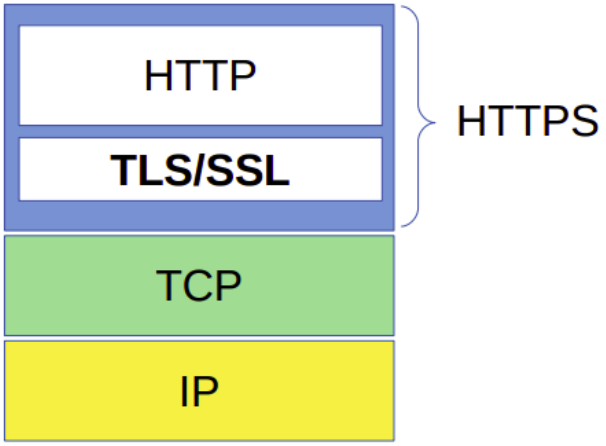
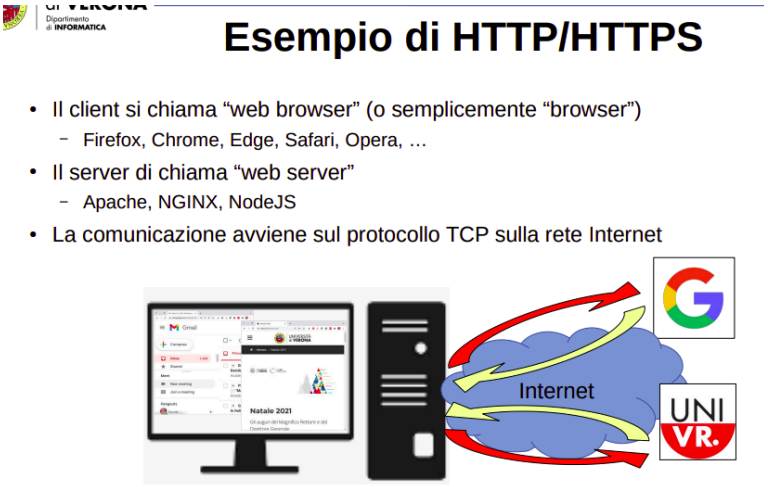
WEBSERVICE

HTTP/HTTPS:

Nato per la fruizione di contenuti in rete (World Wide Web), oggi è usato anche per l'invocazione di funzionalità remote (Webservice). I messaggi che passano nella connessione TCP sono gli stessi dell'HTTP ma vengono sottoposti a cifratura dei dati in transito e autenticazione del server mediante certificato digitale. Il server lavora sulla porta 443 invece che 80.





Questo protocollo è testuale, di tipo client/server formata da:

1. apertura connessione TCP;
2. (se HTTPS) autenticazione del server e negoziazione di una chiave di cifratura;
3. richiesta;
4. risposta;
5. chiusura connessione TCP.

Problema cybersec: nel DNS avviene (solitamente) un controllo IP-nome_host; potrebbe essere infatti che per frode un server si "spacci" per un altro, mentre io client penso di comunicare con qualcuno che invece non è (ma che magari ha scopiazzato la parte estetica del web server originale a posta per truffare);

NB: si può forkare apache per gestire più porte;

HTML:

Il corpo della risposta HTTP può contenere:

- HTML puro;

- HTML + codice Javascript;
- Sequenza binaria che rappresenta un'immagine;
- Cascading Style Sheets (CSS);
- Intere librerie di codice Javascript da eseguire sul browser.

HTML è un linguaggio testuale di descrizione di una pagina, una specializzazione del generico XML (eXtensible Markup Language) , basato su "tag" annidati (costrutto ad albero) eventualmente contenenti attributi.

Il DOM (Document Object Model) è una rappresentazione in forma di albero dell'HTML di una pagina, che il browser crea in memoria quando carica il sito. Ogni elemento HTML (come , <p>, <img>, …) diventa un nodo di quest'albero e può essere letto, modificato o eliminato dinamicamente tramite JavaScript. L'HTML base invece è semplicemente il codice statico scritto dall'utente.

