', response.headers.get('Allow'));
  });
```

Differenza con Altri Metodi

GET /api/users → Restituisce lista utenti
POST /api/users → Crea nuovo utente
DELETE /api/users → Elimina tutti gli utenti
OPTIONS /api/users → "Cosa posso fare qui?" (non tocca i dati)

La Connessione con CORS

Qui entra in gioco il browser. Quando fai richieste cross-origin "complesse", il browser usa automaticamente OPTIONS per chiedere permesso prima di inviare la tua richiesta vera.

Ma questo è il passo successivo. Per ora, ricorda che OPTIONS è semplicemente un metodo HTTP per "fare domande" al server sui suoi permessi e capacità.

ESEMPIO

Tu NON scrivi OPTIONS

```
fetch('/api/users', { method: 'DELETE' })
```

Il Browser Invia OPTIONS Automaticamente

1. Browser vede: "DELETE è pericoloso, devo chiedere permesso"
2. Browser invia automaticamente: OPTIONS /api/users (NON lo hai chiesto tu)
3. Server risponde: "OK, DELETE permesso"
4. Solo DOPO, browser invia la TUA richiesta: DELETE /api/users

Non Controlliamo Questo Processo

Il browser decide autonomamente quando serve OPTIONS:

- Tu scrivi method: 'DELETE' → Browser manda OPTIONS + DELETE
- Tu scrivi method: 'GET' → Browser manda solo GET
- Tu scrivi Content-Type: 'application/json' → Browser manda OPTIONS + POST

L'unico momento in cui scrivi esplicitamente OPTIONS è per debug:

```
fetch('/api/users', { method: 'OPTIONS' })
```

- Non puoi disabilitare il preflight
- È il browser che decide, non tu
- È un meccanismo di sicurezza automatico

Quindi quando parliamo di CORS, OPTIONS è sempre "dietro le quinte" - tu scrivi DELETE, ma il browser gestisce tutto il flusso OPTIONS automaticamente.

Cos'è la Same-Origin Policy?

La Same-Origin Policy è una politica di sicurezza implementata dai browser che impedisce a una pagina web di fare richieste a un dominio diverso da quello che ha servito la pagina.

Due URL hanno la stessa origine se condividono:

- Protocollo (http/https)
- Dominio
- Porta

✅ **STESSA ORIGINE:**
https://example.com/page1
https://example.com/page2

❌ **ORIGINI DIVERSE:**
https://example.com → http://example.com (protocollo diverso)
https://example.com → https://api.example.com (sottodominio diverso)
https://example.com → https://example.com:8080 (porta diversa)

Perché esiste questa limitazione?

Sicurezza: Previene attacchi malevoli dove un sito potrebbe:

- Rubare dati da altri siti aperti nel browser
- Fare richieste non autorizzate a nome dell'utente
- Accedere a risorse private

Cos'è CORS?

CORS (Cross-Origin Resource Sharing) è un meccanismo che permette a un server di indicare esplicitamente quali origini possono accedere alle sue risorse, "rilassando" la Same-Origin Policy in modo controllato.

Simple Requests (Richieste Semplici)

Alcune richieste sono considerate "semplici" e vengono inviate direttamente:

// Esempio di Simple Request
fetch('http://localhost:3001/users', { method: 'GET' });

Criteri per Simple Request:

Metodi: GET, HEAD

Headers standard (Content-Type: text/plain, application/x-www-form-urlencoded, multipart/form-data)

Preflight Requests

Alcune richieste sono considerate "semplici" e vengono inviate direttamente:

