

# Cyberphysical Systems

## Sistemi distribuiti e di transazione

### Sistemi distribuiti

Un sistema informativo è considerato **"distribuito"** quando i suoi moduli operano **simultaneamente su computer indipendenti e interconnessi**, chiamati **"nodi"**.

Questi sistemi vengono progettati per migliorare la **scalabilità**, la **disponibilità** e la **resilienza** delle applicazioni, permettendo di **distribuire il carico** di lavoro su più macchine.

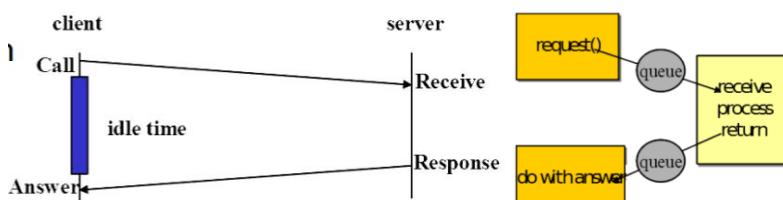
Richiedono determinati standard di **interoperabilità** per garantire che diversi nodi possano **comunicare e collaborare efficacemente**. Questi standard forniscono le modalità con cui i dati e le istruzioni vengono scambiati tra i nodi.

Relativamente alla comunicazione tra i nodi, vengono utilizzate due tipologie di interfacce:

- **API:** basate su **RPC**, permettono a un programma di **eseguire una procedura** su un altro modo come se fosse un nodo locale facilitando la comunicazione tra moduli distribuiti e rendendo il sistema **modulare e facile da gestire**
- **Protocolli di comunicazione e formati di interscambio dati:** forniscono regole per la **trasmissione dei dati** tra moduli; i formati di interscambio dati, come **JSON** e **XML** **standardizzano la struttura** dei dati trasmessi, garantendo che i nodi possano interpretare correttamente le informazioni ricevute

### Modelli di comunicazione

- **Sincroni (bloccanti):** più facili da implementare, performance peggiori
- **Asincroni (non-bloccanti):** basati su code o su eventi, performance migliori, miglior disaccoppiamento, richiede più risorse, più difficile da progettare e implementare

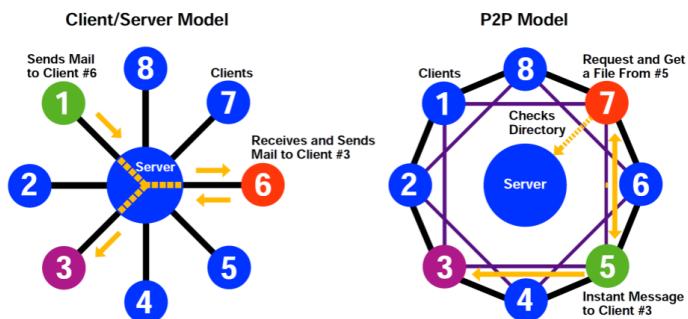


(a SX, modello sincrono; a DX modello asincrono)

### Architetture

- **Client-Server:** Esistono due tipi di nodi

- **Server:** Ricevono richieste e forniscono risposte, possono essere multi-thread o dispatcher-worker
- **Client:** Fanno richieste
- **Peer-to-peer (P2P):** Tutti i nodi sono omogenei, ogni peer può fare sia da client che da server, è possibile dividerli in:
  - **P2P puri:** Senza server centrali o router
  - **P2P ibridi:** Nodo centrale che tiene traccia dei peer per facilitare il bootstrapping della rete



## Peer to Peer

Il computing peer-to-peer (P2P) è un modello di calcolo basato su rete per applicazioni in cui i computer condividono risorse tramite scambi diretti tra i computer partecipanti.

[Barkai, *Peer-to-Peer Computing*, Intel Press, 2001]

La P2P è un'architettura di rete in cui i nodi sono relativamente uguali. Ogni nodo è in grado di svolgere tutte le funzioni necessarie per supportare la rete e, nella pratica, molti nodi svolgono molte di queste funzioni.

[Clarke, *Journal of Theoretical and Applied Electronic Commerce Research*, 1(3), 42-57, 2006]

Questo modello permette ai peer di interagire tra di loro in modo **diretto**, dando all'ambiente la **decentralizzazione**. Il calcolo, pertanto, **viene spostato sull'edge** (sul bordo); se nei modelli classici c'era **un'asimmetria** tra il server e il client, dovuta al numero di richieste in entrata e in uscita, in questo caso **le operazioni di calcolo vengono eseguite sui nodi periferici (peer)** e i **nodi centrali hanno un carico di lavoro molto ridotto**.

La prima infrastruttura P2P è stata ARPANet nel 1969 ma nel 1999 fu Napster a coniare il termine. Grazie a quest'ultimo se ne diffuse anche l'utilizzo ma aveva dei (piccolissimi) problemi con la diffusione di materiale coperto da copyright.

## Stack di P2P

- 1) **Applicazioni P2P:** diverse **applicazioni** per diverse necessità, ovvero:
  - a. **Condivisione di contenuti:** Napster, Gnutella, KaZaA
  - b. **Condivisione di risorse hardware:** SETI@home
  - c. **Comunicazione e messaggistica istantanea:** Skype
- 2) **Interfacce delle Applicazioni P2P:** permettono alle applicazioni di **comunicare** tra di loro e con il middleware, abbiamo anche le **API**
- 3) **Middleware P2P:** software che **gestisce la comunicazione e la gestione delle risorse** tra i nodi della rete
- 4) **Sistemi operativi locali:** presente su ogni nodo, ne **gestisce** le risorse **hardware e software**, devono essere compatibili con il middleware P2P
- 5) **Piattaforme Hardware:** **computer, server** e altri dispositivi che **partecipano alla rete** ospitano tutta l'infrastruttura

## Vantaggi del modello

- **Evitare bottleneck e SPOF** grazie alla decentralizzazione
- **Migliore resilienza e scalabilità**, gestiscono bene anche picchi improvvisi di carico
- **Robustezza** contro DoS
- **Facile condivisione** di risorse e servizi
- **Basso costo di implementazione** della rete (rispetto a una rete centralizzata)

## Sfide

- **Interoperabilità cross-platform** per far comunicare i nodi su diverse piattaforme
- **Gestione di firewall, NAT e IP dinamici**
- **Dinamicità**, ovvero la possibilità da parte dei peer di lasciare la rete con eventuali **difficoltà** relative alla **consistenza**
- **Scoperta di servizi** e risorse particolarmente difficile
- **Implementazione di meccanismi di sicurezza** robusti nei nodi locali
- **Mantenimento della fiducia** tra i peer

## Topologia di rete

Lo **studio** delle **topologie** di reti P2P è **fondamentale** per **migliorare le prestazioni** e le proprietà di **resilienza** di queste reti. Queste vengono modellate come dei **grafi**, dove i **nodi** rappresentano gli **agenti** nella rete e i **collegamenti** rappresentano i **canali di comunicazione**.

Volendo riassumere:

- **Nodi (vertici):** agenti nella rete, ovvero i peer

- **Collegamenti (archi)**: canali di comunicazione tra i nodi
- **Grado di un nodo**: numero di collegamenti che un nodo ha con gli altri nodi
- **Grafo regolare**: grafo in cui ogni nodo ha lo stesso numero di collegamenti

## Modelli di rete

### *Reti Casuali (modello di Erdos-Rényi)*

Si costruisce un **grafo** a partire da una **distribuzione normale**  $G(n,p)$ , dove  $n$  è il numero di nodi e ogni possibile collegamento ha una probabilità  $p$  di esistere e il numero di collegamenti per nodo è approssimativamente costante. Il problema relativo a questo modello è che le reti P2P **raramente si adattano** a questo modello a causa della loro **natura dinamica** e delle **variazioni nella connettività dei nodi**

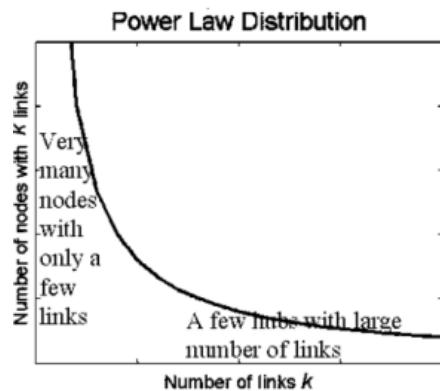
### *Modello di Watts-Strogatz*

Questo modello prevede la **riconnessione casuale** di un grafo regolare per introdurre **piccole variazioni** nella struttura della rete. Il risultato è una **rete con proprietà di "piccolo mondo"**, caratterizzata da **brevi cammini medi** tra i nodi e un **alto grado di clustering**

### Small-world Networks (SWN)

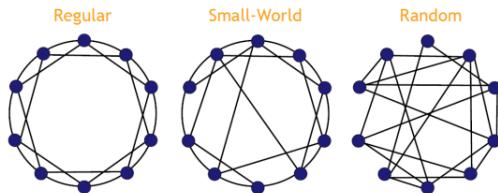
Tipologia di rete caratterizzata da una **struttura** in cui la maggior parte dei nodi ha **pochi collegamenti locali**, mentre un **piccolo numero** di nodi ha **collegamenti estesi**. Questo tipo di rete è **altamente resiliente** ai guasti causati ma **vulnerabile** agli attacchi coordinati contro i **nodi centrali (hub)**. Relativamente alle SWN sono utili i concetti di:

- **Preferential Attachment**: la **probabilità** di nuovi collegamenti è **proporzionale** al **grado** dei **nodi** (modello di Barabàsi-Albert), produce una distribuzione di grado secondo una **legge di potenza**



- **Scale-free Networks**: le **proprietà** della rete **non dipendono** dalla sua **dimensione**, questo concetto è applicabile alle **reti sociali** (esperimento dei sei gradi di Milgram, ogni

persona è collegata a qualsiasi altra attraverso una **catena** di massimo sei passaggi) e alle pagine web



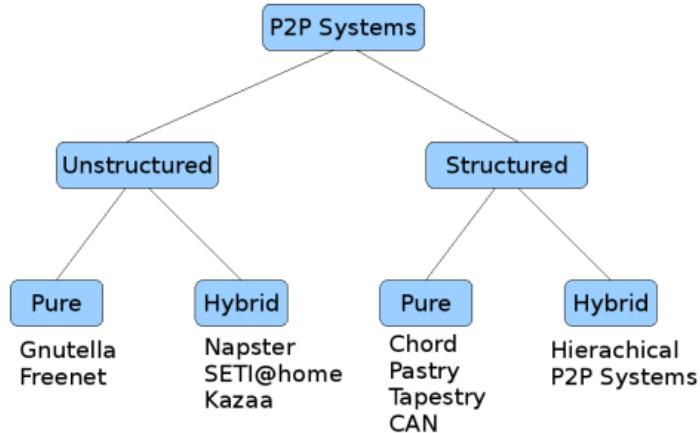
Tra le proprietà delle SWM abbiamo:

- **Lunghezza media** del cammino relativamente **breve** tra due nodi arbitrari
- Gli **hub** forniscono **scorciatoie** che riducono la distanza tra i nodi
- **Altamente resistenti** ai guasti casuali dei nodi
- **Vulnerabili** agli **attacchi coordinati** contro gli hub, possono dividere la rete in **segmenti non comunicanti**

## Overlay Networks

Le reti P2P sono tipicamente **implementate come overlay networks**, ovvero **reti virtuali** di nodi e collegamenti logici che si **sovrappongono** a una **rete esistente** (tipicamente internet). Abbiamo diverse tipologie di overlay:

- **Reti non strutturate:**
  - **Pure:** nessun controllo sulla **topologia** di rete o sulla collocazione di contenuti, le tecniche di routing includono flooding, random walk ed expanding ring, **scarsa efficienza e scalabilità** ma **alta resilienza**
  - **Ibride:** utilizzano **nodi di directory** per cercare peer e/o contenuti (super-peer, tracker)
- **Reti strutturate:**
  - **Pure:** utilizzano **tabelle hash distribuite** (DHT) per collocare i contenuti in posizioni specifiche, non su peer casuali
  - **Ibride:** implementano una **struttura multilivello e gerarchica**, **utilizzata raramente**



## Hash Tables distribuite (DHT)

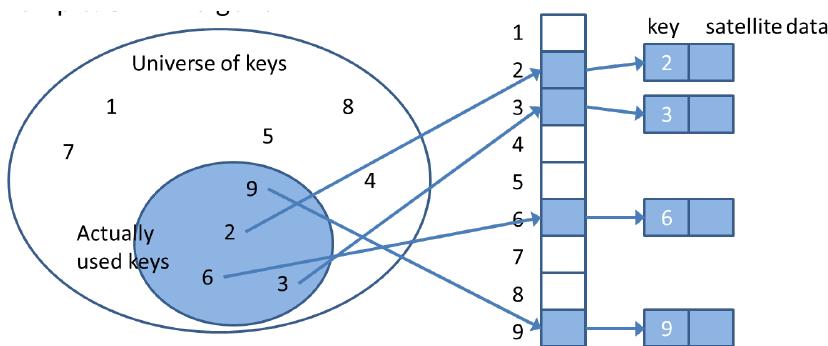
### *Hash Tables*

Una **hash table** è una struttura che permette di **memorizzare e recuperare informazioni** in modo veloce. È contraddistinta da:

- **Array di slot** (chiamati anche bucket)
- **Funzione di hash**, serve per trasformare una chiave (un ID o un nome) in un indice dell'array

Supporta le operazioni di inserimento, cancellazione e ricerca in tempo (medio) costante. Per essere una buona funzione di hash deve:

- **Essere veloce da calcolare**
- **Essere difficile da invertire**
- **Distribuire uniformemente** le chiavi nei bucket



In modo specifico, ci interessano le **DHT**, queste sono una versione **distribuita** delle Hash Table, dove i bucket sono distribuiti tra diversi nodi (computer o dispositivi), in questo caso:

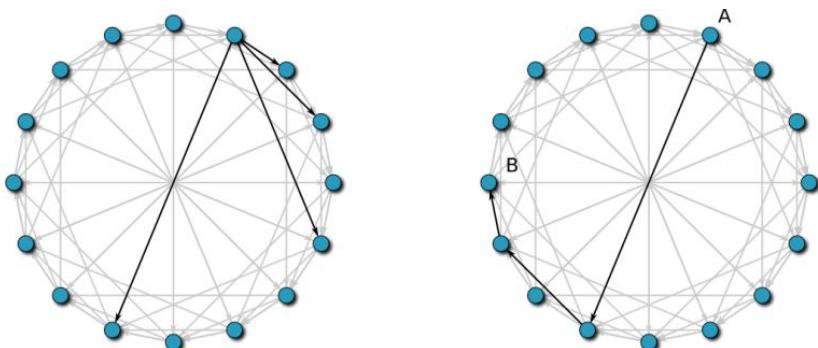
- Ogni **nodo gestisce** uno o più **bucket**
- I nodi possono **entrare o uscire** dalla rete, i **compiti** vengono **ridistribuiti**
- Le operazioni sono le stesse: get e put
- I nodi **collaborano** per trovare chi è **responsabile** di una certa chiave

### Routing

In una rete con N nodi, ogni nodo ha **informazioni minime** per **instradare** le **richieste**. Una richiesta **parte** da un **nodo** e viene **inoltrata** fino al raggiungimento del nodo **responsabile**. Il numero di passaggi (hop) è  $O(\log N)$  quindi **cresce lentamente** se la rete è grande

### Chord

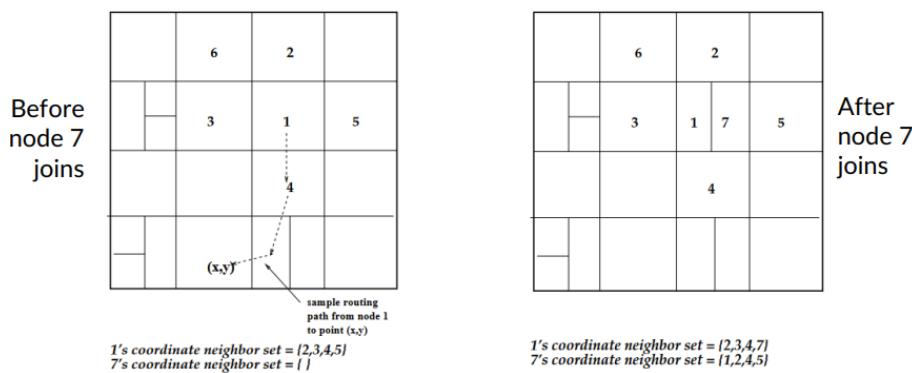
Con una **topologia ad anello**, ogni **nodo** ha un **successore** e un **predecessore** (in senso orario). Il nodo responsabile di una chiave k è il primo nodo con ID  $\geq k$ . Usa una **finger table** per cercare in modo efficiente (sempre  $O(\log N)$ ) e per essere più resiliente ai guasti, i nodi **memorizzano** i nodi **adiacenti**



### CAN

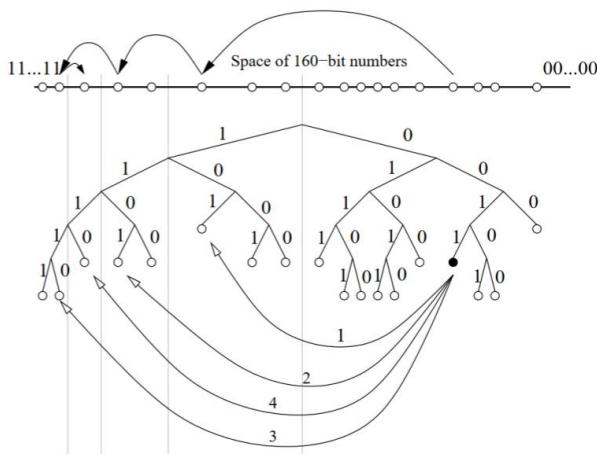
CAN sta per “Content-Addressable Network), la **topologia** può essere uno **spazio a due o più dimensioni**, ogni nodo ha coordinate e conosce i suoi vicini. Le risorse sono **mappate** in punti dello spazio in base all'output della funzione di hashing.

La ricerca ha complessità  $O(\log_d N)$  con d numero di dimensioni.



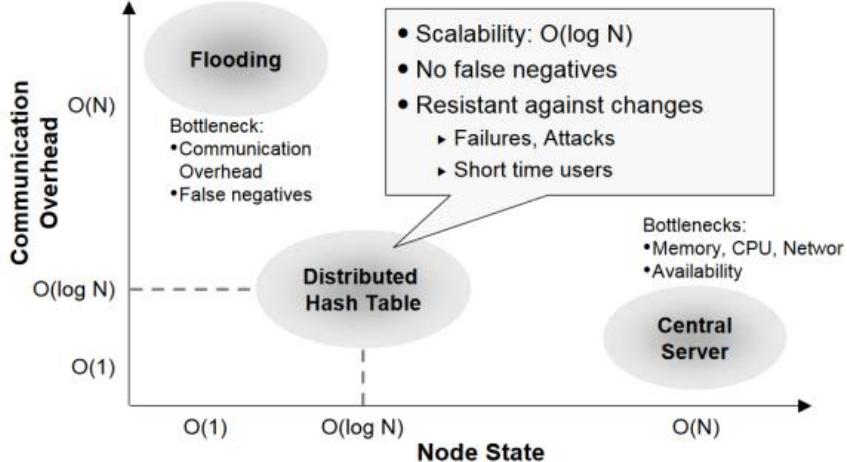
### Kademlia

La topologia utilizzata è quella di un albero binario. La distanza tra i nodi viene calcolata facendo XOR con gli ID. Ogni nodo ha una serie di k-bucket con nodi a distanze diverse e la rete è divisa, via via, in sezioni sempre più piccole. In questo caso la ricerca ha complessità  $O(\log_2 N)$



### Proprietà delle DHT

Sistema	Stato per nodo	Overhead comunicazione	Query complesse	Robustezza
<b>Server centrale</b>	$O(N)$	$O(1)$	✓	✗
<b>Ricerca flooding</b>	$O(1)$	$\geq O(N^2)$	✓	✓
<b>DHT</b>	$O(\log N)$	$O(\log N)$	✗	✓



Per essere affidabile, un sistema distribuito deve essere:

- **Affidabile\*** (**dependability**)
  - **Disponibile:** pronto a fornire servizio
  - **Affidabile:** servizio continuo e corretto
  - **Sicuro:** nessun danno per utenti o ambiente
  - **Integro:** nessuna modifica non autorizzata
  - **Manutenibile:** facile da aggiornare e riparare
- **Resiliente**
  - **Tollerante ai guasti:** continua a funzionare anche se alcuni nodi falliscono
- **Sicuro**
  - **Disponibilità:** accesso garantito
  - **Integrità:** dati non alterati
  - **Confidenzialità:** nessuna divulgazione non autorizzata

## Sistemi transazionali

Un sistema transazionale è un **sistema informatico** che **gestisce operazioni** su risorse in modo **strutturato** e **sicuro**, trattandole come **transazioni**. Una **transazione** è un **insieme** di **operazioni** che devono essere eseguite in **modo unitario**, cioè **o tutte o nessuna**.

Ogni transazione è delimitata da due comandi:

- **BOT** (Begin Of Transaction): ne indica l'**inizio**
- **EOT** (End Of Transaction): ne indica la **fine**

Durante la transazione, il sistema può decidere di:

- **Commit:** **confermare** e rendere **permanenti** le **modifiche**
- **Abort:** **annullare** tutte le **modifiche** effettuate

Questa logica è fondamentale per evitare **incoerenze**.

## Proprietà ACID

Le transazioni devono rispettare quattro proprietà fondamentali, note con l'acronimo **ACID**:

- **Atomicità:** La transazione è **un'unità indivisibile**: o tutte le operazioni vengono eseguite, oppure nessuna
- **Consistenza:** Il sistema deve essere in uno **stato coerente** prima e dopo la transazione. Le **regole** e vincoli del sistema **non** devono essere **violati**
- **Isolamento:** Le **transazioni** devono essere **indipendenti**. Anche se eseguite in parallelo, il risultato deve essere equivalente a una loro esecuzione in sequenza
- **Durabilità:** Una volta confermata (commit), una transazione non può essere persa. I suoi effetti devono **persistere** anche in caso di **guasti**

In generale, ci sono dei modi per garantire queste proprietà:

- **Atomicità e durabilità** sono gestite tramite il **controllo di affidabilità**, che si basa anche su operazioni atomiche a livello hardware
- **Isolamento** è garantito dal controllo della **concorrenza**, che regola l'accesso simultaneo alle risorse
- **Consistenza** dipende dalla **correttezza** delle **operazioni implementate** nel sistema

## Controllo di affidabilità

Il controllo di affidabilità assicura che:

- Le **transazioni** siano **complete** e i loro effetti siano **permanenti**
- I comandi **bot**, **commit** e **abort** siano **correttamente gestiti**
- Il **sistema** possa essere **ripristinato** in **caso di guasto**

Per farlo si utilizza un **log**, ovvero un registro che salva informazioni ridondanti su memoria stabile. Questi possono essere di due tipologie:

- **Log transazionali:** registrano le operazioni delle transazioni
- **Log di sistema:** registrano le operazioni generali del sistema

I meccanismi di supporto, a loro volta, possono essere:

- **Checkpoint:** **salvataggio periodico** dello **stato** corrente delle transazioni attive
- **Dump:** **copia completa** e coerente dello stato del sistema, effettuata solo quando non ci sono transazioni attive

## Controllo di concorrenza

Eseguire le transazioni una alla volta è semplice ma inefficiente. Il controllo della concorrenza permette di eseguire **transazioni in parallelo**, mantenendo però un comportamento equivalente a quello seriale. Questo deve evitare i seguenti problemi:

- **Lost updates**: due transazioni scrivono sullo stesso dato, perdendo una modifica
- **Dirty reads**: una transazione legge dati che poi vengono annullati
- **Unrepeatable reads**: una transazione legge due volte lo stesso dato, ma ottiene risultati diversi
- **Phantom updates/inserts**: una transazione vede un insieme di dati incoerente a causa di modifiche concorrenti

Quando più transazioni accedono alla stessa risorsa, possono verificarsi conflitti. Per gestirli, si usano i lock:

- **s\_lock**: blocco condiviso (lettura)
- **x\_lock**: blocco esclusivo (scrittura)
- **unlock**: rilascio del blocco

Request	Resource state		
	free	s_locked	x_locked
s_lock	OK / s_locked	OK / s_locked *	NO / x_locked
x_lock	OK / x_locked	NO / s_locked	NO / x_locked
unlock	error	OK / depends ^	free

\*: increase s\_lock counter

^: decrease counter, if 0 resource becomes free

#### Blocco a due fasi (2PL) e deadlock

Il protocollo Two-Phase Locking (2PL) impone che una transazione non possa acquisire nuovi lock dopo averne rilasciati. Questo ci garantisce la serializzabilità, cioè l'equivalenza con una esecuzione in sequenza.

Questo potrebbe però portarci a un deadlock, ovvero uno stato in cui due o più transazioni si bloccano a vicenda, aspettando risorse che non verranno mai liberate. Per questo motivo esistono delle soluzioni:

- **Timeout**: se una transazione aspetta troppo, viene interrotta
- **Prevenzione**: politiche che evitano deadlock
- **Rilevamento**: costruzione di un grafo delle attese ed eventuale interruzione

## Sistemi transazionali distribuiti

I sistemi transazionali distribuiti nascono per rispondere a due esigenze fondamentali:

- 1) **Aumentare la velocità di elaborazione**: sfruttando il parallelismo, è possibile eseguire più transazioni contemporaneamente migliorando il numero di operazioni completate per unità di tempo

## 2) Migliorare l'affidabilità e la disponibilità dei dati: distribuendo le risorse su più nodi si riduce il rischio di perdita di dati, garantendo un accesso più continuo

Una transazione distribuita coinvolge più nodi della rete, ma ogni singola operazione all'interno della transazione agisce su un solo nodo. Questo significa che la transazione è composta da sottotransazioni locali, ciascuna gestita da un nodo specifico.

La distribuzione influisce anche sulle proprietà ACID:

- **Consistenza e Durabilità:** gestite localmente da ciascun nodo, non dipendono dalla distribuzione
- **Isolamento:** richiede un controllo della concorrenza distribuito per evitare interferenze tra transazioni parallele
- **Atomicità:** richiede un controllo di affidabilità distribuito per garantire che tutti i nodi coinvolti raggiungano lo stesso esito (commit o abort)

### Controllo della concorrenza distribuito

Quando una transazione è distribuita, questa viene suddivisa in sottotrasazioni eseguite sui vari nodi. Tuttavia, anche se ogni nodo garantisce la serializzabilità locale, questo non basta per assicurare la serializzabilità globale. Per ottenere una corretta esecuzione globale:

- Ogni nodo deve applicare il protocollo Two-Phase Locking (2PL)
- Il commit deve avvenire solo quando tutte le sottotrasazioni hanno acquisito le risorse necessarie

NB.: nei sistemi distribuiti, i deadlock possono verificarsi tra nodi diversi. La soluzione più comune è l'utilizzo dei timeout, poiché la rilevazione distribuita dei deadlock (ad esempio con i grafi di attesa) è complessa e poco efficiente

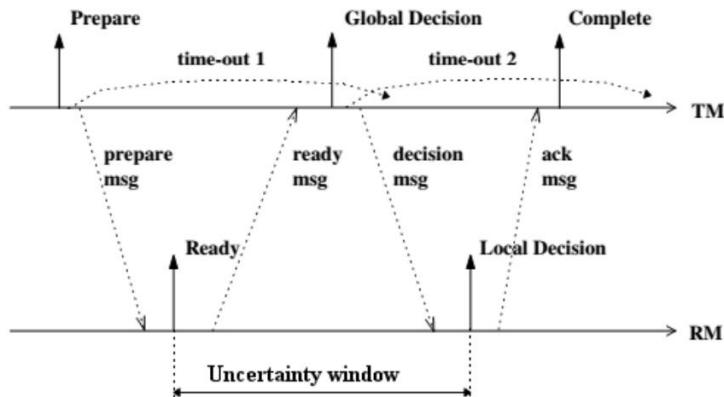
### Controllo di affidabilità distribuito

Per garantire l'atomicità in una transazione distribuita, tutti i nodi devono concordare sull'esito finale, dando come risultato un commit o un abort. Questo viene gestito mediante il protocollo di commit a due fasi (2PC)

Abbiamo due attori:

- **Transaction Manager (TM):** uno per rete, coordina la transazione
- **Resource Manager (RM):** uno per nodo, gestisce le risorse locali

Entrambi scrivono su un log per garantire la persistenza e la possibilità di recupero in caso di guasti



Analizziamo il protocollo 2PC, nello specifico il caso di esecuzione senza guasti:

### 1) Fase 1 – Preparazione

- Il TM scrive un **record** di prepare nel log e invia un **messaggio** ai RMs per iniziare il 2PC
- Ogni RM, se in stato affidabile, scrive un **record ready** e risponde al TM
- Il TM raccoglie le risposte:
  - Se tutti i RMs sono pronti, scrive global commit
  - Altrimenti scrive global abort

### 2) Fase 2 – Decisione

- Il TM comunica la **decisione finale** ai RMs
- Ogni RM scrive il proprio **record** di commit o abort e invia un ack al TM
- Il TM raccoglie gli ack:
  - Se tutti arrivano, scrive complete
  - Altrimenti ripete la trasmissione finché tutti rispondono

Relativamente alla gestione dei guasti, ci potrebbero essere due casi:

- Guasto di un RM:** Il RM recupera il proprio stato dal log, in particolare:
  - Se l'ultimo record è commit, ripete le azioni
  - Se è abort, annulla le azioni
  - Se è ready è in uno stato incerto, bisogna contattare il TM per conoscere lo stato finale
- Guasto di un TM:** il TM può perdere messaggi ma il log conserva lo stato del protocollo, in particolare:
  - Se l'ultimo record è prepare, può decidere un abort o ripetere la fase di preparazione

- Se è global commit o abort, ripete la fase di decisione
- Se è complete, il guasto non ha alcun effetto sulla transazione

Volendo fare ulteriori considerazioni, c'è bisogno di un TM **affidabile** e **fidato** da tutti i RM. Oltre a questo, il TM deve essere anche dimensionato bene, altrimenti potrebbe fare **bottleneck**.

Il protocollo **2PC** è **bloccante** perché richiede **scritture sincrone** su log, per questo motivo si fanno delle **ottimizzazioni**, tra le più comuni abbiamo:

- **Presumed abort**: si presume l'abort in caso di recupero remoto, riducendo il numero di scritture sincrone
- **Presumed commit**: approccio opposto a quello precedente, si presume il commit
- **Read-only**: i RMs che eseguono solo letture non influenzano l'esito della transazione e possono essere ignorati nella fase di decisione

## Guasti e modelli di avversario nei sistemi distribuiti

Per progettare sistemi distribuiti **affidabili**, è fondamentale comprendere che tipo di guasti possono verificarsi. Una classificazione utile considera diverse dimensioni:

- **Fase di creazione**
  - Guasti di **sviluppo**: errori nel progetto o nel codice
  - Guasti **operativi**: si verificano durante l'esecuzione del sistema
- **Confini del sistema**:
  - Guasti **interni**: originati dall'interno del sistema (Es.: bug software)
  - Guasti **esterni**: causati da fattori esterni (Es.: blackout, attacchi)
- **Causa**:
  - **Naturale**: eventi fisici (Es.: fulmini, usura hardware)
  - **Umana**: errori o azioni intenzionali
- **Dimensione**
  - **Hardware**
  - **Software**
- **Obiettivo**
  - **Malevoli**: causati da attacchi o sabotaggi
  - **Non-malevoli**: causati da errori o incidenti
- **Intento**
  - **Deliberati**: azioni intenzionali
  - **Non deliberati**: errori involontari
- **Capacità**
  - **Accidentali**: causati da eventi casuali
  - **Incompetenza**: dovuti a mancanza di competenze
- **Persistenza**
  - **Permanenti**: richiedono intervento per essere risolti

- **Transitori:** si risolvono da soli o con il tempo

#### *Modelli di avversario e tolleranza ai guasti*

Modello	Descrizione	Tolleranza ai guasti	Max f tollerabile
Fail-stop	Un nodo può semplicemente smettere di funzionare	CFT (Crash Fault Tolerant)	N/2
Byzantine	Un nodo può comportarsi in modo arbitrario e malevolo	BFT (Byzantine Fault Tolerant)	N/3

Per garantire **affidabilità** e **sicurezza**, i sistemi distribuiti **replicano dati e operazioni** su più nodi. Tuttavia, la **replica** da sola **non basta**, è necessario un **meccanismo di consenso** per assicurare che tutti i nodi eseguano le stesse operazioni nello stesso ordine. Per questo sono fondamentali:

- Una **macchina a stati deterministica** che definisca il comportamento del servizio
- Un **protocollo di consenso** che coordini i nodi

Per esempio, nei sistemi DLT (Distributed Ledger Technology), come le blockchain, tutti i nodi validano le transazioni per garantire la fiducia nel sistema. La replica non serve alla scalabilità ma alla resilienza

#### Atomic broadcast

Il tipo di consenso più rilevante per sistemi replicati è l'**atomic broadcast**. Questo garantisce che tutti i nodi corretti **ricevano e processino** la stessa **sequenza** di messaggi

Tra le sue proprietà abbiamo:

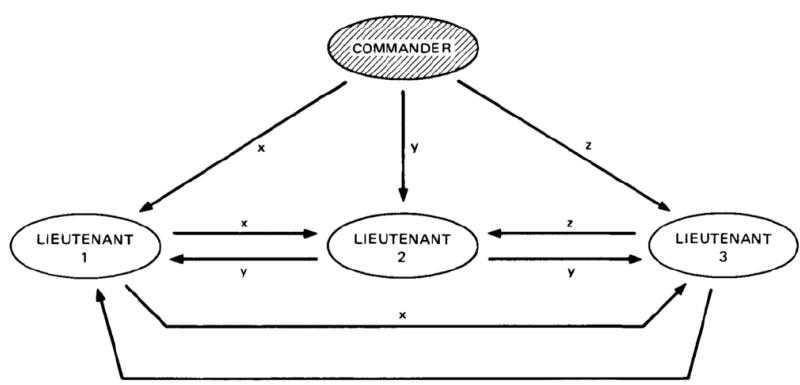
- **Validità:** se un nodo corretto invia un messaggio, lo riceverà
- **Accordo:** se un nodo corretto riceve un messaggio, lo riceveranno anche gli altri
- **Integrità:** nessun messaggio viene ricevuto più di una volta
- **Ordine totale:** tutti i nodi ricevono i messaggi nello stesso ordine

#### Problema dei generali bizantini

Il problema dei generali bizantini è un tipico problema della teoria di sistemi distribuiti, mostra le **difficoltà di coordinamento** in presenza di nodi malevoli.

Si immagini uno scenario in cui dei **generali devono attaccare** insieme e alcuni di questi possono essere **traditori**. L'obiettivo è che tutti i generali devono **concordare** su un **piano**, anche se **alcuni mentono**.

L'algoritmo, anche chiamato **Oral Message (OM)** funziona solo se  $N \geq 3f + 1$ , ovvero può tollerare fino a  $f < N/3$  traditori





## Blockchain e DLT in generale

La blockchain è una tecnologia che consente di creare un registro distribuito e immutabile delle transazioni digitali. In pratica, è una base di dati condivisa tra più partecipanti, dove ogni operazione è registrata in modo permanente e verificabile.

Il termine blockchain può riferirsi a:

- La **struttura dati** che **organizza** le **transazioni** in blocchi concatenati
- Le **piattaforme** e **protocolli** che permettono la condivisione e la gestione di questi registri distribuiti

*La blockchain è nata nel 2008 con la pubblicazione del whitepaper Bitcoin: A Peer-to-Peer Electronic Cash System da parte di Satoshi Nakamoto (pseudonimo). L'implementazione è stata rilasciata nel 2009 come software open source.*

*Sebbene Bitcoin come valuta sia oggetto di dibattito, la tecnologia sottostante ha dimostrato grande efficacia e ha ispirato un'intera generazione di Distributed Ledger Technologies (DLT).*

*Prima di Bitcoin, la ricerca sulle DLT era limitata da problemi tecnici e organizzativi considerati insormontabili. Il successo di Bitcoin ha riacceso l'interesse, portando a miglioramenti e nuove applicazioni.*

## Transazioni

Se volessimo fare un confronto con i sistemi tradizionali (Es.: le banche), una **transazione** implica il **trasferimento** di un **bene** (Es.: denaro) **da un mittente a un destinatario**. Per garantire la sicurezza c'è bisogno di un **terzo fidato** (come una banca) e che questo:

- **Verifichi** la transazione
- **Protegga** i dati da manomissioni
- **Conservi** lo storico delle operazioni

La blockchain elimina la necessità di questo intermediario, in quanto:

- Ogni transazione è **verificata** da un insieme di partecipanti (nodi)
- Una volta **registrata** non può essere **modificata o cancellata**
- La **validazione** avviene tramite **consenso**, secondo regole definite dal protocollo

Per validare una transazione bisogna verificare che tutte le condizioni necessarie siano soddisfatte.

Es.: Se Bob vuole trasferire un asset ad Alice:

- Bob deve essere il legittimo proprietario dell'asset
- Bob deve avere sufficiente disponibilità

Queste verifiche devono essere **automatiche** e **decentralizzate**, grazie alla crittografia e al consenso

In tutto questo, la blockchain opera su una rete peer-to-peer, dove:

- Non esistono autorità centrali
- Ogni nodo è identificato da una **chiave pubblica**
- Le transazioni sono **firmate** con la **chiave privata** del **mittente**
- Ogni transazione è trasmessa a **tutti i nodi**, **validata** e poi **registrata** nel registro pubblico

Questa struttura consente la **collaborazione senza fiducia** tra entità autonome, come le **Decentralized Autonomous Organizations (DAOs)**, che si occupano di gestire fondi, votazioni o smart contract senza bisogno di un ente centrale

### Ordinamento delle transazioni

La blockchain è una struttura composta da una **sequenza di blocchi**, ognuno dei quali contiene un **insieme di transazioni**. Ogni blocco è collegato al precedente tramite un **hash crittografico**, che garantisce l'**integrità** della catena: se si modifica anche solo una transazione in un blocco, l'hash cambia, invalidando tutti i blocchi successivi.

Le transazioni vengono gestite in questo modo:

- 1) Quando una transazione viene **creata**, viene **trasmessa** a tutti i nodi della rete
- 2) Ogni nodo la **memorizza** in una struttura chiamata transaction pool (o mempool in Bitcoin, txpool in Ethereum)
- 3) Periodicamente i nodi **selezionano** le transazioni dal mempool in base a criteri come:
  - a. **Commissioni di transazione** (più alte = maggiore priorità)
  - b. **Ordine di arrivo**
  - c. **Dimensione del blocco**

Queste transazioni vengono poi raggruppate in un nuovo blocco che il nodo propone alla rete

### Costruzione della blockchain: il consenso

Poiché qualsiasi nodo può **proporre** un nuovo blocco, è possibile che più blocchi vengano **creati contemporaneamente**. Tuttavia, la rete deve decidere quale blocco accettare come successivo nella catena.

Questo potrebbe creare dei problemi, in quanto i blocchi possono arrivare in **ordine diverso** a seconda del nodo, a causa di ritardi di rete.

Per evitare confusione e garantire coerenza, la blockchain utilizza un **protocollo di consenso**, cioè un algoritmo distribuito che permette alla rete di **accordarsi** su un solo blocco da aggiungere

## Il problema del double spending

Un rischio concreto nelle criptovalute è il **double spending**, cioè la possibilità che una stessa moneta venga spesa due volte. Questo può accadere se due transazioni concorrenti vengono propagate in momenti diversi ai nodi.

Per evitare questo problema è fondamentale che la rete **concordi** sull'ordine delle transazioni, qui entra in gioco il protocollo del consenso

## Protocolli di consenso

Bitcoin ha introdotto il **Proof of Work** (PoW), un meccanismo basato sulla **crittografia** per raggiungere il **consenso**. I nodi competono per risolvere un problema matematico complesso e il primo che lo risolve può proporre il suo blocco come prossimo nella catena, mentre gli altri nodi verificano facilmente il blocco.

Volendo fare un esempio pratico, il nodo deve trovare un nonce (numero casuale) tale che, una volta combinato con le transazioni e l'hash del blocco precedente, produca un hash inferiore a una soglia prestabilita. Il punto importante di questa logica è che il processo deve essere difficile da risolvere ma facile da verificare, garantendo sicurezza e trasparenza.

La **difficoltà** del problema viene **regolata automaticamente** per mantenere costante il tempo medio di creazione dei blocchi.

Ese.: in Bitcoin 1 blocco ogni 10 minuti, regolato ogni 2016 blocchi

In particolare, **PoW** è BFT, ovvero **tollerante ai guasti bizantini**, può raggiungere il consenso anche se alcuni nodi sono malintenzionati o non funzionano correttamente.

PoW è sicuro ma **energivoro**, pertanto sono stati sviluppati anche altri protocolli:

- **Proof of Stake (PoS)**: chi possiede più monete ha più probabilità di proporre un blocco
- **Delegated Proof of Stake (DPoS)**: gli utenti votano dei delegati che propongono più blocchi
- **Proof of Authority (PoA)**: solo nodi autorizzati possono proporre blocchi
- **Proof of Elapsed Time (PoET)**: selezione casuale basata sul tempo di attesa

Tutti questi protocolli hanno come obiettivo il **ridurre il consumo energetico e aumenta la scalabilità**

## Mining

I nodi che partecipano alla risoluzione del problema PoW sono chiamati **miner**. Questi vengono ricompensati con una piccola quantità di criptovaluta per ogni blocco valido che aggiungono.