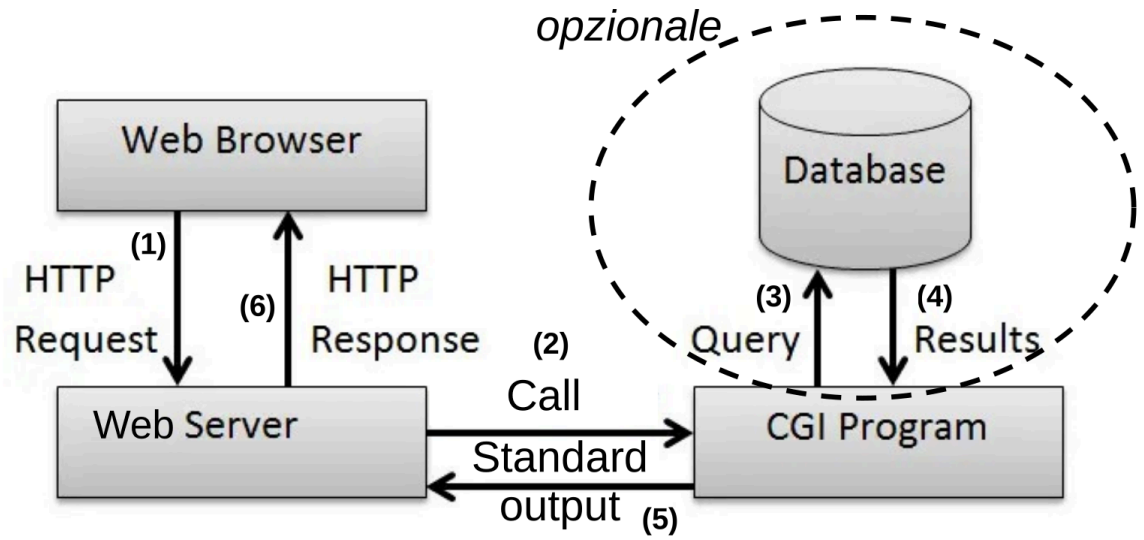
L'URL (Universal Resource Locator) è la stringa che permette di identificare in maniera univoca una risorsa HTTP in qualsiasi parte della rete mondiale:
https://www.univr.it/servizi/studenti/carriera
Struttura:

- per prima cosa il protocollo utilizzato a livello applicazione, che di solito include implicitamente anche il protocollo di livello trasporto + la porta utilizzata (per esempio, HTTP →TCP/80, HTTPS→TCP/443);
- segue nome dell'host (o indirizzo IP) che eroga tale risorsa;
- infine nome della risorsa con suo percorso logico completo (non necessariamente fisico).

La porta può essere indicata esplicitamente se non è quella standard:

https://www.univr.it:**8000**/servizi/studenti/carriera

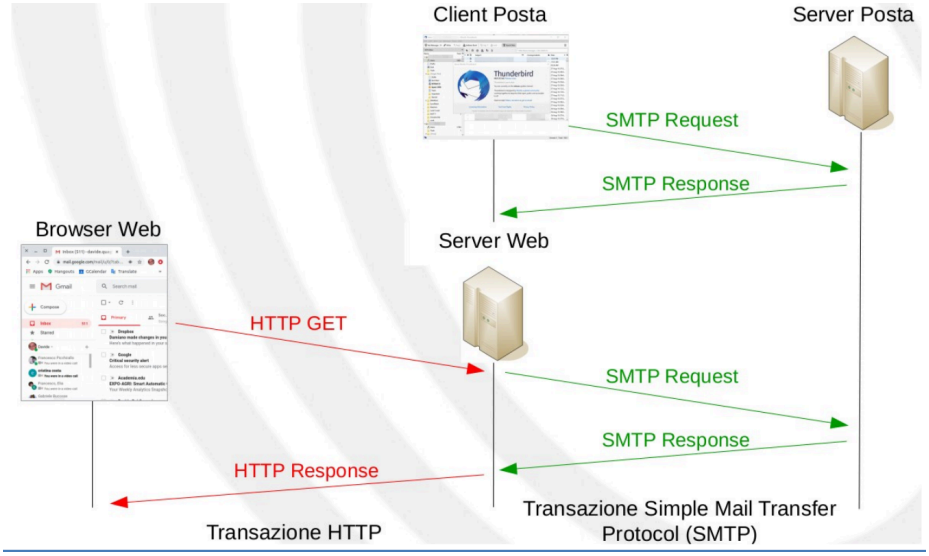
All'inizio il web (detto primo web) era quindi composto da semplici richieste HTTP statiche. Poi, con l'avvento del CGI (common gateway interface) si è aggiunta la possibilità di fare richieste dinamiche. La richiesta del client è sempre la stessa, mentre il server accoglie la richiesta, e vi esegue delle elaborazioni, appunto, dinamiche. Possono essere query in database, elaborazione complessa di dati, ecc. ecc. Il server dunque invoca delle chiamate dietro le quinte (da qui nasce il "backend") verso qualsiasi cosa (il client comunque non sa nulla), e ritorna il risultato finale al client.



L'esempio più stupido possibile è che il server accumuli un contatore ogni volta che il client si collega al server; quel conteggio è dinamico e non statico, per tanto è una prima forma di backend a tutti gli effetti.

Dunque, il programma CGI viene eseguito "lato server". Allo stesso modo, per alcune specifiche operazioni del server, esso può diventare a sua volta un client di molte applicazioni di rete, per esempio per:

- Content Management Systems (CMS);
- Posta elettronica;
- Wiki.



Web-socket:

I WebSockets rappresentano un protocollo di livello applicazione che si pone come un'alternativa significativa a HTTP/HTTPS, progettato specificamente per abilitare una comunicazione simmetrica e persistente tra il browser web (client) e il server.