```
// Questa richiesta triggerà un preflight

fetch('http://localhost:3001/users', {
method: 'DELETE',
headers: { 'Content-Type': 'application/json', 'Authorization': 'Bearer token123' }
});
```

Criteri per Simple Request:

Metodi: PUT, PATCH, POST, DELETE

Headers standard (Content-Type: text/plain, application/x-www-form-urlencoded, multipart/form-data)

Headers CORS Principali

Server Response Headers

Access-Control-Allow-Origin: *
Access-Control-Allow-Origin: https://localhost:3000
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization, X-Requested-With
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 86400

Request Headers (Preflight)

Origin: https://localhost:3000
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: Content-Type, Authorization

Cos'è una Preflight Request?

Una Preflight Request è una richiesta HTTP OPTIONS che il browser invia automaticamente PRIMA della tua richiesta vera e propria, per "chiedere il permesso" al server.

Perché Esistono?

Il browser pensa: "*Questa richiesta potrebbe essere pericolosa, meglio chiedere prima se il server è d'accordo*"

1. Tu scrivi: fetch('/api/users', {method: 'DELETE'})
2. Browser pensa: "DELETE non è sicuro, chiedo prima"
3. Browser invia automaticamente: OPTIONS /api/users Origin: https://myapp.com Access-Control-Request-Method: DELETE
4. Server risponde: Access-Control-Allow-Origin: https://myapp.com Access-Control-Allow-Methods: GET, POST, DELETE
5. Browser: "Ok, il server accetta DELETE da myapp.com" 6. Browser invia la TUA richiesta originale:DELETE /api/users

Esempio Pratico - Passo per Passo

Immaginiamo di avere questa chiamata:

```
fetch('http://localhost:3001/users/1', {
method: 'DELETE',
headers: { 'Authorization': 'Bearer abc123' }
});
```

Quello che succede "dietro le quinte":

STEP 1 - Browser invia preflight automaticamente:

```
OPTIONS /users/1 HTTP/1.1
Host: localhost:3001
Origin: http://localhost:3000
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: Authorization
```

STEP 2 - Server deve rispondere al preflight:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Authorization, Content-Type
```

STEP 3 - Solo ora il browser invia la TUA richiesta:

```
DELETE /users/1 HTTP/1.1
Host: localhost:3001
Origin: http://localhost:3000
Authorization: Bearer abc123
```

Quando Scatta il Preflight?

Il browser usa preflight per richieste "non semplici":

Triggherano Preflight:

```
// Metodi "pericolosi"
fetch('/api', { method: 'DELETE' })     // DELETE
fetch('/api', { method: 'PUT' })        // PUT
fetch('/api', { method: 'PATCH' })      // PATCH

// Content-Type "speciali"
fetch('/api', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' }  // JSON!
})

// Headers custom
fetch('/api', {
    headers: { 'Authorization': 'Bearer token' }  // Auth header
})

fetch('/api', {
    headers: { 'X-Custom-Header': 'value' }  // Qualsiasi X-*
})
```

Quando Scatta il Preflight?

Il browser usa preflight per richieste "non semplici":

NON triggherano preflight (Simple Requests):

```
// Metodi "sicuri"
fetch('/api')                              // GET di default
fetch('/api', { method: 'POST' })          // POST semplice

// Content-Type "sicuri"
fetch('/api', {
    method: 'POST',
    body: new FormData()  // multipart/form-data
})

// Solo headers "standard"
fetch('/api', {
    headers: { 'Accept': 'application/json' }  // Accept è sicuro
})
```

Quando Scatta il Preflight?

Il preflight è quindi un "controllo di sicurezza preventivo" che il browser fa automaticamente per te, per richieste che potrebbero essere potenzialmente pericolose!

Quando fai una chiamata da un client (ad esempio dal tuo componente React) verso un server con CORS abilitato, e quella chiamata richiede un pre-flight, il browser prima invia una richiesta OPTIONS. Questa richiesta OPTIONS è tipo un "chiedo permesso" al server: gli chiede quali metodi, quali header, quali origini sono consentite.

Il server risponde a questa OPTIONS dicendo "ok, queste sono le regole" oppure "no, non è consentito". Solo se il server dà il via libera, il browser allora procede con la vera e propria chiamata effettiva, per esempio un DELETE o un POST con i dati che volevi inviare.