Relativamente al mining, c'è stata una notevole evoluzione nel corso degli anni:

- **CPU mining**: utilizzato solo inizialmente, inefficiente
- **GPU mining**: più potente, principale causa dell'aumento dei prezzi di gpu

- **ASIC mining:** standard attuale, vengono usati dispositivi appositi

Volendo strutturare anche delle eventuali implicazioni, abbiamo:

- |  |
|--|
| • <b>Più sicurezza</b>                                   |
| • <b>Maggiore democrazia (e minore centralizzazione)</b> |
| • <b>Consumo energetico molto elevato</b>                |

L'utilizzo della blockchain porta con sé un gran numero di vantaggi:

- |  |
|--|
| <ul style="list-style-type: none"> <li>• <b>Robustezza e tolleranza ai guasti:</b> la rete p2p non dipende da un singolo punto di controllo, il sistema può funzionare anche se alcuni nodi falliscono</li> <li>• <b>Integrità delle transazioni:</b> garantita dal protocollo di consenso, impedisce delle modifiche non autorizzate</li> <li>• <b>Autenticazione e non ripudio:</b> grazie all'utilizzo della crittografia asimmetrica (chiavi pubbliche e private), ogni transazione è firmata e non può essere negata</li> <li>• <b>Trasparenza e verificabilità:</b> tutte le transazioni sono pubbliche e tracciabili, questo elimina dispute e rende inutile la riconciliazione manuale</li> <li>• <b>Assenza di fiducia</b> tra le parti: non serve una autorità centrale, il sistema è trustless ma prevedibile e sicuro</li> <li>• <b>Immutabilità:</b> nessun nodo singolo o piccolo gruppo può alterare i dati, la sicurezza dipende dal consenso globale</li> <li>• <b>Pseudonimato:</b> gli utenti non sono completamente anonimi ma i loro dati personali non vengono esposti. Le transazioni sono tracciabili, utile per indagini e analisi</li> </ul> |
|--|

## Smart contract e applicazioni delle BC

Uno **smart contract (SC)** è un **protocollo informatico** che esegue automaticamente i termini di un **contratto**. Grazie alla BC, gli smart contract sono diventati pratici e sicuri, questi registrano condizioni da soddisfare e azioni da eseguire appena le condizioni sono soddisfatte. L'esecuzione del contratto avviene **senza intermediari** (che siano banche o avvocati). Praticamente si possono implementare con Solidity (per Eth) o altri linguaggi logici.

La blockchain è general-purpose e trova applicazione in moltissimi settori, alcuni legati al mondo finanziario, altri no:

- Finanziario
  - Criptovalute
  - Private securities
  - Assicurazioni
- Non finanziario
  - Servizi notarili: per verificare l'autenticità dei documenti (Proof of Ownership, Existence, Integrity)
  - Mercati digitali: gestione trasparente dei diritti d'autore
  - Storage decentralizzato: condivisione sicura dei file senza provider centralizzati

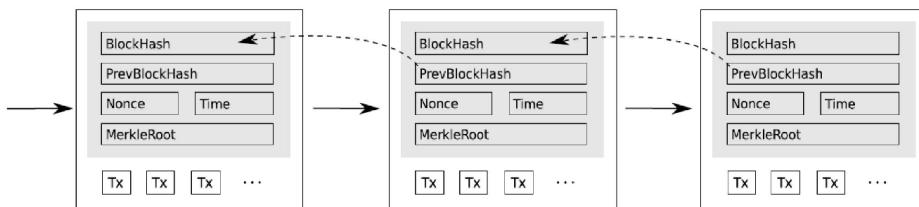
- Internet of Things: scambio sicuro di dati tra dispositivi senza hub centrale
- Anticontraffazione: tracciabilità nella filiera

## Strutture dati della blockchain

Commentato [FP1]: Aggiungere esempi di codice bc

La blockchain può essere vista come un **server distribuito di timestamp** in una **rete p2p**. Il suo scopo principale è creare consenso sulla cronologia delle transazioni, garantendo che:

- Non avvenga double spending (la stessa moneta spesa due volte)
- La catena più lunga (quella con più lavoro computazionale) sia considerata la versione valida



Finché la maggioranza della potenza computazionale è controllata da nodi onesti, la blockchain è resistente agli attacchi.

Per rappresentare le informazioni, la blockchain utilizza tre strutture fondamentali:

- **Transazioni**: rappresentano il **trasferimento di valore** tra utenti
- **Blocchi**: contengono un **insieme di transazioni e metadati**
- **Blockchain**: **catena di blocchi** collegati tra loro tramite hash

### Blocco

Ogni blocco è composto da:

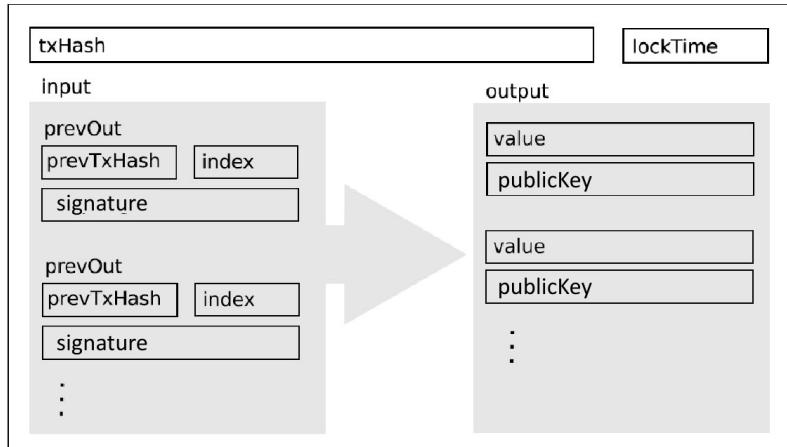
- **Block hash**: identificativo univoco del blocco
- **Previous block hash**: hash del blocco precedente (garantisce il collegamento)
- **Timestamp**: data e ora di creazione
- **Nonce**: numero scelto per soddisfare la condizione del Proof of Work
- **Merkle root**: radice dell'albero di Merkle che rappresenta tutte le transazioni del blocco

### Transazione

Una transazione può avere n input e m output. Nel modello **UTXO (Unspent Transaction Output)**, ogni transazione:

- **Consuma uno o più UTXO** (output non spesi di transazioni precedenti)
- **Genera nuovi UTXO** che potranno essere usati in futuro

Questo garantisce che ogni unità di valore sia spesa una sola volta



## UTXO

- Ogni transazione ha **input** e **output**
- Gli **output** non spesi sono **UTXO** e possono essere usati in nuove transazioni
- Ogni transazione “**consuma**” **UTXO** e ne crea di nuovi (quelli che riceve il destinatario)
- È **semplice e sicuro**
- Ha una **gestione complessa** per gli **smart contract**

## Account-based

- Ogni **account** ha uno **stato** e un **saldo**
- Le **transazioni** **modificano** direttamente il **saldo** degli account
- Supporta pienamente gli **smart contract**
- Lo **stato** viene **gestito** in **DB NoSQL**, la **complessità è maggiore**

Es.: In Bitcoin, per pagare 1 BTC, potresti dover combinare più UTXO (ad esempio 0.6 BTC + 0.4 BTC). In Ethereum, basta sottrarre 1 ETH dal saldo dell'account.

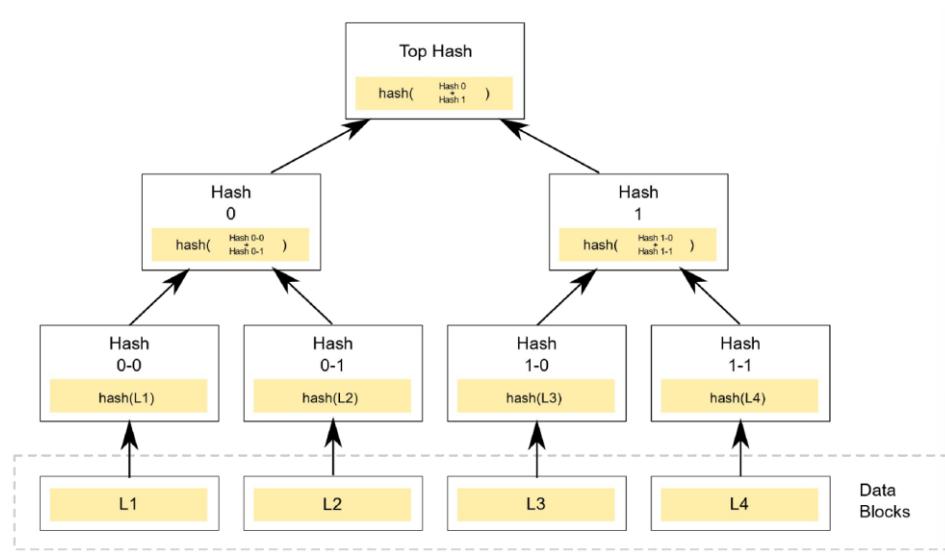
Volendo fare un'analogia, UTXO è come utilizzare delle banconote, possiamo unire banconote diverse per arrivare a un importo specifico. I sistemi account-based sono più simili a un conto corrente

## Merkle Tree

Un **Merkle Tree** (o hash tree) è una **struttura ad albero** in cui ci sono:

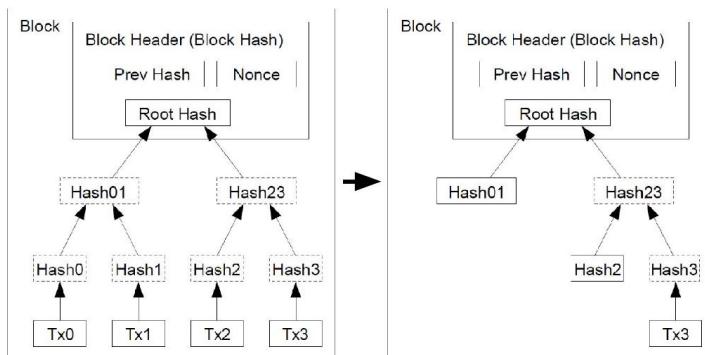
- **Foglie**: ognuno di questi nodi **contiene l'hash** di un **blocco** di dati
- **Nodi interni**: ognuno di questi nodi (non foglia) **contiene l'hash** derivato dai suoi **figli**

Questo permette di **verificare** se un **dato appartiene** a un insieme **senza controllare tutto** in un numero di operazioni di hash di complessità  $O(\log_2 N)$ , con N numero di foglie.



È ideale per applicazioni che gestiscono grandi quantità di dati, in questo caso le blockchain, in cui le transazioni di un blocco vengono organizzate in un Merkle Tree. Nell'hash del blocco è incluso anche il Merkle Root che ci permette di:

- **Verificare l'integrità delle transazioni** solo mediante di esso (senza scaricare tutte le transazioni del blocco)
- **Eliminare** tutte le **transazioni vecchie** senza invalidare l'hash del blocco (pruning)



## Directed Acyclic Graph (DAG)

Oltre alla blockchain, esiste un'altra struttura per registrare transazioni: il **DAG** (grafo aciclico diretto). Ha due semplici regole:

- Tutte le **transazioni** devono **fluire** nella **stessa direzione**
- **Nessuna transazione** può riferirsi a **sé stessa**

In questo caso **non abbiamo** bisogno di **miner** né **consenso** tradizionale, ogni nodo che invia una transazione deve approvare due transazioni precedenti

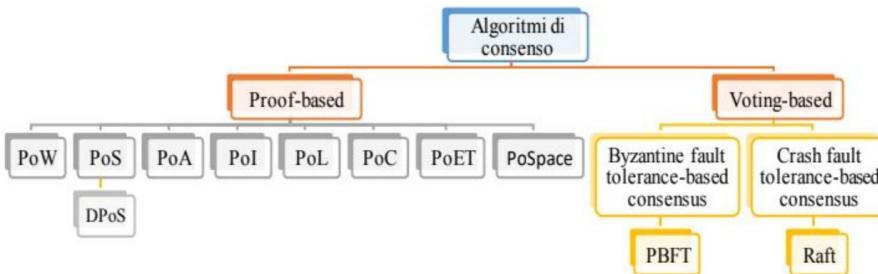
Caratteristica	Blockchain	DAG
Throughput (trans/s)	Basso	Alto
Consumo energetico	Alto	Basso
Latenza transazioni	Alta	Bassa
Scalabilità	Limitata	Elevata
Commissioni	Alte	Basse
Robustezza consenso	Alta	Media
Popolarità	Alta	Minore

# Protocolli di Consenso Blockchain

## Classificazione Generale

I protocolli di **consenso** sono il cuore delle tecnologie blockchain, garantendo l'accordo sullo **stato del sistema distribuito**. Si dividono in due macro-categorie principali:

- **Proof-based:** Richiedono di dimostrare un **dispendio** di **risorse** (lavoro computazionale, possesso di valuta, ecc.). Includono PoW, PoS, PoA, ecc...
- **Voting-based:** Basati sullo **scambio di messaggi e votazioni** tra nodi. Si suddividono in algoritmi tolleranti ai guasti bizantini (BFT) e tolleranti ai guasti di crash (CFT)



## Tipologie di Blockchain

La scelta del protocollo dipende dal tipo di accesso alla rete:

- **Permissionless** (Senza permessi): Chiunque può unirsi, leggere, scrivere e partecipare al consenso senza autorizzazione
  - Alta **esposizione ad attori malevoli** (necessita protocolli robusti), **incentivi economici** per la partecipazione
  - **Protocolli tipici:** Proof of Work (PoW), Proof of Stake (PoS), Delegated Proof of Stake (DPoS)
- **Permissioned** (Con permessi): Solo parti autorizzate possono partecipare
  - Blockchain di consorzio, controllate da un **gruppo predeterminato** di **organizzazioni**
  - **Protocolli tipici:** Practical Byzantine Fault Tolerance (PBFT), usato in Hyperledger Fabric; Federated BFT (Stellar)

## Proof of Work (PoW)

È il protocollo più noto (usato da Bitcoin) e serve a **prevenire gli attacchi Sybil**, dove un'entità **crea false identità** per **sovvertire la rete**. Prima di diffondere un blocco, un nodo deve risolvere una **sfida crittografica** ("Work"). L'assunzione di base è che sia molto più **difficile controllare la maggioranza della potenza** di calcolo della rete piuttosto che creare **identità fittizie**.

Funzionamento in BTC:

- Si usa la **funzione di hash SHA-256**
- Il nodo deve trovare un **numero arbitrario** chiamato **Nonce** tale che l'hash del blocco sia **inferiore** a un valore **target T** (ovvero, l'hash deve iniziare con n zeri)
- Essendo una funzione di hash sicura, l'unico modo per trovare il Nonce è **procedere per tentativi** (brute force)

È come una lotteria. La probabilità di risolvere il blocco per primi è proporzionale alla potenza di calcolo posseduta rispetto al totale della rete ("biglietti della lotteria")

### Aggiustamenti di difficoltà

Per mantenere il **tempo** di generazione dei blocchi **costante** (circa 10 minuti in Bitcoin), il **target** viene **ricalcolato** ogni **2016 blocchi** (circa 2 settimane). La formula di aggiornamento è:

$$T_{new} = T_{old} \cdot \frac{\text{Tempo atteso (2016 min)}}{\text{Tempo effettivo per 2016 blocchi}}$$

Il nuovo target non può **variare** di un fattore maggiore di 4 o minore di 1/4 rispetto al precedente

$$T = \begin{cases} \frac{T_{prev}}{4} & f \leq \frac{1}{4} \\ f \cdot T_{prev} & \frac{1}{4} \leq f \leq 4 \\ 4 \cdot T_{prev} & f \geq 4 \end{cases}$$

### Criticità di PoW

- **Costo Computazionale:** Richiede **incentivi economici** (commissioni di transazione + ricompensa del blocco) per motivare i minatori
- **Scarsità programmata:** Le **ricompense diminuiscono** nel tempo (come l'estrazione dell'oro), aumentando la dipendenza dalle commissioni
- **Attacco del 51%:** Se un nodo controlla la maggioranza della potenza di hash, può **alterare la catena**. Esistono varianti come il Selfish Mining, efficace già con il 25% della potenza
- **Spreco Energetico:** Bitcoin consuma circa 72.99 TWh/anno (0,34% del consumo mondiale)
- **Centralizzazione (Mining Pools):** Per aumentare l'efficienza, i minatori si **aggregano** in "pool". Attualmente, 8 pool controllano oltre il 75% dei blocchi Bitcoin
- **Eclipse Attack:** Mira a **isolare** specifici nodi della rete; la presenza di pool rende la rete più vulnerabile a questo attacco

## Proof of Stake (PoS)

Introdotto con Peercoin per risolvere i problemi energetici del PoW. Invece di usare potenza di calcolo, **elegge il validatore del blocco** in modo pseudo-casuale basandosi sullo "stake" (interesse economico) posseduto

### Metodi di selezione

- **Wealth** (Ricchezza): **Probabilità proporzionale alla quantità di valuta posseduta**
- **Coin Age** (Età della moneta): **Probabilità proporzionale al prodotto tra quantità di valuta e tempo di possesso** ( $c \cdot a$ ). Dopo aver vinto, l'età viene resettata

### Vantaggi e Svantaggi

- Computazionalmente **economico** ed **efficiente**. Penalizza chi attacca la rete (svalutando il proprio capitale)
- **Meno protezione** contro il **double spending** rispetto al PoW. I nodi possono lavorare su più catene contemporaneamente senza costi ("Nothing at Stake"), ostacolando il consenso

## Delegated Proof of Stake (DPoS)

È un approccio basato sulla **democrazia rappresentativa**:

- Gli **utenti votano** usando i propri token per eleggere **Witnesses** (Testimoni) e **Delegates** (Delegati)
- **Witnesses: Creano e validano i blocchi** a turno. I migliori vengono **ricompensati**
- **Delegates: Gestiscono i parametri di governance** (tasse, dimensioni blocchi, intervalli)
- Il **voto è continuo**: chi non performa bene perde il ruolo

Come caratteristiche:

- **Molto scalabile**, alto **throughput** di transazioni, **efficiente**
- **Rischio centralizzazione** (numero limitato di witness) e apatia dei votanti

## Practical Byzantine Fault Tolerance (PBFT)

Utilizzato nelle **blockchain permissioned** (es. Hyperledger Fabric), è un protocollo di **replicazione della macchina a stati**:

- Il sistema è composto da un **insieme R di repliche**
- Per tollerare  $f$  nodi guasti (o malevoli), sono **necessarie  $R = 3f + 1$  repliche**
- Si opera in "**viste**" (views): in ogni vista c'è un **Primary** (leader) e dei **Backups**

Volendo semplificare il flusso:

- 1) Il **Client invia una richiesta al Primary**

- 2) Il Primary invia in **multicast** la **richiesta** ai **Backups**
- 3) Le **repliche eseguono la richiesta e inviano la risposta** al **Client**
- 4) Il **Client attende  $f + 1$  risposte identiche** da repliche diverse per **confermare il risultato**

## Smart Contract

Uno Smart Contract (SC) è definito come:

*"Un protocollo informatico che esegue automaticamente i termini di un contratto"* (N. Szabo, 1994).

Gli obiettivi principali degli smart contract sono:

- **Traduzione di condizioni contrattuali** comuni (pagamenti, NDA, garanzie) in codice
- **Incorporazione** di queste **condizioni** in sistemi che possano applicarle **automaticamente** (implementate via software o hardware)
- **Ridurre eccezioni ed errori** (sia intenzionali che accidentali)
- **MInimizzare la necessità di intermediari**

Ovviamente il loro utilizzo porta con sé notevoli benefici economici, come:

- **Riduzione delle frodi**
- **Minori costi di arbitrato ed enforcement**
- **Riduzione dei costi di transazione**

## Smart Contract nei registri distribuiti (BDLT)

In una blockchain, uno **Smart Contract** è uno **script** memorizzato sul **ledger**:

- È **come** una **stored procedure** in un database relazionale
- Ha un **indirizzo unico** sulla **blockchain**
- Si **attiva inviando una transazione** al suo indirizzo
- Viene **eseguito automaticamente** e in modo identico su tutti i nodi, grazie alla macchina virtuale distribuita (su Ethereum c'è la Ethereum Virtual Machine)

Volendo fare un esempio pratico, abbiamo uno scenario:

- Utenti: Alice e Bob
- Asset: X e Y

Bob crea un contratto con alcune funzioni:

- **Deposit**: Bob deposita unità di X
- **Trade**: scambia 1X per 5Y
- **Withdraw**: Bob ritira tutti gli asset dal contratto

Un flusso verosimile sarebbe:

- Bob invia una transazione a **deposit** con 3X → registrato sulla blockchain
- Alice invia 10Y a **trade** → riceve 2X → registrato sulla blockchain

- Bob chiama `withdraw` → il contratto verifica la firma e restituisce  $1 X + 10 Y$  a Bob

## Proprietà degli Smart Contract

Tra le principali proprietà degli SC abbiamo:

- **Stato proprio:** ogni SC mantiene dati interni
- **Custodia di asset:** può gestire fondi sulla blockchain
- **Business logic** in codice: descrive regole e condizioni
- **Determinismo:** stesso input → stesso output (fondamentale per il consenso)
- **Trasparenza:** codice visibile a tutti i nodi
- **Tracciabilità:** ogni interazione è firmata e verificabile crittograficamente

## Smart Contract e DAOs

Gli SC sono **attori autonomi** e **prevedibili**, da questo nasce il concetto di **Decentralized Autonomous Organization (DAO)**. Queste sono organizzazioni gestite da SC, modificabili solo seguendo regole codificate

Volendo fare un esempio:

- SC1 chiama SC2 per svolgere funzioni
- SC1 mantiene lista di utenti che possono votare
- Maggioranza può cambiare l'indirizzo di SC2

## Linguaggi per Smart Contract

Principali approcci

- **Procedurali:** indicano cosa fare e come farlo.
- **Linguaggi dedicati:**
  - **Solidity** (Ethereum)
- **Linguaggi esistenti:**
  - **Interpretati:** JavaScript, Python
  - **Compilati:** C++, Rust, Go
  - **Ibridi:** Java, C#
- **Approcci alternativi:**
  - **SQL** (Es.: Aergo)
  - **Macchine a stati** (Automata, Petri nets)
  - **Programmazione logica** (Defeasible Logic)

## Sfide nella scrittura di SC complessi

- **Procedurale:** difficile mantenere coerenza con clausole legali.
- **Macchine a stati:** più intuitivo, ma cresce la complessità con il numero di stati.
- **Logica:** utile per contratti normativi, ma richiede motori di inferenza.

## Validità legale

Ad oggi gli SC non hanno ancora pieno riconoscimento a livello legale, questo perché:

- È **difficile adattare** i sistemi giuridici attuali, soprattutto se si pensa al fatto che le leggi variano da paese a paese, le blockchain invece su scala globale o regionale
- **Lasciano questioni aperte**, per esempio come si garantisce che il codice rifletta correttamente le clausole legali? Come si gestiscono errori o bug nel contratto? Chi è responsabile in caso di malfunzionamento?

Non è semplice dire se saranno mai riconosciute del tutto a livello legale ma servirebbe un dialogo tra esperti di tecnologia, diritto ed economia per:

- **Definire regole** chiare, eque e applicabili
- **Garantire** che l'automazione non comprometta **diritti fondamentali**
- **Creare standard internazionali** per interoperabilità e sicurezza

## Ethereum

Ethereum è la più grande piattaforma blockchain completamente programmabile al mondo. È **Open-source** e ha la sua criptovaluta nativa, **Ether**. La sua applicazione spazia in diversi campi, essendo una piattaforma **general-purpose**, supporta wallet di criptovalute, applicazioni finanziarie e mercati decentralizzate. Queste ultime, anche chiamate **dApps**:

- Sono **affidabili**: una volta distribuite, eseguono il codice come programmato
- **Possono gestire** asset digitali tokenizzati
- Sono **decentralizzate**, senza controllo da parte di un singolo ente

Ogni nodo esegue l'**Ethereum Virtual Machine (EVM)**, che garantisce che tutti i nodi eseguano lo stesso codice per validare transazioni e modifiche di stato.

La storia di Ethereum nasce nel 2013, da un di Vitalik Buterin, che successivamente, nel dicembre 2013 crea Ethereum.org, una organizzazione no-profit. Il 30 luglio 2015 viene rilasciata la prima versione stabile, in seguito gli aggiornamenti saranno particolarmente incentrati sulle questioni di sicurezza e sull'aggiunta di feature.

Ethereum nasce con il principale obiettivo di **superare i limiti di Bitcoin**, tra questi:

- **Assenza di Turing-completezza**: i linguaggi di scripting di Bitcoin sono limitati
- **Value-blindness**: nel modello UTXO non è possibile controllare in modo fine l'importo da trasferire
- **Assenza di stato**: UTXO è "tutto o niente", impedendo contratti multi-step
- **Blockchain-blindness**: le transazioni UTXO non possono usare dati della blockchain come fonte di casualità (Es.: nonce, hash precedente)

## Account

In Ethereum abbiamo **due tipologie di account**:

- **Externally owned account (EOA)**:
  - Account utente controllato da **chiave privata**
  - Indirizzo di 20 byte (**hash** della **chiave pubblica**)
  - Può inviare transazioni firmate
- **Contract Account**:
  - Contiene codice di uno **smart contract**
  - Ha **bilancio** in Ether e **memoria interna**
  - Può leggere e scrivere **dati**, inviare **messaggi** e creare altri **contratti**

Ognuno di questi account ha:

- **Bilancio in Ether**
- **Codice** (se presente)
- **Storage** (key-value)
- **Nonce**: contatore per evitare replay e garantire ordine

## Transazioni

Una **transazione** è un **pacchetto di dati firmato** che contiene:

- **Mittente** (derivato dalla firma)
- **Destinatario (to)**
- **Valore** (Ether da trasferire)
- **Data** (parametri per SC)
- **Nonce** (anti-replay)
- **GasLimit (Startgas)**: massimo numero di operazioni consentite
- **GasPrice**: costo per unità di gas

Tipi di transazioni:

- **Regolare**: da un account a un altro
- **Esecuzione di SC**: interazione con un contratto
- **Deploy di SC**: il codice del contratto è nel campo data

## Gas

Il gas misura il **costo computazionale** delle **operazioni** su Ethereum:

- **Ogni istruzione ha un costo in gas**
- **Fee = gasPrice × gasUsato**
- Serve per:
  - **Pagare risorse** (CPU, storage, banda)
  - **Evitare sprechi** (Es.: loop infiniti)

Se il gas si esaurisce:

- **Eccezione Out of Gas**, transazione annullata
- Il **mittente paga comunque** il gas consumato

## Messaggi

I **contratti** possono inviare **messaggi** ad altri **contratti** (il loro opcode è **CALL**). Questi messaggi sono simili alle **transazioni** ma non sono creati da EOAs e non vengono registrati sulla blockchain. La loro struttura è:

- **Destinatario (to)**
- **Valore (Ether)**
- **Dati**
- **Gas**

## Transizione di stato

Ogni blockchain è, in sostanza, un **sistema distribuito di transazione di stato**:

- Lo **stato** rappresenta l'**insieme delle informazioni correnti** (Es.: bilanci, contratti)
- In Ethereum, lo **stato include** tutti gli **account** e i loro **dati** (Es.: saldo, codice, storage)

In particolare, Ethereum utilizza la funzione:

`APPLY(S, TX) → S'`

Dove:

- S = stato attuale
- TX = transazione
- S' = nuovo stato

Volendo strutturare una lista di passaggi:

- 1) **Validazione della transazione:** se non è ben formata c'è un errore
- 2) **Calcolo della fee:** viene **sottratta** dal **saldo del mittente**, se il saldo è insufficiente c'è un errore
- 3) **Gestione del gas:** si inizializza la variabile `gas = startgas` e si detrae gas per ogni byte della transazione
- 4) **Esecuzione della transazione:** avviene il **trasferimento** di **valore** al **destinatario**, se questo è un contratto ne viene eseguito il codice, se fallisce c'è il rollback delle modifiche, tranne il pagamento della fee
- 5) **Rimborsi:**
  - a. Se il gas non viene usato viene restituito al mittente
  - b. Se il gas è consumato viene pagata la fee al miner

## Struttura di un blocco Ethereum

Ogni blocco contiene:

Un **header** con:

- **Hash del blocco padre**
- **Hash degli “uncle blocks”** (blocchi minati quasi contemporaneamente)
- **Radice dell'albero Merkle-Patricia dello stato**
- **Radice dell'albero Merkle-Patricia delle transazioni**

Lista delle transazioni:

- **Codificata** con **RLP** (Recursive Length Prefix)
- **Organizzata** in un **albero** Merkle-Patricia (variante del **radix tree**)

## Sistema di Token

Un **token system** è, in pratica, un **database** che **gestisce operazioni** del tipo:

*Sottrai X unità da A e aggiungi X unità a B, se A ha almeno X unità e approva la transazione*

Ha diverse applicazioni, dalle **criptovalute** agli **asset digitali**, passando per gli **NFT**. Questi sistemi si implementano tramite **smart contract**, seguendo standard che garantiscono interoperabilità

## Standard principali

### ERC-20 (Fungible Tokens)

**Token fungibili** (tutti uguali).

Funzioni principali:

- `name()`, `symbol()`, `decimals()`
- `totalSupply()`
- `balanceOf(address)`
- `transfer(address, value)`

## ERC-721 (Non-Fungible Tokens - NFT)

**Token unici** (Es.: CryptoKitties).

Funzioni principali:

- `balanceOf(address)`
- `ownerOf(tokenId)`
- `safeTransferFrom(...)`
- `approve(...)`
- `getApproved(tokenId)`
- `setApprovalForAll(...)`

## Aggiornamenti recenti e direzioni future

- **The merge** (15 settembre 2022): Passaggio da Proof of Work a Proof of Stake → meno consumo energetico, più scalabilità
- **Sharding**: suddivisione della rete in shard per aumentare la capacità di elaborazione, difficoltà implementative e di gestione

## Ethereum Virtual Machine (EVM)

Ethereum può essere vista come:

- Una **blockchain con linguaggio di programmazione integrato**
- Una **macchina virtuale globale che esegue codice in modo distribuito e basato sul consenso**

La EVM è il componente che si occupa di gestire il deploy e l'esecuzione degli smart contract, oltre che definire come lo stato cambia in seguito alle transazioni.

Bisogna immaginare la EVM come un **computer decentralizzato** con milioni di account che conservano dati, eseguono codice e comunicano tra di loro. È quasi Turing-completa, la computazione è limitata dal gas per evitare loop infiniti e sprechi

### Account e transizioni di stato

Come abbiamo detto ci sono due tipologie di account:

- **Externally Owned Account (EOA):** può inviare Ether e messaggi
- **Contract Account:** contiene lo smart contract e il suo storage

Di default, la rete è "inattiva", questo vuol dire che non accade nulla se non si fa nulla.

Una transazione da un EOA ha due flussi, in base al destinatario:

- Se è un altro **EOA** c'è un **trasferimento di Ether**
- Se è destinata a un **contratto**, questo viene **attivato** e può **leggere il messaggio, scrivere nel proprio storage e inviare messaggi** ad altri contratti

Una volta che la transazione è stata effettuata, la rete torna "in pausa" fino alla successiva

### Contratti: ruoli principali

Gli smart contract possono:

- **Gestire dati** utili ad **altri contratti** o al mondo esterno
- **Agire come forwarder**, ovvero **inoltrare messaggi** solo se certe condizioni sono rispettate
- **Gestire relazioni** tra **utenti**
- **Fornire funzioni** ad altri **contratti**

### Modello di esecuzione EVM

La EVM è una macchina virtuale stack-based con 140 opcode. Ogni opcode ha un costo in gas e il programma ha tre aree di memoria che può utilizzare:

- **Stack:** max 1024 elementi (256 bit ciascuno)
- **Memory:** volatile, usata per variabili locali
- **Storage:** persistente, key-value (256 bit)

Scrivere in storage è molto costoso rispetto alla memoria.

## Categorie di istruzioni

- **Stack:** POP, PUSH, DUP, SWAP
- **Memory:** MLOAD, MSTORE
- **Storage:** SLOAD, SSTORE
- **Aritmetiche:** ADD, SUB, AND, OR
- **Controllo flusso:** JUMP, JUMPI
- **Ambiente:** CALLER, CALLVALUE
- **Halting:** STOP, RETURN, SELFDESTRUCT

0x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	STOP	ADD	MUL	SUB	DIV	SDIV	MOD	SMOD	ADDMOD	MULMOD	EXP	SIGEXTEND				
1	LT	GT	SLT	SGT	EQ	ISZERO	AND	OR	XOR	NOT	BYTE	SHL	SHR	SAR		
2	SHA3															
3	ADDRESS	BALANCE	ORIGIN	CALLER	CALLVALUE	CALLDATACOPY	CALLDATASIZE	CODESIZE	CODECOPY	GASPRICE	EXTCODESIZE	EXTCODECOPY	RETURNDATASIZE	RETURNDATACOPY	EXTCODEHASH	
4	BLOOMHASH	CONBESS	TIMESTAMP	NUMBER	DIFFICULTY	GAUMIT										
5	POP	MLOAD	MSTORE	MSTORE8	SLOAD	SSTORE	JUMP	JUMPI	PC	MSIZE	GAS	JUMPDEST				
6	PUSH1	PUSH2	PUSH3	PUSH4	PUSH5	PUSH6	PUSH7	PUSH8	PUSH9	PUSH10	PUSH11	PUSH12	PUSH13	PUSH14	PUSH15	PUSH16
7	PUSH17	PUSH18	PUSH19	PUSH20	PUSH21	PUSH22	PUSH23	PUSH24	PUSH25	PUSH26	PUSH27	PUSH28	PUSH29	PUSH30	PUSH31	PUSH32
8	DUP1	DUP2	DUP3	DUP4	DUP5	DUP6	DUP7	DUP8	DUP9	DUP10	DUP11	DUP12	DUP13	DUP14	DUP15	DUP16
9	SWAP1	SWAP2	SWAP3	SWAP4	SWAPS	SWAP5	SWAPP	SWAP8	SWAP9	SWAP10	SWAP11	SWAP12	SWAP13	SWAP14	SWAP15	SWAP16
a	LOG0	LOG1	LOG2	LOG3	LOG4											
b																
c																
d																
e																
f	CREATE	CALL	CALLCODE	RETURN	DELEGATECALL	CREATE2			STATICCALL			REVERT	INVALID	SELFDESTRUCT		

## Modello di sicurezza EVM

La **EVM** è particolarmente **orientata alla sicurezza**, pertanto:

- **Ogni passo** computazionale deve essere **pagato in gas**
- **Sandboxing:** un contratto può modificare solo il proprio stato e inviare messaggi
- **Interazioni limitate:** solo tramite byte array, senza accesso diretto allo stato altrui
- **Determinismo:** stesso input → stesso output, per garantire consenso

## Funzionamento del gas

Ogni operazione eseguita nella EVM viene replicata su tutti i nodi della rete. Questo è essenziale per il meccanismo del consenso, quindi tutti devono arrivare allo stesso risultato.

- Per questo motivo **qualsiasi contratto** può **chiamare** un altro, senza costi di comunicazione remota (RPC) ma anche
- Ma **ogni passo computazionale è costoso**

Una regola pratica dice: *non fare sulla EVM ciò che non potresti fare su un vecchio PDA del 1999.*

**Il meccanismo del gas serve a prevenire attacchi DoS (quindi esecuzioni infinite) e limitare l'utilizzo eccessivo di risorse.**

Ogni transazione include:

- **GASPRICE:** costo per unità di gas (in wei), può essere aggiunta anche una mancia per aumentare la priorità della transazione
- **STARTGAS:** quantità massima di gas assegnata alla transazione

Quando la transazione parte:

- Si sottrae subito `STARTGAS × GASPRICE` dal saldo del mittente
- Se il prezzo del gas è troppo basso, i miner possono rifiutare la transazione

All'inizio della transazione si calcola:

```
gas disponibile = STARTGAS - 21000 - (68 × TXDATALEN)
```

Dove:

- 21000 è il costo base di una transazione (anche inviare eth)
- TXDATALEN = numero di byte nei dati della transazione

Attualmente, la formula che viene usata è un'altra:

```
gas disponibile = STARTGAS - [21.000 + (16×byte non-zero) + (4×byte zero)]
```

Durante l'esecuzione ogni istruzione consuma gas. Es.: lo SHA3: 20 gas + 6 gas per parola

Group	Gas cost	Examples
G <sub>zero</sub>	0	STOP, RETURN
G <sub>base</sub>	2	ADDRESS, POP, PC
G <sub>verylow</sub>	3	ADD, SUB, LT, SLT, AND, MLOAD, MSTORE
G <sub>low</sub>	5	MUL, DIV, SDIV, MOD
G <sub>mid</sub>	8	ADDMOD, MULMOD, JUMP
G <sub>high</sub>	10	JUMPI

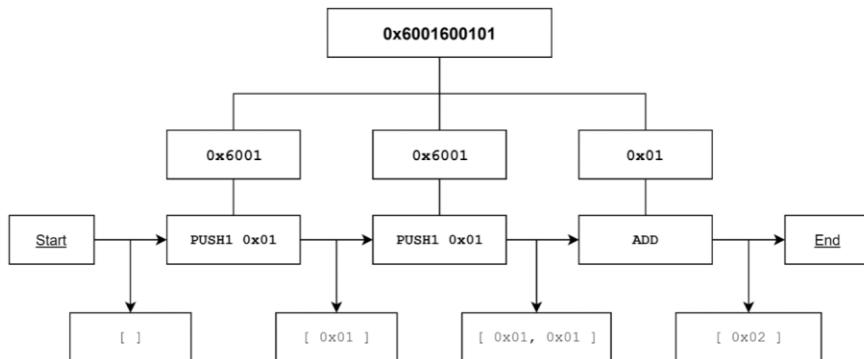
C'è anche da sottolineare che la scrittura in memoria/storage è molto costosa.

Ricordiamo che:

- Se la transazione termina con `gas >= 0`, il gas non usato viene rimborsato al mittente e quello consumato viene pagato al miner
- Se il gas finisce prima c'è la rollback di tutte le modifiche (ma la fee non viene rimborsata)

## Esempio pratico

Somma di due valori, Bytecode: 6001600101



Disassemblato:

```
PUSH1 0x01
```

```
PUSH1 0x01
```

```
ADD
```

```
STOP
```

Con:

- PUSH1: inserisce 1 sullo stack
- ADD: somma i due valori
- Risultato: 0x02 sullo stack

# Solidity

Per **programmare Smart Contract** sulla **Ethereum Virtual Machine (EVM)** non si scrive direttamente in **bytecode** (perché sarebbe troppo complesso) ma si usano **linguaggi ad alto livello** che vengono **compilati in bytecode EVM**.

Tra questi abbiamo:

- **Serpent**: simile a Python, oggi deprecato
- **Vyper**: evoluzione di Serpent, sintassi Python-like e orientato alla sicurezza
- **Solidity**: linguaggio statisticamente tipizzato, orientato agli oggetti, creato per Ethereum

Ad oggi **Solidity** è quello più maturo e diffuso, oltre che adottato anche da altre blockchain.

Ricordiamo che uno **smart contract** è una **collezione** di dati (stato) e funzioni, associato a un **indirizzo** sulla blockchain.

Per lavorare con Solidity ci sono diversi approcci:

- Remix IDE (browser): Permette di scrivere, compilare e testare contratti senza installare nulla (<https://remix.ethereum.org>)
- Compilatore solc: (<https://github.com/ethereum/solidity/releases>)
- Docker image solc: <https://hub.docker.com/r/ethereum/solc>
- solc-js (Node.js): Versione JavaScript del compilatore, utile per integrazione in progetti web

N.B.: **Solidity evolve rapidamente, è consigliabile usare sempre l'ultima versione stabile**

## Hello world!

```
pragma solidity ^0.6.0;

contract HelloWorld {
    function helloWorld() external pure returns (string memory) {
        return "Hello, World!";
    }
}
```

Dove:

- `pragma solidity ^0.6.0;` → indica la **versione del compilatore**.
- `contract HelloWorld { ... }` → **definisce il contratto**.
- La funzione `helloWorld()` **restituisce** una **stringa**.

## Sintassi

### Pragmas

**Direttive del compilatore**, di solito si indicano le versioni con cui si vuole la compatibilità:

```
Es.: pragma solidity >=0.4.0 <0.7.0; → accetta versioni tra 0.4.0 e 0.6.x
```

## Importare file

Solidity **supporta import** per **modularizzare il codice**, simile a JavaScript:

Es.:

```
import "filename";
import "./path/to/file.sol";
```

## Commenti e annotazioni

- **Commenti standard:** // e /\* ... \*/
- **NatSpec: documentazione formale** con /// o /\*\* ... \*/

Es.: /\*\* @title Shape calculator \*/

## Tipi di dato

- **Boolean:** bool (true/false)
- **Interi:**
  - intN / uintN (N =8,16,...256)
  - int / uint = alias per 256 bit
- **Bytes:**
  - bytes1 ... bytes32 (sequenze di byte)
  - byte = alias per bytes1 (bool)

## Indirizzi

Un **Address** è un **tipo di dato** che **rappresenta** un **indirizzo Ethereum** (Hex) e può essere usato per:

- **Leggere il saldo:** address.balance (in wei)
- **Effettuare chiamate a funzioni:** call, delegatecall, staticcall

Il tipo **Address Payable** è **simile** a quello di **Address** ma ha dei **membri in più**, ovvero:

- **Transfer(amount): trasferisce Ether, fa revert se fallisce** (sicuro)
- **Send(amount): trasferisce Ether, non fa revert se fallisce** (ritorna false, non sicuro)

Gli **indirizzi** devono **superare il checksum** test per evitare errori di digitazione

## Tipi di riferimento

Solidity **definisce tre tipi di riferimento:**

- **Array**
- **Mapping**
- **Struct**

A differenza dei tipi di valore, richiedono la specifica della data location:

- **memory:** dati **temporanei** (**durano solo durante la chiamata**)

- **storage**: dati **persistenti** (stato del **contratto**)
- **calldata**: **parametri di funzioni esterne** (solo lettura)

NB.: Copiare dati tra location diverse è costoso in gas

### *Array*

In Solidity gli **array** sono **zero-based** e possono essere di due tipologie:

- **Dimensione fissa** (compile-time)
- **Dinamici**

Tra i metodi abbiamo:

- `Length()`: **numero di elementi**.
- `push()`: **aggiunge elemento**.
- `pop()`: **rimuove ultimo elemento**.

Le **stringhe** vengono **trattate come** degli **array** di **byte UTF-8**, non supporta nativamente l'accesso diretto a lunghezza o indice, per questo bisogna utilizzare delle librerie

### *Mapping*

Un **mapping** è una **struttura chiave-valore** (hash table) e viene **conservata solo in storage**. La sintassi è del tipo: `mapping(KeyType => ValueType) variableName;`

Es.:

```
mapping(address => uint) public balances;

function update(uint newBalance) public {
    balances[msg.sender] = newBalance;
}
```

### *Struct ed Enum*

Le **struct** sono **come** quelle di **C/C++** e servono a **raggruppare dati**, gli **enum** sono dei **tipi definiti dall'utente** con **valori interi sequenziali**

### *Operatori principali*

In base ai tipi di dato possiamo avere diversi operatori:

- **Booleani:** !, &&, ||, ==, !=
- **Interi:**
  - **Comparazione:** <, >, <=, >=
  - **Bitwise:** &, |, ^, ~
  - **Aritmetici:** +, -, \*, /, %, \*\*
- **Array di byte:** operazioni simili agli interi
- **Address:** confronti (==, !=, <, >)

## Strutture di controllo

Le **strutture di controllo** sono le **stesse** che ci sono anche su **altri linguaggi**, come C e JS, ovvero: if, else, while, do...while, for, break, continue, return.