A differenza del modello richiesta-risposta di HTTP, dove il client deve sempre avviare la comunicazione, una volta stabilita una connessione WebSocket, entrambi i processi (client e server) possono prendere l'iniziativa per inviare dati all'altro in qualsiasi momento. Questo elimina il sovraccarico di dover aprire e chiudere una nuova connessione HTTP per ogni scambio di dati, rendendo la comunicazione estremamente efficiente e a bassa latenza.

La connessione WebSocket nasce tipicamente da una sessione HTTP/HTTPS preesistente attraverso un'operazione chiamata "Protocol Upgrade". È come se il browser e il server, dopo un primo "saluto" via HTTP, decidessero di passare a un linguaggio di comunicazione più diretto e continuo per tutto il resto della sessione. Questa capacità di comunicazione bidirezionale e in tempo reale è fondamentale per molte applicazioni moderne.

Un esempio è il refresh asincrono di una pagina web: invece di richiedere al client di "tirare" continuamente per verificare se ci sono nuovi dati, il server può "spingere" (push) proattivamente nuove informazioni o aggiornamenti alla pagina non appena questi diventano disponibili. Questo rende possibili esperienze utente molto più dinamiche e reattive, ideali per chat in tempo reale, giochi multiplayer, dashboard live e notifiche istantanee.

SOA (SERVICE ORIENTED ARCHITECTURE):

Tradizionalmente, le applicazioni venivano concepite come soluzioni "monolitiche". Immaginiamo un unico, grande programma installato direttamente sul computer dell'utente: tutte le funzionalità, dall'interfaccia grafica ai calcoli più complessi, erano strettamente legate e contenute in un singolo eseguibile che "girava" interamente sulla macchina locale. La Service-Oriented Architecture (SOA) ha rappresentato un cambio di paradigma significativo. Invece di un blocco unico, le applicazioni complesse sono ora realizzate combinando diversi programmi che comunicano tra loro, non più all'interno dello stesso sistema, ma attraverso la rete. Con SOA, l'interfaccia utente e alcune funzionalità di base (spesso definite come un "client leggero") risiedono ancora sul dispositivo dell'utente. Tuttavia, il vero cuore dell'applicazione, ovvero le sue funzionalità principali e più elaborate, è fornito da programmi che risiedono su uno o più server remoti. In questo contesto, un Servizio può essere inteso come una specifica funzionalità, concettualmente simile a una chiamata a funzione o a un metodo negli ambienti orientati agli oggetti.

L'adozione di un'architettura SOA porta con sé una serie di notevoli vantaggi. La potenza di calcolo e la memoria non gravano più sul dispositivo dell'utente, ma sono delegate ai

server, permettendo all'applicazione di gestire carichi di lavoro maggiori e di mantenere elevate prestazioni. Questo approccio offre anche una maggiore protezione della proprietà intellettuale, poiché gli algoritmi strategici e il codice critico rimangono residenti sui server. Un altro beneficio cruciale riguarda gli aggiornamenti del software: non c'è più la necessità di distribuire nuove versioni agli utenti ogni volta che le modifiche riguardano solo il codice sul lato server, semplificando enormemente la manutenzione. La SOA ha inoltre aperto la strada a nuovi modelli economici, come il "pay-per-use" (pagamento basato sull'utilizzo effettivo del servizio), e contribuisce significativamente all'eliminazione della pirateria, dato che il cuore dell'applicazione risiede sul server e viene fruito come servizio.

Nonostante i numerosi benefici, l'architettura SOA presenta anche delle sfide intrinseche. Il più evidente è che l'infrastruttura di rete diventa un elemento essenziale e non opzionale. Di conseguenza, se la connessione di rete viene a mancare o è instabile, l'applicazione cessa di funzionare, rappresentando un vincolo significativo per la disponibilità del servizio.

Al centro della SOA risiede un concetto fondamentale, la chiamata di funzione remota. Questo modello è perfettamente conforme al classico paradigma client/server, ma con una complessità e una flessibilità maggiori dettate dalla distribuzione su rete.
Ogni funzione, sia essa locale o remota, è definita da alcuni elementi essenziali:

- nome, che ci indica la sua finalità;
- tipo e il numero dei parametri in ingresso che accetta;
- tipo del valore che restituisce;
- implementazione, ovvero il codice che ne definisce il comportamento.

L'interfaccia di una funzione è proprio la sua "carta d'identità": una descrizione di nome, parametri e tipo di valore di ritorno. Questa interfaccia può essere fornita da una libreria locale, come nelle applicazioni tradizionali, o, nel contesto della SOA, da un server remoto. L'insieme di queste funzioni o metodi esposte da una libreria (locale) o da un server (remoto) prende il nome di Application Program Interface (API).