Riassumendo

1. CORS (Cross-Origin Resource Sharing):
   - Si applica a TUTTE le richieste cross-origin
   - Anche alle GET semplici!
   - Richiede sempre Access-Control-Allow-Origin
2. Preflight (OPTIONS):
   - Solo per richieste "non semplici"
   - GET è semplice = no preflight
   - Ma serve comunque CORS

GET è Simple Request (No Preflight):

```
// Questa richiesta NON trigghera preflight OPTIONS
fetch('http://localhost:3001/libri')

// Flusso:
// 1. Browser invia direttamente: GET /libri
// 2. Server risponde: { libri: [...] }
// 3. Browser controlla: "C'è Access-Control-Allow-Origin?"
// 4. ❌ NO → BLOCCA la risposta
// 5. JavaScript riceve: "Failed to fetch"
```

POST JSON è Non-Simple (Con Preflight):

```
// Questa trigghera preflight
fetch('http://localhost:3001/libri', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({...})
})

// Flusso:
// 1. Browser invia: OPTIONS /libri (preflight)
// 2. Server risponde: 404 (non gestito)
// 3. Browser: "Preflight fallito" → BLOCCA tutto
// 4. POST non viene mai inviata
```

Riassumendo

Simple Request ≠ No CORS needed

Simple Request significa solo:

- Nessun preflight OPTIONS
- Ma CORS è sempre necessario per cross-origin

Abilitare i cors lato client in fase di sviluppo

Con react possiamo usare un proxy.

Abbiamo due modi per farlo

1 - Proxy in package.json

{ "proxy": "http://localhost:3001" }

Prima (senza proxy)

```
const API_BASE = 'http://localhost:3001';

fetch(`${API_BASE}/libri`)          // http://localhost:3001/libri
fetch(`${API_BASE}/libri/1`)        // http://localhost:3001/libri/1
fetch(`${API_BASE}/libri`, {...})   // POST http://localhost:3001/libri
```

Abilitare i cors lato client in fase di sviluppo

Con react possiamo usare un proxy.
Abbiamo due modi per farlo

1 - Proxy in package.json

{ "proxy": "http://localhost:3001" }

Prima (senza proxy)

```
const API_BASE = 'http://localhost:3001';

fetch(`${API_BASE}/libri`)          // http://localhost:3001/libri
fetch(`${API_BASE}/libri/1`)        // http://localhost:3001/libri/1
fetch(`${API_BASE}/libri`, {...})   // POST http://localhost:3001/libri
```

Dopo (con proxy):

```
fetch('/libri')                     // /libri → proxy → localhost:3001/libri
fetch('/libri/1')                   // /libri/1 → proxy → localhost:3001/libri/1
fetch('/libri', {...})              // POST /libri → proxy → localhost:3001/libri
```

Abilitare i cors lato client in fase di sviluppo

Con react possiamo usare un proxy.
Abbiamo due modi per farlo

1 - Proxy in package.json

{ "proxy": "http://localhost:3001" }

Prima (senza proxy)

```
const API_BASE = 'http://localhost:3001';

fetch(`${API_BASE}/libri`)          // http://localhost:3001/libri
fetch(`${API_BASE}/libri/1`)        // http://localhost:3001/libri/1
fetch(`${API_BASE}/libri`, {...})   // POST http://localhost:3001/libri
```

Dopo (con proxy):

```
fetch('/libri')                     // /libri → proxy → localhost:3001/libri
fetch('/libri/1')                   // /libri/1 → proxy → localhost:3001/libri/1
fetch('/libri', {...})              // POST /libri → proxy → localhost:3001/libri
```

Abilitare i cors lato client in fase di sviluppo

2 - Proxy Javascript

Installa dipendenza proxy: **npm install http-proxy-middleware --save-dev**

Crea src/setupProxy.js

```
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  app.use('/api', createProxyMiddleware({
    target: 'http://localhost:3001',
    changeOrigin: true,
    pathRewrite: { '^/api': '' }
  }));
};
```