NB.: Non esiste conversione implicita tra boolean e altri tipi

## Funzioni

Le funzioni possono avere due **modificatori**:

- **view:** non modifica lo stato
- **pure:** non legge né modifica lo stato

Sono supportati overloading e overriding

Le **chiamate**, invece, possono essere:

- **Interne:** via EVM jump (efficiente)
- **Esterne** (meno efficiente)

I **parametri** possono essere **passati** in due modi:

- Per **posizione**
- Per **nome** (in ordine arbitrario)

## Gestione errori

In Solidity le **eccezioni** vengono **risolte facendo** una **revert** dello **state** e possono essere **raccolte** con i **try/catch**. Per dei **test interni** viene usato l'**assert**, per controllare le **condizioni** a **runtime** viene usato **require**

## Struttura dei contratti

I **contratti** in Solidity sono **simili** alle **classi** nei linguaggi orientati agli oggetti. Un contratto può contenere:

- **Variabili di stato:** dati persistenti memorizzati sulla blockchain
- **Funzioni:** logica del contratto, possono modificare lo stato
- **Modificatori di funzione:** controllano l'esecuzione (Es.: restrizioni di accesso)
- **Tipi strutturati:** struct ed enum

- **Eventi:** per notificare cambiamenti agli utenti
- **Ereditarietà:** supporta ereditarietà multipla e un contratto è astratto se ha almeno una funzione non implementata o se è marcato con abstract

## Variabili di stato

Le **variabili di stato** sono **memorizzate** nello **storage** e **persistono** tra le **chiamate**.

Es.:

```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleStorage {
    uint storedData; // Variabile di stato
}
```

## Creare contratti

I **contratti** possono essere **creati**:

- **Dall'esterno:** tramite **transazione Ethereum**
- **Dall'interno:** con la **keyword new**

Quando viene creato un contratto il suo costruttore viene eseguito una sola volta (non è possibile fare overloading, massimo un costruttore)

Dopo il deploy il codice del contratto (quindi funzioni pubbliche ed esterne) viene memorizzato sulla blockchain. Di questo, il **codice del costruttore** e delle **funzioni interne chiamate solo da esso, non vengono inclusi**.

NB.: Un contratto che ne crea un altro deve conoscere il suo codice, non sono ammesse dipendenze cicliche

## Visibilità

Ci sono **quattro livelli di visibilità**:

- **external:** chiamabile solo da fuori (via transazioni o altri contratti)
- **public:** accessibile internamente ed esternamente
- **internal:** solo all'interno del **contratto** e dei **derivati**
- **private:** solo nel **contratto stesso**

NB.: Tutto il codice è visibile sulla blockchain, la visibilità private ne impedisce solo l'accesso da altri contratti, non la lettura esterna

## Getter e setter

Il **compilatore genera automaticamente getter per variabili public**

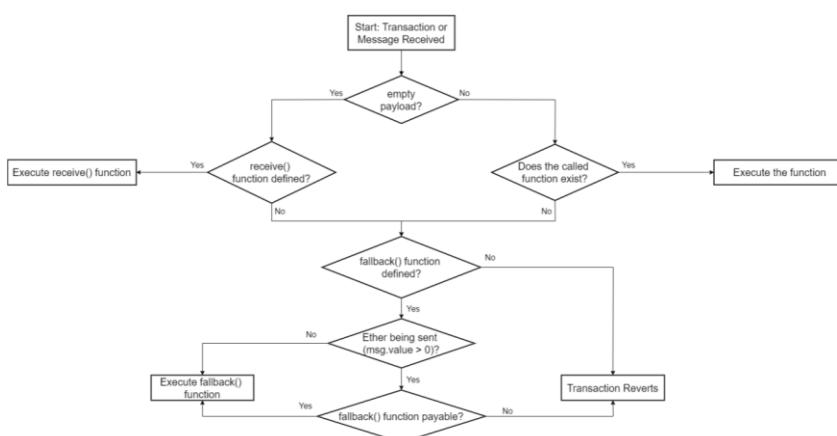
## Funzioni di Receive e Fallback

In Solidity, i **contratti possono definire due funzioni speciali** per gestire **chiamate generiche** e trasferimenti di Ether:

- **Receive function:**
  - La sintassi è del tipo `receive() external payable { ... }`
  - Viene **chiamata quando la transazione non contiene dati** (calldata vuoto) o **quando viene inviato Ether**
  - È **sempre payable**
- **Fallback function**
  - La sintassi è del tipo `fallback() external [payable] { ... }`
  - Viene **chiamata quando la funzione richiesta non esiste, quando calldata è vuoto e receive() non è definita**
  - Per **ricevere Ether** deve essere **payable**

Entrambe queste funzioni possono essere presenti per massimo una volta per contratto.

NB.: Se usata per ricevere Ether, la **fallback function** ha solo **2300 gas** disponibili, **sufficiente** per **operazioni minime** (come il logging), **fallirebbero** tutte le **operazioni costose** come scrittura in storage, creazione di contratti o chiamate esterne



Scenario	C'è <code>receive()</code> ?	C'è <code>fallback()</code> ?	Cosa viene eseguito?
Invio solo ETH (no dati)	Sì	-	<code>receive()</code>
Invio solo ETH (no dati)	No	Sì (payable)	<code>fallback()</code>
Invio solo ETH (no dati)	No	No	<b>Revert (errore)</b>
Chiamo funzione inesistente	-	Sì	<code>fallback()</code>
Chiamo funzione inesistente	-	No	<b>Revert</b>

## Invio di Ether

Tre modi principali:x

- `transfer(amount)`:
  - **Invia Ether e 2300 gas**
  - **Sicuro:** revert in caso di errore

- Attiva `receive()` o `fallback()`
- Potrebbero cambiare i costi degli opcode, 2300 gas potrebbero non essere sufficienti
- `send(amount)`:
  - **Simile** a `transfer()`, ma **non reverte** in caso di **errore** (ritorna false)
  - **Meno sicuro**
- `call{value: amount, gas: gasAmount}("")`:
  - **Metodo consigliato**
  - Permette di **specificare il gas**
  - Può **chiamare altre funzioni** (con ABI encoding)
  - **Rischio di re-entrancy attack** → usare pattern di sicurezza (Es.: Checks-Effects-Interactions)

Caratteristica	<code>transfer()</code>	<code>send()</code>	<code>call()</code>
<b>Limite di Gas</b>	Fisso (2300)	Fisso (2300)	Tutto il gas disponibile (o custom)
<b>In caso di errore</b>	Fa il <b>Revert</b> automatico	Restituisce <b>false</b>	Restituisce <b>false</b>
<b>Sicurezza</b>	Alta (previene reentrancy)	Alta (previene reentrancy)	Bassa (richiede controlli manuali)
<b>Consigliato?</b>	No (obsoleto)	No (obsoleto)	<b>Si (Best Practice attuale)</b>

## Eventi

Gli **eventi** sono un **meccanismo** per registrare informazioni nei **log** della **blockchain**, utili per:

- **Notificare applicazioni esterne**
- **Tracciare operazioni senza memorizzare dati** nello storage

Tra le loro caratteristiche abbiamo:

- **Accessibilità tramite RPC**, non dai contratti
- **Permanenti nella blockchain**
- Sintassi del tipo:

```
event Deposit(address indexed _from, bytes32 indexed _id, uint _value);

function deposit(bytes32 _id) public payable {
    emit Deposit(msg.sender, _id, msg.value);
}
```

Con `indexed` che permette la ricerca rapida nei log

## Interfacce

Sono **simili** a **contratti astratti**, ma:

- **Nessuna implementazione**
- **Nessuna variabile di stato**

- **Nessun costruttore**
- **Tutte le funzioni** devono essere **external**
- La sintassi è del tipo:

```
interface MyInterface {
    function doSomething() external;
}
```

## Librerie

Le librerie sono **moduli riutilizzabili** che **necessitano il deploy solo una volta**. Queste **eseguono codice nel contesto del contratto chiamante** (utilizzando this per riferirsi a quest'ultimo). Possono **accedere allo storage solo se esplicitamente passato** e devono essere **stateless**

## Variabili e funzioni speciali

- `block.number`: numero del blocco corrente
- `block.timestamp`: timestamp UNIX
- `msg.sender`: indirizzo del chiamante
- `msg.value`: Ether inviato
- `tx.origin`: indirizzo originario della transazione (attenzione: vulnerabile a phishing)
- `gasleft()`: gas residuo

## Deploy di un contratto

The diagram illustrates the deployment process. On the left, a black box contains Solidity code for a contract named `ExampleContract` with a single variable `number` set to 1. An arrow points from this box to a red-bordered box on the right, which displays the generated Ethereum bytecode in hexadecimal format.

```
pragma solidity 0.5.3;

contract ExampleContract {
    uint256 number = 1;
}
```

```
0x608060405260016000553480156014
57600080fd5b50603580602260003960
00f3fe6080604052600080ffdfe16562
7a7a723058204e048d6cab20eb0d9f95
671510277b55a61a582250e04db7f658
7a1bebc134d20029
```

Avviene tramite una **transazione senza indirizzo to**, il bytecode è diviso in:

- **Constructor** (eseguito una volta)
- **Runtime codice** (funzioni pubbliche)
- **Metadata**

## ABI (Application Binary Interface)

L'**Application Binary Interface (ABI)** è lo **standard fondamentale** per l'**interazione** con gli **smart contract** nel mondo Ethereum. Poiché la Ethereum Virtual Machine (EVM) non comprende il codice sorgente (Solidity), ma solo il **bytecode** (una sequenza di numeri esadecimali), l'ABI funge da "ponte" o manuale di istruzioni per codificare i dati in modo che il contratto possa interpretarli correttamente

L'ABI definisce come devono essere strutturate le chiamate alle funzioni e come devono essere formattati i dati restituiti. È essenziale in due scenari:

- **Interazione Esterna:** Quando un'applicazione (es. un sito web tramite `ethers.js`) vuole "parlare" con un contratto.
- **Interazione Interna:** Quando un contratto chiama una funzione di un altro contratto (Inter-contract interaction).

Es.:

```
address(nameReg).call{gas: 1000000, value: 1 ether}(
    abi.encodeWithSignature("register(string)", "MyName")
);
```

Ogni volta che invochiamo una funzione, il "messaggio" (`calldata`) inviato alla EVM è composto da due parti principali:

- La EVM non sa cosa sia `register(string)`. Per **identificare la funzione**, calcola l'hash **Keccak-256** della "firma" della funzione (**nome della funzione e tipi di argomento**, senza spazi)
  - Per la funzione `HelloWorld()`, si calcola l'**hash** e si prendono i primi **4 byte**
  - Se l'**hash** è `0x7fffb7bd...`, la transazione deve iniziare esattamente con `0x7fffb7bd`. Questi **4 byte** dicono al contratto: "Esegui questa specifica funzione"
- Dopo il selettore, vengono inseriti i **parametri passati** alla funzione
  - Ogni argomento viene **convertito** e "impacchettato" in blocchi da **32 byte** (parole della EVM). Se un dato è più piccolo (come un `uint8`), viene aggiunto del "padding" (zeri extra) per raggiungere i 32 byte
  - Per dati di **lunghezza variabile** (come `string` o `bytes`), l'ABI utilizza un sistema di **puntatori** per indicare dove inizia il dato effettivo all'interno del messaggio

Sebbene a livello di protocollo l'ABI sia un concetto logico di codifica binaria, per noi sviluppatori si presenta solitamente come un **file JSON**. Questo file elenca tutte le funzioni pubbliche, gli eventi e i costruttori del contratto, specificando nomi, tipi di input e tipi di output. Senza il file JSON dell'ABI, un'applicazione frontend non saprebbe quali tasti o moduli mostrare all'utente, né come trasformare un "clic" su un pulsante in una sequenza di byte comprensibile dalla blockchain

# Argomentazione computazionale

## Cos'è un'argomentazione?

L'argomentazione è presente in quasi tutti gli ambiti della vita, dal diritto (in cui si interpretano delle leggi, spesso contraddittorie) alla vita quotidiana (con decisioni semplici)

Per **argomentazione** si intende lo **studio dei processi e attività che riguardano la produzione e lo scambio di argomenti**. Gli obiettivi principali:

- **Identificare, analizzare e valutare argomenti**
- **Formalizzare il ragionamento** in modo intuitivo
- **Fornire procedure per prendere decisioni e spiegarle**

In AI:

- **Dare ragioni a supporto di affermazioni dubbie**
- **Difendere queste affermazioni da attacchi** (contro-argomenti)

## Cos'è un argomento

Secondo logica, linguistica computazionale e filosofia:

- È un **insieme di proposizioni** da cui si può **trarre una conclusione**
- È **attivato** da **un'azione linguistica** (monologo o dialogo)
- Ha una **intenzione comunicativa**
- Può essere **valutato** e avere **strutture interne diverse**

In Knowledge Representation:

- Un **argomento** è un **tentativo di persuasione tramite ragioni**
- **Defeasible reasoning**: la **validità** può essere **contestata**
- **Nonmonotonic reasoning**: **conclusioni** possono essere **ritirate** se **emergono nuove informazioni**

# Argomentazione computazionale

**Obiettivo: progettare macchine che argomentano e aiutano le persone a:**

- **Gestire informazioni conflittuali**
- **Prendere decisioni complesse rapidamente**
- **Evitare errori e ottenere informazioni accurate**
- **Usare argomenti persuasivi**

Gli aspetti chiave di un'argomentazione sono:

- **Strutturale**: come sono costruiti gli argomenti?
- **Relazionale**: quali relazioni esistono tra argomenti?
- **Dialogico**: come si svolge il confronto in dialoghi?
- **Valutativo**: come valutare un insieme di argomenti interagenti?
- **Retorico**: come renderli persuasivi per un pubblico?

## Modelli formali

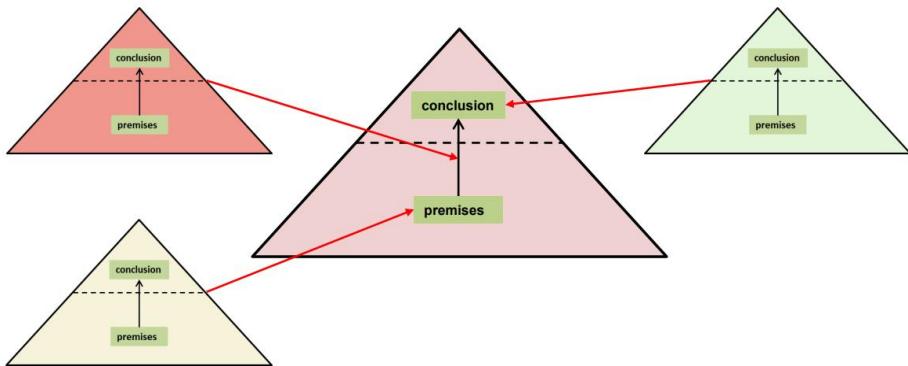
### Argument Mining

Si estrae con la **sentiment analysis** (che servono a comprendere gli atteggiamenti degli utenti, se sono positivi, neutri e negativi) o con **opinion mining** (per capire le opinioni su un tema). Le applicazioni sono molteplici:

- **Visualizzare pro e contro** su un argomento
- **Supporto alla ricerca**
- **Valutazione automatica di saggi**
- **Tecnologie per dibattiti**
- **Analisi social media e consultazioni pubbliche**

### Structured Argumentation

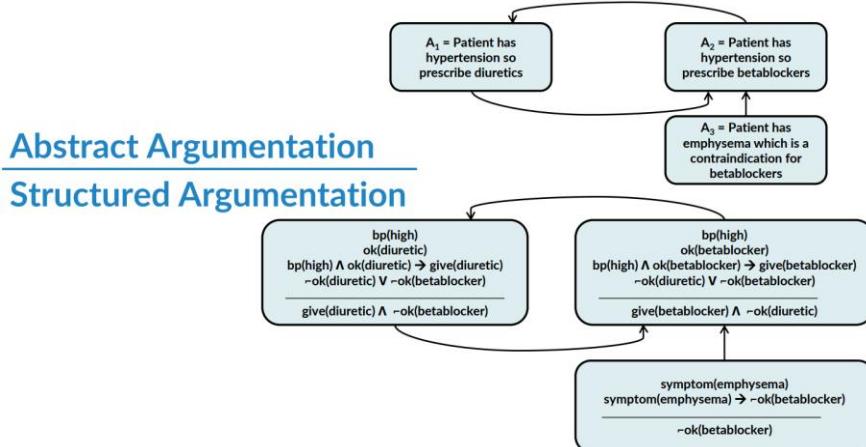
Gli **argomenti** vengono visti come degli **alberi di affermazioni** (premesse → conclusione) e supporta logiche formali. Possono esserci tipi di conflitto (rebuttal, undercutting, undermining), come limite ha la gestione dell'incertezza poco chiara



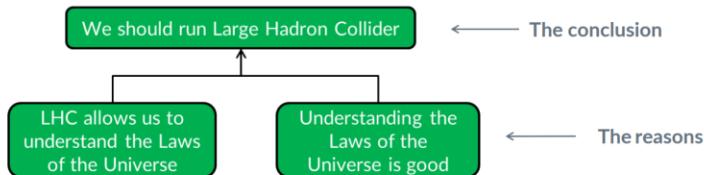
### Abstract Argumentation

Gli **argomenti** sono come **nodi di un grafo**, si **astraggono i contenuti** e si **considerano solo le relazioni** (attacco). **Risolvere i conflitti** vuol dire effettuare un **calcolo** per **determinare** quali **argomenti** sono **accettabili**. Ammette diverse semantiche nella selezione di insiemi di argomenti giustificati

## Esempio pratico



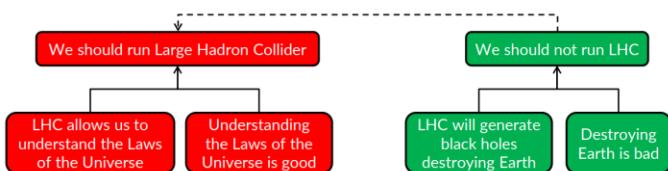
Da Argomentazione Strutturata ad Argomentazione Astratta



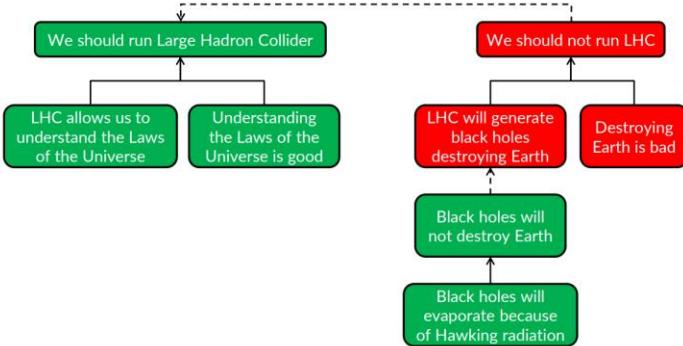
We are justified in believing that we should run LHC 😊

In questo caso potremmo pensare che le argomentazioni siano valide ma:

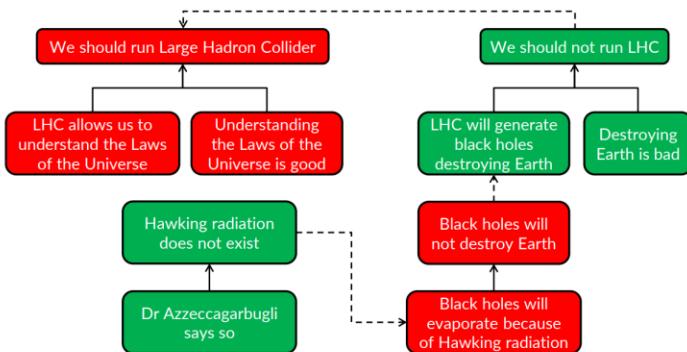
- **Le ragioni non sono sempre conclusive**
- **Argomenti e conclusioni possono essere ritrattati se emergono contro-argomenti**



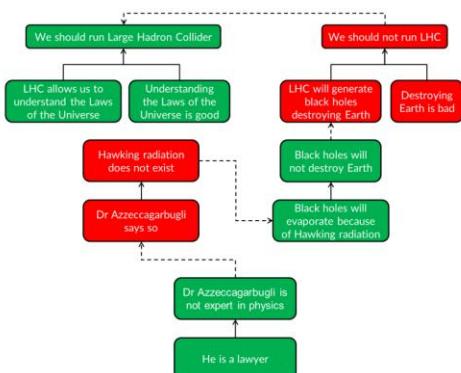
Now we are justified in believing that we should not run LHC 😢



Now we are again justified in believing that we should run LHC 😊



Now we are again justified in believing that we should not run LHC 😞



Now we are again justified in believing that we should run LHC 😊

## Dung's Argumentation Framework (AF)

Un **argumentation framework** è un **modello formale** per **rappresentare e valutare argomenti** in conflitto. Per definizione:

Un **AF** è una **coppia** tale che:

$$AF = (A, R)$$

dove:

- **A = insieme finito di argomenti**
- **R ⊆ A × A = relazione di attacco tra argomenti** (Es.: “A1 attacca A2”)

## Semantiche: come valutare gli argomenti

Le **semantiche** definiscono **insiemi** di **argomenti** (detti **estensioni**) che “**sopravvivono insieme**” al conflitto. Nelle proprietà fondamentali sono:

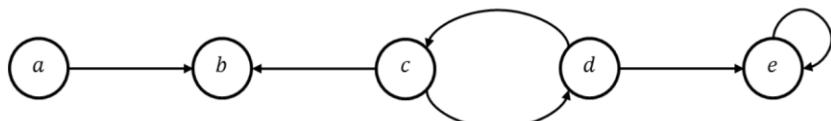
- **Conflict-freeness:** due **argomenti in conflitto non possono** stare nello **stesso insieme**
- **Defense:** un **argomento deve difendersi da ogni attacco** (con un contro-attacco)
- **Admissibility:** un **insieme è ammissibile se è conflict-free e si difende**
- **Strong-Admissibility:** nessun **argomento che si auto-sconfigge**
- **Reinstatement:** se **difendi un argomento, devi includerlo**
- **I-Maximality:** nessuna **estensione è sottoinsieme di un'altra**

## Tipi di semantiche (Extension-based)

- **Naive:** insiemi massimi conflict-free
- **Complete:** insiemi conflict-free che includono tutti gli **argomenti difesi**
- **Grounded:** l'estensione completa minima (più cauta)
- **Preferred:** estensioni complete massime (più estese)
- **Stable:** estensioni che **attaccano** tutti gli **argomenti esterni**

## Esempio pratico

Prendendo come esempio questo AF:



## Conflict-free set

Sono **insiemi di argomenti** che **non contengono coppie in conflitto**, cioè **nessun argomento attacca un altro presente nello stesso insieme**.

Es.:

$$cf(F) = \{a, c\}, \{a, d\}, \{b, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \emptyset$$

NB.: L'insieme vuoto è sempre conflict-free per definizione

## Admissible Sets

Sono **insiemi conflict-free** che **riescono a difendersi da ogni attacco**.

Significa che, se un argomento del set è attaccato, esiste un altro argomento nello stesso set che contrattacca.

Es.:

$$adm(F) = \{a, c\}, \{a, d\}, \{a\}, \{c\}, \{d\}, \emptyset$$

### Naive Extensions

Sono **insiemi massimi conflict-free**, cioè **non esiste un altro insieme conflict-free che li contenga**.

Es.:

$$naive(F) = \{a, c\}, \{a, d\}, \{b, d\}, \emptyset$$

### Complete Extensions

Sono **insiemi ammissibili che includono tutti gli argomenti che difendono**. Hanno come proprietà l'**ammissibilità e il reinserimento degli argomenti difesi**.

Es.:

$$comp(F) = \{a, c\}, \{a, d\}, a$$

NB.: L'insieme vuoto è completo solo se non ci sono argomenti attaccati

### Grounded Extensions

È l'estensione più **cauta**: il **minimo insieme completo**. **Include** solo gli **argomenti sicuramente giustificati** (non attaccati o difesi senza ambiguità).

Es.:

$$grd(F) = a$$

### Preferred Extensions

Sono **insiemi ammissibili massimi**, cioè il più **grande insieme possibile** che **rispetta l'ammissibilità**.

Esempio:

$$pref(F) = \{a, c\}, a, d$$

### Stable Extensions

Sono **insiemi conflict-free** che **attaccano tutti gli argomenti esterni** (quelli non inclusi).

Es.:

$$stb(F) = a, d$$

### Ranking-based Semantics

Quando si analizzano argomentazioni (ad esempio in un dibattito), non è sempre sufficiente dire che un argomento è “accettato” o “rifiutato”. A volte è utile **ordinare gli argomenti in base alla loro forza o**

**accettabilità.** Questo approccio si chiama **Ranking-based semantics**. L'idea di base è quella per cui **un attacco forte può avere lo stesso effetto di più attacchi deboli.** In alcune applicazioni questo approccio è sensato, in altre un po' meno. Un esempio pratico potrebbe essere:

- a: "Lei è la migliore candidata per la posizione"
- p: "Non ha abbastanza esperienza di insegnamento"
- q: "Non ha mai pubblicato in questo settore"
- r: "Non parla inglese"

Un singolo attacco potrebbe non distruggere completamente l'argomento, più attacchi potrebbero indebolirlo progressivamente

Extension-based	Ranking-based
<b>Argomenti: accettati o rifiutati</b> (status assoluto)	<b>Livelli graduali di accettabilità</b>
<b>Argomenti "sconfitti" vengono esclusi</b>	<b>Argomenti non vengono eliminati, ma indeboliti</b>
<b>Numero di attacchi non conta</b>	<b>Più attacchi = minore accettabilità</b>
<b>Stato indipendente dal confronto</b> con altri	Stato relativo: ha senso solo <b>confrontando gli argomenti</b>
Tutti gli <b>accettati</b> hanno lo stesso livello	<b>Infiniti gradi di accettabilità</b>

Volendo sintetizzare ulteriormente:

- **Extension-based**: "Bianco o nero", accettato o rifiutato
- **Ranking-based**: "Graduale", più attacchi = meno forza, ma non necessariamente escluso

## Proprietà

- **Abstraction (Abs):** La classifica degli argomenti dipende solo dagli **attacchi tra argomenti**
  - Es.: Non consideriamo chi ha detto l'argomento, ma solo la struttura degli attacchi
- **Independence (In):** Il confronto tra due argomenti non deve dipendere da **argomenti non collegati**
  - Es.: Se A e B non hanno legami con C, C non influenza il ranking tra A e B
- **Void Precedence (VP):** Un argomento non attaccato è sempre più forte di uno attaccato
  - Es.: Se X non ha critiche e Y sì, X è più accettabile
- **Self-Contradiction (SC):** Un argomento che si attacca da solo è meno accettabile di uno che non lo fa
  - Es.: "Io non sono affidabile" → si auto-smina
- **Cardinality Precedence (CP):** Più attacchi diretti riceve un argomento, meno è accettabile
  - Es.: Se Z ha 3 critiche e W ne ha 1, Z è più debole

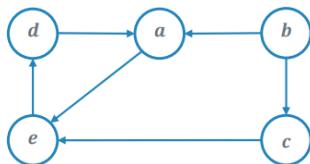
## Categorizer Semantics

Il **Categorizer** è una funzione che assegna un valore numerico a ogni argomento, indicando quanto è accettabile rispetto agli altri. Più alto è il valore, più forte è l'argomento

$$Cat(a) = \begin{cases} 1 & \text{if } a \text{ is unattacked} \\ \frac{1}{1 + \sum_{b \in \mathcal{R}_1^-} Cat(b)} & \text{otherwise} \end{cases}$$

Volendo fare un esempio pratico:

$$\begin{aligned} Cat(a) &= \frac{1}{(1 + Cat(d) + Cat(b))} \approx 0,38 \\ Cat(b) &= 1 \\ Cat(c) &= \frac{1}{(1 + Cat(b))} = 0,5 \\ Cat(d) &= \frac{1}{(1 + Cat(e))} \approx 0,65 \\ Cat(e) &= \frac{1}{(1 + Cat(a) + Cat(c))} \approx 0,53 \end{aligned}$$



$b >^{Cat} d >^{Cat} e >^{Cat} c >^{Cat} a$

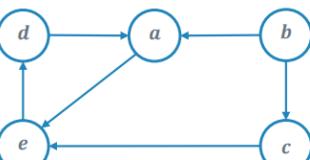
## Discussion-based Semantics (Dbs)

Questa semantica **confronta gli argomenti** contando il **numero di percorsi (paths)** che terminano su **di essi**. Un **percorso** rappresenta una **discussione**: una **sequenza di attacchi** tra argomenti. Se due argomenti hanno lo stesso numero di attaccanti diretti, si considerano percorsi più lunghi (di lunghezza i) finché emerge una differenza. **Non si guarda solo chi attacca direttamente un argomento ma anche gli attacchi indiretti.** Un argomento che è al centro di molte **discussioni** (cioè molti percorsi terminano su di lui) è **più debole perché più contestato**.

Volendo semplificare il modo in cui funziona:

- Si calcola una **funzione**  $Dis(a)$  che misura il “**peso**” delle discussioni che terminano in  $a$
- Si ordina poi la **lista** degli argomenti in base a questo valore: più percorsi → minore accettabilità

$$\begin{array}{c|ccc} & i = 1 & i = 2 & i = 3 \\ \hline Dis(a) & -1 & 2 & -1 \\ Dis(b) & -1 & 0 & 0 \\ Dis(c) & -1 & 1 & 0 \\ Dis(d) & -1 & 1 & -2 \\ Dis(e) & -1 & 2 & -3 \end{array}$$



$b >^{Dbs} d >^{Dbs} c >^{Dbs} e >^{Dbs} a$

## Burden-based Semantics (Bbs)

Questa semantica **assegna** a ogni argomento un **numero di burden** (onere), calcolato in modo ricorsivo:

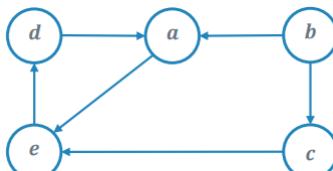
- Se un **argomento non ha attaccanti**, il suo **burden** è 1
- Se è **attaccato**, il **burden aumenta** in base ai burden dei suoi attaccanti

Più un **argomento** è **attaccato** da **argomenti forti**, più il suo **burden** cresce → diventa **meno accettabile**.

Semplificando l'algoritmo:

- Si **calcola** Burden(a):
  - Se a **non ha attaccanti**, **burden = 1**
  - Altrimenti, **burden = 1 + somma** degli **attaccanti**
- Si **ordina la lista** degli argomenti in base ai burden

	$i = 0$	$i = 1$	$i = 2$
$Bur(a) =$	1	3	2,5
$Bur(b) =$	1	1	1
$Bur(c) =$	1	2	2
$Bur(d) =$	1	2	1,33
$Bur(e) =$	1	3	1,83



$b >^{Bbs} d >^{Bbs} c >^{Bbs} e >^{Bbs} a$

## Limiti del modello di Dung

Il framework classico di Dung è **molto potente** ma ha alcune **limitazioni**:

- **Rappresenta** solo **attacchi** tra **argomenti**, in modo **simbolico**
- Non considera **aspetti sociali** (fiducia, reputazione) o **preferenze** tra **argomenti**
- Non gestisce **gradi di forza** degli attacchi
- Può produrre più soluzioni **senza indicare quale scegliere**, o addirittura **nessuna soluzione**
- Il **supporto** tra **argomenti** è solo **implicito** (tramite difese)

## Generalizzazioni del framework

Per superare questi limiti, sono stati proposti modelli più ricchi:

### Value-based AF (VAF)

- Possono essere aggiunte delle **preferenze** sugli **argomenti**
- Es.: In una smart home, la sicurezza può avere più valore del risparmio energetico
- **Gestisce le preferenze**, è utile in contesti con **valori diversi** (confort vs sicurezza)

### Bipolar AF (BAF)

- Introduce anche **relazioni di supporto** oltre agli attacchi
- Es.: "Usare sensori termici supporta l'idea di risparmio energetico."
- **Permette interazioni complesse**
- Assume che il **conflitto** sia sempre **più forte del supporto**

### Weighted AF (WAF)

- **Assegna pesi** agli **attacchi** per indicarne la forza
- Es.: Un attacco basato su dati scientifici è più forte di uno basato su opinioni

- Consente **tolleranza** agli **errori**
- Non **sfrutta** pienamente i **pesi** nelle **estensioni**

### Bipolar Weighted AF (BWAF)

- **Combina supporti e attacchi pesati** per una valutazione più realistica
- Il più **completo**
- **Complesso da calcolare**

## Hyperledgers e IOTA

Commentato [VF2]: Da ampliare e aggiungere immagini

Hyperledger è un **progetto** della **Linux Foundation** che riunisce aziende e sviluppatori per **creare piattaforme, librerie e strumenti** basati su **blockchain** e tecnologie di registro distribuito. L'obiettivo è **fornire soluzioni open source** per **applicazioni aziendali**, garantendo **sicurezza, scalabilità e modularità**.

Un **Distribuite Ledger (registro distribuito)** è una **base dati condivisa** tra più nodi di una rete, dove ogni **modifica è tracciata e verificabile**. Hyperledger propone diversi approcci per blockchain aziendali, sviluppati da una comunità globale.

### Hyperledger Fabric

In particolare, Hyperledger Fabric è uno dei framework più usati per diversi motivi:

- **Struttura modulare** grazie a moduli plug-and-play
- **Resilienza per ecosistemi complessi**
- **Implementazioni “pluggabili”**
- **Opzioni personalizzabili** per la memorizzazione dei dati
- **Controlli granulari** grazie alla Membership Service Providers (MSP) e **canali privati** per transazioni riservate

### Componenti

- **Ledger (L): Registro completo** di tutte le **transazioni, immutabile e verificabile**
- **World State (W): Stato corrente del ledger, memorizzato** come coppie **chiave-valore**
- **Peer (P): nodo** che mantiene **copie** del **ledger** esegue smart contract (**chaincode**), **propone** e **verifica** transazioni
- **Ordering Service (O): ordina le transazioni e le raggruppa in blocchi** per distribuirle ai peer
- **Channel (C): “Sotto-rete” privata** per isolare dati e garantire riservatezza

### Membership e sicurezza

- **Certificate Authority (CA): Emissione** dei **certificati** per **utenti e organizzazioni**
- **Membership Service Provider (MSP): Gestisce identità e autenticazione:**
  - **Local MSP:** definisce **permessi** per **nodi e utenti**
  - **Channel MSP:** condiviso tra i **membri del canale**, garantisce **privacy e politiche comuni**
- **Organizzazione (Org): Partecipante** alla **rete**, rappresentato da peer e certificati

### Configurazione della rete

Ogni organizzazione ha:

- **Peer** per interagire con il ledger
- **Certificati per nodi, admin e applicazioni**

Le applicazioni client comunicano con la rete tramite gateway service (gRPC)

## Chaincode (Smart Contract)

- Codice che **implementa** la **logica** di business
- Sono supportati linguaggi come: Go, Node.js, Java
- **Interagisce** con il **world state** per leggere e aggiornare dati
- Da Fabric v2.0 c'è il **supporto** per l'**esecuzione esterna** (Kubernetes) e **maggior flessibilità** nella gestione

## Endorsement

- **Processo** in cui i **peer** selezionati **eseguono il chaincode** e **restituiscono** una **risposta firmata**
- Endorsement:
  - **Definisce** quali **peer** devono **approvare** una **transazione**
  - È **specificata** nella **definizione del chaincode**
  - **Garantisce integrità e consenso** tra i membri del canale

## Transaction Flow

Prima di **avviare** una **transazione**, devono essere **soddisfatte** alcune **condizioni**:

- Il **canale** deve essere **attivo e configurato**
- L'**utente** dell'applicazione è **registrato** presso la **Certificate Authority (CA)** dell'organizzazione e ha ricevuto le **credenziali crittografiche** (certificati e chiavi) per autenticarsi
- Il **chaincode** (smart contract) è **installato** sui **peer** e **distribuito** sul **canale**
- Il **Gateway Service** è **attivo** sui peer (introdotto da Fabric ≥ 2.4) e semplifica le transazioni

Per cominciare la transazione:

- 1) Il **client** utilizza l'**SDK** (Software Development Kit) per **generare** una **transaction proposal** invocando una **funzione** del **chaincode** con i parametri necessari
- 2) L'**SDK** **firma** la **proposta** usando le **credenziali crittografiche** dell'utente
- 3) La **proposta** viene **inviata** ai **peer** per la fase di **endorsement**

NB.: La firma garantisce autenticità e integrità, evita che la proposta venga alterata

## IOTA

IOTA è una **tecnologia pensata** per l'**IoT** che funge da:

- Layer di regolamento delle transazioni
- Sistema di trasferimento dati tra persone e macchine

Le sue caratteristiche principali sono:

- **Basato su Tangle**, un **registro distribuito** innovativo che supera i limiti delle blockchain tradizionali (in quanto a scalabilità e costi) e introduce un nuovo meccanismo di consenso decentralizzato
- **Zero commissioni**: i micropagamenti sono possibili senza intermediari
- **Ambiente trustless e permissionless**

Riguardo a Tangle, abbiamo due "elementi":

- **Vertice (nodo)**: ovvero una **transazione**, ha un peso  $w=3^n$  con ( $n \geq 0$ )
- **Arco**: se esiste un arco da A a B si dice che “A **approva** B”, se esistesse un percorso da A a C (con lunghezza  $> 1$ ), A approva C direttamente

Ci sono altre definizioni utili, come:

- **Tip**: **transazione** non ancora approvata
- **Peso cumulativo**: **peso** della **transazione** + **somma** dei **pesi** di tutte le **transazioni** che la approvano (direttamente e non)
- **Height**: **lunghezza** del **percorso** più lungo verso il **genesis**
- **Depth**: **lunghezza** del **percorso** più lungo verso un **tip**

## Tip selection

Utilizzando una tip selection basata su dei valori random potremmo avere dei problemi:

- Dei **nodi pigri** possono **approvare** sempre **transazioni vecchie**, **rallentando il sistema**
- Dei **nodi malevoli** potrebbero creare molte **tip artificiali**, **attaccando la rete**

Per questo motivo si utilizza la **Catena di Markov Monte Carlo** (MCMC), ovvero si eseguono più **random walk** da una **transazione confermata** verso un **tip** e ad ogni passo la **scelta del nodo** è **probabilistica**. La probabilità è proporzionale a  $e^{-\alpha(H_x - H_y)}$  dove  $H$  è il **peso cumulativo** e  $\alpha > 0$  è un **parametro**.

Questo approccio **favorisce tip con peso maggiore**, scoraggiando comportamenti malevoli

## Proof of Work

IOTA utilizza **Proof of Work** come **algoritmo** per il **consenso** e serve fondamentalmente come **meccanismo antispam**. Per questo motivo ogni **transazione include** un **hash** che soddisfa un certo livello di difficoltà. Il calcolo può essere fatto in diversi modi:

- **Calcolo locale**: sicuro ma richiede risorse
- **Calcolo remoto**: più veloce ma richiede fiducia
- **Servizi a pagamento**: alta performance e alti costi

## Coordinator

Attualmente la **rete** è **vulnerabile** a un **Sybil attack** e **attacco** del **34%**. Per questo motivo, per garantire la **finalità**, IOTA utilizza un **coordinator**, ovvero un **nodo speciale** gestito direttamente dalla IOTA Foundation e **ogni minuto** emette una **transazione “milestone”** (valore 0). Una transazione è **definitiva** se **approvata** (direttamente o indirettamente) da una **milestone**. Questo coordinator è **temporaneo**, sarà **rimosso** quando la **rete** sarà **abbastanza grande**

# Web Intelligence

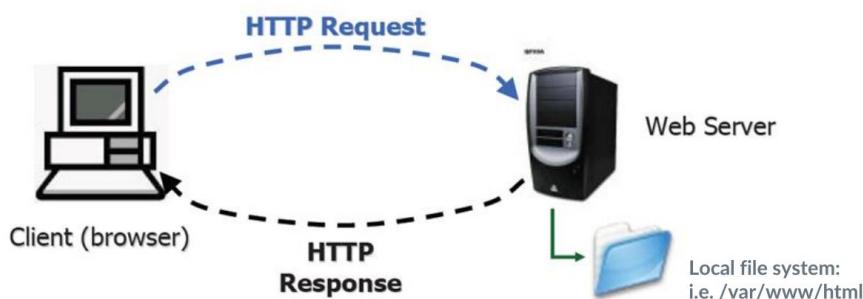
## HTTP

### Introduzione

L'HTTP è un **protocollo di livello applicativo** progettato per sistemi informativi **ipermediati, distribuiti e collaborativi**. Sebbene nato per lo scambio di ipertesti (HTML), oggi è un protocollo **generico** utilizzato per trasmettere **qualsiasi tipo** di dato (immagini, binari, oggetti distribuiti).