In un'architettura SOA, il componente server espone una API che descrive una serie di funzioni invocabili da un componente client, che in questo specifico contesto, non è un semplice web browser. L'implementazione vera e propria di queste funzioni risiede interamente sul server. Ciò che rende la SOA così potente è la sua agnosticità tecnologica: il codice che invoca la funzione sul client e quello che la implementa sul server possono essere scritti in linguaggi di programmazione completamente diversi e girare su architetture di calcolo molto differenti senza alcun problema di compatibilità. Per far sì che questa comunicazione avvenga senza intoppi, entrano in gioco due componenti chiave:

- sul lato client, la funzione chiamata dall'applicazione apparentemente realizza la funzionalità desiderata. In realtà, il suo compito principale è codificare i parametri di input in un formato adatto per la trasmissione in rete e, una volta ricevuta la risposta, decodificare il valore di ritorno. Questa funzione speciale sul client, che implementa la stessa interfaccia della funzione remota, è chiamata STUB;
- sul lato server, esiste un componente chiamato SKELETON. Questo si occupa di decodificare i parametri di input che arrivano dalla rete, di passarli alla funzione che contiene l'implementazione logica vera e propria (spesso definita BUSINESS LOGIC), di prendere il risultato generato da quest'ultima, di codificarlo e infine di spedirlo indietro al client.

Per il trasporto effettivo di questi dati codificati attraverso la rete, è necessario un protocollo di rete (come TCP/IP o HTTP). Il processo di codifica dei dati per la trasmissione è spesso definito SERIALIZZAZIONE, mentre il processo inverso di decodifica è la deserializzazione.

Sono quindi state sviluppate diverse tecnologie e standard, ognuno con le proprie caratteristiche e ambiti di applicazione:

- Remote Procedure Call (RPC): RPC è un concetto fondamentale che permette a un programma di eseguire una procedura (o subroutine) su un computer remoto come se fosse locale. Sebbene sia un concetto più generale, è alla base di molte delle tecnologie successive;
- Java Remote Method Invocation (Java RMI): specifico per l'ecosistema Java, Java RMI consente a un oggetto in una macchina virtuale Java (JVM) di invocare metodi su un oggetto che risiede in un'altra JVM, anche se su una macchina diversa. Utilizza TCP come protocollo di trasporto sottostante, sfruttando le capacità native di Java per la serializzazione degli oggetti.
- Common Object Request Broker Architecture (CORBA): CORBA è uno standard indipendente dal linguaggio di programmazione e dal protocollo di trasporto di livello inferiore. Permette a oggetti software scritti in linguaggi diversi e distribuiti su diverse piattaforme di comunicare tra loro. Storicamente, CORBA ha mirato a fornire un interoperabilità universale, spesso utilizzando TCP per il trasporto dei messaggi;
- Web Services: i Web Services rappresentano uno degli approcci più diffusi e standardizzati per l'implementazione della SOA, sfruttando ampiamente i protocolli e le tecnologie del web:

    - HTTP/HTTPS vengono utilizzati come protocollo di trasporto principale per scambiare gli elementi della funzione (richieste e risposte). Questo li rende facilmente accessibili attraverso firewall e reti esistenti;
    - Per il formato dei dati scambiati, si sono affermati due approcci principali:
        - Protocollo XML SOAP (Simple Object Access Protocol): questo è stato uno dei primi e più robusti standard per i Web Services. Utilizza XML per la messaggistica ed è spesso associato a protocolli più complessi per la descrizione dei servizi (WSDL) e la loro scoperta (UDDI). Sebbene sia molto potente e fornisca funzionalità avanzate (come transazioni e sicurezza a livello di messaggio), è percepito come pesante e talvolta datato a causa della verbosità di XML e della sua complessità;
        - Metodologia REST (Representational State Transfer): attualmente è l'approccio più usato e preferito per la costruzione di Web Services moderni, spesso chiamati API RESTful. REST è uno stile architetturale che sfrutta appieno i principi di HTTP, trattando le risorse come URL e utilizzando i metodi HTTP standard (GET, POST, PUT, DELETE) per operare su di esse. Il formato dei dati più comune per le API REST è JSON, grazie alla sua leggerezza e facilità di parsing, sebbene possa supportare anche XML o altri formati.

I Web Services basati su REST rappresentano oggi l'approccio predominante per la costruzione di API e servizi distribuiti. La loro efficacia deriva dal saper sfruttare appieno la semplicità e la robustezza del protocollo HTTP/HTTPS come veicolo principale per la comunicazione. In questo modello, la chiamata a una funzione remota viene gestita in modo intelligente:

- il nome della funzione non è più un riferimento diretto a una procedura, ma viene mappato su una URL (Uniform Resource Locator), identificando la "risorsa" su cui si vuole operare;
- il passaggio dei parametri avviene in due modi principali: o direttamente nella URL (tipico per metodi GET o DELETE), oppure inclusi nel corpo della richiesta, ovvero dopo l'header HTTP (come accade con i metodi POST o PUT);
- la scelta del metodo HTTP da utilizzare è cruciale e segue una semantica ben definita legata all'azione che si intende compiere sulla risorsa:
    - POST è impiegato per funzioni che creano un NUOVO oggetto sul server;
    - PUT è destinato a funzioni che aggiornano un oggetto ESISTENTE sul server;

    - GET è utilizzato per recuperare informazioni di un oggetto ESISTENTE sul server;
    - DELETE serve per le funzioni che hanno il compito di distruggere un oggetto esistente sul server.
- Il valore di ritorno della funzione remota viene generalmente inserito nel corpo della risposta HTTP;
- Un'evoluzione significativa rispetto ai protocolli più datati è la sostituzione di HTML come formato di scambio dati. Nei Web Services REST, il contenuto del corpo della risposta è tipicamente testo puro o, molto più frequentemente, JSON (JavaScript Object Notation), grazie alla sua leggerezza e facilità di elaborazione.

Metodi della Richiesta HTTP:

Il protocollo HTTP definisce diversi metodi che indicano l'azione desiderata sulla risorsa identificata dalla URL. I più comuni, già menzionati nel contesto REST, sono:

- GET
- POST
- PUT
- DELETE

L'adozione dei Web Services, in particolare quelli basati su REST, porta con sé notevoli benefici:

- compatibilità con l'Infrastruttura Internet Esistente: L'intera infrastruttura di Internet è già nativamente predisposta all'uso di HTTP/HTTPS. Questo significa che i Web Services funzionano senza problemi attraverso elementi come i firewall e le soluzioni NAT (Network Address Translation), che altrimenti complicherebbero la comunicazione;
- facilità di Debugging: L'utilizzo di contenuti testuali (come JSON o testo puro) nelle transazioni rende estremamente più semplice ispezionare e comprendere il flusso di dati, facilitando notevolmente il debugging delle applicazioni SOA;
- flessibilità del Client: Lo stesso insieme di servizi può essere richiamato e utilizzato sia da un programma client dedicato (come un'applicazione desktop o mobile) sia direttamente da un browser web. Un esempio calzante è quello dei Web Services associati a un'applicazione di Internet Banking: gli stessi servizi possono essere invocati sia dal sito web bancario sia dall'app mobile dedicata, garantendo coerenza e riuso del backend.

JSON:

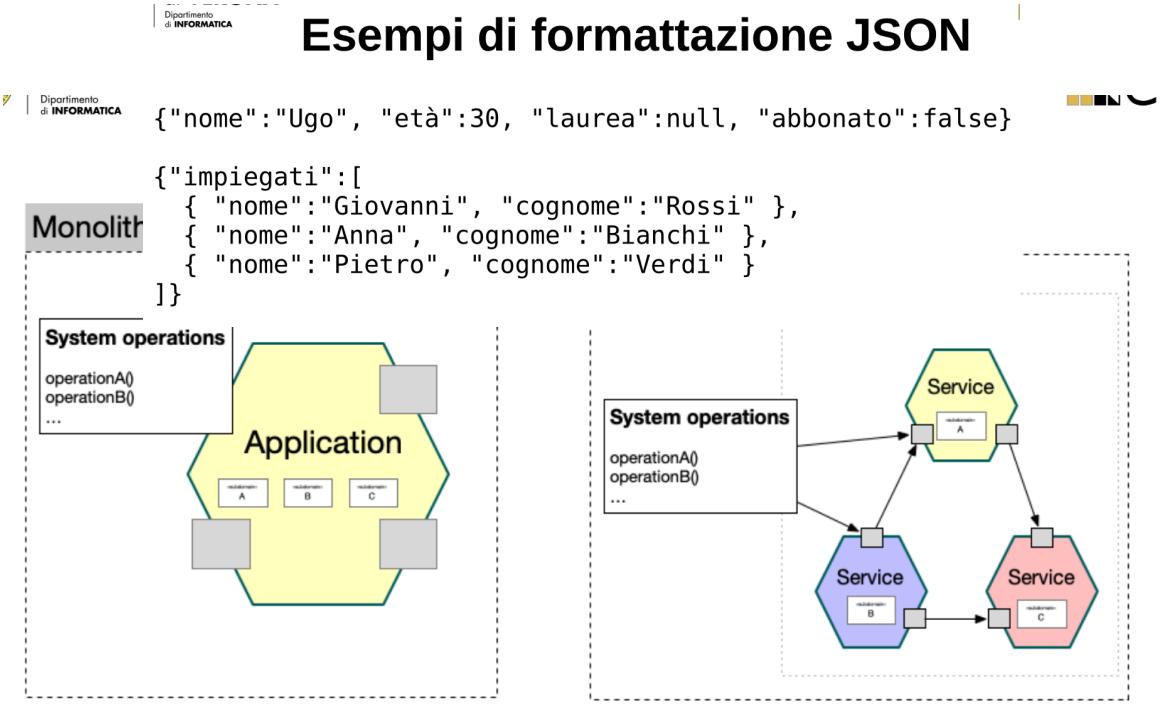
JSON (JavaScript Object Notation) è un formato di dato testuale estremamente versatile, nato in origine con JavaScript ma che oggi gode di un supporto universale in praticamente tutti i linguaggi di programmazione. La sua popolarità deriva dalla capacità di combinare leggibilità umana con una facilità di parsing eccezionale per i programmi.