Tra le sue caratteristiche chiave:

- **Modello Client-Server:** Il **client invia richieste, il server genera risposte**
- **Stateless:** Il server **non mantiene memoria** delle richieste precedenti dello stesso client, ogni richiesta è indipendente e deve contenere tutte le informazioni necessarie per essere soddisfatta
- **Negoziazione del formato:** Client e Server possono **accordarsi sul formato dei dati** (MIME type)



### Versioning

La storia di HTTP mostra l'evoluzione delle esigenze del Web:

- HTTP/0.9 (1991): Supportava solo il metodo GET e trasferiva solo HTML grezzo senza metadati
- HTTP/1.0 (1996): Prima versione standard (RFC 1945). Introduce gli header, i codici di stato e supporta diversi formati multimediali
- HTTP/1.1 (1997): La versione più diffusa per anni. Introduce connessioni persistenti, caching sofisticato e il multi-homing (più siti sullo stesso server fisico)
- HTTP/2 (2015): Ottimizza la velocità modificando il modo in cui i dati vengono incapsulati e trasportati (binary framing)

### Entità

Oltre a Client e Server, l'architettura HTTP prevede diversi intermediari:

- **User Agent (Client): applicazione** che inizia la richiesta (Es.: browser, bot, spider)

- **Origin Server:** server dove **risiede** fisicamente la **risorsa**
- **Proxy:** **intermediario** che agisce sia da server (per il client) che da client (per il server di origine), può sia filtrare richieste che fare caching
- **Gateway (Reverse Proxing):** **Intermediario** che appare al client come se fosse il server di origine, può essere usato per load balancing o sicurezza
- **Tunnel:** **trasmettitore passivo** di dati, non altera la richiesta
- **Cache:** memoria locale di un'applicazione (sia lato server che client)

**ha formattato:** Tipo di carattere: Grassetto

## Struttura dei messaggi

I messaggi http (richieste e risposte) sono testo leggibile (questo fino a HTTP/1.1) strutturato in:

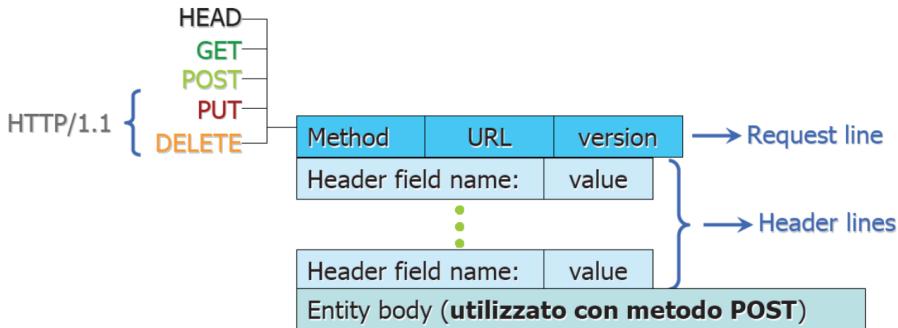
- Start-line: **Tipo di messaggio** (Request-Line o Status-Line)
- Header: **Metadati del messaggio** (coppie nome-valore)
- Body (opzionale): **Contenuto** vero e proprio (Es.: file HTML o i dati di un form)

Un esempio di richiesta HTTP potrebbe essere:

```
POST /api/user/register HTTP/1.1
Host: www.esempio.it
Content-Type: application/json
Content-Length: 45

{
    "username": "MarioRossi",
    "email": "mario@email.it"
}
```

- **Start-line (Request-Line):** POST è il metodo (l'azione che vogliamo compiere)
  - /api/user/register è il percorso (URI) della risorsa
  - HTTP/1.1 è la versione del protocollo utilizzata
- **Header (Metadati):** Host indica il dominio del server a cui ci stiamo rivolgendo
  - Content-Type: specifica che il contenuto nel body è in formato JSON
  - Content-Length: indica la dimensione del body in byte
- **Body (Contenuto):** Separato dagli header da una **riga vuota** (fondamentale per il parsing del protocollo)
  - Contiene i dati effettivi inviati al server (in questo caso, le credenziali dell'utente)



```
GET /index.html HTTP/1.1
Referer: http://www.poliba.it/pippo.html
User-Agent: Mozilla/4.61
Host: www.microsoft.com:80
Accept: image/gif, image/jpeg, image/png, */
Accept-Encoding: gzip
Accept-Language: it
Accept-Charset: iso-8859-1,* ,utf-8
```

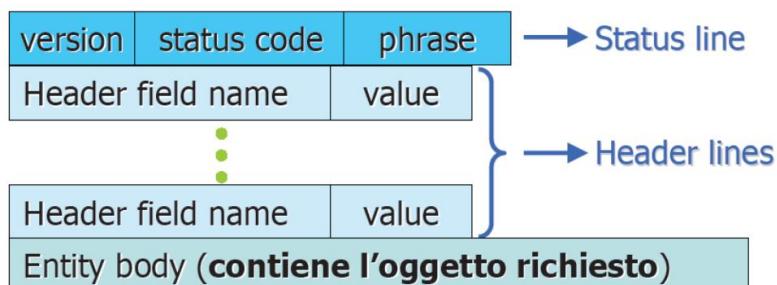
Una risposta, invece:

```
HTTP/1.1 201 Created
Date: Mon, 26 Jan 2026 10:55:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: application/json
Content-Length: 52

{
    "status": "success",
    "message": "Utente registrato con successo"
}
```

- **Start-line (Status-Line):**
  - **HTTP/1.1:** La versione del protocollo
  - **201:** Il **Codice di Stato** (Status Code). In questo caso, 201 significa "**Created**" (Creato), ottimo per una registrazione
  - **Created:** La "Reason Phrase", una breve descrizione testuale del codice
- **Header (Metadati della risposta):**

- **Date**: Il momento esatto in cui è stata generata la risposta
- **Server**: Indica quale software sta usando il server (es. Apache, Nginx)
- **Content-Type**: Fondamentale, dice al client: "Ehi, ti sto mandando un file JSON, trattalo di conseguenza"
- **Body**:
  - È separato dagli header dalla solita **riga vuota**.
  - Contiene la conferma effettiva dell'operazione, spesso usata dal frontend per mostrare un messaggio di successo all'utente.



```
HTTP/1.1 200 OK
Date: Fri, 11 Mar 2005 11:42:03 GMT
Server: Apache/1.3.3
Last-Modified: Tue, 1 Feb 2005 08:21:02 GMT
Content-Length: 127
Content-Type: text/html

<HTML> ... </HTML>
```

## Metodi di richiesta

Definiscono l'azione che il client vuole eseguire:

- **GET**: **Richiede una risorsa**, è **sicuro** (non modifica lo stato del server) e **idempotente** (ripetere la richiesta produce lo stesso effetto). Può essere a sua volta: **assoluta**, **condizionale** o **parziale**
- **HEAD**: Simile a GET ma il server **restituisce** solo gli **header** (senza body), utile per controllare l'esistenza o la validità di una risorsa (Es.: cache check)
- **POST**: **Invia** dati al server per essere processati, non è né sicuro né idempotente (Es.: Database record, invio form)
- **PUT**: **Carica** una **risorsa** sul server restituendo quella esistente all'URL specificato, è idempotente ma non sicuro

- **DELETE**: Rimuove la **risorsa** specificata

## Metodi di risposta

Numeri a tre cifre che indicano l'esito della richiesta

- **1xx (Informational)**: Risposta provvisoria.
- **2xx (Successful)**: Successo (Es.: 200 OK, 201 Created).
- **3xx (Redirection)**: La risorsa è stata spostata (Es.: 301 Moved Permanently, 304 Not Modified).
- **4xx (Client Error)**: Errore nella richiesta (Es.: 400 Bad Request, 401 Unauthorized, 404 Not Found).
- **5xx (Server Error)**: Problema lato server (Es.: 500 Internal Server Error).

## Campi Header e MIME

Il MIME è uno **standard di comunicazione** (RFC 822, 1341) nato originariamente per permettere l'**invio di dati binari** tramite posta elettronica, estendendo le capacità dei sistemi SMTP. Successivamente, è stato adottato dal Web (via HTTP) per **gestire il trasferimento di contenuti multimediali**.

Ogni **flusso** di dati viene **associato** a un **header Content-Type** che segue la sintassi **oggetto/formato**:

- **Oggetto**: Specifica la **categoria macroscopica del contenuto** (Es.: text, image, application)
- **Formato**: Specifica la **struttura precisa dei dati** (Es.: html, jpeg)
- La **coppia oggetto/formato** **costituisce** il **MIME type**
- La **lista ufficiale** dei tipi è **gestita** dalla **IANA** (Internet Assigned Numbers Authority). Per i dati non standardizzati o sconosciuti si utilizza il tipo generico **application/octet-stream**

L'header **Content-Transfer-Encoding** indica **come** l'oggetto è stato **codificato** per il **trasporto**. Le codifiche standard più diffuse sono **7-bit, quoted-printable e base64**

## Header Generici

Questi campi si **applicano** al **messaggio stesso** (sia esso una richiesta o una risposta) e **riguardano** la **trasmissione, non il contenuto** specifico

- **Date**: Data e ora di generazione del messaggio
- **Connection**: Specifica come gestire la connessione (Es.: **Keep-Alive** per mantenerla attiva, o **close** per chiuderla dopo la risposta)
- **Cache-Control**: Direttive per i meccanismi di **caching** (Es.: "non salvare in cache")
- **Transfer-Encoding**: La codifica usata per il trasferimento sicuro del messaggio
- **Via**: Traccia i proxy e i gateway attraversati dal messaggio
- **MIME-Version**: Indica la versione MIME utilizzata (sempre 1.0)

## Header di Entità

Forniscono **informazioni specifiche** sul **corpo** del **messaggio** (body) o sulla **risorsa** richiesta se il body è assente.

- **Content-Type**: Il tipo MIME dell'entità. È **obbligatorio** in ogni messaggio che contiene un **body**
- **Content-Length**: La lunghezza in byte del **body**. Anch'esso **obbligatorio** se c'è un **body**

- **Last-Modified:** Data e ora dell'ultima modifica della risorsa. È fondamentale per la gestione della cache, permettendo al client di sapere se la sua copia locale è ancora valida
- **Expires:** Una data assoluta dopo la quale la risorsa non è più considerata valida e deve essere richiesta nuovamente
- Altri: Content-Encoding (compressione), Content-Language (lingua), Content-Location, Content-MD5 (hash per integrità), Content-Range

## Header specifici di Richiesta

Sono impostati dal client per fornire al server informazioni sulla richiesta e sul client stesso

### *Identificazione e Contesto*

- **User-Agent:** Stringa che identifica il software del client (browser, versione, sistema operativo)
- **Referer:** L'URL della pagina da cui proviene la richiesta (quella che conteneva il link). Se l'indirizzo viene digitato manualmente o aperto dai preferiti, questo campo è assente

NB.: È scritto Referer (errore ortografico storico) invece di Referrer. È usato spesso per profilazione utente e statistiche pubblicitarie

### *Indirizzamento e Autorizzazione*

- **Host:** Specifica il nome di dominio e la porta del server. È obbligatorio in HTTP/1.1
  - Funzione chiave: Permette il Multi-homing (o virtual hosting basato sul nome), ovvero ospitare più siti web (Es.: a.com, b.com) sullo stesso indirizzo IP. Il server usa questo header per capire a quale sito è destinata la richiesta
- **Authorization / Proxy-Authorization:** Contiene le credenziali per accedere a risorse protette
- **From:** Email del richiedente (raramente usato e richiede approvazione utente per privacy)

### *Negoziazione del contenuto*

Il client specifica quali formati è in grado di elaborare, permettendo al server di inviare la versione migliore disponibile. Si usa un "fattore di qualità" (q) da 0 a 1 (dove 1 è il massimo/default) per indicare la preferenza.

- **Accept:** Tipi MIME accettati (Es.: text/html, image/gif)
- **Accept-Charset:** Set di caratteri (Es.: utf-8, iso-8859-1)
- **Accept-Encoding:** Codifiche (Es.: gzip per la compressione)
- **Accept-Language:** Lingua preferita (Es.: it, en)

Es.: text/html; q=1.0, text/plain; q=0.5 significa "Preferisco HTML, ma se non c'è va bene anche testo semplice"

### *Richieste Condizionali e Parziali*

- **Range:** Richiede solo una parte della risorsa (byte range). Usato dai download manager per riprendere download interrotti
- **If-Modified-Since:** Rende la richiesta condizionale ("Inviami la pagina solo se è cambiata dopo questa data")
  - Se la risorsa è cambiata: Il server risponde 200 OK e invia il nuovo body

- Se la **risorsa non è cambiata**: Il server risponde **304 Not Modified** e **non invia il body**, risparmiando banda

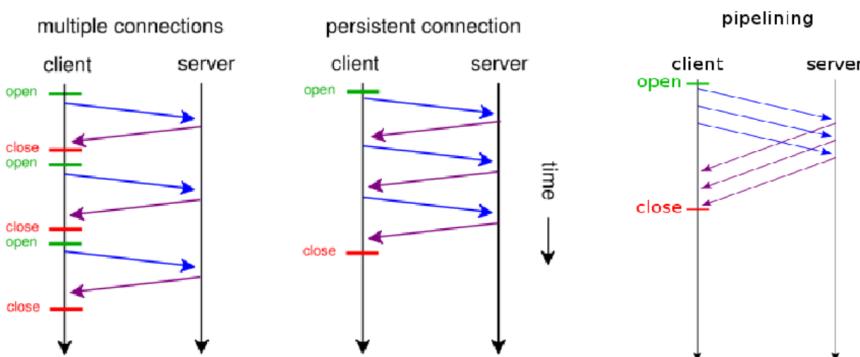
### *Header specifici di Risposta*

Impostati dal **server** per fornire **informazioni** sulla **risposta** e sulle **capacità** del server stesso.

- Server:** Informazioni sul **software del server** (Es.: Apache, versione, OS)
- Accept-Ranges:** Indica se il **server supporta le richieste parziali** (valori: bytes o none)
- WWW-Authenticate:** Usato nelle **risposte di errore 401** per indicare al client il **metodo di autenticazione richiesto**

### Gestione della connessione

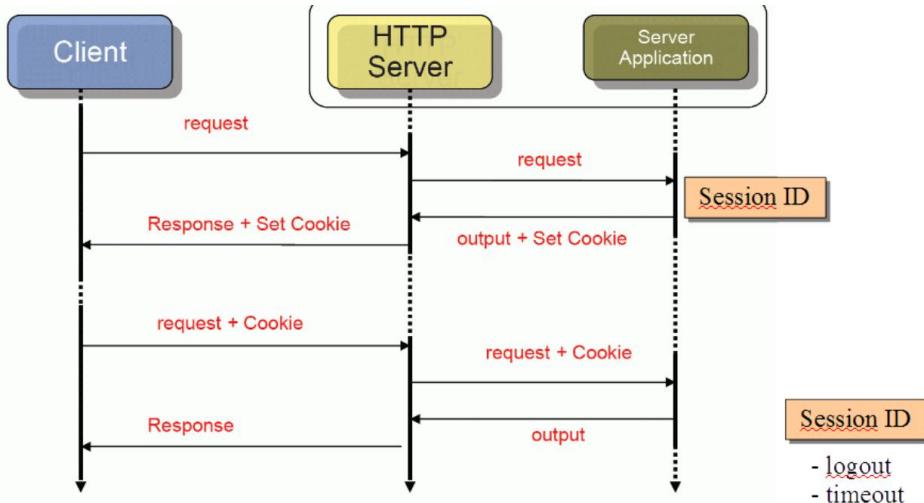
- Connessioni Persistenti (Keep-Alive):** In HTTP/1.1, la connessione TCP rimane aperta per più richieste/risposte, **riducendo l'overhead** di apertura/chiusura socket. È il **comportamento di default**
- Pipelining:** Il client invia più **richieste consecutive senza attendere la risposta della precedente**. Riduce la **latenza**, ma le risposte devono arrivare nello **stesso ordine** delle richieste



### Gestione dello stato (cookies)

Poiché HTTP è **stateless**, i **Cookie** sono stati introdotti per **mantenere la sessione** (stato) tra le diverse richieste. Funziona in questo modo:

- Il **server invia l'header Set-Cookie** con un ID o dati
- Il **client salva il cookie** localmente
- Nelle richieste successive allo stesso dominio, il **client invia l'header Cookie** con quei dati



Relativamente alla privacy ci sono dei cookies che vengono impostati da domini diversi da quello visitato (Es.: banner pubblicitari) e servono a profilare l'utente su più siti. Spesso vengono bloccati dai browser moderni o regolati da header come DNT (Do Not Track)

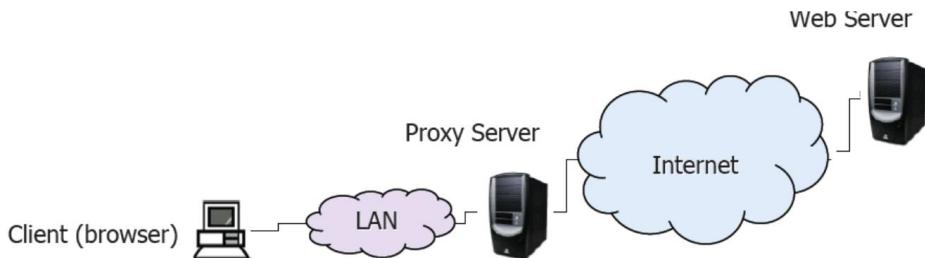
Al posto dei cookies, possono essere utilizzati diversi approcci (ma ognuno di questi ha difetti strutturali):

- **Associazione tramite Indirizzo IP**
  - Fallisce in contesti multi-utente (dove più persone condividono lo stesso IP tramite NAT)
  - con indirizzi IP dinamici, che possono cambiare durante la sessione o essere riassegnati a macchine diverse
- **Campi Hidden nei Form HTML**
  - È possibile nascondere informazioni di stato all'interno di campi `<input type="hidden">`
  - Richiede che ogni singola pagina sia generata dinamicamente dal server e rende la sessione estremamente fragile: se l'utente naviga al di fuori del flusso dei form (ad esempio usando il tasto "Indietro" del browser), lo stato viene irrimediabilmente perso
- **URL Rewriting**
  - Consiste nell'appendere l'ID di sessione direttamente nell'URL (Es.: `sito.com/pagina?id=123`)
  - Esteticamente sgradevole, complica la gestione delle cache dei proxy e rende la stringa di sessione facilmente manipolabile o intercettabile

## Proxy

Nel percorso tra client e server possono trovarsi dei server intermediari che ottimizzano o filtrano il traffico

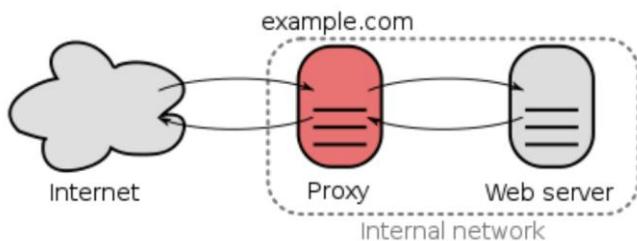
## Server Proxy



Un proxy classico agisce per conto dei client, solitamente all'interno di una rete locale (LAN). Le sue funzioni principali sono:

- **Caching:** Memorizza le **risposte** a richieste **comuni**. Se più utenti nella stessa rete richiedono la stessa risorsa (es. la home page di un giornale), il proxy la serve direttamente dalla sua memoria interna senza occupare banda esterna
- **Filtraggio:** Implementa **politiche di sicurezza** aziendali, **permettendo o negando l'accesso** a specifici domini tramite white list o black list

## Reverse Proxy



A differenza del proxy standard, il **Reverse Proxy** agisce per conto dei server. Il client contatta il reverse proxy convinto che sia il server di origine, ignaro che la richiesta verrà poi inoltrata a un server interno. Questo schema è fondamentale per:

- **Load Balancing:** Distribuire il **carico** di lavoro tra più server fisici per evitare sovraccarichi
- **Sicurezza e Firewall:** Nascondere l'**architettura** interna della rete agli attacchi esterni
- **SSL Termination:** Gestire il **carico computazionale** della crittografia SSL/TLS su hardware dedicato, liberando risorse sui web server principali
- **Indirizzamento IP:** Permettere a più web server di **condividere** un **unico indirizzo** IP pubblico globale

## Caching

Il caching riduce il traffico di rete e la latenza salvando copie locali delle risorse.

Esistono tre livelli principali dove il caching può essere implementato:

- **Client-side (Browser):** La risorsa è salvata direttamente sul dispositivo dell'utente
- **Server-side (Cache server):** Il server memorizza internamente dati computati pesanti (come query al database) per rispondere più velocemente. NB.: questa tecnica riduce il tempo di elaborazione del server, ma non influenza sul carico della rete esterna
- **Intermediate (Proxy Cache):** Situata tra client e server, questa cache serve più utenti contemporaneamente, riducendo drasticamente il traffico sulla rete geografica

## Evoluzione

Inizialmente, il controllo era rudimentale e si basava su tre pilastri:

- **Expires:** Il server comunica una **data esatta di scadenza**. Oltre quella data, la risorsa è considerata vecchia
- **Pragma: no-cache:** Un'istruzione tassativa che **impedisce** alla risorsa di essere **memorizzata** in qualsiasi forma
- **If-Modified-Since:** Un meccanismo di **verifica condizionale**; il client richiede la risorsa solo se questa è stata modificata dopo una data specifica. **Presentava criticità legate alla sincronizzazione dei cronometri tra client e server**

Con HTTP/1.1 è stato introdotto l'header **Cache-Control**, molto più **flessibile**, che permette di definire **politiche di scadenza sia esplicite che euristiche**

### Scadenza (Expiration)

#### Specificata da server

Il server può stabilire quanto a lungo una risorsa sia valida tramite l'header **Expires** o, più modernamente, tramite la direttiva **max-age** all'interno di **Cache-Control** (espressa in secondi). Se la data di scadenza è passata, la risorsa diventa "**stale**" (non fresca). In casi eccezionali, come l'irraggiungibilità del server di origine, una cache potrebbe comunque restituire un dato scaduto accompagnato dallo status code **110 (Response is stale)**

Tuttavia, il server può imporre vincoli più rigidi:

- **must-revalidate:** Se la risorsa è scaduta, la cache deve obbligatoriamente parlare con il server di origine. Se il server non risponde, la cache restituirà un errore **504 (Gateway Time-out)** invece del dato vecchio
- **no-cache:** Contrariamente al nome, non impedisce il salvataggio, ma impone alla cache di validare la risorsa con il server di origine prima di ogni singolo utilizzo

#### Euristica

Quando un server non fornisce indicazioni esplicite sulla durata di una risorsa, il gestore della cache applica degli algoritmi predittivi basati sulla "storia" del file. L'assunzione logica è che una risorsa che non cambia da molto tempo rimarrà probabilmente stabile ancora per un po'

Il calcolo segue solitamente questa formula:

$$\text{expiry} - \text{period} = (\text{current} - \text{date} - \text{last-modified} - \text{date}) \times \text{factor}$$

$$\text{expiry} - \text{date} = \text{current} - \text{date} + \text{expiry} - \text{period}$$

Un valore tipico per il fattore è 0.1. Ad esempio, se un file non viene modificato da 10 ore, la cache assumerà che resterà valido per un'ulteriore ora. Sebbene efficiente, questo approccio è ottimistico e può portare a servire dati obsoleti; in tali casi, viene inviato lo status code **113 (Heuristic expiration)**

## Convalida (Validation)

Anche quando una risorsa scade, non è detto che sia cambiata sul server. Riscaricare l'intero file sarebbe uno spreco di risorse. La validazione serve a verificare se la copia in cache è ancora identica all'originale.

Esistono due metodi principali per validare una risorsa:

1. **Metodo HEAD:** Il client chiede solo gli header della risorsa per controllare la data di ultima modifica (`Last-Modified`). Se la data coincide, evita il download del corpo.
2. **Richiesta Condizionale:** Il client effettua una GET includendo l'header `If-Modified-Since`. Se il server vede che il file è identico, risponde con un leggerissimo codice **304 (Not Modified)** senza inviare il corpo del file, risparmiando banda

## ETag

Gli ETag rappresentano l'evoluzione della validazione basata sul tempo. Un ETag è un identificatore univoco (spesso un hash del contenuto) associato a una specifica versione di una risorsa.

Il server invia l'ETag nella prima risposta. Nelle richieste successive, il client utilizza gli header:

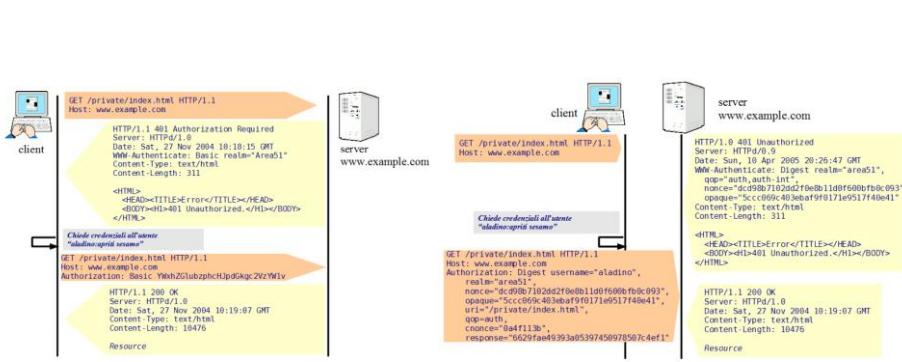
- `If-None-Match`: "Inviami il file solo se il mio ETag non corrisponde più a quello che hai tu". Se corrispondono, il server risponde con **304 Not Modified**.
- `If-Match`: Utilizzato principalmente per operazioni di scrittura (come `PUT`), per assicurarsi di non sovrascrivere modifiche fatte da altri (controllo della concorrenza). Se l'ETag non corrisponde, il server restituisce **412 Precondition Failed**.

Questa metodologia risolve i problemi legati alla precisione del secondo delle date (i file possono cambiare più volte in un secondo) e garantisce che la validazione avvenga sul contenuto effettivo piuttosto che sui metadati temporali.

## Autenticazione e sicurezza

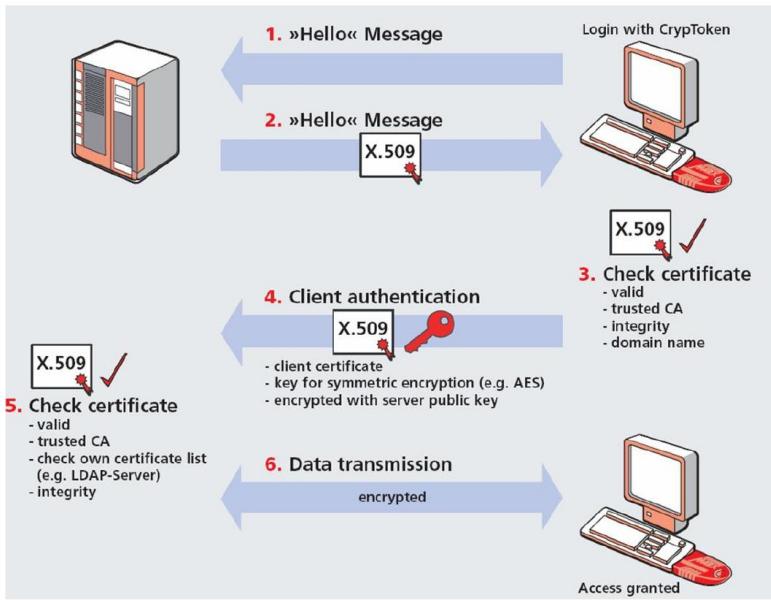
HTTP offre **metodi nativi** per **controllare l'accesso**, ma ha **limiti di sicurezza** se non usato su **canale criptato**. In particolare, ha due metodi per l'autenticazione:

- **Basic Authentication:** Invia `username` e `password` codificati in **Base64** (facilmente decodificabili). Sicuro solo su HTTPS (a sinistra)
- **Digest Authentication:** Non invia la password, ma un **hash** (MD5) calcolato su password, **nonce** (numero casuale) e **altri dati**. Evita l'invio della password in chiaro e protegge dai replay attack (a destra)



Per garantire **riservatezza e integrità**, HTTP viene **incapsulato in TLS** (Transport Layer Security), creando **HTTPS**, con le seguenti caratteristiche:

- **Porta: 443** (invece della 80)
- **Certificati X.509**: Emessi da una **CA (Certification Authority)** per **autenticare il server** (e optionalmente il client)
- **Crittografia Ibrida**: Usa **crittografia asimmetrica** (chiave pubblica/privata) per l'handshake e lo scambio delle chiavi, poi **crittografia simmetrica** (Es.: AES) per la velocità nella trasmissione dei dati



## WWW e W3C

Il World Wide Web (o W3) non è semplicemente "Internet", ma **un'iniziativa di recupero informazioni ipermateriale** su vasta scala (WAN) che ha come obiettivo **fornire accesso universale** a un vasto numero di **documenti**.

La sua definizione concettuale: "È l'universo delle informazioni accessibili via rete, l'incarnazione della conoscenza umana"

### Storia ed Evoluzione

La nascita del Web è legata al CERN e a Tim Berners-Lee:

- Marzo 1989: Berners-Lee scrive la proposta per un sistema informativo distribuito per il laboratorio
- Dicembre 1990: Vengono definiti i concetti base. Berners-Lee scrive il primo browser e il primo software server. Il primo sito web al mondo girava sul computer NeXT di Berners-Lee
- Aprile 1993: Il CERN rilascia il codice sorgente nel pubblico dominio, permettendo l'esplosione del fenomeno
- Crescita: Alla fine del 1994 si contavano già 10.000 server (di cui 2.000 commerciali) e 10 milioni di utenti

### Concetti chiave

Il passaggio fondamentale segnato dal Web è il **cambiamento** nel **modo** di **accedere** alle **informazioni**. Se prima del Web era necessario utilizzare **diversi programmi** per accedere a dati su computer diversi, dopo il Web tutte le informazioni sono accessibili da **qualsiasi tipo di computer**, in qualunque paese, grazie ad un unico programma semplice, il **browser**.

Un concetto cardine è l'ipertesto (Hypertext):

- **Testo** che contiene **collegamenti** (link) ad altri testi
- **Rottura della sequenzialità**: A differenza di un libro, il lettore non è vincolato a una lettura sequenziale ma può "saltare" da un documento all'altro o a parti diverse dello stesso documento
- **Ipermedia**: Quando i documenti non contengono solo testo ma anche grafica, video e suoni

### Protocolli del Web

Il funzionamento del Web si basa su tre pilastri standardizzati

#### URL

È l'**indirizzo univoco** di un **documento** nella **rete**. Dietro ogni link c'è un **URL** che permette di **recuperare** una **risorsa**, indipendentemente dal protocollo usato (Es.: HTTP o FTP). Deve garantire **risolvibilità, persistenza, univocità e leggibilità**

#### HTTP

È il protocollo di trasferimento:

- Caratteristiche: È **veloce, stateless** (senza memoria delle richieste passate) ed **estensibile**

- **Negoziazione:** Permette di superare i problemi legati ai diversi tipi di dati attraverso la negoziazione della rappresentazione (il client dice cosa può leggere, il server risponde di conseguenza)

## HTML

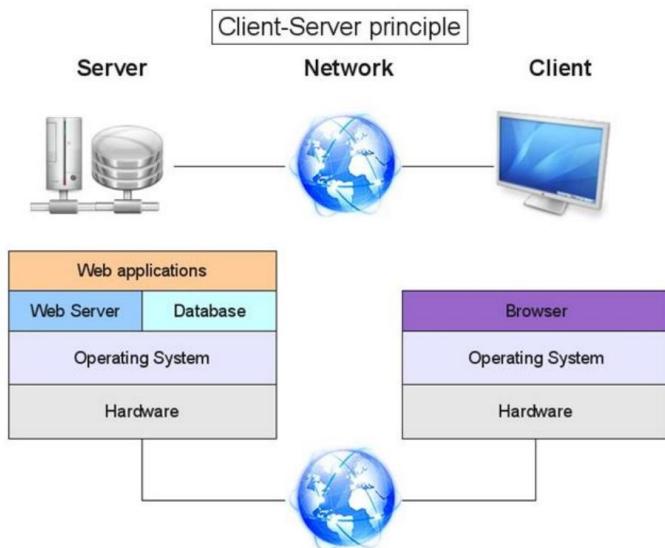
È il formato base dei documenti Web:

- **Struttura Logica:** L'HTML descrive la struttura logica del documento (titoli, paragrafi, liste), **non** la sua **formattazione grafica**
- **Indipendenza dalla piattaforma:** Poiché descrive la struttura, il documento può essere visualizzato in modo ottimale su schermi e sistemi diversi, che applicheranno i propri font e convenzioni

## Architettura

Come abbiamo detto, il Web segue il principio Client-Server:

- **Client (Browser):** L'**utente interagisce** con l'**applicazione** (browser) che gira sul sistema operativo del suo hardware
- **Rete:** La **richiesta viaggia** attraverso la rete
- **Server (Web Server):** Riceve la **richiesta**, **interagisce** con eventuali **database** o applicazioni web, e **restituisce la risorsa**



## World Wide Web Consortium

Fondato nel 1994 da Tim Berners-Lee, è l'organismo che **guida lo sviluppo** del **Web** e del **Web Semantico**. È coordinato da tre poli principali: MIT (USA), ERCIM (Europa) e Keio University (Giappone).

I suoi obiettivi sono:

- 1) **Web for Everyone:** Accesso universale indipendentemente da cultura, abilità o disabilità
- 2) **Web for Everything:** Accesso da qualsiasi dispositivo (PC, mobile, TV, auto)
- 3) **Knowledge Base:** Risolvere problemi complessi attraverso la condivisione della conoscenza
- 4) **Trust and Confidence:** Promuovere tecnologie per transazioni sicure e fiducia nella rete

## Attori del W3C

Il W3C è composto da diverse figure e gruppi che collaborano alla creazione degli standard:

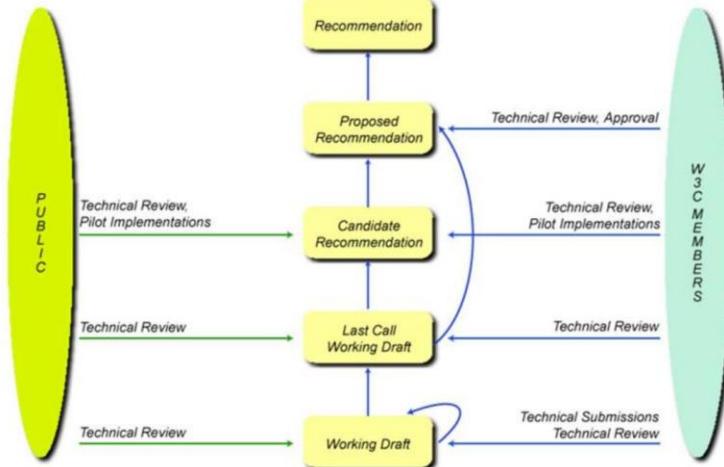
- **Membri (Members):** Organizzazioni (aziende, università) che pagano una quota per far parte del consorzio. Non ci sono membri individuali (tranne casi particolari come "Affiliate")
- **Advisory Committee (AC):** Composto da un rappresentante per ogni organizzazione membro. Devono seguire politiche sui conflitti di interesse
- **Team:** Lo staff retribuito del W3C, inclusi il Direttore e il CEO. Forniscono la leadership tecnica e gestiscono le attività
- **Director:** L'architetto tecnico principale (storicamente Berners-Lee), valuta il consenso sulle scelte architettoniche
- **Advisory Board (AB):** Fornisce guida strategica e legale al Team e risolve conflitti
- **Technical Architecture Group (TAG):** Si occupa dei principi generali dell'architettura Web e coordina lo sviluppo tecnologico tra i vari gruppi
- **Working Groups (WG):** Sono il cuore operativo. Producono le specifiche tecniche (standard), software e suite di test
- **Interest Groups (IG):** Forum di discussione per valutare nuove tecnologie o idee. Non producono standard tecnici

## Processo di standardizzazione (Recommendation Track)

Quando il W3C definisce uno **standard**, questo non si chiama "legge" ma **Raccomandazione**. Il processo per arrivarci è rigoroso:

- 1) **Working Draft (WD - Bozza di Lavoro):** Un documento pubblicato per essere revisionato dalla comunità. Viene ripubblicato ogni volta che ci sono cambiamenti significativi
- 2) **Candidate Recommendation (CR):** Il documento soddisfa i requisiti tecnici del gruppo di lavoro e ha ricevuto un'ampia revisione. È pronto per essere testato
- 3) **Proposed Recommendation (PR):** Il Direttore del W3C accetta il documento come di qualità sufficiente per diventare uno standard
- 4) **W3C Recommendation (REC):** Lo standard finale. Ha ricevuto l'approvazione dei membri e del Direttore. Il W3C ne raccomanda l'adozione universale

NB.: Esistono anche le "Working Group Note", documenti che non sono destinati a diventare standard formali



# HTML

L'HTML (HyperText Markup Language) è il **linguaggio fondamentale** del **Web** per la **creazione** di **contenuti** accessibili a tutti e ovunque:

- È il **linguaggio di marcatura standard** per descrivere la **struttura** delle **pagine** web
- Utilizza i "**tag**" (etichette) per **definire** gli **elementi** della pagina (paragrafi, intestazioni, tabelle, ecc.). I **browser** non visualizzano i tag, ma li **interpretano** per effettuare il rendering del contenuto visibile
- Inventato da Tim Berners-Lee nel 1989 come **linguaggio di pubblicazione** per il Web

## Evoluzione storica

L'HTML ha subito **diverse iterazioni** per adattarsi alle nuove esigenze del web:

- HTML 1.0 - 2.0 (1992-1994): Dalle prime proposte sperimentali alla standardizzazione delle funzionalità base
- HTML 3.2 - 4.01 (1996-1999): Introduzione di tabelle, applet, CSS e stabilizzazione dello standard (W3C Recommendation)
- XHTML (2000-2002): Una riformulazione di HTML come applicazione XML, più rigida e modulare
- HTML5 (2014): Raccomandazione W3C che introduce nuove API, semantica e supporto multimediale nativo
- Living Standard (2019): Attualmente il WHATWG gestisce lo standard come un'entità in continua evoluzione

## Struttura del documento

Un documento HTML moderno segue una **struttura gerarchica** ad **albero** ben definita:

```
<!DOCTYPE html>

<html>
  <head>
    <title>Titolo della Pagina</title>
  </head>
  <body>
    <h1>Intestazione Principale</h1>
    <p>Il contenuto visibile va qui.</p>
  </body>
</html>
```

Dove:

- `<!DOCTYPE html>`: Dichiara il **tipo di documento** e la **versione** HTML al browser. Deve essere la **prima riga** assoluta. Serve a garantire che il browser esegua il rendering in standard mode

anziché in quirks mode. In HTML5 la sintassi è semplificata rispetto alle lunghe dichiarazioni DTD di HTML 4.01 (Strict, Transitional, Frameset)

- `<html>`: L'elemento radice che racchiude tutta la pagina
- `<head>`: Contiene i metadati, il titolo, collegamenti a script e fogli di stile (CSS). Non viene visualizzato direttamente nel viewport
- `<body>`: Contiene tutto il contenuto visibile (testi, immagini, link)

## Sintassi: Elementi, Tag e Attributi

- **Elementi e Tag**: Un elemento è solitamente composto da un tag di apertura (Es.: `<p>`), il contenuto e un tag di chiusura (Es.: `</p>`). Esistono elementi "vuoti" o self-closing che non hanno contenuto né tag di chiusura, come `<img>` o `<br>`
- **Entità: Codici speciali** per rappresentare caratteri riservati (che altrimenti verrebbero interpretati come codice) o simboli
  - Esempio: < si scrive `&lt;` o `&#60;`;
- **Attributi**: Forniscono informazioni aggiuntive agli elementi. Si trovano nel tag di apertura come coppie `nome="valore"`
  - **Attributi Globali**: Utilizzabili su tutti gli elementi. Es.: `id` (identificativo univoco), `class` (per raggruppare elementi, utile per CSS), `style` (stile CSS in linea), `lang` (lingua), `title` (tooltip)
  - **Attributi di Evento**: Innescano script JavaScript al verificarsi di azioni (Es.: `onclick`, `onload`, `onsubmit`, `onkeydown`)

## Head (Metadati)

Situata tra `<html>` e `<body>`, gestisce informazioni cruciali per il browser e i motori di ricerca (SEO). Elementi comuni:

- `<title>`: Il titolo visualizzato nella scheda del browser
- `<meta>`: Definisce charset (Es.: UTF-8), descrizione pagina, autore, viewport
- `<link>`: Collega risorse esterne, tipicamente fogli di stile CSS (`rel="stylesheet"`)
- `<script>`: Contiene o collega codice JavaScript

## Body (Contenuti)

Gli elementi nel body si dividono principalmente in due categorie di visualizzazione:

- **Block-level Elements**: Iniziano sempre su una nuova riga e occupano tutta la larghezza disponibile (Es.: `<div>`, `<h1>`-`<h6>`, `<p>`, `<form>`)
- **Inline Elements**: Non iniziano su una nuova riga e occupano solo la larghezza necessaria al contenuto (Es.: `<span>`, `<a>`, `<img>`, `<b>`)

Esistono anche degli elementi specifici:

- **Testo**: Intestazioni da `<h1>` (più grande) a `<h6>`. Formattazione come `<b>` (grassetto), `<em>` (enfasi), `<strong>` (importante)
- **Liste**:
  - **Non ordinate** (punti elenco): `<ul>` con elementi `<li>`

- o **Ordinate** (numeri/lettere): `<ol>` con elementi `<li>`. L'attributo type definisce lo stile (1, A, a, I, i)
- **Link (Hyperlink)**: Definiti dal tag `<a>`. L'attributo href specifica la destinazione. L'attributo target="\_blank" apre il link in una nuova scheda . Supportano anche i "segnalibri" interni alla pagina (`href="#id"`)
- **Immagini**: Tag `<img>`. Richiede `src` (percorso) e alt (testo alternativo). Dimensionabile con `width` e `height`
- **Tabelle**: Tag `<table>`. Definita riga per riga (`<tr>`). Le celle possono essere di intestazione (`<th>`) o dati (`<td>`). Attributi `colspan` e `rowspan` permettono di unire celle

## Form (Moduli)

I form (`<form>`) servono a **raccogliere input dall'utente** e inviarli al server:

- **Attributi del form**: `action` (URL di destinazione dati) e `method` (GET o POST)
- **Tag `<input>`**: Il più versatile, cambia comportamento in base all'attributo `type` (Es.: `text`, `radio`, `submit`, `checkbox`)
- **Altri elementi form**: `<textarea>` (testo multilinea), `<select>/<option>` (menu a tendina), `<button>`

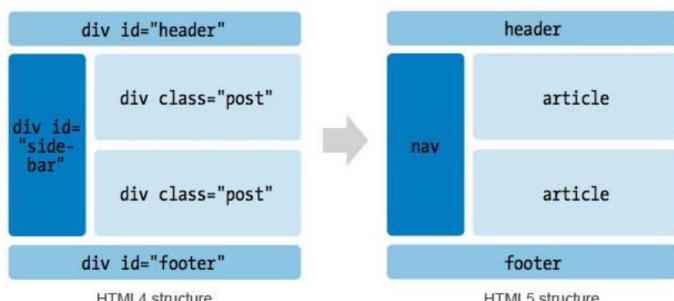
## HTML5

HTML5 ha introdotto cambiamenti significativi per modernizzare il Web

### Semantica

Sono stati introdotti **nuovi tag semantici** per sostituire l'uso **generico** dei `<div>`, migliorando l'accessibilità e la SEO :

- `<header>`: **Intestazione della pagina o sezione**
- `<nav>`: **Menu di navigazione**
- `<section>` e `<article>`: **Sezioni di contenuto** tematico o indipendente
- `<aside>`: **Contenuto laterale/correlato**
- `<footer>`: **Piè di pagina**



### Multimedia e Grafica

HTML5 supporta **nativamente audio** e **video** senza plugin esterni (come Flash):

- `<audio>` e `<video>`: Supportano **controlli nativi** (play, pause, volume) e **formati multipli** tramite il tag `<source>` (MP4, WebM, OGG)
- `<canvas>`: Un  **contenitore** per **disegnare** grafica raster (2D/3D) dinamicamente tramite JavaScript
- `<svg>`: **Supporto** per **grafica** vettoriale scalabile

## Nuovi Input e API

- **Input Form**: Nuovi tipi come date, email, url, number, range che offrono validazione nativa e controlli UI specifici (Es.: calendari)
- **API Avanzate**: Geolocalizzazione, Web Storage (alternativa ai cookie), Web Sockets (comunicazione full-duplex), Multithreading (Web Workers)

## Best practices

Per scrivere codice pulito e compatibile:

- **Dichiarare sempre il** `<!DOCTYPE html>`
- **Usare tag e attributi in minuscolo**
- **Chiudere sempre tutti gli elementi** (anche se i browser moderni sono permissivi)
- **Racchiudere i valori degli attributi tra virgolette**
- **Includere sempre l'attributo alt per le immagini** (accessibilità)
- **Separare lo stile** (usare CSS esterni) e gli script (caricare JS alla fine del body se possibile)

# CSS

Il **CSS (Cascading Style Sheets)** è un **linguaggio progettato** per **descrivere lo stile** e la presentazione di un documento HTML. Questo offre:

- **Separazione tra Contenuto e Stile:** Mentre l'HTML descrive la struttura e il contenuto (paragrafi, intestazioni), il CSS definisce **come** questi elementi **devono essere visualizzati** (colori, font, spaziature)
- **Vantaggi:** Le definizioni di stile sono normalmente **salvate** in **file esterni** (.css). Questo permette di modificare l'aspetto di un intero sito web **aggiornando un solo file**, senza toccare la struttura HTML

## Evoluzione e Livelli (Levels)

Il CSS non ha versioni monolitiche (come 1.0, 2.0) nel senso tradizionale del software moderno, ma si evolve per "Livelli", dove ogni livello costruisce sul precedente:

- Livello 1 (1996): Meccanismi semplici per font, colori e spaziature
- Livello 2 (1998/2011): Introduce il posizionamento dei contenuti, font scaricabili, layout tabellari e supporto per media specifici
- Livello 3: Modula la specifica in parti più piccole e indipendenti (moduli), permettendo un'evoluzione più rapida e flessibile di singole funzionalità (Es.: Colori, Selettori, Box Model)
- Livello 4 e oltre: Non esiste un "CSS 4" unico; i singoli moduli indipendenti possono raggiungere il livello 4 o superiore autonomamente

Esistono sottoinsiemi di CSS definiti per dispositivi specifici con vincoli particolari, come CSS Mobile Profile (smartphone), CSS Print Profile (stampanti low-cost) e CSS TV Profile

## Integrazione HTML-CSS e “Cascading”

Ci sono tre modi per applicare stili a una pagina, elencati qui in ordine di priorità (dal più basso al più alto):

- a) **Esterno** (<link>): Un file .css separato collegato nell'head. È il **metodo raccomandato**
- b) **Interno** (<style>): Definito all'interno del tag <head> della pagina HTML
- c) **Inline** (attributo style): Applicato direttamente sul tag HTML (Es.: <h1 style="color: blue;">). **Sconsigliato** perché mischia contenuto e presentazione

**Ordine a Cascata (Cascading Order):** Se più regole definiscono la stessa proprietà per lo stesso elemento, "vince" l'ultima letta dal browser. La priorità generale è:

- 1) **Browser default** (minore priorità)
- 2) **Fogli di stile Esterni e Interni** (vince l'ultimo dichiarato nell'head)
- 3) **Stile Inline** (massima priorità)

## Sintassi

Una regola CSS è **composta** da un **Selettori** e un **Blocco di dichiarazione**:

```
h1 { color: blue; font-size: 12px; }
```

Dove:

- **Selettori** (`h1`): Indica l'elemento HTML da colpire
- **Dichiarazione** (`color: blue`): Composta da una **Proprietà** e un **Valore**, separati da due punti.  
Ogni dichiarazione termina con punto e virgola

## Tipologie di Selettori

- **Elemento**: Seleziona per **nome** del tag (Es.: `p { ... }`)
- **ID** (`#`): Seleziona un **elemento specifico** con quell'attributo **id univoco** (Es.: `#myid { ... }`)
- **Classe** (`.`): Seleziona tutti gli elementi con **quella classe** (Es.: `.myclass { ... }`)
- **Attributo** (`[]`): Seleziona elementi con un **attributo specifico** (Es.: `p[name] { ... }`)
- **Prossimità/Relazione**:
  - `div h1`: **Discendenti** (`h1` dentro `div`)
  - `div > p`: **Figli diretti**
  - `div + p`: **Fratelli adiacenti** (subito dopo)
- **Pseudo-classi** (`:`): Stato **speciale** di un elemento (Es.: `a:hover` quando il mouse è sopra)
- **Pseudo-elementi** (`::`): Parti di un **elemento** (Es.: `p::first-letter` per la prima lettera)
- **Raggruppamento** (`,`): Applica lo stesso stile a **più selettori** (Es.: `h1, h2, h3 { ... }`)

## Proprietà e valori

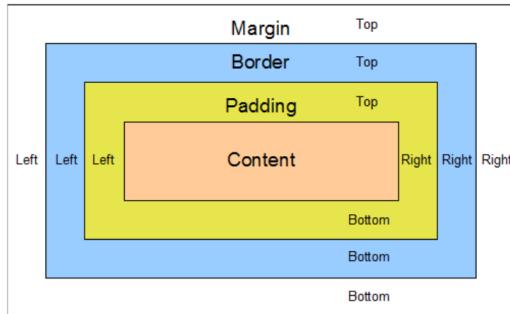
Le proprietà controllano aspetto come colore, sfondo, testo, font, ecc. I valori possono essere:

- **Unità Assolute**: `px` (pixel logici), `pt` (punti), `cm`, `mm`
- **Unità Relative**: `%` (rispetto al contenitore), `em` (rispetto al font-size dell'elemento)
- **Colori**: Possono essere definiti tramite Nome (Es.: `red`), RGB (`rgb(255, 0, 0)`) o Esadecimale (`#FF0000`)

## Box Model

Ogni elemento HTML è considerato come una **scatola rettangolare** composta da **quattro strati concentrici** (dall'interno all'esterno):

- 1) **Content**: Il **contenuto** vero e proprio (testo, immagini)
- 2) **Padding**: **Spaziatura interna trasparente** attorno al **contenuto**
- 3) **Border**: **Bordo** che **circonda** il **padding** (può essere `solid`, `dashed`, `dotted`, ecc.)
- 4) **Margin**: **Spaziatura esterna trasparente** che **separa** l'elemento dagli altri elementi vicini



## Posizionamento

La proprietà `position` definisce **come** un elemento si **colloca** nella **pagina**:

- `static`: Comportamento predefinito, segue il **flusso normale** della pagina
- `relative`: Posizionato **relativamente** alla sua **posizione normale** (spostato con `top`, `left`, ecc.)
- `fixed`: Fissato rispetto alla finestra del browser (viewport); **non scorre** con la pagina
- `absolute`: Posizionato **rispetto al primo genitore posizionato** (non statico) più vicino
- `z-index`: Gestisce la **sovraposizione** (l'ordine sull'asse Z). Valori più alti stanno "sopra"

## CSS3 e Responsive Web Design (RWD)

Il **Responsive Web Design** permette di **usare lo stesso codice** HTML e CSS per **adattare il sito a dispositivi diversi** (Desktop, Tablet, Smartphone)

### Media Queries

Le Media Queries **estendono il concetto** di **tipi di media**, permettendo di applicare stili diversi in base alle **capacità** del **dispositivo** (Es.: larghezza dello schermo, orientamento).

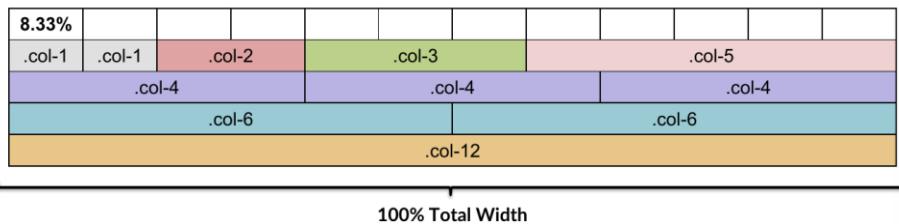
La sintassi è del tipo: `@media type and (expression) { ... }`

Es.:

```
@media screen and (max-width: 480px) {
    /* Stili per smartphone */
    body { background-color: lightblue; }
}
```

### Viewport e Griglie

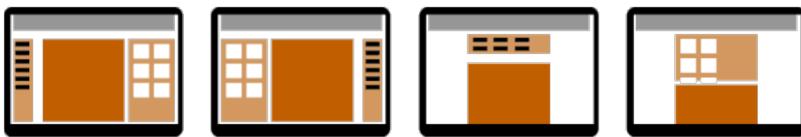
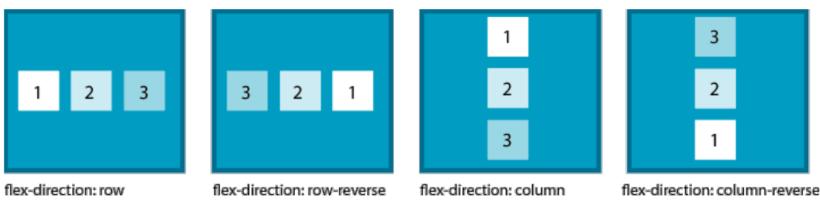
- **Viewport Meta Tag**: Essenziale per il mobile, controlla le **dimensioni** e lo **zoom** della pagina
  - Es.: `<meta name="viewport" content="width=device-width, initial-scale=1.0">`
- **GridView**: Suddivide la pagina in **colonne** (solitamente 12) per facilitare il **layout responsivo** (Es.: classi come `.col-6` per occupare metà larghezza)



## Flexbox

Un **modello di layout monodimensionale** introdotto con CSS3 per allineare e distribuire lo spazio tra gli elementi in un contenitore in modo efficiente. Tra le proprietà abbiamo:

- **Concetti:** C'è un **Flex Container** (genitore) e dei **Flex Items** (figli)
- **Proprietà del Container:**
  - `display: flex;` (**Attiva Flexbox**)
  - `flex-direction`: Definisce **l'asse** (riga o colonna)
  - `justify-content`: **Allineamento orizzontale** (Es.: center, space-between)
  - `align-items`: **Allineamento verticale** (Es.: center, stretch)
- **Proprietà degli Item:**
  - `flex-grow`: Quanto l'elemento può **crescere** per **riempire** lo **spazio**
  - `order`: **Cambia l'ordine** visivo degli **elementi**



## Framework: Bootstrap

Bootstrap è citato come **esempio di framework front-end** gratuito che facilita lo sviluppo responsivo:

- **Approccio Mobile-First:** Gli stili base sono pensati per il mobile e scalati verso l'alto
- **Sistema a Griglia:** Usa classi predefinite per le dimensioni dello schermo (xs per telefoni, sm per tablet, md per desktop, lg per desktop grandi)

# XML

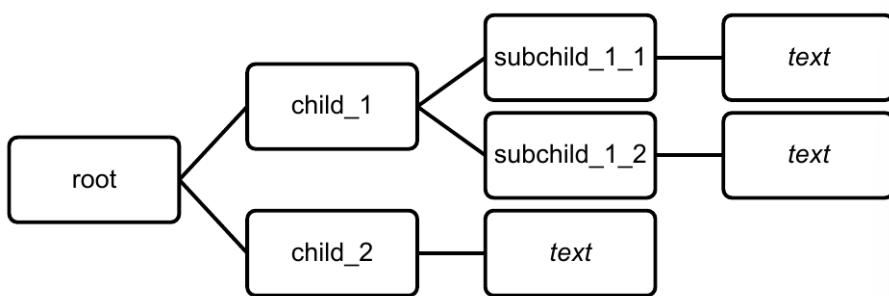
L'XML (eXtensible Markup Language) è un sottoinsieme dell'SGML progettato per permettere la **gestione, la ricezione e l'elaborazione di documenti generici** sul Web. A differenza dell'HTML, che si occupa della **visualizzazione**, l'XML è progettato per **memorizzare e trasportare dati**. Le sue caratteristiche principali sono:

- **Auto-descrittivo:** I tag **non** sono **predefiniti** (come in HTML) ma vengono "**inventati**" dall'autore per descrivere il contenuto
- **Leggibilità:** È **leggibile** sia dalle **macchine** che dagli **umani** (formato testo semplice)
- **Interoperabilità:** **Facilita lo scambio di dati** tra sistemi incompatibili

## Struttura e Sintassi

### Struttura ad Albero

Un documento XML segue una **struttura rigida ad albero** e ogni documento deve avere un **elemento radice** (root) da cui si diramano tutti gli altri elementi (figli e sotto-figli)



### Regole sintattiche fondamentali

Un documento XML "**ben formato**" (well-formed) deve rispettare queste regole:

- **Prolog:** **Opzionale**, ma se presente deve essere la prima riga (Es.: `<?xml version="1.0" encoding="UTF-8" ?>`)
- **Radice:** Deve esistere un unico elemento radice che racchiude tutto
- **Chiusura Tag:** Tutti i tag devono essere **chiusi** (Es.: `<p>...</p>` o `<empty/>` per elementi vuoti)
- **Case Sensitive:** I tag `<Messaggio>` e `<messaggio>` sono diversi
- **Annidamento:** Gli elementi devono essere annidati correttamente (non si possono accavallare i tag)
- **Attributi:** I valori degli attributi devono essere sempre tra virgolette

### Elementi VS Attributi

- **Elementi:** Possono contenere testo, altri elementi o attributi. Sono estensibili e flessibili
- **Attributi:** Forniscono metadati sull'elemento. Non possono contenere strutture ad albero o valori multipli e sono difficilmente espandibili

- o Esempio: <person gender="female">...</person> ("gender" è un attributo)

## Gestione dei nomi (Namespaces)

Quando si **combinano documenti XML diversi**, possono nascere **conflitti** se si usano gli stessi **nomi** per i **tag** (Es.: un tag <table> può riferirsi a una tabella di mobili o a una tabella HTML). Per risolvere questo problema si usano i **Namespace**:

- Si assegna un **prefisso univoco** al tag (Es.: <lif:course>)
- Il prefisso viene **associato** a un **URI** tramite l'attributo `xmlns` (Es.: `xmlns:lif="http://www.poliba.it..."`)

## Validazione: DTD e XSD

Un documento XML è:

- **Well-formed**: Se rispetta la **sintassi di base**
- **Valid** (Valido): Se è well-formed e **rispetta la struttura definita** in uno **schema**

Esistono due linguaggi principali per definire schemi: DTD e XSD

### DTD

È il **metodo più vecchio**, usa una sintassi **compatta** ma non XML:

- **Definizione Elementi**: <!ELEMENT nome (contenuto)>
  - **Quantificatori**: ? (0 o 1), \* (0 o più), + (1 o più)
  - **Contenuto**: #PCDATA (testo parsabile), EMPTY, ANY
- **Definizione Attributi**: <!ATTLIST elemento nome tipo default>
  - **Tipi**: CDATA (testo), ID (identificativo univoco), IDREF (riferimento a ID), (v1|v2) (lista enumerata)
  - **Obbligatorietà**: #REQUIRED (obbligatorio), #IMPLIED (opzionale), #FIXED (valore fisso)

### XSD

È l'**alternativa moderna, basata su XML** (quindi processabile dagli stessi parser) e **molto più potente** del DTD. Supporta **tipi di dati complessi e namespace**

#### *Tipi di dati (Data Types)*

XSD distingue tra:

- **Tipi Semplici** (Simple Types): Elementi che contengono **solo testo, numeri o date**, senza attributi o altri elementi figli. Esempi: `xs:string`, `xs:integer`, `xs:date`, `xs:boolean`
- **Tipi Complessi** (Complex Types): Elementi che **contengono altri elementi o attributi**

#### *Restrizioni (Facets)*

XSD permette di **limitare i valori accettabili** per un tipo semplice:

- `xs:minInclusive / xs:maxInclusive`: Range **numerici**
- `xs:enumeration`: Lista di **valori accettabili**

- **xs:pattern:** Espressioni regolari per il formato (Es.: email)
- **xs:length:** Numero esatto di caratteri

### *Indicatori (Indicators)*

Definiscono come gli elementi **figli** devono **comparire** nei tipi **complessi**:

- Ordine:
  - **xs:sequence:** Gli elementi devono apparire **nell'ordine specificato**
  - **xs:choice:** Deve apparire **solo uno** degli **elementi elencati**
  - **xs:all:** Gli elementi possono apparire in **qualsiasi ordine**
- Occorrenza:
  - **minOccurs:** Numero **minimo** di **volte** (default 1)
  - **maxOccurs:** Numero **massimo** di **volte** (può essere "unbounded")

### *Esempio di sintassi*

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

# DNS

Il DNS è un sistema **essenziale** per il **funzionamento** di **Internet** che gestisce la **corrispondenza** tra **indirizzi numerici** (indirizzi IP) e **nomi logici** (hostname), più facili da memorizzare per gli esseri umani:

- Es.: Converte `sisinflab.poliba.it` in `193.204.49.75`.
- **Mail Alias:** Fornisce l'indirizzo IP del **server di posta** per un **dato dominio** (tramite query MX)

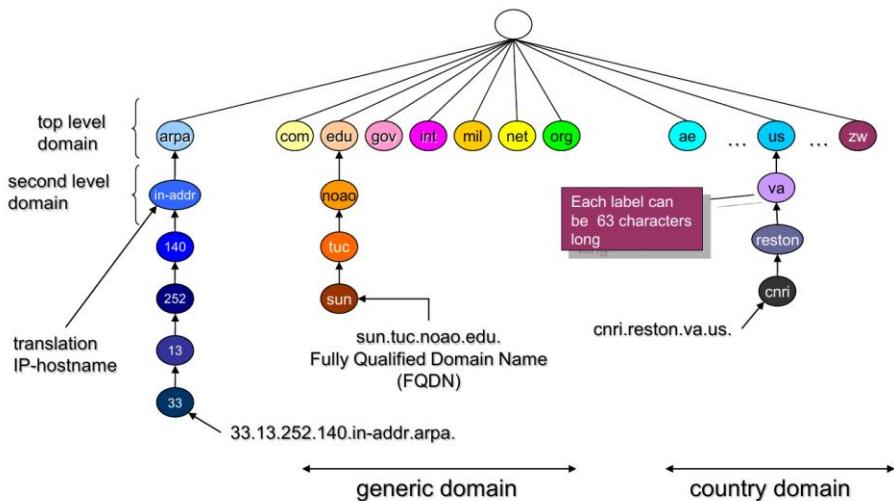
Il sistema si compone di tre elementi principali:

- **Schema di denominazione gerarchico:** Basato sul concetto di "dominio"
- **Database distribuito:** Implementa lo **schema di denominazione**
- **Protocollo:** Gestisce la **manutenzione** e la **distribuzione** delle **informazioni** (query client/DNS o tra DNS)

## Organizzazione dello Spazio dei Nomi (Namespace)

Il DNS utilizza una **struttura ad albero invertito**. Ogni nodo dell'albero è separato da un punto:

- **Root (.)**: La **radice dell'albero** (nodo di livello più alto)
- **Top Level Domain (TLD)**: I **domini di primo livello**, che si dividono in:
  - **Generic Domain**: Es.: `.com`, `.edu`, `.gov`, `.net`, `.org`
  - **Country Domain**: Codici nazionali ISO 3166, Es.: `.it`, `.us`, `.uk`
  - **Infrastructure Domain**: `.arpa` (usato per il reverse lookup)
- **Second Level Domain**: **Sottodomini registrati** sotto i TLD (Es.: `poliba` in `poliba.it`)
- **FQDN (Fully Qualified Domain Name)**: Il **nome completo** di un host che include tutti i livelli fino alla radice (Es.: `sun.tuc.noao.edu.`)
- **Vincoli**: Ogni **etichetta** (label) può essere **lunga** al massimo **63 caratteri**



Riguardo la gestione dei domini:

- **NIC** (Network Information Center): **Gestisce** direttamente i **TLD nazionali** (Es.: il TLD `.it`). Per le altre zone, il NIC delega la gestione a singole organizzazioni
- **Server Autorevoli** (Authoritative):
  - **Primary Server**: È l'**autorità principale** per una zona, **gestisce** direttamente le **corrispondenze**
  - **Secondary Server**: **Mantiene copie dei dati** per **ridondanza, aggiornandole periodicamente** dal server primario

## Architettura e Root Servers

**Esistono 13 Root Name Servers** logici nel mondo (etichettati da a a m), che **gestiscono il livello più alto** del dominio (Root). Essi mantengono le **corrispondenze** per gli **indirizzi** dei **TLD**. Sono **distribuiti globalmente** per garantire **affidabilità e velocità** (Es.: m WIDE Tokyo, k RIPE London, a NSI Herndon VA)

## Meccanismo di Risoluzione

Quando un client deve risolvere un nome (Es.: `host2.mydomain.eu`), avviene un processo che può essere **ricorsivo o iterativo**

### Risoluzione Ricorsiva

Il **client interroga il server DNS locale** (`dns.poliba.it`). Se questo non ha la risposta:

- Il server locale si fa **carico del lavoro** e **interroga la gerarchia** (Root -> TLD -> Authoritative)
- Restituisce al client la **risposta finale**
- **Semplifica il lavoro** del client
- **Carico computazionale maggiore** sul server

### Risoluzione Iterativa

Il **server interrogato risponde** con il riferimento al **prossimo server** DNS che potrebbe conoscere la risposta (Es.: "Non lo so, ma chiedi al server del dominio `.eu`"):

- Il **client** deve **effettuare nuove query** sequenziali
- **Riduce il carico** sui server
- **Aumenta la latenza e l'uso di banda** lato client

## Caching

I server DNS **memorizzano** (cache) le **risposte precedenti** per un certo periodo di tempo (**TTL** - Time To Live):

- **Authoritative Answer**: Risposta proveniente **direttamente** dal **server** che gestisce la zona
- **Non-Authoritative Answer**: Risposta proveniente dalla **cache** di un **server intermedio**. È fondamentale distinguere le due per evitare dati obsoleti o fraudolenti

## Struttura del messaggio DNS

I messaggi DNS (sia query che risposte) hanno un **formato standard** composto da un **Header** e **sezioni variabili**

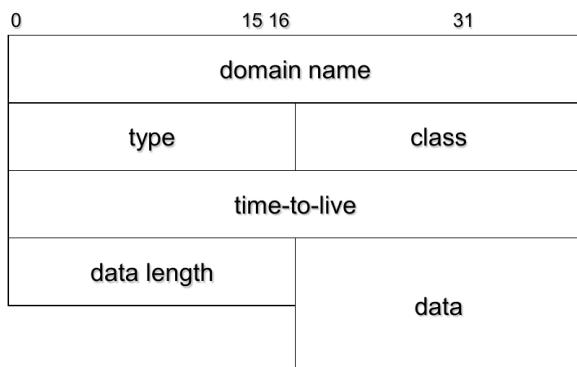
## Header

QR	opcode	AA	TC	RD	RA	Z	AD	CD	rcode
1	4	1	1	1	1	1	1	4	bit

Contiene **campi fissi**:

- **Identification** (Transaction ID): **Intero univoco per accoppiare richiesta e risposta**
- **Flags**: Bit di controllo:
  - QR: 0 = **Query**, 1 = **Response**
  - Opcode: **Tipo di operazione** (0 = standard, 1 = reverse, etc.)
  - AA (Authoritative Answer): 1 se il server è **autorevole** per il **dominio**
  - TC (Truncated): 1 se il messaggio **superà i 512 byte** (limite UDP standard)
  - RD (Recursion Desired): Il client **richiede la ricorsione**
  - RA (Recursion Available): Il server **supporta la ricorsione**
  - Rcode: Codice di **errore** (Es.: 0 = No Error, 3 = Name Error/NXDOMAIN)
- **Contatori**: Numero di record nelle **sezioni** Question, Answer, Authority, Additional

## Resource Record



È l'unità base dei dati nel DNS. Un RR contiene:

- **Domain Name**: Il **nome del dominio**
- **Type**: **Tipo di risorsa** (vedi sotto)
- **Class**: **Solitamente IN** (Internet)
- **TTL** (Time-to-Live): **Tempo in secondi di validità** in cache
- **Data Length & Data**: I **dati effettivi** (Es.: l'indirizzo IP)

## Tipi di Query e Record

- **Tipo A**: Risoluzione Hostname → Indirizzo IP (IPv4)
- **Tipo CNAME** (Canonical Name): Definisce un **alias** per un **host** (Es.: `www.google.it` è un alias per `www.google.com`)
- **Tipo NS** (Name Server): Indica i **server DNS autorevoli** per un dominio

- **Tipo MX** (Mail Exchange): Indica i **server di posta** per il **dominio**. Include un numero di "preferenza" (più basso = priorità più alta) per gestire server multipli
- **Tipo PTR** (Pointer Query): **Risoluzione inversa** Indirizzo IP → Hostname
  - Utilizza il dominio speciale `in-addr.arpa`
  - L'IP viene rovesciato: per trovare il nome di `140.252.13.33`, si interroga `33.13.252.140.in-addr.arpa`
- **Tipo SOA** (Start of Authority): Fornisce **informazioni tecniche** sulla zona (Server primario, contatto email, serial number per aggiornamenti, timer di refresh/retry/expire)

## Strumenti di Diagnostica (Tools)

### nslookup

Tool **interattivo o a riga di comando** per query DNS:

- Es.: (Non-authoritative): `$ nslookup microsoft.com` restituisce gli **indirizzi IP** dalla **cache**
- Es.: (SOA): `$ nslookup -type=soa microsoft.com`

### host

Tool semplice per **conversioni rapide**:

- **Diretta**: `$ host microsoft.com`
- **Inversa**: `$ host 193.204.59.227`
- **MX**: `$ host -t MX poliba.it`

### dig (Domain Information Groper)

Tool **flessibile e dettagliato**, ideale per il **debugging** perché mostra **l'intera struttura del pacchetto** (Header, Flags, Sezioni)

- **Sintassi**: `dig name type`
- **Output**: Mostra le **sezioni** QUESTION, ANSWER, AUTHORITY, ADDITIONAL e i **tempi di risposta**

### Whois

**Interroga i database** dei Registrar (NIC) per ottenere **informazioni amministrative** su un dominio (proprietario, contatti admin/tech, date di creazione/scadenza)

## Server DNS: BIND

Il software server più diffuso è **BIND** (Berkeley Internet Name Domain), il cui demone si chiama **named**.

File di configurazione principale: `/etc/named.conf`

Configurazioni tipiche:

- **Authoritative-only**: Risolve solo i **domini** per cui è **responsabile**
- **Caching server**: Risolve **tutto** e **mantiene i dati** in cache per **velocizzare** le **richieste** della rete locale

**Configurazione di Zona (Zone File)**: Un file di zona definisce i **record** per un **dominio**. Include tipicamente:

- **\$TTL**: Time to Live di default
- **Record SOA**: Parametri vitali della zona
- **Record NS**: Elenco dei nameserver
- **Record A e PTR**: Mappature indirizzi/nomi

# Javascript

JavaScript (JS) nasce nel 1995 (introdotto con Netscape Navigator 2.0) come **linguaggio di scripting** client-side per creare **pagine web attive e dinamiche**. Tra le sue caratteristiche abbiamo:

- **Piattaforma-Indipendente:** L'interprete JavaScript viene eseguito all'interno del browser, rendendolo indipendente dal sistema operativo sottostante
- **Tipizzazione Dinamica:** Non è necessario specificare il tipo di dato alla dichiarazione; i tipi possono cambiare a runtime
- **Orientato agli Oggetti (Prototype-based):** A differenza di linguaggi come Java o C++, JS utilizza i prototipi invece delle classi
- **Relazione con Java:** Non esiste alcuna relazione tecnica tra Java e JavaScript; sono linguaggi completamente diversi

## Paradigmi a Confronto: Class-based vs Prototype-based

È fondamentale comprendere la distinzione tra i linguaggi orientati agli oggetti classici e JavaScript.

I linguaggi Class-based (come Java e C++) si basano su due entità distinte:

- **Classe:** Definisce le **proprietà** (attributi e metodi) ed è **astratta**
- **Istanza:** È l'**oggetto concreto** creato a partire dalla classe

Nei linguaggi Prototype-based, invece:

- **Non esiste distinzione tra classe e istanza:** esistono solo **oggetti**
- **Prototipo:** È un oggetto che funge da "modello" (template). I nuovi oggetti **ereditano** le proprietà direttamente da un oggetto prototipo

Qualsiasi oggetto può **definire** le proprie **proprietà** durante o dopo la creazione e può fungere da prototipo per altri oggetti

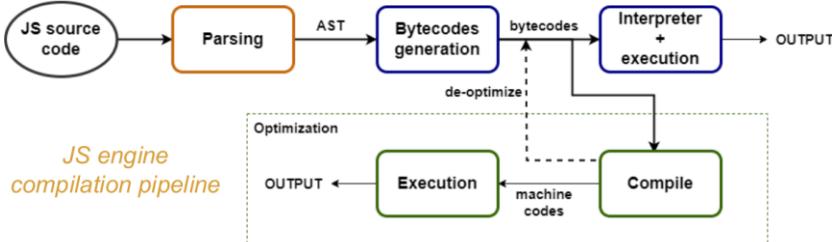
## Il Motore JavaScript (JS Engine)

Il codice JavaScript viene elaborato da un **motore** (JS Engine) presente nel browser. I motori moderni (conformi allo standard ECMAScript) utilizzano la **compilazione JIT** (Just-In-Time) per migliorare le performance.

Tra i principali Engine abbiamo:

- **V8:** Google Chrome, Chromium
- **SpiderMonkey:** Mozilla Firefox
- **JavaScriptCore:** Apple Safari
- **Chakra:** Microsoft Edge (versioni legacy)

Pipeline di Compilazione ed Esecuzione: Il processo segue questi step logici:



- 1) **Parsing:** Il codice sorgente viene **analizzato** sintatticamente e **trasformato** in un **AST** (Abstract Syntax Tree)
- 2) **Generazione Bytecode:** L'AST viene convertito in bytecode
- 3) **Interpretazione ed Esecuzione:** Il bytecode viene **interpretato** ed **eseguito**
- 4) **Ottimizzazione (JIT):** Il motore monitora il codice "caldo" (eseguito spesso), lo compila in codice macchina ottimizzato per velocizzarlo. Se le assunzioni cambiano, può avvenire una "de-ottimizzazione"

## Integrazione in HTML

JavaScript può essere **inserito** in un **documento HTML** tramite il tag `<script>`.

Esistono due modi per inserirlo:

- **Inline:** il codice è scritto **direttamente** nella **pagina**

```
<script type="text/javascript">
  // codice JavaScript
</script>
```

- **Esterno:** il codice risiede in un **file separato** (best practice per modularità)

```
<script type="text/javascript" src="myfunctions.js"></script>
```

**Gestione fallback:** È possibile mostrare **contenuti alternativi** se l'utente ha **disabilitato** JavaScript tramite il tag `<noscript>`

## Sintassi e Struttura

JS è un linguaggio C-Like e case-sensitive (distingue tra maiuscole e minuscole). Le istruzioni terminano (opzionalmente ma consigliato) con ; o con un'interruzione di riga.

I commenti possono essere di due tipologie:

- Singola riga: `// testo`
- Multi-riga: `/* testo */`

## Variabili e Scope

Le variabili sono **contenitori di dati con tipizzazione dinamica**. I nomi possono iniziare con **lettere o underscore**.

Esistono tre modi per dichiarare variabili, con differenze sostanziali nello scope (visibilità) e nel comportamento:

Keyword	Scope	Riassegnabile?	Note Importanti
<b>var</b>	Function-level	Sì	Soggetta a <i>hoisting</i> con valore iniziale <code>undefined</code> . Può essere ridichiarata.
<b>let</b>	Block-level	Sì	Scope limitato al blocco <code>{}</code> . Non ridichiarabile nello stesso scope. Hoisting senza inizializzazione (Temporal Dead Zone).
<b>const</b>	Block-level	No	Deve essere inizializzata alla dichiarazione. Non riassegnabile.

Es.:

```
var x = 10; // vecchio standard
let y = 20; // standard moderno, modificabile
const PI = 3.14; // costante
```

## Tipi di Dato

JavaScript possiede sette tipi di dati **primitivi** e un tipo **complesso** (Object).

I primitivi sono:

- **Number**: Numeri **intei** e virgola mobile (es. 42, 3.14)
- **String**: **Sequenza di caratteri** (es. "Ciao", 'Mondo')
- **Boolean**: Valori **logici** true o false
- **Undefined**: **Variabile dichiarata** ma a cui non è stato assegnato un valore
- **Null**: **Assenza** intenzionale di valore (è un oggetto vuoto)
- **Symbol**: Valore **unico e immutabile** (introdotto in ES6)
- **BigInt**: **Intei** molto **grandi** oltre i limiti di Number (introdotto in ES2020)

Tutto ciò che non è primitivo è un Oggetto (array, funzioni, oggetti built-in)

## Operatori

Gli operatori permettono di manipolare i valori

- **Aritmetici:** `+, -, *, /, %` (modulo), `**` (esponente)
- **Assegnazione:** `=, +=, -=, ecc...`
- **Confronto:**
  - `==` (uguaglianza **debole**, effettua coercione del tipo)
  - `===` (uguaglianza **stretta**, controlla valore e tipo)
  - `!=, !==, >, <, >=, <=`
- **Logici:** `&&` (AND), `||` (OR), `!` (NOT)
- **Bitwise:** `&, |, ~, <<, >>` (operazioni a livello di bit)
- **Ternario:** `condizione ? vero : falso` (shorthand per `if-else`)
- **Tipo:** `typeof` (restituisce il tipo), `instanceof`

## Control Flow (Flusso di Controllo)

Strutture per gestire l'ordine di esecuzione:

- **Condizionali**
  - `if / else`: Esegue blocchi basati su condizioni booleane
  - `switch`: Seleziona tra molteplici casi (utile per valori discreti)
- **Cicli**
  - `while`: Ripete finché la condizione è vera
  - `do-while`: Esegue almeno una volta, poi controlla la condizione
  - `for`: Ripete per un numero definito di volte (inizializzazione; condizione; incremento)

## Funzioni

Blocchi di codice **riutilizzabili**. Possono essere definiti in due modi principali:

### Function Declaration

```
function somma(a, b) {
    return a + b;
}
```

### Function Expression

```
const saluta = function(nome) {
    return "Ciao " + nome;
};
```

## Callback Functions

In JS, le funzioni sono "**cittadini di prima classe**", il che significa che **possono** essere **passate** come **argomenti** ad altre **funzioni**. Una funzione passata come parametro è detta **callback**.

Es.: Una funzione `hello` accetta un'altra funzione `callbackFunc` e la esegue al suo interno per generare parte del risultato

## Stringhe (Strings)

In JavaScript, una stringa è una sequenza di caratteri racchiusa tra virgolette.

Ci sono due tipologie di dichiarazione:

- **Apici singoli (' ')** e **doppi (" ")**: Vengono utilizzati per **stringhe semplici**. Non esiste alcuna differenza funzionale tra i due
  - Es.: `let s1 = 'Ciao'; let s2 = "Mondo";`
- **Backticks (` `)**: Introdotti per gestire **formattazioni complesse**. Permettono **l'interpolazione** di variabili tramite la sintassi  `${variabile}` e supportano stringhe su più righe
  - Es.: `let saluto = `Ciao, ${nome}`;`

Gestione caratteri speciali: Si utilizza il carattere di escape backslash \ per includere caratteri speciali (come \n per una nuova riga o \" per inserire virgolette in una stringa delimitata da virgolette)

**Concatenazione:** Le **stringhe** possono essere **unite** utilizzando l'operatore + oppure, in modo più moderno, tramite i template **literals** (backticks)

## Array

Un array è una struttura dati **ordinata** che memorizza una **sequenza** di valori, alcune sue caratteristiche sono:

- **Dinamicità**: La dimensione di un array non è fissa; può **cambiare** a runtime aggiungendo o rimuovendo elementi
- **Natura "Sparse" (Sparsa)**: Gli array possono contenere **elementi vuoti** (`undefined`). Questi "buchi" non occupano memoria fisica allo stesso modo degli elementi definiti
  - Es.: `var colors = ["Yellow", "Red", , "Blu"];` (l'elemento all'indice 2 è `undefined`)
- **Array Annidati**: È possibile avere array all'interno di altri array (matrici o strutture multidimensionali)
- **Proprietà Length**: Restituisce la lunghezza dell'array. Se inizializzato con `new Array(7)`, la lunghezza sarà 7 anche se vuoto

I metodi principali per la manipolazione sono:

- **sort()**: **Ordina** l'array **alfabeticamente** per default.
  - Per ordinare numeri correttamente, è necessario passare una funzione di comparazione:  
`myArray.sort(function(a, b) { return a - b; })`
- **reverse()**: **Inverte** l'ordine degli elementi
- **push(...)**: **Aggiunge** un elemento alla **fine** dell'array, aumentandone la dimensione
- **pop()**: **Rimuove** e **restituisce** l'ultimo elemento, riducendo la dimensione
- **toString()**: **Converte** l'array in una **stringa** con i valori separati da virgole

Quelli di iterazione, invece:

- **forEach()**: **Itera** sull'array ed esegue una funzione (callback) per ogni elemento. Viene usato per eseguire azioni (side effects). La callback accetta fino a tre parametri:
  - `element`: Il **valore** corrente

- o `index`: L'indice della posizione corrente
- o `array`: Un riferimento all'array completo (utile per modificarlo)
- `map()`: Simile a `forEach`, ma con uno scopo diverso: **crea e restituisce un nuovo array basato sui risultati della funzione callback applicata a ogni elemento**
  - o Es.: Convertire tutte le stringhe di un array in maiuscolo

## Keyword this e il Contesto

La keyword `this` è uno dei concetti più complessi in JS. Essa fa sempre riferimento al "**proprietario**" della funzione in cui viene utilizzata

### Il Problema del Contesto

Quando una funzione è un metodo di un oggetto, `this` punta all'oggetto stesso. Tuttavia, se si definisce una funzione interna (come una callback dentro un `forEach`), il contesto cambia: `this` non punta più all'oggetto ma al **contesto globale** (`window`) o diventa `undefined` (in strict mode).

Es.: In un metodo `show`, `this.message` è accessibile. Ma dentro `forEach(function() {...})`, `this.message` diventa `undefined`

### Le Soluzioni

Esistono tre modi principali per mantenere il riferimento corretto all'oggetto:

- 1) **Salvataggio del contesto** (Pattern "self" o "that"): Si **salva il riferimento a this** in una variabile (spesso chiamata `that` o `self`) all'inizio del metodo, quando si è ancora nello scope corretto
  - a. Es.: `var that = this;` poi si usa `that.message` nella callback
- 2) **Utilizzo di bind()**: Si forza il contesto della funzione interna agganciandola all'oggetto esterno tramite `.bind(this)` alla fine della definizione della callback
- 3) **Arrow Functions** (Soluzione Moderna): Le arrow functions (`=>`) utilizzano lo **scope lessicale**. Non definiscono un proprio contesto, ma **ereditano** il `this` dal blocco di codice genitore (parent scope)
  - a. Es.: `var that = this;` poi si usa `that.message` nella callback

## Manipolazione Esplicita del Contesto

È possibile impostare **manualmente** il valore di `this` per una funzione usando tre metodi:

- `call(thisArg, arg1, arg2...)`: Invoca la funzione immediatamente, passando gli argomenti uno a uno
- `apply(thisArg, [args])`: Invoca la funzione immediatamente, ma accetta gli argomenti come un array
- `bind(thisArg)`: Non invoca la funzione. Restituisce una nuova funzione con il contesto `this` fissato permanentemente all'oggetto specificato

## Oggetti (Objects)

Un oggetto è una collezione di coppie chiave-valore:

- **Chiave** (Property): Una **stringa** o un **Symbol**

- **Valore:** Qualsiasi tipo di dato (inclusi altri oggetti o funzioni)

## Creazione e Accesso

- Si possono definire **letteralmente**: `let person = { name: "Alice", age: 30 };`
- Accesso tramite **dot notation** (`person.name`) o **bracket notation** (`person["age"]`)
- È possibile **eliminare proprietà** con `delete person.prop`

## Costruttori (Object Constructor)

Per creare **molti oggetti** con la stessa struttura, si usano funzioni **costruttori**:

- Per convenzione hanno la lettera **maiuscola** (es. `Person`)
- L'uso di `this` è **obbligatorio** per assegnare le proprietà all'istanza che verrà creata
- Si **istanziano** con la **keyword new**: `var bruce = new Person(...)`

## Metodi

Poiché un oggetto è essenzialmente un **array associativo**, è possibile aggiungere metodi **assegnando una funzione a una proprietà** dell'oggetto, anche dinamicamente o dall'esterno

Es.: `this.setYear = setYearOfBirthday;` (dove `setYearOfBirthday` è una funzione definita altrove)

## Oggetti Comuni (Built-in)

Oggetto	Descrizione
<b>Object</b>	L'oggetto base da cui ereditano tutti gli altri.
<b>Array</b>	Collezione ordinata indicizzata numericamente.
<b>Function</b>	Blocco di codice progettato per eseguire un task.
<b>Date</b>	Gestione di date e orari ( <code>new Date()</code> ).
<b>RegExp</b>	Espressioni regolari per il pattern matching nelle stringhe.
<b>Math</b>	Costanti e funzioni matematiche (Es.: <code>Math.PI</code> ).
<b>JSON</b>	Metodi per il parsing e la stringificazione dei dati JSON.

## Prototype Chain (Catena dei Prototipi)

In JavaScript, l'ereditarietà è **basata sui prototipi**. Ogni oggetto possiede un **riferimento interno** a un altro oggetto, chiamato prototipo, dal quale **eredita proprietà e metodi**

## Meccanismo di Lookup

Quando si cerca di accedere a una proprietà di un oggetto (es. `obj1.propB`), l'interprete segue questi passaggi:

- 1) **Controlla** se l'oggetto stesso **possiede la proprietà**

- 2) Se non la trova, **risale la Prototype Chain**, ispezionando il prototipo dell'oggetto
- 3) Il processo continua fino a **trovare la proprietà** o finché la catena non termina (solitamente con `null`). Se non trovata, restituisce `undefined`

## Copia vs Delega (Delegation)

È fondamentale **distinguere** tra la **copia** delle **proprietà** e la **delega** tramite prototipo:

- **Copia** (`Object.assign`): Crea una **copia istantanea** delle **proprietà** ("one-time property copying"). Se l'oggetto sorgente cambia successivamente, la **copia non viene aggiornata**
- **Delega** (`Object.create`): Crea un **nuovo oggetto** utilizzando un **oggetto esistente** come **prototipo**. Questo instaura una relazione dinamica ("ongoing lookup-time delegation"). Se il prototipo cambia, le **modifiche si riflettono** sugli oggetti che delegano a esso

Es.: Delega

```
var obj1 = { propA: 1 };

var obj3 = Object.create(obj1); // obj3 delega a obj1

obj3.propB = 2; // Proprietà diretta di obj3
// obj3 non ha propA, la cerca nel prototipo (obj1) e la trova
console.log(obj3.propA); // Output: 1
```

## Hoisting (Sollevamento)

L'Hoisting è il comportamento per cui le **dichiarazioni** di variabili e funzioni **vengono spostate** ("sollevate") in **cima** al loro scope corrente prima dell'esecuzione del codice

### Hoisting delle funzioni

Le dichiarazioni di funzione vengono **sollevate completamente**. È possibile invocare una funzione **prima** che essa sia **definita** nel codice

Es.:

```
printMessage();

function printMessage() {
  console.log("Hello");
}
```

### Hoisting delle variabili (var)

Per le variabili dichiarate con `var`, avviene il sollevamento solo della dichiarazione, non dell'assegnazione (inizializzazione):

- La variabile **esiste** in **memoria** ma il suo valore è `undefined` fino alla riga di **assegnazione effettiva**

- Accedere alla **variabile prima dell'assegnazione** non genera un **errore** di "non definito", ma restituisce il valore `undefined`

## Runtime Environment di JavaScript

JavaScript viene eseguito su un **singolo thread** (Single Threaded), utilizzando un **modello basato su Heap** (allocazione memoria) e **Call Stack** (gestione delle chiamate)

### Execution Context (Contesto di Esecuzione)

Il motore JS **crea** un **ambiente** chiamato **Execution Context** (EC). Esistono il **Global EC** (principale) e i **Function EC** (creati a ogni invocazione di funzione).