La sua struttura si basa su una gerarchia intuitiva, fondata sull'elemento base: la coppia attributo : valore. I tipi di valore che JSON può rappresentare sono semplici e diretti:

- stringhe: sempre racchiuse tra virgolette doppie (es. "nome");
- numeri: interi o decimali (es. 123, 45.6);
- booleani: true o false;
- null: per indicare l'assenza di un valore.

Per organizzare dati più complessi, JSON offre due strutture principali:

- array [...]: permettono la composizione di elementi omogenei, rappresentando liste o sequenze di valori (es. ["mela", "pera", "banana"]);
- strutture dati {...} (oggetti): consentono la composizione di elementi eterogenei, mappando attributi a valori per formare oggetti complessi (es. {"nome": "Mario", "età": 30}).



I microservizi rappresentano un'evoluzione delle architetture SOA, spingendo il concetto di modularità al massimo. Anziché un'applicazione composta da pochi servizi di grandi dimensioni, in un'architettura a microservizi, l'applicazione è un insieme di servizi molto piccoli, indipendenti e focalizzati su una singola funzionalità di business.

Vantaggi dei microservizi:

Il principale punto di forza dei microservizi è la loro capacità di rendere estremamente modulare la parte di backend di un'applicazione. Questo porta a numerosi benefici operativi e di sviluppo:

- scalabilità migliorata: ogni micro servizio può essere assegnato a macchine diverse, permettendo di aumentare la scalabilità orizzontale. Se, ad esempio, il servizio di autenticazione è sotto carico elevato, si possono aggiungere più istanze solo di quel servizio, senza dover scalare l'intera applicazione. Questo si lega strettamente al concetto di Load Balancing, che distribuisce le richieste tra le varie istanze per ottimizzare le prestazioni;
- facilità di sviluppo e manutenzione: la dimensione ridotta e l'indipendenza di ogni micro servizio rendono lo sviluppo e la manutenzione molto più agili. I team possono lavorare su singoli servizi in modo indipendente;
- cicli di sviluppo indipendenti: è possibile programmare interventi di manutenzione o l'introduzione di nuove funzionalità su specifici moduli software lasciando attivi gli altri. Questo riduce drasticamente i tempi di inattività e permette rilasci continui e più rapidi;
- riduzione del rischio di effetti collaterali: data l'indipendenza tra i servizi, un errore o un problema in un micro servizio è meno probabile che si propaghi e comprometta l'intera applicazione, isolando così i guasti;
- agnosticismo tecnologico: team diversi possono scegliere le tecnologie e i linguaggi di programmazione più adatti per ogni singolo micro servizio, senza essere vincolati da un'unica scelta per l'intera applicazione.

Svantaggi e complessità dei microservizi:

Nonostante i numerosi pro, l'adozione dei microservizi introduce anche delle complessità significative:

- gestione del database: per massimizzare l'indipendenza dei microservizi, spesso si suddivide il database in più database distinti, uno per ogni servizio o per gruppi di servizi correlati. Questo però elimina la possibilità di eseguire JOIN ottimizzati direttamente a livello di database relazionale, come si farebbe con un unico database monolitico;
- incrocio dati manuale: se i dati necessari per una certa operazione sono distribuiti su database diversi gestiti da microservizi distinti, l'incrocio delle tabelle deve essere implementato manualmente a livello applicativo attraverso chiamate tra i servizi. Ciò richiede un'attenta progettazione per dividere la base dati in punti dove l'esigenza di

incrocio tra diversi database è minima, minimizzando così la complessità e l'overhead di rete;
- complessità operativa e di gestione: un'architettura a microservizi è intrinsecamente più complessa da gestire, monitorare e dispiegare. Richiede strumenti avanzati per l'orchestrazione, il logging, il monitoring e la tracciabilità distribuita;
- comunicazione di rete aumentata: con più servizi che comunicano tra loro, aumenta il traffico di rete e la potenziale latenza tra i componenti.

Realizzare e Mettere Online i Servizi:

Una volta che abbiamo ben chiaro cosa fa ogni servizio e come comunica, possiamo passare alla fase di implementazione. Scegliamo il linguaggio di programmazione che preferisci (Python, Java, Node.js, ecc.) e un buon framework che ti aiuti a creare il server e gestire tutta la logica. Dopodiché, dobbiamo mettere online questi servizi. Questo significa "istanziare" (avviare) i tuoi applicativi server su delle macchine. Pensiamo a come abbiamo fatto con i tre server sulla stessa macchina, su porte diverse (8000, 8001, 8002). In un ambiente reale, questi server potrebbero essere su macchine fisiche diverse, su macchine virtuali o, sempre più spesso, all'interno di container come Docker. L'ottimizzazione iniziale, in questa fase, è più che altro una buona configurazione di base del server per assicurarsi che funzioni senza intoppi. Il backend è progettato e online, ma cosa succede quando arrivano migliaia o milioni di utenti? È qui che l'ottimizzazione diventa cruciale, e il concetto di Load Balancing è il tuo migliore amico. Il Load Balancing è come avere un direttore d'orchestra per il traffico web. Invece di far arrivare tutte le richieste a un unico server, che potrebbe andare in crisi, un Load Balancer le distribuisce intelligentemente tra più istanze del tuo servizio. Immaginiamo di avere tre server che gestiscono gli ordini. Se tutte le richieste arrivano al server A, gli altri due restano a guardare. Un Load Balancer le smista: una al server A, una al B, una al C, e così via. Permette quindi di:

- gestire più utenti: l'applicazione può elaborare molte più richieste contemporaneamente;
- essere più veloce: ogni richiesta viene servita più rapidamente perché il carico è distribuito;
- non andare in crash: nessun server viene sovraccaricato fino al punto di bloccarsi;
- essere sempre disponibile: se un server si guasta, il Load Balancer smette semplicemente di inviargli richieste e le reindirizza agli altri server funzionanti. Questa è la base dell'alta disponibilità;
- la scalabilità orizzontale: il Load Balancing è la chiave per la scalabilità orizzontale. Invece di comprare un computer sempre più potente (scalabilità verticale, che ha un limite), puoi semplicemente aggiungere più server "normali" e farli lavorare insieme.

Per rendere il Load Balancing davvero efficace in un ambiente dinamico, ci sono due tecnologie fondamentali:

- la containerizzazione (es. Docker): pensa a Docker come a una piccola scatola portatile che contiene la tua applicazione e tutto ciò di cui ha bisogno per funzionare (librerie, configurazioni, ecc.). Questo significa che puoi prendere questa scatola e farla girare su qualsiasi macchina, ed essa si comporterà sempre allo stesso modo. È il modo perfetto per creare repliche identiche e affidabili dei tuoi servizi da distribuire;
- l'orchestrazione (es. Kubernetes): se Docker ti dà le scatole, Kubernetes (o altri orchestratori) è la fabbrica che gestisce quelle scatole. Automatizza il piazzamento delle scatole sui tuoi server, decide quante repliche avviare (anche in base al carico, aggiungendone o togliendone automaticamente), e se una scatola si rompe, la sostituisce subito. È come avere un team di operai robot che gestiscono la tua infrastruttura.

Ci sono diversi modi in cui il load balancer può decidere dove inviare una richiesta:

- servizi diversi su istanze diverse: questo, come accennato, non è "load balancing" in senso stretto, ma piuttosto una divisione del lavoro per servizio. Se il tuo servizio utenti è su un set di server e il tuo servizio prodotti su un altro, il traffico per gli utenti non andrà mai ai server prodotti. Aiuta a distribuire il carico tra i servizi, ma non all'interno di un singolo servizio;
- il load balancer centrale (reverse proxy): questa è la soluzione più diffusa. Hai un unico punto di accesso (un IP o un nome a dominio) che è il tuo Load Balancer. I client si connettono a lui, e lui, come un vigile urbano, smista le richieste ai vari server reali che stanno "dietro le quinte". Il Load Balancer può usare vari criteri per smistare, ad esempio mandare la richiesta al server con meno carico, o distribuire a turno;
    - "session stickiness" (o affinità di sessione): a volte, per applicazioni che mantengono informazioni specifiche dell'utente sul server (come un carrello della spesa o una sessione di login), è utile che tutte le richieste di quello stesso utente vadano sempre allo stesso server. Il Load Balancer può essere configurato per farlo, garantendo che l'utente non perda i suoi dati di sessione. Il rovescio della medaglia è che un server potrebbe ritrovarsi con più "utenti appiccicati" e quindi un carico maggiore;
- load balancing basato su DNS (DNS Round Robin): un metodo più semplice è far sì che il tuo server DNS (quello che traduce nomi come google.com in indirizzi IP) risponda con IP diversi ogni volta che qualcuno chiede l'indirizzo del tuo servizio. Quindi, il primo utente riceve l'IP del server A, il secondo l'IP del server B, e così via.

    - il problema del caching DNS: la grande debolezza di questo approccio è che client e altri server DNS in giro per internet tendono a memorizzare queste risposte per un certo tempo (fanno "caching"). Se un server va giù o ne aggiungi di nuovi, i client che hanno in cache la vecchia informazione continueranno a provare a connettersi ai server non più validi o non verranno a conoscenza dei nuovi server fino alla scadenza della loro cache. Questo lo rende meno efficace per la gestione dinamica del carico e l'alta disponibilità.

Cloud Computing:

Immaginiamo di non dover più comprare, installare e mantenere server fisici nel tuo ufficio o data center. Il Cloud Computing è esattamente questo: un servizio che ti permette di usare risorse di calcolo (server, storage, database, software) tramite internet, senza che tu debba preoccuparti dell'infrastruttura sottostante. È un po' come l'elettricità: non costruisci una centrale elettrica a casa tua, semplicemente attacchi la spina e usi l'energia che ti viene fornita. Nel cloud, paghi per le risorse che consumi, proprio come con la bolletta della luce, oppure con un costo fisso, a seconda del fornitore. Tra i giganti di questo settore troviamo nomi come:

- Amazon Web Services (AWS);
- Google Cloud;
- Microsoft Azure;
- servizi come Dropbox, sebbene non siano piattaforme di calcolo complete, sono un esempio di "archiviazione nel cloud".