La creazione dell'EC avviene in **due fasi distinte**:

- 1) **Fase di Creazione** (Memory Creation)
  - a. Viene **creato l'oggetto globale** (window nei browser)
  - b. La keyword `this` viene **legata all'oggetto globale**
  - c. Viene **allocata memoria** per variabili e funzioni (qui avviene **l'hoisting**: le funzioni sono memorizzate interamente, le variabili `var` come `undefined`)
- 2) **Fase di Esecuzione**:
  - a. Il **codice** viene **eseguito** riga per riga
  - b. Vengono **assegnati i valori** reali alle **variabili**
  - c. Le **funzioni** invocate vengono **inserite** nel **Call Stack** (pila delle chiamate). Quando una funzione termina, viene rimossa dallo stack (pop)

## Asincronia ed Event Loop

Essendo JavaScript single-threaded (ha un solo Call Stack), operazioni lunghe bloccerebbero il browser. Per gestire la concorrenza, il browser fornisce meccanismi aggiuntivi

### Componenti del Runtime Asincrono

- 1) **JS Engine**: Heap e Call Stack
- 2) **Web APIs**: Funzionalità fornite dal browser (DOM, AJAX, `setTimeout`) che **operano esternamente** al thread principale
- 3) **Callback Queue** (Coda): Dove vengono **parcheggiate** le **callback** (le funzioni da eseguire al termine di un'operazione asincrona)
- 4) **Event Loop**: Il "vigile" che **coordina** il tutto

### Funzionamento dell'Event Loop

L'Event Loop **controlla costantemente** il **Call Stack** e la **Callback Queue**. Se (e solo se) il Call Stack è vuoto, l'Event Loop preleva il primo task dalla Callback Queue e lo **spinge** nello **Stack** per l'esecuzione

Es.:

```
console.log('10');           // 1. Eseguito subito (Stack)
setTimeout(function() {      // 2. Inviato alle Web API (Timer)
  console.log('cb');        // 4. Messo in Queue dopo 2s -> Stack (quando vuoto)
```

```
, 2000);
console.log('hi');           // 3. Eseguito subito (Stack)
// Output: 10 -> hi -> cb
```

## Promises

Una Promise è un **oggetto** che **rappresenta** l'eventuale **completamento** (o fallimento) di **un'operazione asincrona**. È un'alternativa più pulita alle callback nidificate:

- **Costruzione:** Accetta una funzione con due argomenti: `resolve` (successo) e `reject` (fallimento)
- **Consumo:**
  - `.then()`: Gestisce il caso di **successo** (fulfilled)
  - `.catch()`: Gestisce **l'errore** (rejected)
- I metodi possono essere **concatenati** (chained)

## Closures (Chiusure)

Le Closure sono un concetto derivato dal **Lexical Scoping** (scoping statico). In JS, l'**accessibilità** delle variabili è determinata dalla loro **posizione fisica** nel codice sorgente (annidamento). Uno scope interno ha accesso alle variabili dello scope esterno.

Una Closure è una funzione che "ricorda" le variabili del suo ambito lessicale (dove è stata definita), anche quando viene eseguita al di fuori di quell'ambito

Es.:

```
function outer() {
    var counter = 0;
    function inner() {
        console.log(counter); // inner "chiude" su counter
    }
    return inner;
}

const myFunc = outer(); // outer termina l'esecuzione qui
myFunc(); // 0
```

In questo caso, `inner` mantiene un riferimento vivo alle variabili di `outer` anche dopo che `outer` è stata rimossa dal Call Stack

## Object Decorator Pattern

Il pattern "Decorator" si verifica quando una funzione **accetta un oggetto esistente come input e lo arricchisce** (augment) aggiungendo nuove proprietà o metodi

Es.:

```
var buildcar = function(obj, position) {  
    obj.position = position;  
    obj.move = function() { obj.position++; }; // Metodo aggiunto  
    return obj;  
};  
  
var carA = buildcar({}, 1); // Passo un oggetto vuoto da decorare
```

Se il metodo `move` è definito dentro la funzione `buildcar`, viene creata una nuova funzione in memoria per ogni oggetto creato (nuovo scope di closure). **Questo porta a un elevato spreco di memoria se si creano molti oggetti!**

## Functional Classes (Classi Funzionali)

A differenza del decorator, una classe funzionale **costruisce l'oggetto internamente** invece di riceverlo come input. Le funzioni che producono flotte di oggetti simili sono dette **Constructor Functions**

### Functional Shared Pattern

Per risolvere il problema della memoria del pattern precedente (duplicazione dei metodi), si utilizza il **Functional Shared Pattern**. L'idea è **definire i metodi una sola volta e condividerli** tra tutte le istanze tramite riferimento.

**Problema del contesto (`this`):** Spostando la funzione `move` fuori dal costruttore, essa perde l'accesso alla variabile `obj` (non è più in closure). Si risolve usando la keyword `this`, che viene legata all'oggetto al momento della chiamata (`carA.move()`)

Es.:

```
var move = function() {  
    this.position++; // 'this' si riferisce all'oggetto che invoca il  
    metodo  
};  
  
var Car = function(position) {
```

```

var obj = { position: position };

obj.move = move; // Condivisione del riferimento
return obj;
};

```

## Prototypal Classes (Classi Prototipali)

Questo pattern sfrutta la **catena dei prototipi** per la **delega**, **invece di copiare** i riferimenti ai metodi su ogni oggetto:

- Si crea un **oggetto contenitore** per i metodi (es. `Car.methods`)
- Si usa `Object.create(Car.methods)` nel costruttore per **creare** un nuovo oggetto che **delega** a quel **contenitore**

## Convenzione Standard (.prototype)

Dato che questo pattern è molto comune, JavaScript fornisce un supporto nativo:

- **Ogni funzione** possiede **automaticamente** una **proprietà** `.prototype` (un oggetto vuoto)
- Questo oggetto ha una **proprietà** `.constructor` che punta alla funzione stessa

Es.:

```

var Car = function(position) {
    var obj = Object.create(Car.prototype); // Delega automatica
    obj.position = position;
    return obj;
};

Car.prototype.move = function() { // Aggiunta metodi al prototipo
    this.position++;
};

```

## Pseudoclassical Pattern

È un pattern che tenta di **somigliare** alla sintassi "**class-based**" di altri linguaggi (come Java o C++) aggiungendo uno strato di "**zucchero sintattico**".

Si basa sull'uso della **keyword new**. Quando una funzione viene invocata con `new` (Modalità **Costruttore**), l'interprete esegue automaticamente tre operazioni "invisibili":

- 1) **Crea** un **nuovo oggetto** che delega al `prototype` della funzione
- 2) **Lega** `this` a quel nuovo oggetto
- 3) **Restituisce** automaticamente l'**oggetto** creato

Es.:

```

var Car = function(position) {
    this.position = position; // 'this' è il nuovo oggetto
    // Non serve 'return obj'
};

Car.prototype.move = function() {
    this.position++;
};

var carA = new Car(1); // Uso di 'new'

```

## Superclassi e Sottoclassi (Ereditarietà)

L'obiettivo è creare **nuovi tipi di oggetti** (es. `Van`, `Cop`) che **condividono caratteristiche** con una classe base (`Car`) ma hanno **funzionalità specifiche**, evitando **duplicazione** di codice

### Implementazione Pseudoclassica

Per implementare l'ereditarietà correttamente nel pattern pseudoclassico, bisogna **gestire** sia il **contesto** di esecuzione che la **catena dei prototipi**

#### *Esecuzione del Costruttore Base (Call Context)*

Nel costruttore della sottoclasse (`Van`), bisogna invocare il costruttore della superclasse (`Car`) applicandolo all'istanza corrente (`this`)

- **Sintassi:** `Car.call(this, position);`
- Se usassimo `new Car()`, creeremmo un oggetto separato inutile. `call` permette di "prestare" la logica di inizializzazione di `Car` al nuovo oggetto `Van`

#### *Collegamento dei Prototipi (Delegation)*

Bisogna far sì che `Van.prototype` deleghi a `Car.prototype`, altrimenti i metodi definiti su `Car` (come `move`) non saranno accessibili

- **Sintassi:** `Van.prototype = Object.create(Car.prototype);`
- Questo sovrascrive l'oggetto `prototype` originale di `Van`

#### *Ripristino del Costruttore*

Poiché il passaggio 2 sovrascrive il prototipo, si perde il riferimento corretto al costruttore originale. Bisogna riassegnarlo manualmente

- **Sintassi:** `Van.prototype.constructor = Van;`

Esempio completo:

```

// Superclasse
var Car = function(pos) { this.position = pos; };

```

```
Car.prototype.move = function() { this.position++; };

// Sottoclassa

var Van = function(pos) {
    Car.call(this, pos); // 1. Eredita proprietà
};

Van.prototype = Object.create(Car.prototype); // 2. Eredita metodi
Van.prototype.constructor = Van; // 3. Fix costruttore

var myVan = new Van(1);
myVan.move();
```

# WebAPI e JSON

## JSON (JavaScript Object Notation)

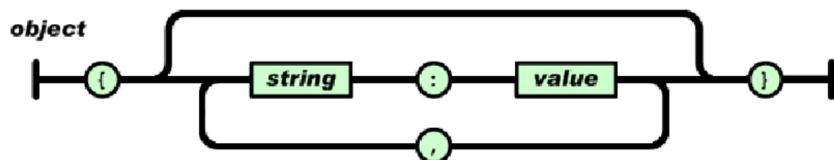
Il JSON è un **formato semplice e versatile** per lo **scambio di dati**, basato su un **sottoinsieme** della **sintassi di JavaScript** (RFC 7159). È **indipendente dal linguaggio** (usabile da Python, Java, C++, ecc.) ed è diventato lo standard per la comunicazione web:

- MIME type: `application/json`
- Estensione file: `.json`

### Struttura dei Dati

Il JSON permette di rappresentare **strutture dati complesse** combinando due strutture fondamentali:

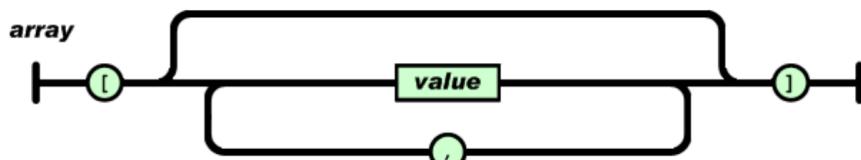
#### Oggetto (Object)



Un insieme **non ordinato** di **coppie** nome/valore:

- Inizia con `{` e finisce con `}`
- I **nomi** sono **stringhe** (tra doppi apici)
- **Separati da virgole**
- **Corrispettivi**: Record, struct, dizionario, hash table

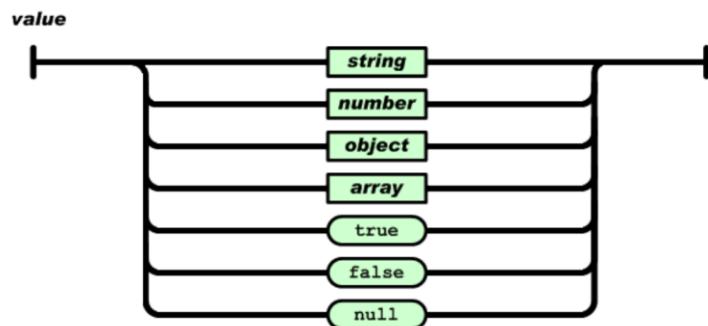
#### Array



Una **lista ordinata** di valori:

- Inizia con `[` e finisce con `]`
- **Separati da virgole**
- **Corrispettivi**: Vettore, lista, sequenza

## Tipi di Valore



Un valore in JSON può essere:

- **Stringa** (tra doppi apici "")
- **Numero** (intero o decimale)
- **Oggetto** (JSON annidato)
- **Array**
- **Booleano** (true, false)
- **Null** (null)

## Esempio di struttura JSON

```
{  
  "glossary": {  
    "title": "example glossary",  
    "GlossDiv": {  
      "title": "S",  
      "GlossList": {  
        "GlossEntry": {  
          "ID": "SGML",  
          "Acronym": "SGML",  
          "Abbrev": "ISO 8879:1986",  
          "GlossSeeAlso": ["GML", "XML"]  
        }  
      }  
    }  
  }  
}
```

## Serializzazione e Deserializzazione in JavaScript

Poiché JSON è un sottoinsieme di JavaScript, i browser moderni offrono un **oggetto nativo JSON** per la conversione:

- **Deserializzazione (Input):** `JSON.parse(stringa)` converte una **stringa JSON** in un **oggetto JavaScript** utilizzabile
- **Serializzazione (Output):** `JSON.stringify(oggetto)` converte un **oggetto JavaScript** in una **stringa JSON** (utile per inviare dati al server)

## Web API e JavaScript nel Browser

Le Web API forniscono **interfacce** per interagire con l'ambiente del **browser** e creare **contenuti dinamici**. Esempi chiave includono il **DOM**, **AJAX**, **Web Storage** e **Geolocation**

## Inserimento ed Esecuzione degli Script

Il codice JavaScript può essere **incluso** nell'`<head>` o nel `<body>`. Tuttavia, se inserito normalmente, il browser **blocca** il **rendering** della pagina finché lo script non è **scaricato** ed **eseguito**

Per evitare il blocco del rendering si usano due attributi nel tag `<script>`:

- `defer`: Lo script viene **scaricato in background** ma **eseguito** solo quando il **DOM** è **pronto** (mantiene l'ordine di esecuzione relativo degli script)
- `async`: Lo script è **completamente indipendente**; viene **scaricato** ed **eseguito** appena possibile, **senza attendere** il **DOM** o altri script

## Gestione degli Eventi

JavaScript **risponde** alle **azioni dell'utente** tramite gli **eventi**. Esistono diverse categorie:

- **Mouse:** `onClick` (click), `onMouseOver` (passaggio del mouse), `onMouseOut`
- **Form:** `onSubmit` (invio modulo), `onFocus` (elemento selezionato), `onBlur` (elemento deselezionato), `onChange` (valore modificato)
- **Documento:** `onLoad` (pagina caricata), `onUnload`

## Modelli a Oggetti: BOM e DOM

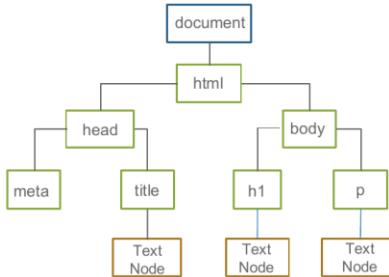
### BOM (Browser Object Model)

Permette l'**interazione** con la **finestra del browser** (oltre il contenuto della pagina):

- **Oggetto window:** L'oggetto **globale**. Contiene metodi come `alert()`, `open()`, e proprietà come `location`
- **Oggetto location:** Gestisce l'**URL corrente** (`href`, `hostname`, `protocol`)
- **Oggetto history:** Permette la **navigazione nella cronologia** (`back()`, `forward()`)

## DOM (Document Object Model)

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Example Page</title>
  </head>
  <body>
    <h1>DOM tree</h1>
    <p>Hello!</p>
  </body>
</html>
```



È la **rappresentazione strutturata** (ad albero) del **documento HTML**. Definisce come **accedere e manipolare gli elementi** della pagina:

- **Oggetto document:** Rappresenta l'**intera pagina**
  - **Attributi:** `forms, images, links, cookie, title`
  - **Selezione Elementi:**
    - `getElementById("id")`: Seleziona per **ID univoco**
    - `querySelector("cssSelector")`: Seleziona il **primo elemento** che corrisponde al **selettori CSS**
    - `querySelectorAll("cssSelector")`: Seleziona **tutti gli elementi corrispondenti**
  - **Creazione:** `createElement("tag")` **crea un nuovo nodo HTML**
- **Manipolazione Elementi** (Interfaccia Element):
  - `innerHTML`: Legge o scrive il **contenuto HTML** interno di un nodo
  - `style`: Accede alle **proprietà CSS** (Es.: `el.style.marginTop = "20px"`)
  - `getAttribute(name) / setAttribute(name, value)`: **Gestione attributi HTML**
  - `appendChild(child)`: Aggiunge un **nodo figlio**

## API di Storage e Geolocalizzazione

### Storage API

Permette di **salvare dati** nel **browser** sotto forma di **copie chiave-valore**:

- **LocalStorage: Persistente.** I dati rimangono anche dopo la chiusura del browser
  - `localStorage.setItem("key", "value")`
- **Session Storage: Temporaneo.** I dati vengono cancellati alla chiusura della scheda/browser

### Geolocation API

Permette di accedere alla **posizione dell'utente** (richiede consenso esplicito):

- Metodo principale: `navigator.geolocation.getCurrentPosition(callback)`

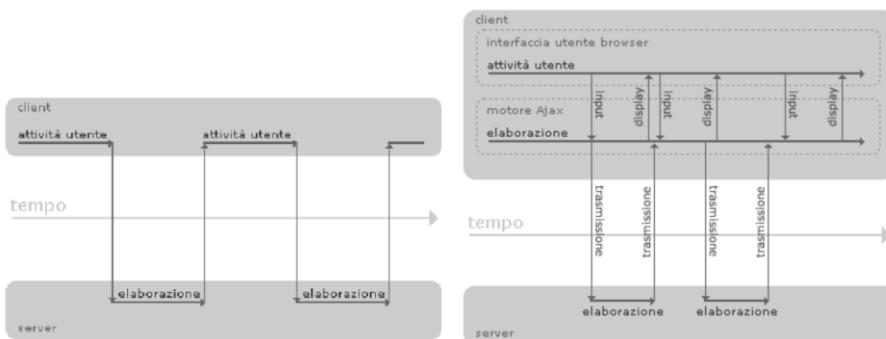
## AJAX (Asynchronous JavaScript And XML)

AJAX è una **tecnica** che permette di **recuperare dati** dal **server** in **background** senza **ricaricare** l'intera **pagina**:

- **Modello Sincrono** (Classico): Richiesta utente → Attesa → Ricaricamento intera pagina
- **Modello Asincrono** (AJAX): Il motore AJAX gestisce la richiesta **indipendentemente dall'interfaccia utente**, che rimane **reattiva**. Quando i dati arrivano, si aggiorna solo una parte della pagina

I vantaggi sono: **Interattività, usabilità, risparmio di banda**

Gli svantaggi, invece: **Gestione complessa della cronologia/preferiti, problemi SEO** (indicizzazione)



## CORS (Cross-Origin Resource Sharing)

Per **sicurezza**, i browser **bloccano le richieste** verso **domini diversi** (Same-Origin Policy). Il CORS è il meccanismo che permette al server di **autorizzare richieste** da **altri domini** tramite **header specifici** come Access-Control-Allow-Origin

## Effettuare Richieste: XMLHttpRequest vs Fetch

### XMLHttpRequest (L'approccio storico)

Oggetto fondamentale per AJAX. **Gestisce la richiesta** tramite **stati**:

#### Ciclo di vita

- **OPENED**: Chiamata a `open()`
- **HEADERS\_RECEIVED**: Chiamata a `send()`, header ricevuti
- **LOADING**: Ricezione dati in corso
- **DONE**: Operazione completata

#### Metodi principali

- `open(method, url, async)`: **Inizializza la richiesta** (Es.: `GET` o `POST`)
- `setRequestHeader(header, value)`: **Imposta header** HTTP (Es.: `Content-Type`)
- `send(data)`: **Invia la richiesta** (con eventuale body)

#### Gestione risposta

Si usa l'evento `onreadystatechange` per **verificare** quando `readyState == 4` e lo status HTTP è `200` (OK). I dati si trovano in `responseText` o `responseXML` (o `response` se parsato come JSON)

## Fetch API (L'approccio moderno)

Un'alternativa più moderna basata sulle **Promise**. Ha una sintassi più **pulita** ed evita il "**callback hell**".

La Fetch API permette facilmente di **configurare metodo**, **header** e **body** passando un oggetto di configurazione come secondo parametro

### Esempio di Fetch

```
fetch('https://api.example.com/data')

  .then(response => {
    if (!response.ok) throw new Error('Errore');

    return response.json() // Parsing automatico JSON
  })

  .then(data => console.log(data)) // Gestione dati

  .catch(error => console.error('Errore:', error)); // Gestione errori
```

## Web Browser

Un web browser è un **programma** software che **recupera documenti** da server web remoti e li **visualizza sullo schermo**. Tra le funzioni principali abbiamo:

- **Visualizzare pagine HTML** direttamente o tramite applicazioni esterne di supporto
- **Gestire le richieste** di risorse tramite URL esplicativi o collegamenti ipertestuali (link)
- **Mantenere traccia della cronologia** e gestire i preferiti (bookmark)
- **Memorizzare dati utente** come password e campi dei moduli (form)
- **Fornire funzionalità di accessibilità** per utenti con disabilità (Es.: ipovisione, disabilità motorie)

## Storia ed Evoluzione

L'evoluzione dei browser ha attraversato diverse fasi cruciali:

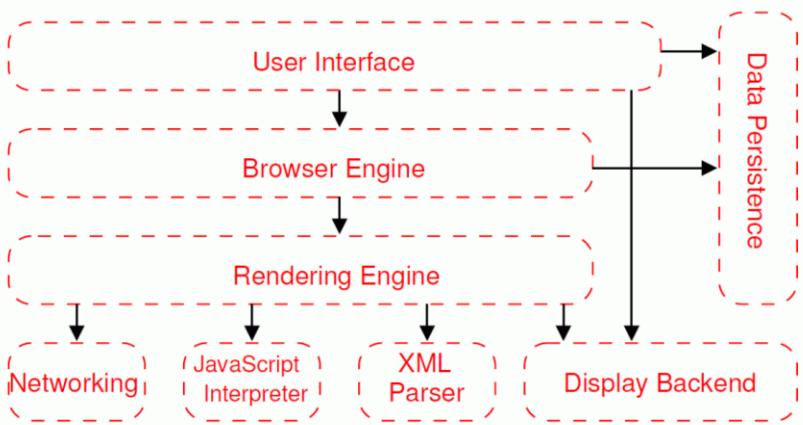
- Gli inizi (1991-1994): Tim Berners-Lee crea il primo browser (grafico e editor). Nel 1993 nasce Mosaic (NCSA), il primo a visualizzare immagini nel testo. Nel 1994 nasce Netscape
- La guerra dei browser (1995-1998): Microsoft rilascia Internet Explorer (basato su Spyglass Mosaic), iniziando una competizione con Netscape. Nel 1998, Netscape rilascia il codice sorgente come open source, dando vita al progetto Mozilla
- L'era moderna e mobile (2007-2008): Con l'iPhone (Mobile Safari) nel 2007, i browser mobili diventano potenti quanto quelli desktop. Nel 2008 Google lancia Chrome, puntando su velocità (motore JS V8), sicurezza e standard W3C. Chrome introduce il progetto open-source Chromium

Un **fork**, in ingegneria del software, avviene quando gli **sviluppatori copiano il codice sorgente** di un progetto per **svilupparne uno distinto** (Es.: Blink è un fork di WebKit)

## Architettura di Riferimento

I browser moderni, sebbene diversi, condividono una struttura a componenti comune. I componenti principali sono:

- **User Interface (UI)**: L'**interfaccia** visibile all'utente
- **Browser Engine**: Il **gestore** delle **azioni** tra UI e motore di rendering
- **Rendering Engine**: **Responsabile** della **visualizzazione** del **contenuto** richiesto
- **Networking**: Gestisce le **chiamate** di rete (HTTP/FTP)
- **JavaScript Interpreter**: Esegue il codice **JS**
- **XML Parser**: Analizza i documenti **XML**
- **Display Backend**: Disegna le primitive grafiche interfacciandosi con il sistema operativo
- **Data Persistence**: Gestisce il **salvataggio** dei **dati** (cookie, cache, ecc.)



## Analisi dettagliata dei componenti

### User Interface

È la **cornice** del **browser**. Include la barra del titolo, la barra degli indirizzi, i pulsanti (indietro/avanti), i segnalibri e le preferenze.

- **Caratteristiche:** Non esiste una **specifica formale**; il design deriva da "**buone pratiche**" consolidate negli anni (Es.: navigazione a schede/tab, barra di ricerca integrata)

### Browser Engine

Agisce come un "**maresciallo**" che **coordina UI e Rendering Engine**.

- **Funzioni:** Avvia il **caricamento** degli **URL**, **gestisce le azioni di navigazione** (reload, back), **comunica errori e progressi di caricamento** alla UI
- **Gestione Plugin:** Gestisce **librerie di terze parti** (Es.: Flash, Silverlight, PDF readers), anche se queste tecnologie sono in fase di abbandono per motivi di sicurezza e performance

### Rendering Engine

È il **cuore** del **browser**: **interpreta HTML e CSS** per produrre la **rappresentazione visiva**:

- **Processo Multi-processo:** I browser moderni (come Chrome e IE) usano **processi separati** per **ogni scheda** (tab) per evitare che il crash di una pagina blocchi l'intera applicazione
- **Motori principali:**
  - Blink: Chrome, Edge, Opera (Open Source)
  - Gecko: Firefox (Open Source)
  - WebKit: Safari (Open Source)
  - Trident: Internet Explorer (Closed Source)

Riguardo al **Flusso di Rendering (Basic Flow)**:

- 1) **Parsing:** L'**HTML** viene **analizzato e convertito** nell'albero **DOM** (Document Object Model). Il **CSS** viene analizzato per creare le regole di stile

- 2) **Render Tree:** Il **DOM** e le **regole di stile** vengono **combinati** per creare il Render Tree. Questo albero contiene solo gli oggetti che devono essere visualizzati (Es.: head non c'è, elementi display: none non ci sono)
- 3) **Layout** (o Reflow): Viene **calcolata** la **posizione esatta** e le coordinate di ogni nodo sullo schermo
- 4) **Painting:** L'albero di rendering viene "dipinto" pixel per pixel usando il Display Backend

### *Javascript Interpreter*

Esegue il **codice JavaScript** incorporato nelle pagine. Per risolvere la lentezza dell'interpretazione classica, i motori moderni (come V8 di Chrome o SpiderMonkey di Firefox) **compilano** il codice **JS** in **codice macchina nativo** durante l'esecuzione, rendendolo estremamente veloce



### *Networking e XML parser*

- **Networking:** Gestisce protocolli (HTTP, FTP), risoluzione dei set di caratteri e tipi MIME. Implementa la cache per ridurre il traffico di rete
- **XML Parser:** Analizza documenti XML per trasformarli in alberi DOM, fondamentale per il funzionamento di AJAX

### *Data Persistence*

#### Gestisce i dati utente:

- **Storage:** Include bookmark, impostazioni, cookie, cache e certificati di sicurezza
- Evoluzione: Utilizzo di database embedded (Es.: SQLite) e sincronizzazione in Cloud (per avere cronologia e password su più dispositivi)

### *Display Backend*

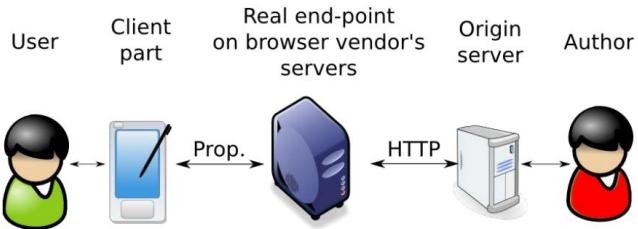
Fornisce **primitive di disegno** (finestre, font):

- **Accelerazione Hardware:** I browser moderni usano la **GPU** per il **rendering**. Il Render Tree viene diviso in **layer grafici** caricati come texture sulla GPU. Questo permette **trasformazioni** (Es.: 3D o animazioni) molto **fluide** senza dover ridipingere l'intera pagina

NB.: Caricare troppe texture può saturare la memoria e causare crash, specialmente su mobile

### **Browser Proxy-Based**

Una categoria particolare di browser progettata per **ridurre l'uso** della **banda** e **accelerare** la **navigazione** su dispositivi limitati



Invece di connettersi direttamente al sito web, il **client** si collega a un **Server Proxy** del **produttore** del browser:

- 1) Il **Proxy scarica la pagina dal server originale**
- 2) Il **Proxy esegue il rendering, elabora il JavaScript e comprime la pagina in un formato leggero**
- 3) Il **Proxy invia la pagina compressa al client**

L'obiettivo è **minimizzare il tempo totale ( $T'$ ) rendendolo inferiore a quello di un browser standard ( $T$ )**:

$$T' = Lcp + Lps + Tcp + Tps + S' + C'$$

Dove il trasferimento dati compresso tra Proxy e Client ( $Tcp$ ) è molto più veloce del trasferimento diretto ( $Tcs$ ), e il tempo di elaborazione del client ( $C'$ ) è ridotto perché riceve dati già processati

# Nginx

NGINX (pronunciato "engine-x") è un **Web server** open source progettato per gestire **un'alta concorrenza di connessioni**. Sviluppato inizialmente da Igor Sysoev nel 2004, è oggi mantenuto da Nginx Inc. NGINX è estremamente versatile e può fungere da:

- **Web Server:** Per servire **contenuti** statici e dinamici
- **Reverse Proxy:** Per protocolli HTTP, HTTPS, ma anche per protocolli di posta come SMTP, POP3 e IMAP

È disponibile per sistemi operativi UNIX-like (Linux, Mac OS X, Solaris) e Windows

## Architettura e Confronto con Apache

Una delle caratteristiche distintive di NGINX è la sua **architettura**, che **differisce radicalmente** da quella dei web server tradizionali come Apache HTTPd (versioni pre-2.4)

### Il Modello Apache (Tradizionale)

Apache utilizzava primitive di **comunicazione sincrone (bloccanti)**. La sua architettura prevedeva la creazione di una **copia del processo** (o un thread) per ogni nuova connessione in arrivo.

Il problema è che questo approccio soffre di problemi di **scalabilità**. All'aumentare del carico, il **consumo** di memoria **RAM cresce linearmente**, richiedendo spesso l'uso di molte macchine fisiche per gestire il carico

### Il Modello NGINX (Event-Driven)

NGINX adotta **un'architettura basata su eventi** e primitive di comunicazione **asincrone** (non bloccanti). Come vantaggio ha il fatto che **non crea un nuovo processo/thread** per ogni connessione. Di conseguenza, il **consumo** di memoria **cresce molto lentamente** anche quando le connessioni concorrenti aumentano drasticamente

## Struttura dei Processi

L'architettura di NGINX si basa su due tipi di processi:

- **Master Process:** È il **supervisore**. Si occupa di **leggere e validare i file di configurazione** e di gestire (avviare, arrestare, mantenere) i processi worker. Permette la **riconfigurazione** del **server senza interrompere** il servizio
- **Worker Processes:** Sono i processi che **gestiscono** effettivamente il **traffico**. Ogni worker può gestire migliaia di connessioni concorrenti

NB.: Nota di configurazione: Poiché i worker sono molto efficienti, non conviene configurarne un numero superiore ai core della CPU disponibili

## Installazione e Gestione

Su Linux, NGINX si trova solitamente in /usr/sbin. Il file di configurazione principale si trova di default in /etc/nginx/nginx.conf

## Comandi Principali

L'eseguibile richiede privilegi di root:

- **Avvio:** `/usr/sbin/nginx`
- **Stop Immediato:** `/usr/sbin/nginx -s stop`
- **Stop "Graceful" (ordinato):** `/usr/sbin/nginx -s quit`
- **Ricarica Configurazione** (senza stop): `/usr/sbin/nginx -s reload`

## Configurazione (`nginx.conf`)

Il file `nginx.conf` è composto da **direttive**. Esistono direttive semplici (terminano con `;`) e **direttive di blocco** (racchiuse in `{ }`) che definiscono dei contesti

```
server {  
    location / {  
        root /data/www;  
    }  
    location /images/ {  
        root /data;  
    }  
}
```

Per esempio posso richiedere un file: `http://localhost/images/example.png`

## Gerarchia dei Contesti

Le direttive vengono applicate in modo gerarchico:

- 1) `http`: Configurazione valida per **l'intera istanza** del web server
- 2) `server`: **Definisce** un **virtual host** specifico (distinto per nome o porta)
- 3) `location`: **Definisce** le **regole** per un **set** specifico di **risorse** (URI) all'interno di un virtual host

## Esempio di Virtual Hosting

È possibile configurare un server che risponde su una porta specifica e serve file da una directory:

```
server {  
    listen 4000;                      # Porta di ascolto  
    server_name www.example.com;      # Nome del dominio  
    location / {  
        root /usr/share/nginx/html/vhost; # Directory radice locale  
        index index.html index.htm;     # File di default  
    }  
}
```

```
}
```

## Logica di Selezione "Location"

Quando NGINX riceve una richiesta, **confronta l'URI** con le **direttive location**. Se ci sono più match, viene selezionato il **blocco** con il **prefisso più lungo** (il match più specifico)

## NGINX come Reverse Proxy e Load Balancer

### Reverse Proxy

NGINX può **inoltrare le richieste** a un **altro server** (Es.: un'applicazione backend o un server Apache). Si usa la direttiva `proxy_pass` all'interno di una location

### Load Balancing (Bilanciamento del Carico)

NGINX può **distribuire il traffico** su più **server backend** per migliorare **performance, scalabilità e affidabilità** (fault-tolerance). Si definisce un gruppo di server nel contesto upstream e poi si usa `proxy_pass` per inoltrare il traffico a quel gruppo

### Metodi di Bilanciamento

- **Round-Robin** (Default): Le **richieste** sono **distribuite ciclicamente** tra i server
- **Least-Connected**: La richiesta viene inviata al server con il **minor numero di connessioni attive** (utile per evitare sovraccarichi)
- **IP-Hash**: Seleziona il server basandosi **sull'hash** dell'indirizzo **IP** del client
  - Garantisce che lo **stesso client** finisce sempre sullo **stesso server** (utile per la **persistenza della sessione**)

### Esempio di Configurazione Load Balancer