Il cloud computing non è magia, ma il risultato di diverse tecnologie mature che lavorano insieme:

- connessione internet costante: è il requisito base. Se non fossimo perennemente connessi, l'idea di accedere a risorse remote sarebbe impraticabile;
- virtualizzazione: questa è la vera star. Permette di far girare più "macchine virtuali" (sistemi operativi completi, ognuno con le sue applicazioni) su un singolo server fisico. In questo modo, i fornitori di cloud possono ottimizzare l'uso delle loro potenti macchine fisiche, affittando "fette" ad n clienti contemporaneamente;
- Web Services: sono il "linguaggio" che permette alle tue applicazioni di sfruttare le risorse di calcolo remote. Tramite API standard (come quelle RESTful che abbiamo visto), il tuo software può chiedere al cloud di eseguire un calcolo, salvare un dato o recuperare un file.

Il cloud non è un servizio unico, ma una scala di opzioni che ti danno più o meno controllo sull'infrastruttura. Vediamole, partendo dal controllo totale fino all'uso puro del software.

- On-site (o On-premise): questo è il punto di partenza, il "non-cloud". Significa che l'hardware è tuo, fisicamente presente nella tua azienda, e devi gestire tutto tu: comprare i server, installare il sistema operativo, le applicazioni, la sicurezza, i cavi;
- Infrastructure as a Service (IaaS): entriamo nel cloud vero e proprio. Con IaaS, è come se affittassi un "terreno" digitale vuoto. Ti viene fornita una macchina virtuale (VM) su cui non c'è nulla, se non l'hardware virtuale. Sei tu a scegliere e installare il sistema operativo (Windows, Linux), e poi tutte le tue applicazioni sopra di esso. Hai molto controllo, ma devi comunque gestire il sistema operativo e ciò che ci gira sopra;
- Platform as a Service (PaaS): Questo è un passo avanti in termini di comodità. In PaaS, non affitti solo il "terreno", ma un "terreno già preparato" con la fondazione. Il fornitore del cloud ti offre già un ambiente di lavoro completo: il sistema operativo è pre-installato, ci sono già i runtime per il tuo linguaggio di programmazione (es. Java, Python) e magari strumenti come Docker. Tu devi solo caricare il tuo codice o le tue applicazioni, senza preoccuparti del server sottostante. È ideale per gli sviluppatori che vogliono concentrarsi solo sul codice;
- Software as a Service (SaaS): questo è il livello con più "chiavi in mano". Qui, è come se affittassi l'uso di un "appartamento già arredato". Non gestisci né l'hardware né il software di base, usi direttamente l'applicazione già pronta. Esempi classici sono Gmail, Dropbox (per l'archiviazione), o servizi cloud che ti offrono direttamente un web server (come Apache o Nginx gestito), un database server (come MySQL o PostgreSQL) o un broker di messaggi (per la comunicazione tra applicazioni), senza che tu debba installarli o configurarli. Li usi e basta.

Function-as-a-Service (FaaS):

Il FaaS è una delle offerte più recenti e interessanti del cloud, simile al SaaS ma con una granularità ancora più fine. Sostanzialmente ti permette di creare e caricare direttamente delle singole funzioni (piccoli blocchi di codice) nel cloud, scegliendo il linguaggio di programmazione che preferisci. Queste funzioni non sono sempre attive, ma vengono associate a degli eventi. Ad esempio, una funzione potrebbe avviarsi quando un'immagine viene caricata su un servizio di storage, o quando arriva un messaggio in una coda. Vengono eseguite solo quando c'è bisogno, allocando al volo le risorse di calcolo necessarie (CPU, RAM) solo per il tempo di esecuzione della funzione stessa. Il vantaggio enorme è che tu, come utente, non devi preoccuparti nemmeno del processo server che fornisce quella funzione. Non devi creare server virtuali, container, o configurare nulla. Scrivi la tua funzione, la carichi, e il cloud pensa al resto.

Il FaaS è spesso chiamato anche "serverless computing". Questo nome è un po' fuorviante, perché i server ci sono eccome! Semplicemente, tu, l'utente, non li vedi e non li gestisci. È il fornitore del cloud che si occupa di tutta l'infrastruttura sottostante, liberandoti da questo onere.

Tra i servizi commerciali più famosi troviamo:

1. AWS Lambda
2. Google Cloud Functions
3. Microsoft Azure Functions
4. IBM Cloud Functions (basato su Apache OpenWhisk)
5. Oracle Cloud Functions

Granularità estrema:

Il FaaS rappresenta una divisione del calcolo ancora più fine rispetto ai microservizi. Con i microservizi hai un intero servizio (es. "gestione ordini"), con FaaS hai una singola funzione all'interno di quel servizio (es. "calcola totale ordine" o "invia email di conferma"). È ideale per gestire task specifici e occasionali con massima efficienza e con costi a consumo molto bassi (paghi solo quando la funzione viene eseguita).