```
http {  
    upstream myapp1 {  
        ip_hash;                      # Metodo scelto (opzionale)  
        server srv1.example.com;  
        server srv2.example.com;  
    }  
    server {  
        listen 80;  
        location / {  
            proxy_pass http://myapp1; # Inoltro al gruppo upstream  
        }  
    }  
}
```

## Integrazione con Docker

NGINX è ampiamente utilizzato in ambienti containerizzati:

- **Docker Compose:** Si può definire un **servizio nginx mappando le porte** (Es.: `8080:80`) e **montando volumi** per i file HTML (`nginx_root`) e per la configurazione (`nginx_conf`)
- **Dockerfile:** È possibile **creare immagini custom** partendo da `FROM nginx:latest` e copiando i file di configurazione e i contenuti statici all'interno dell'immagine

## Performance: NGINX vs Apache oggi

Nelle versioni moderne (Apache 2.4.x), le **prestazioni** si sono **avvicinate**, ma rimangono delle distinzioni:

- **Risorse Statiche:** NGINX è leggermente più veloce
- **Risorse Dinamiche:** NGINX è più lento o non le gestisce nativamente
- **Configurazione Tipica:** Si usa spesso NGINX come **server di front-end** per servire i file statici e come reverse proxy verso Apache per gestire i contenuti dinamici

## HTTP/2

L'HTTP/2 è la **seconda versione** principale del **protocollo** di rete utilizzato dal World Wide Web. È stato standardizzato dall'IETF nel 2015 con l'RFC 7540. Questa evoluzione nasce dalla necessità di **superare** le **limitazioni prestazionali** della versione precedente, ormai datata, per adattarsi al web moderno

### Limiti di HTTP/1.1

Sebbene HTTP/1.1 abbia servito il web per decenni (standardizzato nel 1997), la sua progettazione privilegiava la **semplicità implementativa** a scapito delle **prestazioni pure**. I principali difetti strutturali identificati sono:

- **Traffico di rete inefficiente:** Gli **header** sono in **formato testo semplice** (plain text) e non compressi, **aumentando** inutilmente la **dimensione** dei dati trasmessi
- **Problemi di latenza e concorrenza:** Per scaricare più risorse contemporaneamente (concorrenza), i **client** sono costretti ad aprire **più connessioni TCP** verso lo stesso server, sprecando risorse
- **Mancanza di priorità:** Non esiste un meccanismo per dire al server quali risorse sono più urgenti di altre, portando a un **uso povero** della **connessione TC**
- **Modello richiesta-risposta rigido:** Il server può inviare dati solo se esplicitamente richiesti dal client. **Manca** il supporto per il **Server Push**, ovvero l'invio non richiesto di risorse (utile, ad esempio, per inviare file CSS o JS che il server sa già serviranno al client)

### Storia ed Evoluzione: Da SPDY a HTTP/2

Il percorso verso HTTP/2 è stato accelerato dalle iniziative di Google:

- 2009 (SPDY): Google annuncia SPDY, un protocollo sperimentale pensato per migliorare HTTP/1.1
- 2012 (Adozione): SPDY viene supportato dai principali browser e server. Nello stesso anno, il gruppo di lavoro HTTP dell'IETF prende SPDY come base di partenza per la bozza di HTTP/2.0
- 2015 (Standardizzazione): Vengono pubblicati l'RFC 7540 (HTTP/2) e l'RFC 7541 (HPACK, specifico per la compressione degli header)
- Adozione globale: Già alla fine del 2015 i principali browser (Chrome, Firefox, Edge, Safari) e server (Apache, Nginx, IIS) supportavano il protocollo. Nel 2017, circa il 18% dei primi 10 milioni di siti web utilizzava già HTTP/2

### Obiettivi Principali di HTTP/2

L'obiettivo non è sostituire le semantiche di HTTP, ma **renderne il trasporto più efficiente**. Gli obiettivi chiave sono:

- **Riduzione della latenza:** Ottenuta tramite il **multiplexing** completo di **richieste e risposte** (più scambi dati contemporanei sulla stessa connessione)
- **Efficienza: Compressione** degli **header** HTTP per ridurre l'overhead
- **Prioritizzazione: Supporto** alla **definizione di priorità** per le richieste
- **Server Push:** Capacità del server di **inviare risorse proattivamente**

- **Retrocompatibilità semantica:** HTTP/2 estende HTTP/1.1, non lo sostituisce. **Metodi** (GET, POST), **codici di stato** (200, 404) e **URI** rimangono **invariati**, garantendo che le applicazioni web esistenti continuino a funzionare senza modifiche

## Architettura Tecnica: Il Binary Framing Layer

La differenza fondamentale tra HTTP/1.1 e HTTP/2 risiede nel modo in cui i **messaggi vengono formattati e trasmessi**. HTTP/2 introduce il **Binary Framing Layer**.

### Perché non si chiama HTTP/1.2?

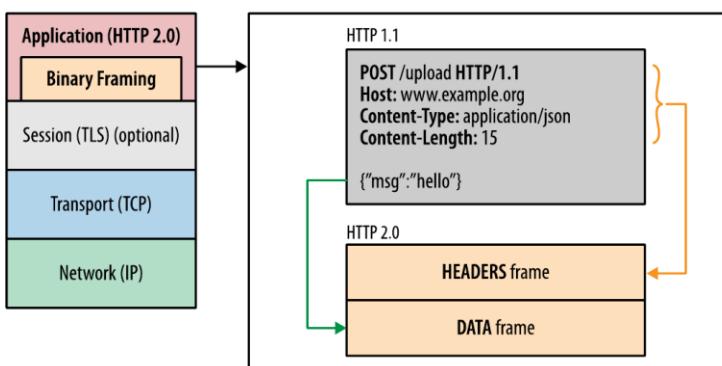
Perché introduce un **cambiamento radicale** nel **trasporto**: si passa dal **testo al binario**. Questo non è retrocompatibile con i vecchi parser HTTP/1.x. Tuttavia, questa modifica è **trasparente** per gli **sviluppatori web**; sono i browser e i server (che lavorano con i socket TCP grezzi) a dover gestire questa complessità

### Come funziona il Binary Framing

In HTTP/1.1, un **messaggio** era un **blocco di testo continuo**. In HTTP/2, i **messaggi** vengono **suddivisi** in **unità più piccole** chiamate **Frame** e **codificati in binario**.

La pila protocollare diventa:

- **Application:** HTTP 2.0 (Semantic)
- **Binary Framing:** Incapsulamento e trasferimento
- **Session:** TLS (Opzionale ma raccomandato/usato de-facto)
- **Transport:** TCP
- **Network:** IP



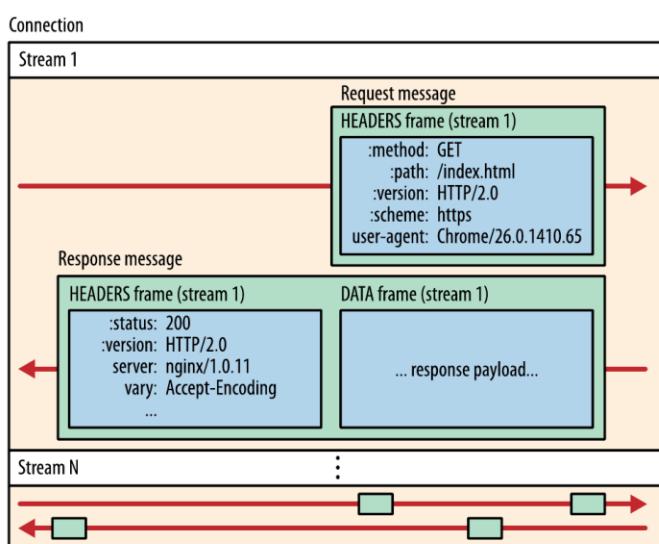
### Esempio di differenza

- HTTP 1.1: Un messaggio POST è un **blocco di testo leggibile** che include header (`Host`, `Content-Type`) e body (`{"msg": "hello"}`)
- HTTP 2.0: Lo stesso messaggio viene **diviso e convertito in binario**: un `HEADERS` frame per le intestazioni e un `DATA` frame per il contenuto JSON

## Concetti base: Stream, Messaggi e Frame

Per comprendere il funzionamento di HTTP/2, è essenziale distinguere queste entità:

- **Connessione:** Il canale TCP sottostante
- **Stream:** Un **flusso bidirezionale** di dati all'interno della connessione. Ogni stream ha un **identificatore univoco**
- **Messaggio:** Un **messaggio logico** HTTP (una Richiesta o una Risposta). Un messaggio è **composto da uno o più frame**
- **Frame:** È l'**unità atomica** (più piccola) di comunicazione. Ogni frame **trasporta un tipo specifico di dati** (Es.: intestazioni HTTP o il payload vero e proprio)

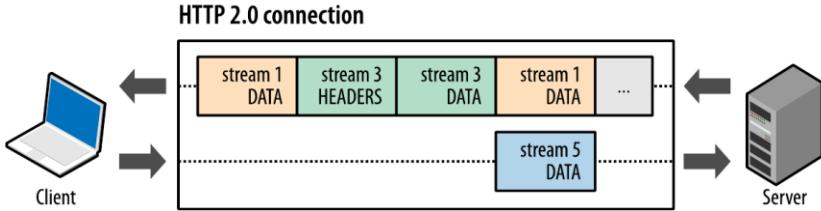


## Interleaving (Interlacciamento)

I frame appartenenti a stream diversi **possono** essere inviati in **modo disordinato** e **alternato** sulla connessione. Grazie all'identificatore di stream presente nell'intestazione di ogni frame, il **ricevente** è in grado di **riassemblarli nell'ordine corretto** per **ricostruire i messaggi** originali

## Multiplexing

Il multiplexing è la **caratteristica più potente** abilitata dal framing binario. In HTTP/1.1, le **richieste erano sequenziali** (o limitatamente parallele tramite più connessioni TCP). In HTTP/2, client e server possono scambiarsi **frame di stream diversi** contemporaneamente sulla stessa connessione



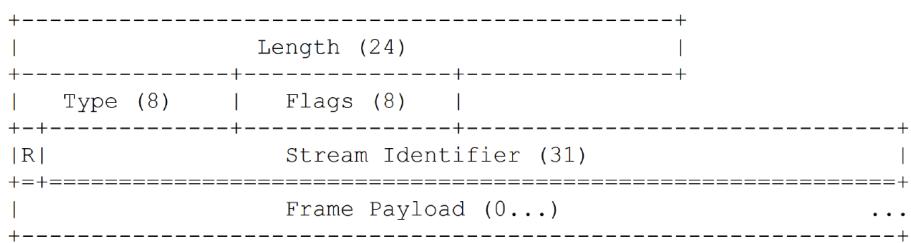
Es.: Immaginiamo una connessione attiva:

- Il Client sta inviando un frame DATA per lo stream 5
- Il Server, nello stesso momento, sta inviando una sequenza alternata di frame per rispondere allo stream 1 e allo stream 3
- Risultato: Ci sono tre stream paralleli "in volo" senza che nessuno blocchi gli altri

L'uso di una singola connessione multiplexata porta vantaggi significativi:

- **Parallelismo reale:** Richieste e risposte multiple vengono **gestite in parallelo senza bloccarsi a vicenda** (risolve il problema dell'Head-of-Line blocking a livello applicativo)
- **Efficienza di rete:** Si utilizza **una sola connessione TCP**, ottimizzando l'uso della **banda** e riducendo il carico di **handshake**
- **Eliminazione dei workaround:** Tecniche obsolete usate in HTTP/1.1 per simulare il parallelismo, come la concatenazione di file (unire tutti i CSS/JS in un file), gli image sprites o il domain sharding (usare sottodomini finti per aprire più connessioni), non sono più necessarie e anzi sconsigliate
- **Latenza ridotta: Caricamento delle pagine più veloce**

## Struttura del Frame



Ogni frame HTTP/2 ha una **struttura fissa** che permette un **parsing efficiente**. L'intestazione (header) del frame è composta dai seguenti campi:

- **Length** (24 bit): Indica la **lunghezza** del **payload** del frame (escludendo l'header di 9 byte)
- **Type** (8 bit): Specifica il **tipo** di **frame** (Es.: DATA, HEADERS, PING), determinandone la semantica e **come leggere il payload**
- **Flags** (8 bit): **Interruttori booleani** specifici per il **tipo** di frame (Es.: per indicare la fine di uno stream)
- **R** (1 bit): Campo **riservato** (attualmente deve essere 0)

- Stream Identifier (31 bit): Identifica a quale stream appartiene il frame. Se è 0, il frame riguarda l'intera connessione e non uno stream specifico
- Frame Payload: Il contenuto variabile del frame

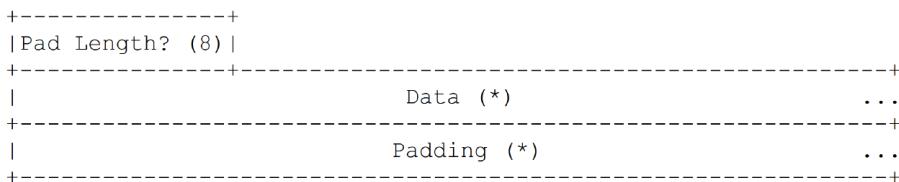
## Tipologie di Frame

HTTP/2 definisce diversi tipi di frame, ognuno identificato da un codice numerico:

- DATA (0): Trasporta il corpo del messaggio (payload)
- HEADERS (1): Trasporta le intestazioni HTTP (compresse)
- PRIORITY (2): Definisce la priorità di uno stream
- RST\_STREAM (3): Termina immediatamente uno stream (errore)
- SETTINGS (4): Parametri di configurazione della connessione
- PUSH\_PROMISE (5): Notifica al client che il server sta per inviare una risorsa (Server Push)
- PING (6): Misura il tempo di round-trip e verifica la connessione
- GOAWAY (7): Chiede la chiusura della connessione
- WINDOW\_UPDATE (8): Gestione del controllo di flusso
- CONTINUATION (9): Usato se le intestazioni sono troppo grandi per un solo frame HEADERS

## Analisi Specifica: Frame DATA e HEADERS

### Frame DATA (Tipo 0)



Trasporta dati arbitrari (Es.: l'HTML, un'immagine, un JSON). Nei flag importanti abbiamo:

- END\_STREAM (0x1): Indica che questo è l'ultimo frame per quello stream (chiusura stream)
- PADDED (0x8): Indica la presenza di padding

La struttura del payload può includere un campo Padding (per riempire dei byte nulli) ed è opzionale.

Il suo scopo è oscurare la dimensione reale del messaggio per migliorare la sicurezza contro attacchi di analisi del traffico

## Frame HEADERS (Tipo 1)

+-----+   Pad Length? (8)		
+-----+   E   Stream Dependency? (31)		
+-----+   Weight? (8)		
+-----+   Header Block Fragment (*) ...		
+-----+   Padding (*) ...		

Trasporta le **intestazioni** HTTP (metodo, status, path, ecc.). Nei flag importanti abbiamo:

- **END\_STREAM** (0x1): Se impostato, lo **stream si chiude dopo** queste intestazioni (Es.: una richiesta GET senza body)
- **END\_HEADERS** (0x4): Indica che questo frame contiene **l'intero blocco di intestazioni** (non ci sono frame CONTINUATION successivi)
- **PADDED** (0x8): **Presenza di padding**
- **PRIORITY** (0x20): Presenza di **informazioni sulla priorità** nel payload

Relativamente alla struttura del payload, oltre al frammento di intestazioni (compresso con **HPACK**) e all'eventuale padding, può contenere campi per la **Prioritizzazione (Stream Dependency e Weight)**, permettendo al client di dire al server quali risorse caricare prima

## Gestione dei Stream e Frame di Continuazione

In HTTP/2, i frame **HEADERS** e **PUSH\_PROMISE** trasportano **blocchi di intestazioni** che possono essere di **grandi dimensioni**. Se un blocco non entra in un singolo frame, vengono utilizzati i frame **CONTINUATION**:

- **CONTINUATION** Frame: Viene usato per **continuare** una **sequenza di frammenti** di intestazioni
- Flag **END\_HEADERS** (0x4): Indica che il frame corrente **contiene l'ultima parte** del blocco di **intestazioni**

Gli stream sono flussi bidirezionali identificati da un intero univoco:

- **ID Dispari**: Stream **avviati dal Client**
- **ID Pari**: Stream **avviati dal Server**
- **Consumo**: Una **coppia richiesta/risposta** "consuma" completamente uno stream. Gli ID non possono essere riutilizzati. Se si esauriscono (overflow), la connessione deve essere chiusa e riaperta

## Prioritizzazione degli Stream

+-----+   E   Stream Dependency (31)		
+-----+   Weight (8)		
+-----+		

Dato che tutte le risorse viaggiano sulla stessa connessione, è fondamentale dire al server quali sono **più importanti** (Es.: il CSS è più urgente di un'immagine di sfondo). HTTP/2 permette di costruire un **Albero di Priorità**.

Ogni stream può avere:

- **Peso** (Weight): Un **intero** da 1 a 256. Stream fratelli (allo stesso livello) ricevono **risorse** (banda/CPU) **proporzionalmente** al loro **peso**
- **Dipendenza** (Dependency): Uno stream può dipendere da un altro. Gli stream "figli" dovrebbero essere processati solo dopo che i "padri" hanno ricevuto le risorse

Questa struttura permette al client di guidare il server **nell'allocazione ottimale** di **CPU, memoria e banda**

## Gestione errori e controllo di flusso

### Frame RST\_STREAM

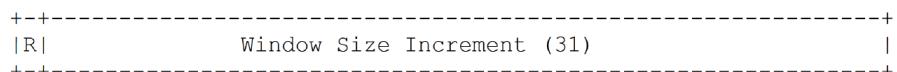


Permette di **terminare immediatamente** uno **stream** specifico **senza chiudere** l'intera **connessione TCP**. Utile, ad esempio, se l'utente annulla il caricamento di una singola immagine

### Controllo di Flusso

Impedisce che un **endpoint invii dati più velocemente** di quanto il **ricevente** possa **processarli**.

**Differenza con TCP:** Il controllo di flusso TCP lavora sulla connessione intera. HTTP/2 necessita di un **controllo granulare per singolo stream**.

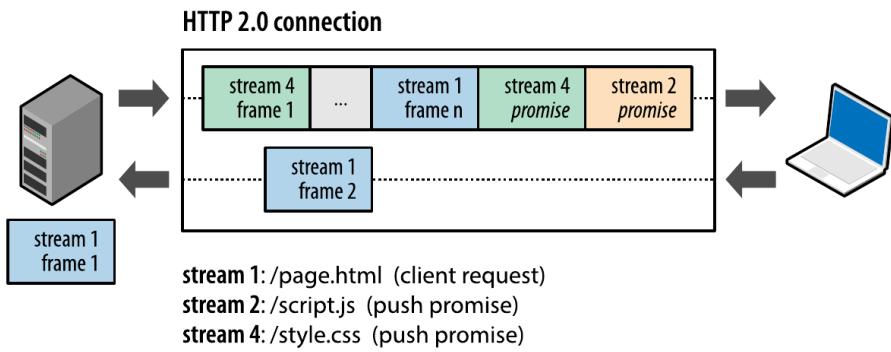


Il suo funzionamento è:

- Si applica **solo ai frame DATA**
- Funziona a **crediti** (finestra iniziale di 65.535 byte)
- Il ricevente **invia frame WINDOW\_UPDATE** per dire "sono pronto a ricevere altri X byte"

## Server Push

Questa è una delle novità più significative. Invece di aspettare che il client analizzi l'HTML e richieda le risorse collegate (JS, CSS, Immagini), il **server** può **inviarle proattivamente** (push) **anticipando la richiesta**



Funziona in questo modo:

- 1) Il client richiede `page.html` (Stream 1)
- 2) Il server sa che per visualizzare `page.html` servono `script.js` e `style.css`
- 3) Il server invia un frame `PUSH_PROMISE` al client (su Stream 1). Questo frame dice: "Sto per inviarti una risorsa che non hai chiesto, immagina di averla richiesta tu". Contiene gli header della "finta" richiesta
- 4) Il server inizia a inviare i dati delle risorse promesse su nuovi stream (Es.: Stream 2 e 4), interlacciandoli con la risposta principale

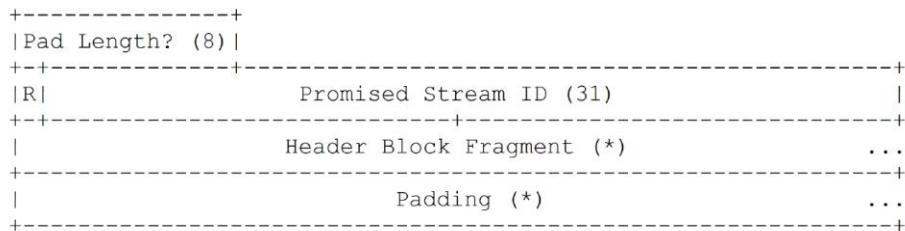
Nelle caratteristiche di una push abbiamo:

- Le **risorse pushate** possono essere **cacheate** dal client
- Possono essere **rifiutate** dal **client** (Es.: se le ha già in cache) tramite `RST_STREAM`
- Devono essere sempre **associate** a una **richiesta esplicita originale**

## Frame di controllo e gestione della connessione

Oltre ai frame per il trasporto dati (`DATA`) e intestazioni (`HEADERS`), HTTP/2 **definisce frame specifici per la gestione della connessione, del server push e della configurazione**

### Frame PUSH\_PROMISE (Tipo 5)



Utilizzato nel contesto del Server Push. Il **server** lo **invia** al **client** per "promettere" l'invio di una risorsa futura senza una richiesta esplicita.

Nella struttura del payload:

- **Promised Stream ID** (31 bit): L'identificativo del **nuovo stream** su cui il server invierà la risorsa (stream "riservato")
- **Header Block Fragment**: Contiene le **intestazioni** della **richiesta "simulata"** (come se il client avesse richiesto quella risorsa)

Il **padding** è **opzionale**, si usa per sicurezza

### Frame PING (Tipo 6)

+-----		+-----
	Opaque Data (64)	
+-----		+-----

Utilizzato per **verificare** se una **connessione inattiva** (idle) è ancora **funzionante** e per **misurare** il tempo di **Round-Trip** (RTT). Non ha flag nell'header.

Il **payload** contiene esattamente **8 byte** di dati **opachi** (privi di significato semantico per il protocollo, ma che devono essere **restituiti identici** nella risposta di PONG).

### Frame GOAWAY (Tipo 7)

+--		+--
R	Last-Stream-ID (31)	
+--		+--
	Error Code (32)	
+--		+--
	Additional Debug Data (*)	
+--		+--

Serve per **avviare la chiusura controllata** (**graceful shutdown**) della connessione, ovvero chiudere la connessione senza perdere le richieste già in volo.

Relativamente al payload:

- **Last-Stream-ID** (31 bit): Indica l'**ID dell'ultimo stream** che il mittente ha processato completamente. Gli stream con ID superiore a questo numero non sono stati elaborati e possono essere tentati su una nuova connessione
- **Error Code** (32 bit): **Motivo della chiusura**
- **Debug Data**: Dati aggiuntivi per il debug

### Frame SETTINGS (Tipo 4)

+-----		+-----
	Identifier (16)	
+-----		+-----
	Value (32)	
+-----		+-----

Utilizzato per **negoziare i parametri di configurazione** della connessione. Il mittente invia delle **preferenze**; il ricevente deve applicarle e rispondere con un frame **SETTINGS** con flag **ACK** (bit 0 impostato a 1) per confermare la ricezione.

Il payload è una sequenza di coppie Identificativo-Valore, i principali parametri configurabili sono:

- `SETTINGS_ENABLE_PUSH` (2): Per **abilitare/disabilitare il Server Push**
- `SETTINGS_MAX_CONCURRENT_STREAMS` (3): **Numero massimo di stream aperti simultaneamente**
- `SETTINGS_INITIAL_WINDOW_SIZE` (4): **Dimensione della finestra di controllo di flusso** (default 65.535 byte)
- `SETTINGS_MAX_FRAME_SIZE` (5): **Dimensione massima del payload di un frame** (default 16.384 byte)

## Compressione degli Header (HPACK)

In HTTP/1.1, gli header (`User-Agent`, `Cookie`, ecc.) venivano inviati come testo ripetitivo ad ogni richiesta, sprecando banda. HTTP/2 introduce **HPACK** (RFC 7541), un algoritmo di **compressione specifico per gli header**.

HPACK combina due meccanismi principali per l'efficienza:

- **Codifica di Huffman Statica:** I valori degli **header** vengono **compressi** utilizzando un **dizionario statico** predefinito (basato sulla frequenza d'uso dei caratteri negli header HTTP)
- **Tabelle di Indicizzazione** (Compression Context): Client e Server **mantengono** una "memoria" degli **header già visti**
  - Invece di inviare nuovamente User-Agent: Mozilla/5.0..., si invia solo un **indice numerico** che fa riferimento a quell'header nella tabella condivisa

### Esempio Pratico

- Richiesta #1: Invia tutti gli header completi (`GET`, `https`, `example.com`, `/resource`, `image/jpeg`). Questi vengono memorizzati nel contesto
- Richiesta #2: Per richiedere una nuova risorsa (`/new_resource`) sullo stesso sito, si inviano solo le differenze (il nuovo `:path`). Gli altri campi (`:method`, `:scheme`, `:host`, ecc.) sono impliciti perché già presenti nella tabella dinamica

## Negoziazione e upgrade del protocollo

Poiché non tutti i server supportano HTTP/2, i client devono poter **negoziare la versione** del protocollo. Il meccanismo cambia a seconda se si usa HTTP (in chiaro) o HTTPS (crittografato).

### URI "http" (Cleartext)

Il client non sa a priori se il server supporta HTTP/2. Usa il meccanismo **HTTP Upgrade**:

- 1) Il client **invia** una **richiesta** HTTP/1.1 con l'header:
  - a. **Upgrade:** h2c (indica HTTP/2 Cleartext)
  - b. **HTTP2-Settings:** Un payload Base64 contenente le impostazioni iniziali HTTP/2
- 2) Se il server supporta HTTP/2: Risponde con status `101 Switching Protocols` e passa al framing binario
- 3) Se il server NON supporta HTTP/2: Ignora l'header Upgrade e risponde normalmente in HTTP/1.1

## URI "https" (Encrypted)

Per **connessioni sicure**, non si usa l'header `Upgrade`. La **negoziazione** avviene durante **l'handshake** TLS utilizzando l'estensione ALPN (Application-Layer Protocol Negotiation).

- Il **client segnala il supporto** inviando **l'identificatore di protocollo "h2"**
- Questo è il **metodo de-facto utilizzato** dai **browser moderni**, che supportano HTTP/2 solo su connessioni cifrate (TLS)

# Apache

Apache HTTP Server è, insieme a Nginx, uno dei **server web** più popolari e utilizzati al mondo. Nato nel 1995 come evoluzione del server NCSA HTTPd (tramite una raccolta di "patches", da cui il nome), è gestito dalla Apache Software Foundation.

Tra le sue caratteristiche principali:

- **Licenza:** Apache 2.0 (gratuita, open source, compatibile con GPLv3)
- **Portabilità:** Funziona su sistemi UNIX-like e Windows grazie alla libreria APR (Apache Portable Runtime), che astrae le differenze tra i sistemi operativi
- **Modularità:** Funzionalità aggiuntive possono essere caricate dinamicamente
- **Concorrenza:** Gestisce un numero di client limitato solo dall'hardware

A gennaio 2024, Nginx detiene circa il 23% del mercato, seguito da Apache con il 21% e Cloudflare all'11%

## Architettura e Gestione dei Processi (MPM)

L'architettura di Apache si è evoluta per gestire meglio le risorse:

- **Process-based** (Apache 1.3): Creava copie (fork) del processo padre. Costoso in termini di risorse e context switching
- **Multi Processing Modules (MPM)** (Apache 2.0+): Astrae l'architettura di elaborazione delle richieste. Permette di scegliere tra:
  - **Modello a Processi:** Isolamento maggiore, stabilità (se un processo fallisce, gli altri continuano), ma alto consumo di risorse
  - **Modello a Thread:** I thread condividono memoria e codice all'interno di un processo. Sono più leggeri e scalabili, ma meno affidabili (un thread instabile può bloccare l'intero processo)

## Installazione e Gestione del Servizio

Su sistemi Debian/Ubuntu, i file principali sono organizzati come segue:

- Binario: /usr/sbin/apache2
- Configurazione: /etc/apache2
- Log: /var/log/apache2
- Directory Web: /var/www/html

L'amministratore può gestire il demone tramite **service** o **apache2ctl**:

- **Start:** sudo service apache2 start : Avvia il server (spesso come root per la porta 80, ma i processi figli girano come utente www-data per sicurezza)
- **Stop:** sudo service apache2 stop : Invia il segnale TERM, terminando immediatamente tutti i processi e le richieste in corso
- **Restart:** sudo service apache2 restart : Uccide i figli e riavvia il padre, rileggendo la configurazione

- **Reload** (Graceful Restart): `sudo service apache2 reload`: **Consiglia ai figli di terminare** dopo aver **completato la richiesta** corrente. Il padre sostituisce gradualmente i vecchi figli con nuovi processi aggiornati alla nuova configurazione, senza interrompere il servizio

## Configurazione

La **configurazione** di Apache avviene tramite **direttive inserite in file di testo**. Esistono tre livelli di configurazione:

- 1) **Build-time: Scelta dei moduli** in fase di **compilazione**
- 2) **Global Configuration: File principali (`httpd.conf` o `apache2.conf`) processati all'avvio**
- 3) **Local Configuration (`.htaccess`)**: **File distribuiti** nelle directory web, letti a ogni richiesta

## Sintassi e Contesti

Le **direttive** sono **processate riga per riga**. Alcune direttive sono valide solo in specifici **Contesti** (Container):

- **Global Context**: Applicato **all'intero server**
- **<VirtualHost>**: Applicato a uno **specifico sito virtuale**
- **<Directory> / <DirectoryMatch>**: Applica **regole** a una **directory** del **file system** e **sottodirectory**
- **<Files> / <FilesMatch>**: Applica **regole a specifici file**
- **<Location>**: Applica **regole** a uno **specifico URL** (non necessariamente legato al file system)

## Esempio di direttive globali

- **ServerRoot**: **Percorso base** dei **file di configurazione**
- **Timeout**: **Tempo di attesa** prima di fallire una richiesta
- **KeepAlive**: **Mantiene aperta** la **connessione TCP** per richieste multiple
- **MaxKeepAliveRequests**: **Limite richieste** per connessione

## Moduli

I **moduli estendono le funzionalità** (Es.: SSL, PHP, Cache). Possono essere **Statici** (compilati dentro) o **Loadable** (caricati dinamicamente con `LoadModule`).

Comando per listare i moduli: `apache2ctl -M`

## Controllo Accessi e Sicurezza

### Contesto Directory e Opzioni

Il blocco **<Directory>** è **fondamentale** per la **sicurezza**:

- **Options**: **Definisce le feature** per la **directory** (Es.: `Indexes` per mostrare l'elenco file se manca `index.html`). Le opzioni possono essere **unite** con `+` o `-`
- **AllowOverride**: Determina quali **direttive** in un file `.htaccess` possono **sovrascrivere** la **configurazione globale**

## Controllo Accessi (Differenza Apache 2.2 vs 2.4)

È importante notare il cambio di sintassi tra le versioni:

- Apache 2.2: Usava `Order, Allow, Deny`
  - Es.: `Order allow,deny` (default `deny`, processa `allow` poi `deny`)
- Apache 2.4: Usa la direttiva `Require`
  - Es.: `Require all granted` (permetti a tutti) o `Require all denied`
  - Es.: `Require host example.org` o `Require ip 192.168.1.1`

## File .htaccess

Permette **configurazioni decentralizzate** per directory:

- Utile per **hosting condivisi** dove gli utenti non hanno accesso root
- **Calo di performance** (il server cerca il file in ogni directory del percorso) e **rischi di sicurezza**
- Di default, Apache blocca l'accesso via browser ai file che iniziano con `.ht` (come `.htaccess` o `.htpasswd`)

## Autenticazione (Basic Auth)

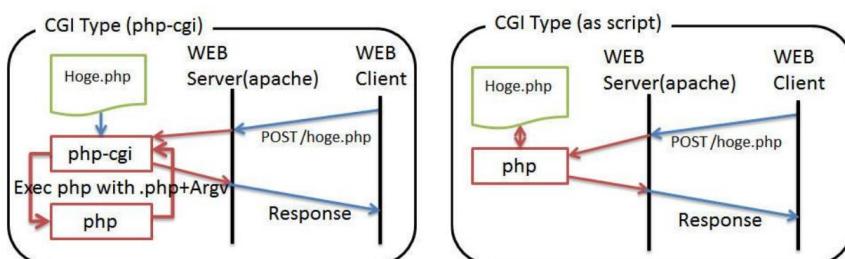
Per proteggere una directory con password:

Es.:

```
AuthType Basic  
AuthName "Area Riservata"  
AuthUserFile /etc/apache2/.htpasswd  
Require valid-user
```

NB.: `AuthType Basic` invia password in Base64 (non criptate), quindi è **sicuro solo su HTTPS**

## Contenuti Dinamici: CGI vs PHP



## CGI (Common Gateway Interface)

**Metodo classico.** Il server **avvia un programma esterno** per ogni richiesta.

- Direttiva: `ScriptAlias /cgi-bin/ "/path/to/scripts/"`
- Inefficiente, crea un nuovo processo per ogni esecuzione

## Integrazione PHP

- 1) `mod_php`: L'interprete PHP è incorporato nel processo Apache
  - Molto veloce, non crea processi esterni
  - Aumenta il consumo di RAM di ogni worker Apache. Ideale per CMS come WordPress
- 2) `FastCGI (php-cgi)`: Esegue script tramite un interprete esterno. Separa l'esecuzione dal server web

## Virtual Hosting

Permette di ospitare più siti sullo stesso server:

- **IP-based**: Un IP diverso per ogni sito
- **Name-based**: Stesso IP, siti distinti in base all'header `Host` della richiesta HTTP (metodo più comune)

Es.:

```
<VirtualHost *:80>
    ServerName www.example.com
    DocumentRoot /var/www/example
</VirtualHost>
```

## Funzionalità Avanzate

### Redirect

Mappa un vecchio URL su uno nuovo.

Es.: `Redirect 301 /old /new` (indica al client di aggiornare i bookmark).

### Caching

Apache gestisce una cache "a tre stati" (Fresh, Stale, Non-existent) per migliorare le performance.

- **Moduli**: `mod_cache` (gestore), `mod_cache_disk` (storage su disco)
- **Header**: `mod_expires` imposta gli header `Expires` e `Cache-Control`. `FileETag` gestisce la validazione dei file statici

### Proxying

Apache può agire come proxy tramite `mod_proxy`:

- 1) **Forward Proxy**: Agisce per conto dei **client interni** per accedere a internet. Richiede `ProxyRequests On`
- 2) **Reverse Proxy / Gateway**: Agisce per conto dei **server backend**, spesso come **Load Balancer**
  - a. Direttiva: `ProxyPass` e `ProxyPassReverse`

- b. **Load Balancing:** Si definisce un gruppo `<Proxy balancer://...>` con i membri `(BalancerMember)` e l'algoritmo di bilanciamento

## SSL/TLS

Il modulo `mod_ssl` interfaccia Apache con OpenSSL per supportare HTTPS (porta 443)

Una configurazione di base è:

```
<VirtualHost *:443>
    SSLEngine on
    SSLCertificateFile "/path/server.crt"
    SSLCertificateKeyFile "/path/server.key"
</VirtualHost>
```

Relativamente ai certificati, questi possono essere:

- **Self-signed:** Generati autonomamente. Utili per test, ma generano avvisi nel browser perché non validati da una CA terza
- **Real CA:** Richiedono la **creazione** di un **CSR** (Certificate Signing Request) da inviare a un'autorità (Es.: Verisign, Let's Encrypt) per ottenere un **certificato fidato**

# PHP

PHP è un **linguaggio di scripting** lato server (server-side), open source e ampiamente utilizzato, progettato per la **creazione di pagine Web** dinamiche e interattive.

## Concetti Fondamentali:

- **Esecuzione lato server:** Gli script PHP vengono **eseguiti dal server Web**. Il **risultato** di questa esecuzione (il codice generato) viene **invitato al browser** del client esclusivamente come HTML semplice
- **Integrazione:** Un file PHP può **contenere codice** PHP intervallato a HTML, CSS e JavaScript
- **Compatibilità:** PHP è supportato da vari sistemi operativi (Windows, Linux, macOS) e compatibile con i server Web più diffusi come Apache e Nginx
- **Storia:** Nato nel 1995 come "Personal Home Page Tools", divenne l'acronimo ricorsivo "PHP: Hypertext Preprocessor" nel 1998

## Sintassi e Struttura di Base

### Delimitatori e Output

Il codice PHP deve essere sempre **racchiuso** all'interno di specifici **delimitatori**:

```
<?php  
    // Il codice PHP va qui  
?>
```

"HelloWorld.php"

```
<!DOCTYPE html>  
<html>  
<body>  
    <h1>My first PHP page</h1>  
  
    <?php  
        echo "Hello World!";  
    ?>  
</body>  
</html>
```

Plain HTML

```
<!DOCTYPE html>  
<html>  
<body>  
    <h1>My first PHP page</h1>  
  
    Hello World!  
  
</body>  
</html>
```



Per l'output sul browser si usano principalmente **due costrutti**:

- `echo`: Utilizzato per **stampare dati**. Non ha valore di ritorno ed è leggermente più **veloce**
- `print`: Utilizzato per **stampare dati**. Ritorna sempre 1

Es.:

```
<?php  
    echo "<h1>Benvenuto nella mia pagina dinamica!</h1>";  
    print "Questo testo è stato generato dal server.";  
?>
```

### Sensibilità alle Maiuscole (Case Sensitivity)

- **Variabili:** Sono **sensibili alle maiuscole** (Case-sensitive). Es.: \$nome, \$Nome e \$NOME sono tre variabili distinte
- **Parole chiave, Classi, Funzioni:** Non sono **sensibili alle maiuscole** (Case-insensitive). Es.: IF, If e if sono trattati allo stesso modo

## Variabili e Tipi di Dato

### Variabili

Le variabili in PHP sono dichiarate utilizzando il prefisso dollaro (\$):

Tipo di Scope	Definizione
<b>Global</b>	Variabile dichiarata al di fuori di qualsiasi funzione. Accessibile solo all'esterno della funzione.
<b>Local</b>	Variabile dichiarata all'interno di una funzione. Accessibile solo all'interno di quella funzione.
<b>global</b>	Keyword usata all'interno di una funzione per accedere a una variabile definita nello scope globale <sup>11</sup> .

### Tipi di Dato Principali

PHP supporta diversi tipi di dati fondamentali:

- **String:** Sequenza di caratteri, racchiusa tra virgolette singole o doppie.
- **Integer (int):** Numero intero senza punto decimale
- **Float (double):** Numero con punto decimale o in forma esponenziale
- **Boolean:** Può assumere solo i valori TRUE o FALSE
- **Array:** Consente di memorizzare più valori in una singola variabile. Può essere numerico, associativo o multidimensionale
- **Object:** Un'istanza di una classe. Richiede la previa definizione di una classe

Es.: (Array associativo):

```
<?php  
$persona = array(  
    "nome" => "Mario",
```

```

"eta" => 30,
"citta" => "Bari"
);
echo $persona["nome"]; // Output: Mario
?>

```

## Operatori e Strutture di Controllo

### Operatori

PHP fornisce una gamma completa di operatori, tra cui:

- **Aritmetici:** +, -, \*, /, % (modulo), \*\* (potenza)
- Di **Assegnazione:** =, +=, -=, \*= (per abbreviare operazioni e assegnazione)
- Di **Confronto:** == (uguale), != (diverso), > (maggiore), < (minore), === (identico, stesso valore E stesso tipo)
- **Logici:** && (AND), || (OR), ! (NOT)
- **Incremento/Decremento:** ++\$a (pre-incremento), \$a++ (post-incremento)

### Strutture Condizionali

- **if...else / if...elseif...else:** Esegue un blocco di codice solo se una condizione è vera
- **switch:** Alternativa pulita a lunghe catene di **if...elseif** per eseguire codice in base a condizioni multiple sulla stessa variabile

### Cicli (Loops)

I cicli permettono di eseguire un blocco di codice ripetutamente:

- **while:** Ripete il blocco finché la condizione è vera
- **do...while:** Esegue il blocco almeno una volta, poi ripete finché la condizione è vera
- **for:** Usato quando si conosce in anticipo il numero di iterazioni
- **foreach:** Il ciclo più comune per gli array, itera su ogni elemento di un array

Es.:

```

<?php
$colori = array("rosso", "verde", "blu");
foreach ($colori as $colore) {
    echo $colore . "<br>";
}
?>

```

## Funzioni

Le funzioni consentono di raggruppare istruzioni per il riutilizzo del codice.

Es.: Funzione con argomento di default

```
<?php  
    // Se $b non è specificato, usa il valore di default 5  
    function somma($a, $b = 5) {  
        return $a + $b;  
    }  
  
    echo "Somma (10, 2): " . somma(10, 2) . "<br>";    // Output: 12  
    echo "Somma (7): " . somma(7) . "<br>";            // Output: 12 (7 +  
5)  
?>
```

## Gestione dei Form (Form Handling)

PHP **gestisce** i **dati** inviati da un **form** HTML utilizzando due **superglobali** (variabili integrate e sempre accessibili):

- **`$_GET`: Array associativo** che contiene i dati inviati tramite il metodo HTTP **GET**. I dati vengono **accodati** all'URL (visibili all'utente)
- **`$_POST`: Array associativo** che contiene i dati inviati tramite il metodo HTTP **POST**. I dati vengono **invia**ti nel corpo della richiesta HTTP (non visibili nell'URL)

In entrambi i casi, la chiave dell'array corrisponde all'attributo name del campo del form, e il valore è l'input dell'utente.

Es.: Se in un form hai `<input type="text" name="nomeUtente">`, in PHP recupera il valore con:

```
<?php  
    $nome = $_POST["nomeUtente"];  
    echo "Ciao, " . $nome;  
?>
```

## Validazione dei Form: Server-side vs. Client-side

Tipo di Validazione	Vantaggi	Svantaggi e Dettagli
Client-side (HTML5 + JS)	Migliore <i>User Experience</i> , risposta immediata, basso carico sul server <sup>26</sup> .	<b>Non sicura.</b> Può essere facilmente bypassata da utenti malintenzionati. Non sostituisce la validazione lato server.

<b>Server-side (PHP)</b>	<b>Essenziale per la sicurezza.</b> Protegge il server da input pericolosi o dannosi <sup>27</sup> .	Più lento (richiede un round-trip di rete). L'esito viene restituito tramite una nuova pagina Web dinamica <sup>28</sup> .
--------------------------	---	--

## Gestione dello Stato e Persistenza

### PHP Cookies

Un Cookie è un piccolo **file di testo memorizzato** sul **computer** dell'utente (client) che il server invia ad ogni successiva richiesta al dominio

- **Scopo:** **Memorizzare informazioni sull'utente** (Es.: preferenze, identificativi di sessione, carrelli acquisti) per un certo periodo di tempo
- **Creazione/Modifica:** Si usa la funzione `setcookie()`
  - Sintassi: `setcookie(name, value, expire, path, domain, secure, httponly);`
  - Esempio: `setcookie("utente", "MarioRossi", time() + (86400 * 30), "/" );` (Crea un cookie valido per 30 giorni per l'intero dominio)
- **Accesso:** Il valore del cookie viene **letto** tramite la **superglobale** `$_COOKIE`
- **Eliminazione:** Si imposta la **data di scadenza** del cookie nel passato (Es.: `time() - 3600`)

### PHP Sessions

Le **Sessions** sono il **metodo preferito** per **memorizzare informazioni** utente attraverso le **pagine, mantenendole sul lato server**:

- **Scopo:** **Memorizzare dati sensibili o grandi quantità di dati in modo più sicuro e persistente rispetto ai cookies**
- Funzionamento:
  - Viene **avviata la sessione** con `session_start()`
  - PHP crea un **ID di sessione univoco**
  - Questo **ID** viene **invia**to al **client** (solitamente tramite un cookie di sessione temporaneo)
  - Il **server memorizza i dati della sessione** (variabili) in un file temporaneo utilizzando l'**ID** come chiave
- **Variabili di Sessione:** Sono **memorizzate** nella **superglobale** `$_SESSION`

Es.:

```
<?php
    session_start();
    $_SESSION["colore_preferito"] = "blu";
    echo "La sessione è stata impostata.";
```

?>

## Riutilizzo del Codice (Include & Require)

PHP fornisce **costrutti** per includere il **contenuto** di un **file** all'interno di un altro, **facilitando la gestione di header, footer o barre laterali comuni**

Costrutto	Descrizione	Gestione Errore
include	Include e valuta il file specificato.	Genera un <b>Warning</b> (l'esecuzione dello script continua).
require	Include e valuta il file specificato.	Genera un <b>Fatal Error</b> (l'esecuzione dello script si interrompe).
include_once	Uguale a include, ma assicura che il file venga incluso al massimo una volta.	Utile per prevenire problemi di ridefinizione di funzioni o classi.

## Connessione a Database (MySQLi)

L'interazione con database relazionali è fondamentale per le applicazioni web dinamiche. PHP offre diverse API, tra cui MySQLi (MySQL Improved Extension)

### Interfaccia MySQLi

MySQLi supporta due approcci:

- **Procedurale:** Utilizza funzioni con il prefisso `mysqli_` e richiede che la **connessione** sia passata come **argomento**
  - Es.: `$conn = mysqli_connect($server, $user, $pass, $db);`
- **Orientato agli Oggetti (OO):** Crea un **oggetto connessione** e invoca metodi su tale oggetto.  
**Questo è l'approccio raccomandato** per codice più pulito e manutenibile
  - Es.: `$conn = new mysqli($server, $user, $pass, $db);`

### Esempio di Query

## Architetture di Sviluppo Web

### Modello Model-View-Controller (MVC)

Il pattern MVC separa l'applicazione in tre componenti logici interconnessi, **facilitando la modularità, lo sviluppo parallelo e la manutenibilità**

Componente	Ruolo	Descrizione
Model (Modello)	Gestione Dati	Contiene la logica di accesso e manipolazione dei dati (Es.: interazione con il DB). <b>Non</b> contiene elementi di presentazione.
View (Vista)	Presentazione	Si occupa dell'interfaccia utente (HTML, CSS). Visualizza i dati forniti dal

		Model. <b>Non</b> contiene la logica di business.
<b>Controller</b>	<b>Controllo/Logica</b>	Riceve le richieste utente, interroga il Model per ottenere/modificare i dati e seleziona la View appropriata per la risposta. Fa da mediatore.

#### Flusso Esempio (Richiesta Utente):

- 1) L'utente invia una richiesta HTTP (Es.: /utenti/list)
- 2) Il Controller intercetta la richiesta
- 3) Il Controller chiede al Model di recuperare l'elenco degli utenti dal database
- 4) Il Model restituisce i dati grezzi
- 5) Il Controller passa i dati alla View listaUtenti.php
- 6) La View genera l'HTML finale
- 7) Il Controller invia la risposta HTML al browser

#### Framework PHP

I **Framework** sono **strumenti** che **forniscono** una **struttura** (spesso basata su MVC), **librerie** e **convenzioni** per **accelerare** e **standardizzare** lo **sviluppo**:

- **Codice più pulito, meno bug** (grazie alle funzionalità di sicurezza integrate), **riutilizzo** del codice, e **mantenibilità**
- Es.:
  - Symfony: Framework molto strutturato e modulare, basato sul concetto di bundle (pacchetto MVC)
  - Zend Framework: (Ora Laminas Project) Una collezione di componenti
  - CakePHP: Framework più semplice, con regole di base chiare

#### Conclusioni sul Ruolo di PHP

PHP rimane un **pilastro** del Web Intelligence, specialmente nelle **fasi** di **backend** e **interazione dinamica** grazie a:

- **Generazione di Contenuti:** Capacità di creare HTML on-the-fly in base a dati, input utente o logiche di business
- **Integrazione Dati: Gestione nativa ed efficiente** delle **connessioni** a database (MySQLi, PDO)
- **Gestione dello Stato: Meccanismi robusti** come Sessioni e Cookies per mantenere il **contesto utente** in un ambiente stateless

Tutto il codice PHP viene eseguito sul server; il client riceve solo l'output HTML finale, mantenendo la logica di business protetta

# Web Accessibility

## Definizioni fondamentali

### Disabilità

La disabilità non è una caratteristica intrinseca della persona, ma un **concetto** in evoluzione. Secondo la Convenzione ONU sui diritti delle persone con disabilità (2006), essa risulta **dall'interazione** tra **persone con menomazioni e barriere comportamentali o ambientali** che ne **impediscono** la piena ed effettiva **partecipazione alla società** su base di **uguaglianza** con gli altri.

### Accessibilità

L'accessibilità è la misura in cui prodotti, sistemi, servizi e ambienti possono essere utilizzati da persone con la più ampia gamma di esigenze, caratteristiche e capacità per raggiungere obiettivi identificati in specifici contesti d'uso. Include l'accesso all'ambiente fisico, ai trasporti, e crucialmente alle tecnologie dell'informazione e della comunicazione (ICT)

### Tecnologia assistiva

Si definisce come **hardware** o **software aggiunto** o incorporato in un sistema ICT che ne **aumenta l'accessibilità** per un individuo. In questa categoria rientrano gli **screen reader**, le **tastiere alternative** e i **software di scansione**. Possono essere software all'interno dello stesso sistema operativo, estensioni/plugin del browser o script inclusi direttamente nel sito web

## Approcci al design e normative

La progettazione per l'accessibilità si è evoluta nel tempo attraverso diversi paradigmi:

- **Barrier-free design:** Focalizzato sul **rispetto** delle **normative** (Es.: abbattimento barriere architettoniche)
- **Universal Design:** Introdotto da Ronald L. Mace. Abbandona l'idea della "persona standard" per considerare **contemporaneamente i bisogni di tutti gli utenti**
- **Design for All:** Progettazione per la **diversità umana, l'inclusione sociale e l'uguaglianza**

## 7 Principi dello Universal Design

Per creare prodotti universalmente accessibili, si seguono questi principi:

- 1) **Uso equo: Utile e vendibile** a persone con diverse abilità
- 2) **Flessibilità d'uso:** Si adatta a un'ampia gamma di preferenze e abilità individuali
- 3) **Semplice e intuitivo:** Facile da capire, indipendentemente dall'esperienza o dalle conoscenze dell'utente
- 4) **Informazione percettibile:** Comunica le informazioni necessarie in modo efficace all'utente (indipendentemente dalle condizioni ambientali o dalle capacità sensoriali)
- 5) **Tolleranza all'errore:** Minimizza i rischi e le conseguenze di azioni accidentali
- 6) **Basso sforzo fisico:** Può essere usato in modo efficiente e comodo
- 7) **Dimensioni e spazio per l'approccio e l'uso:** Spazio adeguato per l'accesso e la manipolazione

Riguardo al contesto normativo:

- **USA:** Section 508 del Rehabilitation Act (1998)
- **UE:** Direttiva 2102/2016 (accessibilità siti web e app mobili degli enti pubblici)
- **Italia:** Legge 4/2004 ("Legge Stanca") e linee guida AgID

## Accessibilità web

Tim Berners-Lee ha affermato: "Il potere del Web sta nella sua universalità. L'accesso da parte di chiunque, indipendentemente dalla disabilità, è un aspetto essenziale".

L'obiettivo è permettere alle persone con disabilità di percepire, capire, navigare, interagire e contribuire al Web.

L'accessibilità non aiuta solo chi ha disabilità permanenti (visive, uditive, cognitive, motorie), ma **porta benefici a tutti gli utenti**, inclusi:

- **Disabilità temporanee:** Es.: un braccio rotto o occhiali persi
- **Limitazioni situazionali:** Es.: luce solare intensa sullo schermo o ambiente silenzioso dove non si può ascoltare audio
- **Limitazioni tecnologiche:** Connessioni lente o dispositivi con schermi piccoli (smartwatch, mobile)
- **Invecchiamento:** Cambiamento delle abilità dovuto all'età

## Ecosistemi

L'accessibilità **non dipende solo** dallo **sviluppatore** del sito, ma da un **ecosistema di componenti interconnessi**. Le componenti principali sono:

- **Contenuto:** Informazioni (testo, immagini, suoni) e codice/markup
- **User Agents:** Browser, media player
- **Tecnologie Assistive:** Screen reader, tastiere alternative
- **Utenti:** Con le loro conoscenze, esperienze e strategie di adattamento
- **Sviluppatori:** Designer, programmatore, autori
- **Authoring Tools:** Software per creare siti web (CMS, editor di codice)
- **Evaluation Tools:** Strumenti di validazione (HTML, CSS, accessibilità)

Esiste un'interdipendenza critica tra i componenti. Se una funzionalità di accessibilità non è supportata da un componente, il ciclo si rompe:

- Se i **browser supportano** una **feature**, gli **utenti la richiederanno** e gli **sviluppatori la implementeranno**
- Se gli **authoring tools rendono facile** implementare una **feature**, gli **sviluppatori la useranno**
- Se un anello manca (Es.: i browser non supportano bene una feature), gli **utenti o gli sviluppatori** devono ricorrere a **workaround** (soluzioni alternative) **costosi** e spesso **inefficaci**

## Standard e linee guida (W3C WAI)

Il W3C Web Accessibility Initiative (WAI) definisce gli standard tecnici:

- **WCAG** (Web Content Accessibility Guidelines): Per i **contenuti web** (usato da sviluppatori e tool di valutazione)
- **ATAG** (Authoring Tool Accessibility Guidelines): Per gli **strumenti di creazione contenuti**
- **UAAG** (User Agent Accessibility Guidelines): Per **browser** e **media player**
- **ARIA** (Accessible Rich Internet Applications): **Specifica tecnica** per rendere accessibili **applicazioni web dinamiche** (AJAX, JavaScript)

## WCAG (Web Content Accessibility Guidelines)

Le **WCAG** sono lo **standard di riferimento principale**. La versione attuale stabile è la 2.2 (2023), con la 3.0 in bozza

### Architettura

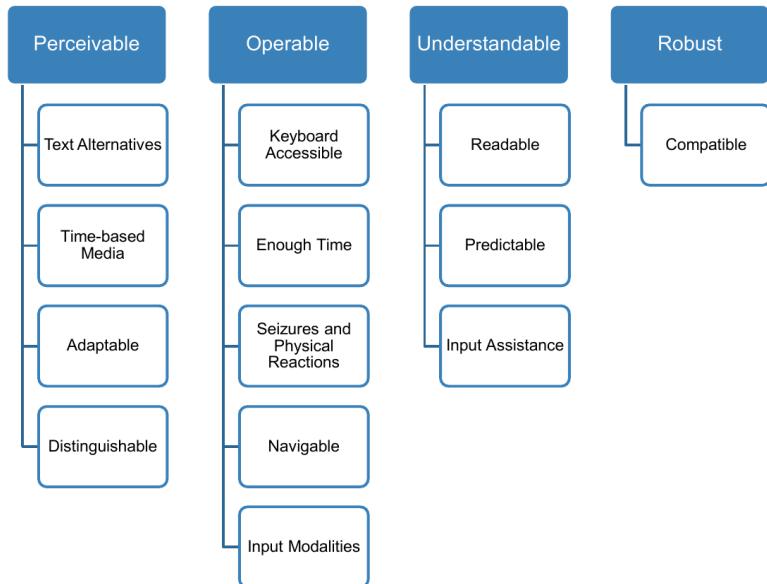
Le WCAG sono strutturate gerarchicamente in:

- 1) **4 Principi: Le fondamenta dell'accessibilità**
- 2) **13 Linee Guida: Obiettivi generali per gli autori**
- 3) **Criteri di Successo: Testabili su tre livelli di conformità** (A = base, AA = standard richiesto dalle normative, AAA = massimo)

### Principi

Il contenuto Web deve essere:

- 1) **Perceivable** (Percepibile): Le informazioni e l'interfaccia **non** devono essere **invisibili a tutti i sensi dell'utente** (Es.: testo alternativo per chi non vede)
- 2) **Operable** (Utilizzabile): L'utente deve poter **operare l'interfaccia** (Es.: navigazione da tastiera)
- 3) **Understandable** (Comprensibile): Le **informazioni** e le **operazioni** devono essere **chiare** (Es.: linguaggio semplice, prevedibilità)
- 4) **Robust** (Robusto): Il **contenuto** deve essere **interpretabile affidabilmente** da una **vasta gamma di user agent**, incluse le tecnologie assistive



## Percepibile

### *Alternative testuali*

Tutto il **contenuto non testuale** (immagini, grafici) deve avere **un'alternativa testuale** che serva allo stesso scopo. Es.: Controlli di input, CAPTCHA, decorazioni pure

### *Media visivi e sonori*

Bisogna fornire **alternative** per i **media preregistrati** o in **diretta**:

- **Sottotitoli** (Captions)
- **Audiodescrizioni** (per i video)
- **Lingua dei segni** (per audio e video)

### *Adattabile (Adaptable)*

Il **contenuto** deve poter essere **presentato in modi diversi senza perdere informazioni** o struttura:

- La **sequenza di lettura** deve essere **determinabile** via software
- Non basarsi solo su **caratteristiche sensoriali** (Es.: "clicca il pulsante rosso" o "il pulsante tondo") per dare istruzioni
- **Supportare l'orientamento** sia **verticale** che **orizzontale** del display
- Lo **scopo** dei **campi** di input deve essere **determinabile** via software

### *Distinguibile (Distinguishable)*

Facilitare la **visione** e l'**ascolto** dei **contenuti**, separando il primo piano dallo sfondo:

- **Colore:** Non usare il **colore** come unico mezzo per **veicolare informazioni**
- **Audio:** Meccanismo per **fermare** o **regolare** il **volume** dell'audio che parte in automatico
- **Contrasto:** Rapporto di contrasto sufficiente tra testo e sfondo

- **Ridimensionamento:** Il testo deve poter essere **ingrandito** fino al **200% senza perdita di contenuto**
- **Testo vs Immagini:** Usare **testo vero invece di immagini** di testo (salvo loghi)
- **Spaziatura:** Rispettare **specifiche minime** di spaziatura del testo

## Utilizzabile

### *Accessibilità da tastiera (Keyboard Accessible)*

Tutte le **funzionalità** devono essere **disponibili** tramite **tastiera**, senza richiedere tempi specifici per la pressione dei tasti. Il focus non deve rimanere intrappolato in un componente (keyboard trap)

### *Tempo sufficiente (Enough Time)*

Gli utenti devono avere **abbastanza tempo** per **leggere e usare il contenuto**:

- Se ci sono **limiti** di tempo, devono poter essere **disattivati o estesi**
- **Contenuti in movimento, lampeggianti** o che si aggiornano automaticamente devono poter essere **messi in pausa o nascosti**
- Le **sessioni autenticate** devono permettere la **continuazione senza perdita di dati** dopo la ri-autenticazione

### *Convulsioni e reazioni fisiche (Seizures)*

Le pagine non devono contenere nulla che **lampeggi** più di **tre volte al secondo** (per evitare crisi epilettiche). Le **animazioni** di movimento innescate dall'interazione devono poter essere **disabilitate**

## *Navigabile (Navigable)*

Fornire modi per **aiutare gli utenti a navigare**, trovare contenuti e determinare dove si trovano:

- Meccanismi per **saltare blocchi** di contenuto **ripetuti** (Es.: "Skip to content")
- **Titoli di pagina descrittivi**
- L'**ordine del focus** deve preservare il **significato e l'operabilità**
- Lo **scopo** di un **link** deve essere **comprendibile dal testo del link stesso**
- **Indicatori visivi del focus** della **tastiera** ben visibili

## *Modalità di Input (Input Modalities)*

**Facilitare l'uso** attraverso **input diversi** dalla tastiera (Es.: touch, puntatori):

- **Target tattili** sufficientemente **grandi** (almeno 44x44 pixel CSS)
- **Gesture complesse** (multipoint) devono avere **alternative semplici**
- **Evitare attivazioni accidentali** al **tocco** (gestione eventi down/up)

## Comprensibile

### *Leggibile (Readable)*

Il contenuto testuale deve essere **leggibile**:

- La **lingua** della pagina (Es.: lang="it") deve essere **determinabile via software**
- **Spiegare abbreviazioni, gergo o parole** con **pronuncia ambigua**
- Fornire **versioni del testo semplificate** per chi ha bassi livelli di istruzione

## *Prevedibile (Predictable)*

Le pagine devono operare in modi **prevedibili**:

- Il **cambio di focus** non deve causare **cambi di contesto inaspettati**
- La **navigazione** deve essere **coerente** tra le diverse **pagine**
- I **cambi di contesto** devono avvenire solo su **richiesta esplicita** dell'utente

## *Assistenza all'input (Input Assistance)*

Aiutare gli utenti a evitare e correggere errori:

- **Identificare e descrivere gli errori di input in testo**
- **Fornire etichette (labels) e istruzioni chiare**
- Per **transazioni legali/finanziarie: permettere la reversibilità, il controllo degli errori e la conferma** prima dell'invio finale

## Robusto

## *Compatibile (Compatible)*

Bisogna **massimizzare la compatibilità**:

- **Markup corretto:** Elementi con tag di apertura/chiusura **completi, annidamento corretto, ID univoci e assenza di attributi duplicati** (parsing corretto)
- **Componenti UI:** Per ogni componente (Es.: form, buttoni), **nome, ruolo e valore** devono essere **determinabili** via software e le **modifiche di stato** devono essere **notificate** alle tecnologie assistive

## UAAG (User Agent Accessibility Guidelines)

Le UAAG si rivolgono agli **sviluppatori di browser, media player e tecnologie assistive**. L'obiettivo è rendere il **software** che fruisce del web **accessibile**. L'architettura è simile alle WCAG (4 principi, linee guida, livelli A/AA/AAA).

Alcuni esempi di requisiti UAAG:

- Permettere la **configurazione del testo** (zoom) e del **volume**
- Garantire l'**accesso completo** da **tastiera** all'interfaccia del browser
- **Comunicare** correttamente con le **tecnologie assistive** (API di accessibilità)

## ATAG (Authoring Tool Accessibility Guidelines)

Le ATAG si rivolgono agli **sviluppatori di strumenti di creazione** (CMS come WordPress, editor di codice, software per corsi online). Si dividono in due parti:

### *Parte A: Interfaccia dello strumento*

L'**interfaccia dell'editor** stesso deve essere **accessibile**. Un autore con disabilità deve poter usare lo strumento per creare contenuti

Es.: editing accessibile da tastiera, anteprime accessibili, prevenzione di flash/convulsioni nell'interfaccia

## *Parte B: Supporto alla produzione di contenuti accessibili*

Lo strumento deve aiutare l'autore a generare output conforme alle WCAG:

- **Processi automatici:** Il codice generato **automaticamente** deve essere **accessibile**
- **Supporto all'autore: Guidare l'autore** (Es.: chiedere il testo alternativo quando si carica un'immagine), **fornire template accessibili e strumenti di controllo** (checker) integrati

## **WAI-ARIA (Accessible Rich Internet Applications)**

Le specifiche WAI-ARIA (o semplicemente ARIA) sono cruciali per le moderne applicazioni web dinamiche (basate su AJAX/JavaScript). L'HTML standard, essendo **statico**, spesso **non riesce a comunicare alle tecnologie assistive** (come gli screen reader) cosa sta succedendo in un'applicazione complessa (Es.: un widget, un menu a tendina custom, aggiornamenti live)

### **Modello "Contract"**

ARIA agisce come un **ponte semantico**. Il browser **espone** alle tecnologie assistive (AT) un **Accessibility Tree** (simile al DOM) arricchito da **informazioni semantiche**:

- 1) **Ruolo (Role):** Cos'è questo elemento? (Es.: `role="slider", role="dialog"`). Definisce il tipo di widget
- 2) **Stato (State):** In che condizione si trova? Cambia dinamicamente (Es.: `aria-checked="true", aria-expanded="false"`)
- 3) **Proprietà (Property):** Attributi essenziali (Es.: `aria-required="true", aria-label="..."`). Hanno tutte il prefisso `aria-`

## **Mapping delle API**

Gli attributi ARIA e HTML vengono **mappati dal browser** sulle API di accessibilità native del sistema operativo (come Microsoft Active Accessibility su Windows o ATK su Linux/GNOME), permettendo allo screen reader di **"leggere"** l'interfaccia web come se fosse **un'applicazione nativa desktop**

## **RDF e RDFS**

Il Web Semantico rappresenta **un'evoluzione del World Wide Web tradizionale**. Secondo Tim Berners-Lee, esso è **un'estensione** del Web che permette alle persone di **condividere contenuti oltre i confini delle applicazioni e dei siti web**.

Il Web attuale è progettato principalmente per la **comprendione umana**. Le macchine faticano a interpretare il significato dei dati a causa di problemi linguistici come:

- **Polisemia (Stesso nome, significati diversi):** La parola "Roma" può riferirsi alla capitale d'Italia, alla squadra di calcio (AS Roma) o alla storia antica. Un motore di ricerca tradizionale potrebbe faticare a distinguere il contesto senza un aiuto semantico
- **Sinonimia (Nomi diversi, stesso significato):** Concetti come "automobile", "auto", "autovettura" o "macchina" indicano lo stesso oggetto, ma sono stringhe di testo diverse

L'obiettivo è fornire un **modo standardizzato** per esprimere **relazioni** tra pagine web e permettere alle macchine di **comprendere il significato** delle **informazioni** collegate. Le specifiche fondamentali sono:

- **RDF** (Resource Description Framework): Usato per **esprimere conoscenza** in un **mondo decentralizzato**; è la fondazione del Web Semantico
- **OWL** (Web Ontology Language): **Estende l'espressività** di **RDF** aggiungendo un livello ulteriore di **semantica (ontologie)**

## Architettura: Il "Semantic Web Layer Cake"

L'architettura del Web Semantico è organizzata a **livelli** (stack), dove ogni livello costruisce sulle capacità di quelli sottostanti:

- **URI/IRI**: Identificatori univoci per le risorse
- **XML & Namespaces**: Sintassi per creare **documenti strutturati**
- **RDF**: Intercambio di **dati**; rappresenta i dati riguardo alle risorse
- **RDFS** (RDF Schema): Fornisce il **vocabolario di base** per RDF
- **OWL**: **Ontologie** per descrivere **relazioni complesse** e logica unificante
- **SPARQL**: Linguaggio di **interrogazione** per recuperare informazioni dalle applicazioni semantiche
- **RIF** (Rule Interchange Format): **Describe relazioni e regole non descrivibili** direttamente con OWL
- **Livelli superiori** (Non ancora pienamente realizzati):
  - **Crypto**: Per **verificare** che le **dichiarazioni** provengano da **fonti fidate**
  - **Trust**: Verifica delle **dichiarazioni** derivate
  - **Proof**: Fornisce **spiegazioni** sulle **risposte** date dagli agenti automatici

## RDF: Resource Description Framework

RDF è un **framework** per **rappresentare informazioni** nel Web. Permette di **aggiungere informazioni leggibili dalle macchine** alle pagine web e fornisce uno standard per lo **scambio di dati**

### Modello di dati RDF

La sintassi astratta di RDF si basa su tre strutture chiave:

- 1) **RDF Statement** (Tripla): L'**unità base** dell'informazione
- 2) **RDF Graph**: Un **insieme di triple** RDF
- 3) **RDF Dataset**: Una **collezione di grafi** RDF

### Tripla RDF

Un'asserzione RDF esprime una **relazione** tra due **risorse** ed è composta da tre parti:

- **Soggetto** (Subject): La **risorsa** di cui si sta parlando (identificata da IRI)
- **Predicato** (Predicate): La **natura** della **relazione** (identificato da IRI). La relazione è **direzionale** (dal soggetto all'oggetto)
- **Oggetto** (Object): Il **valore** della **proprietà**. Può essere **un'altra risorsa** (IRI) o un **valore letterale** (Literal). I Literal possono apparire solo in questa posizione

Es.: Il film "The Matrix" (Soggetto) ha come attore ("starring" - Predicato) "Keanu Reeves" (Oggetto)

## Serializzazioni RDF

Le triple RDF sono un **modello astratto** che può essere **salvato e condiviso** (serializzato) in diversi formati

### RDF/XML

È la **sintassi originale** basata su XML (sviluppata a fine anni '90). È spesso considerata **prolixa** (verbose):

- Le triple sono specificate dentro l'elemento `<rdf:RDF>`
- Si usa `<rdf:Description>` con l'attributo `rdf:about` per definire il Soggetto
- I figli di questo tag rappresentano Predicati e Oggetti

Es.:

```
<rdf:Description rdf:about="dbpedia:The_Matrix">
    <dbpedia-owl:starring rdf:resource="dbpedia:Keanu_Reeves"/>
</rdf:Description>
```

### N-Triples

È un **formato semplice**, basato su **linee di testo**. Ogni linea rappresenta una **tripla completa** e termina con un punto. Gli IRI completi sono racchiusi tra parentesi angolari `< >`

Es.:

```
<http://dbpedia.org/resource/The_Matrix>
<http://dbpedia.org/ontology/starring>
<http://dbpedia.org/resource/Keanu_Reeves> .
```

### Turtle (Terse RDF Triple Language)

Un'estensione di N-Triples che introduce **scorciatoie sintattiche** per **migliorare la leggibilità e facilitare la scrittura**:

- Supporta i prefissi per i namespace (Es.: PREFIX dbpedia: ...)
- Notazione a virgola (,): Per stesso soggetto e stesso predicato, ma oggetti diversi
  - Es.: `dbpedia:The_Matrix dbpedia-owl:starring dbpedia:Keanu_Reeves, dbpedia:Carrie-Ann_Moss`
- Notazione a punto e virgola (;): Per stesso soggetto, ma prediciati e oggetti diversi
  - Es.: `dbpedia:The_Matrix dbpedia-owl:starring ... ; dbpedia-owl:director ... .`

### Dataset e Grafi Multipli (TriG e N-Quads)

Questi formati estendono le **sintassi precedenti** per supportare dataset RDF che contengono **grafi multipli**:

- **TriG**: Estensione di Turtle. Permette di **raggruppare triple** in un **grafo** nominato usando la sintassi `GRAPH <iri> { ... }`

- **N-Quads:** Estensione di N-Triples Aggiunge un quarto elemento alla linea che rappresenta l'IRI del grafo: <subject> <predicate> <object> <graph-IRI> .

## JSON-LD

Una sintassi JSON per RDF, progettata per **trasformare documenti JSON** esistenti in **RDF** con modifiche minime:

- Usa un oggetto `@context` per mappare le chiavi JSON a IRI RDF
- Usa `@id` per identificare la risorsa (Soggetto) e `@type` per definire il tipo

## RDFA

Utilizzato per **incorporare dati RDF direttamente dentro documenti** HTML e XML. Permette ai motori di ricerca di aggregare dati durante il crawling. Utilizza attributi speciali nell'HTML: `resource`, `property`, `typeof`, `prefix`

Es.:

```
<div resource="http://example.org/bob#me" typeof="foaf:Person">
    Bob knows <a property="foaf:knows" href="...>Alice</a>
</div>
```

## Costrutti RDF avanzati

### Tipi e Proprietà

- `rdf:type`: È il **predicato fondamentale** utilizzato per indicare che una risorsa è **un'istanza** di una **classe**
  - Es.: `dbpedia:The_Matrix rdf:type dbpedia-owl:Film`
- `rdf:Property`: Definisce che una **risorsa** è una **proprietà RDF** (un predicato)
  - Es.: `dbpedia-owl:starring rdf:type rdf:Property`

### Container RDF (Gruppi di risorse)

RDF offre **tre tipi di contenitori** per descrivere gruppi di risorse:

- `rdf:Bag`: Una **lista di valori non ordinata** (Es.: i membri di una band, file in una cartella)
  - Es.: I membri degli U2 (Bono, The Edge, ecc.) sono raggruppati in una Bag poiché l'ordine non è rilevante per definire il gruppo
- `rdf:Seq`: Una **lista di valori ordinata** (Es.: elenco alfabetico, capitoli di un libro)
- `rdf:Alt`: Una **lista di valori alternativi** (Es.: diversi formati di file per lo stesso contenuto, siti mirror)

### Valori Letterali (Literals)

I literal (oggetti delle triple che non sono URI) possono essere **arricchiti** in due modi:

- **Typed Literal:** Associano una **stringa** a uno specifico **tipo** di **dato** (solitamente XML Schema Datatypes)
  - Sintassi: `"valore"^^xsd:tipo`

- Es.: "1216"^^xsd:integer (indica che 1216 è un numero intero, non solo testo)
- **Language-tagged String:** Associano una **stringa** a un **identificativo di lingua**
  - Sintassi: "stringa"@lang
  - Es.: "The Lord of the Rings"@en, "Il Signore degli Anelli"@it

## Blank Nodes (Nodi Anonimi)

I Blank Nodes (o B-Nodes) sono **risorse private** di **URI** (anonime). Vengono utilizzati per:

- **Raggruppare dati strutturati** senza dover creare un URI globale per il contenitore
- **Aggregare statement complessi**
- **Rappresentazione:** Spesso indicati con \_:id (identificativo locale)
- Es.: Un indirizzo fisico composto da via, città e stato può essere rappresentato come un Blank Node collegato a una persona, invece di creare tre triple separate dirette dalla persona

## Reificazione (Reification)

La reificazione è il processo di **fare asserzioni** su **altre asserzioni**. Serve per **descrivere metadati** di una tripla (chi l'ha creata, quando, grado di fiducia). Poiché una tripla RDF non ha un ID intrinseco, per parlarne bisogna trasformarla in una risorsa composta da quattro nuove triple:

- 1) rdf:type -> rdf:Statement
- 2) rdf:subject -> Il soggetto della tripla originale
- 3) rdf:predicate -> Il predicato della tripla originale
- 4) rdf:object -> L'oggetto della tripla originale

Es.: Per dire che "DBpedia afferma che The Matrix ha come attore Keanu Reeves", si reifica la tripla originale e si aggiunge la proprietà dc:creator

## RDFS (RDF Schema)

RDF Schema è **un'estensione semantica** di RDF. Mentre RDF descrive i dati (istanze), RDFS **fornisce il vocabolario** per **descrivere tassonomie** di classi e proprietà

## Classi Principali

Tutto in RDFS è definito usando la sintassi RDF stessa:

- rdfs:Resource: La **classe di tutte le risorse** (tutto è una risorsa)
- rdfs:Class: La **classe delle classi**
- rdfs:Literal: La **classe dei valori letterali**
- rdfs:Datatype: La **classe dei tipi di dato**

## Tassonomia e Gerarchie

RDFS permette di **creare gerarchie** (tassonomie) tramite:

- rdfs:subClassOf: Indica che una **classe** è **sottoclasse di un'altra**. La proprietà è **transitiva**
  - Es.: MiniVan è sottoclasse di Van, che è sottoclasse di MotorVehicle

## Definizione delle Proprietà

RDFS permette di **restringere e definire il significato** delle **proprietà**:

- **rdfs:domain** (Dominio): Specifica la classe del Soggetto che può avere questa proprietà
  - Es.: se `hasSon` ha dominio `Person`, allora qualsiasi risorsa che usa `hasSon` come predicato è implicitamente una `Person`
- **rdfs:range** (Codominio/Range): Specifica la classe (o datatype) dell'oggetto che questa proprietà può accettare
  - Es.: se `hasSon` ha range `Man`, allora l'oggetto della relazione deve essere un `Man`
- **rdfs:subPropertyOf**: Crea una gerarchia tra proprietà (Es.: `hasFather` è sottoproprietà di `hasParent`)

## Metadati e Documentazione

Proprietà per rendere i dati comprensibili agli umani:

- **rdfs:label**: Nome leggibile della risorsa (Es.: "Juventus F.C."@en)
- **rdfs:comment**: Descrizione estesa o definizione
- **rdfs:seeAlso**: Link a ulteriori informazioni
- **rdfs:isDefinedBy**: Link al vocabolario che definisce la risorsa

## Evoluzione: Da RDF a OWL

Le tecnologie del Web Semantico si evolvono in termini di espressività:

- **RDF**: Crea un grafo diretto per collegare risorse. Non ha gerarchie native
- **RDFS**: Aggiunge un vocabolario per creare tassonomie (classi, sottoclassi) e definire vincoli basilari (dominio, range)
- **OWL (Web Ontology Language)**: Aggiunge ricchezza semantica. Introduce restrizioni logiche formali, cardinalità, uguaglianza tra classi, ecc. È necessario per il ragionamento automatico complesso (reasoning)

## SPARQL

**SPARQL** (acronimo ricorsivo per SPARQL Protocol and RDF Query Language) è il linguaggio standard del W3C progettato per interrogare dataset rappresentati tramite il modello **RDF** (Resource Description Framework)

### Caratteristiche Principali

- **Graph Pattern Matching**: A differenza dell'SQL che lavora su tabelle relazionali, SPARQL si basa sul confronto di pattern all'interno di grafi RDF
- **Nessuna inferenza nativa**: Di base, SPARQL interroga i dati così come sono ("asserted"), senza applicare ragionamenti logici automatici (built-in inferences)

### SPARQL Endpoints

Un endpoint SPARQL è un servizio web che accetta query e restituisce risultati tramite protocollo HTTP:

- **Formati di output**: I risultati possono essere restituiti in formati processabili dalle macchine (XML, JSON, RDF) o leggibili dall'uomo (HTML)

- **Tipologie di Endpoint:**

- **Specifici:** Interrogano un **dataset predefinito** (Es.: DBpedia)
- **Generici:** Possono interrogare **qualsiasi dataset** RDF accessibile via Web

## Il Meccanismo di Query: Triple Patterns

Il cuore di una query SPARQL è il **Triple Pattern** (pattern di tripla). È strutturalmente **identico** a una tripla RDF (Soggetto, Predicato, Oggetto), con la differenza che una o più parti possono essere **sostituite da variabili**.

- **Variabili:** Sono indicate dal prefisso `?`  (Es.: `?subject, ?actor`). Agiscono come **incognite**
- **Funzionamento:** Il motore di query cerca nel grafo RDF tutte le triple che **corrispondono** al **pattern**, sostituendo le variabili con i termini RDF reali (URI o letterali) trovati nel dataset

Un esempio potrebbe essere:

```
?movie dbpedia-owl:starring ?actor
```

Questo pattern cerca tutte le risorse (`?movie`) che hanno una relazione `starring` con un'altra risorsa (`?actor`)

## Struttura di una Query SPARQL

Una query SPARQL standard è composta da diverse sezioni in un ordine preciso:

- **Dichiarazione dei Prefissi (PREFIX / BASE):** Serve a definire **abbreviazioni** per gli **IRI** (Internationalized Resource Identifiers) lunghi, migliorando la leggibilità
  - **PREFIX:** Associa un'etichetta a un namespace (Es.: `dbpedia:` per `<http://dbpedia.org/resource/>`)
  - **BASE:** Definisce l'IRI di base per i percorsi relativi
- **Definizione del Dataset (FROM / FROM NAMED):** Specifica su quali grafi eseguire la query
  - **FROM <iri>:** Indica il grafo di default. Se specificato più volte, i grafi vengono uniti (merge)
  - **FROM NAMED <iri>:** Indica un grafo con nome, utile per query contestuali specifiche (vedi sezione GRAPH)
- **Forma della Query:** Determina il tipo di output (Es.: `SELECT, CONSTRUCT`)
- **Pattern di Query (WHERE { ... }):** Contiene i criteri di ricerca veri e propri
- **Modificatori:** Clausole per ordinare o limitare i risultati (Es.: `ORDER BY, LIMIT`)

## Pattern Complessi e Operatori Logici

All'interno della clausola `WHERE`, i **pattern** possono essere **combinati** per esprimere logiche complesse

### Congiunzione (AND implicito)

Scrivere più triple pattern separati da un punto `(.)` richiede che **tutti i pattern siano soddisfatti contemporaneamente**

Es.: Trova film E i loro registi

```
?movie dbpedia-owl:starring ?actor .
```

```
?movie dbpedia-owl:director ?director
```

## OPTIONAL (Left Join)

Permette di **recuperare informazioni aggiuntive** se esistono, senza escludere il risultato se mancano.  
È fondamentale per gestire dati incompleti

Es.: Trova film e attori e, se disponibile, il paese di produzione

```
OPTIONAL { ?movie dbpprop:country ?country }
```

## UNION (OR logico)

Definisce **alternative**. La query restituisce risultati che soddisfano **almeno uno** dei **gruppi** di pattern

Es.: Trova persone che sono attori OPPURE registi del film

```
{ ?movie dbpedia-owl:starring ?person }
UNION
{ ?movie dbpedia-owl:director ?person }
```

## FILTER (Restrizioni)

Applica **vincoli** ai **valori** delle **variabili** (numerici, stringhe, date)

Es.: FILTER (?budget > 1.0E8)

## GRAPH (Contesto)

Permette di **dirigere la ricerca** verso un **grafo specifico** (definito con `FROM NAMED`) invece che sul grafo di default

Es.: Cerca l'email di Bob solo all'interno del grafo bobFoaf

```
GRAPH <http://example.org/foaf/bobFoaf> { ... }
```

## Operatori e Modificatori

SPARQL include funzioni per **manipolare i dati e l'output**

### Modificatori di Risultato

- `ORDER BY`: **Ordina i risultati** (Es.: `DESC(?var)` per discendente)
- `LIMIT n`: **Restituisce al massimo n risultati**
- `OFFSET n`: **Salta i primi n risultati** (utile per la **paginazione**)

### Operatori (Funzioni)

- **Unari**: `bound(?var)` (vero se la variabile ha un valore), `isIRI`, `isLiteral`, `isBlank`
- **Binari**: `regex(?string, pattern)` per **espressioni regolari** su stringhe, `langMatches` per filtrare per lingua

## Forme di Query

SPARQL non serve solo a estrarre tabelle di dati, ma offre quattro forme di **interrogazione principali**:

- **SELECT**: Restituisce una **tabella di valori** (bindings) per le **variabili specificate**. È la forma più simile a SQL
- **CONSTRUCT**: Restituisce un nuovo **grafo RDF**. Le variabili trovate nel **WHERE** vengono usate per riempire un template di triple definito dall'utente. Utile per **trasformare dati** da un'ontologia all'altra
- **ASK**: Restituisce un **valore booleano (TRUE/FALSE)**. Verifica semplicemente se **esiste almeno una soluzione** per il pattern specificato, **senza restituire i dati**
- **DESCRIBE**: Restituisce un grafo RDF che "describe" le **risorse trovate**. La struttura esatta delle informazioni restituite dipende dall'implementazione del server (SPARQL engine), ma generalmente include le triple in cui la risorsa è soggetto od oggetto

## Formato dei Risultati XML

Quando un endpoint restituisce risultati in XML (spesso il formato di default per l'elaborazione automatica), la struttura tipica è:

- **<head>**: Contiene l'**elenco** delle **variabili richieste**
- **<results>**: Contiene una **lista di tag <result>**
- **<binding>**: Associa una **variabile (name="actor")** al **valore** trovato (**<uri>** o **<literal>**)

## OWL

OWL (Web Ontology Language) è un **linguaggio** del Semantic Web progettato per **rappresentare conoscenza ricca e complessa** riguardo a **entità, gruppi** di entità e le **relazioni** che intercorrono tra loro.

A differenza di RDF/RDFS, che permettono di creare grafi e tassonomie semplici, OWL è un **linguaggio basato** sulla **logica computazionale**. Questo permette agli agenti software (reasoners) di:

- **Verificare la consistenza della conoscenza** (trovare contraddizioni)
- **Estrarre nuova conoscenza implicita** (inferenza) a partire da quanto esplicitamente dichiarato

La versione attuale è OWL 2 (rilasciata nel 2012), che estende e raffina la specifica originale del 2004

## Struttura e Sintassi

Un'ontologia OWL 2 può essere vista **concretamente** come un **grafo RDF**. Tuttavia, per gestire la complessità semantica, OWL **supporta diverse sintassi** concrete per la memorizzazione e lo scambio dei dati:

- **RDF/XML**: È la **sintassi obbligatoria** che tutti i tool devono supportare per **garantire l'interoperabilità**. È prolissa e difficile da leggere per gli umani
- **OWL/XML**: Ottimizzata per essere processata da tool XML generici
- **Functional Syntax**: Ideale per visualizzare la **struttura formale dell'ontologia**
- **Manchester Syntax**: Una sintassi progettata per essere **leggibile dall'uomo e facile da scrivere**. È quella utilizzata di default in editor come Protégé per definire espressioni logiche
- **Turtle**: Molto usata per la sua **compattezza** nella **rappresentazione** di triple RDF

Es.: Definire che "Mary" è una "Persona"

- Functional: ClassAssertion( :Person :Mary )
- Turtle: :Mary rdf:type :Person .
- Manchester:

Individual: Mary

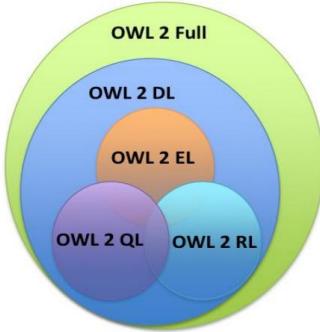
Types: Person

**Commentato [VF3]:** vedere

## Profili di OWL

Dato che il ragionamento automatico su ontologie complesse può essere computazionalmente oneroso, OWL 2 definisce dei Profili (o sotto-linguaggi). Ogni **profilo** è un **sottoinsieme** del **linguaggio completo** (OWL 2 Full) ottimizzato per specifici scenari applicativi:

- **OWL 2 EL** (Existential Logic)
  - Uso: **Ontologie** con un **numero enorme** di **classi e proprietà**
  - Es.: SNOMED CT (ontologia medica)
  - Performance: Ragionamento in tempo polinomiale
- **OWL 2 QL** (Query Language)
  - Uso: Accesso a **grandissimi volumi** di **dati** (istanze) tramite **query**, simile ai database relazionali
  - Performance: Permette la riscrittura delle query (Query Rewriting)
- **OWL 2 RL** (Rule Language)
  - Uso: Scenari che richiedono **regole scalabili** senza sacrificare troppa espressività
  - Performance: Ottimizzato per motori di regole
- **OWL 2 DL** (Description Logic)
  - Il **profilo standard** che garantisce la decidibilità computazionale pur mantenendo la massima espressività possibile



## Nozioni di Base: Assiomi ed Entità

Un'**ontologia** è composta da **Axiomi** (le dichiarazioni di base). Gli **assiomi descrivono le Entità**:

- **Individui** (Individuals): Gli **oggetti** del mondo reale (Es.: :John, :Mary)
- **Classi** (Classes): **Categorie** di oggetti (Es.: :Person, :Woman)
- **Proprietà** (Properties): **Relazioni**. Si dividono in:
  - **Object Property**: Collegano due oggetti (Es.: :John :married :Mary)

- **Datatype Property:** Collegano un oggetto a un valore dati (Es.: `:John :hasAge 35`)
- **Annotation Property:** Metadati non logici (Es.: `rdfs:seeAlso`, commenti)

## Assiomi di Base

OWL permette di definire relazioni strutturali semplici:

- **Gerarchia:**
  - `SubClassOf`: Tassonomia (Es.: `Woman` è sottoclasse di `Person`)
  - `SubPropertyOf`: Gerarchia di relazioni (Es.: `hasWife` è sottoproprietà di `hasSpouse`)
- **Equivalenza e Disgiunzione:**
  - `EquivalentTo`: Due classi sono identiche (Es.: `Person = Human`)
  - `DisjointClasses`: Un individuo non può appartenere a entrambe (Es.: `Man` e `Woman` sono disgiunti)
- **Restrizioni di Dominio e Range:**
  - Dominio: A che tipo di soggetto si applica la proprietà (Es.: `hasWife` si applica al dominio `Man`)
  - Range: Che tipo di valore accetta la proprietà (Es.: `hasWife` punta verso il range `Woman`)
- **Uguaglianza tra Individui:**
  - `SameAs`: Due nomi diversi indicano lo stesso oggetto reale (James `SameAs` Jim)
  - `DifferentFrom`: Esplicita diversità (John `DifferentFrom` Bill)

## Assiomi Avanzati (Costruttori di Classi)

La potenza di OWL sta nella capacità di **definire Classi Complesse combinando classi semplici** tramite operatori logici e restrizioni sulle proprietà

## Operatori Logici (Insiemistica)

- **AND** (`IntersectionOf`): L'intersezione di due insiemi
  - Esempio: `Mother` è equivalente a `Woman AND Parent`
- **OR** (`UnionOf`): L'unione di due insiemi
  - Esempio: `Parent` è equivalente a `Mother OR Father`
- **NOT** (`ComplementOf`): La negazione
  - Esempio: `ChildlessPerson` è equivalente a `Person AND (NOT Parent)`

## Restrizioni sulle Proprietà (Quantificatori)

Questi costrutti definiscono **classi anonime** basate sulle **relazioni** che gli **individui intrattengono**:

- **Quantificazione Esistenziale** (`some / exist`):
  - Descrive individui che hanno almeno una relazione di un certo tipo
  - Es.: `Parent` è equivalente a qualcuno che ha la relazione `hasChild` verso qualche (some) `Person`
- **Quantificazione Universale** (`only / all`):
  - Descrive individui le cui relazioni di un certo tipo sono tutte verso una specifica classe
  - Es.: `HappyPerson` è equivalente a qualcuno che ha la relazione `hasChild` solo (only) verso `HappyPerson` (tutti i suoi figli devono essere felici)

- **Cardinalità** (`min`, `max`, `exactly`):
  - Vincola il numero di relazioni
  - Es.: `hasChild max 4` (avere al massimo 4 figli)

## Gestione delle Ontologie e Strumenti

- **Gestione:** Ogni ontologia ha un **IRI univoco**. Può usare **prefissi** per **abbreviare** i namespace e può importare altre ontologie intere tramite Import
- **Editor:** Lo strumento più diffuso e open-source è Protégé (Stanford University)
- **Reasoners:** Motori di inferenza utilizzati per validare e ragionare sulle ontologie. I più noti sono FaCT++, HermiT, Pellet e Tiny-ME (sviluppato dal Politecnico di Bari)

## LinkedData

Il concetto fondamentale dei Linked Data è **l'evoluzione** dalla semplice pubblicazione di dati in formati leggibili dalle macchine alla creazione di una **rete di dati interconnessi**:

- **RDF come Data Model:** RDF (Resource Description Framework) non va inteso semplicemente come un formato di file, ma come un **modello di dati**. È un **formalismo basato su triple** Soggetto-Predicato-Oggetto che formano un grafo
- **Ruolo degli URI:** Al centro di questa visione ci sono gli URI (Uniform Resource Identifiers). Ogni URI non è solo un identificativo, ma corrisponde alla **descrizione completa** di una **risorsa**
- **Interoperabilità:** L'obiettivo è permettere che dataset prodotti indipendentemente da enti diversi possano essere **collegati liberamente** da **terze parti**, creando un **grafo globale di conoscenza**

## Principi dei Linked Data

Tim Berners-Lee ha stabilito quattro principi fondamentali affinché i dati possano essere considerati "Linked Data":

- **Usare URI per i nomi:** Ogni **risorsa** deve essere **identificata univocamente** da un **URI**
- **Usare URI HTTP:** Gli **URI** devono essere **accessibili** tramite protocollo http, permettendo a persone e agenti software di "cercarli" (differenziarli)
- **Fornire informazioni utili:** Quando un **URI** viene **consultato** (lookup), deve **restituire** una **descrizione utile** della risorsa utilizzando standard come RDF o SPARQL
- **Includere link ad altri URI:** Le **descrizioni** devono contenere **collegamenti** ad altre **risorse**, permettendo la scoperta di nuovi dati (effetto rete)

## Vocabolari Standard

Per valutare la qualità dell'apertura dei dati, esiste una **scala a cinque stelle**:

- ★ **On the Web:** Dati disponibili sul web con licenza aperta (qualsiasi formato, es. PDF, immagine)
- ★★ **Machine-readable:** Dati strutturati in modo che una macchina possa leggerli (es. file Excel .xls), ma in formato proprietario
- ★★★ **Non-proprietary format:** Dati strutturati in formato aperto e non proprietario (es. CSV)
- ★★★★ **RDF Standards:** Dati identificati tramite URI e descritti usando gli standard W3C (RDF), rendendoli accessibili in modo granulare
- ★★★★★ **Linked RDF:** Come sopra, ma i dati contengono link verso altri dataset esterni, contestualizzando l'informazione nel Web Semantico

## SKOS (Simple Knowledge Organization System)

È uno **standard W3C** basato su RDF/RDFS per **rappresentare sistemi di organizzazione** della **conoscenza** come **tesauri, tassonomie e schemi di classificazione**:

- Classe principale: `skos:Concept` (l'unità fondamentale di senso)
- Proprietà principali:
  - `skos:prefLabel`: L'etichetta preferita (il nome principale del concetto)

- **skos:altLabel**: Etichette alternative (sinonimi, acronimi)
- **skos:broader**: Indica un concetto più generale (simile a una superclasse, ma semantico)
- **skos:narrower**: Indica un concetto più specifico
- **skos:related**: Indica una relazione associativa tra concetti
- Es.: Utilizzato dal New York Times o dalla Library of Congress per organizzare gli argomenti (Es.: "Science fiction films")

## Dublin Core (DC)

Un **set** di **metadati** standard per **descrivere risorse digitali e fisiche** (originariamente testi):

- **Proprietà principali** (15 elementi): **dc:title**, **dc:creator**, **dc:subject**, **dc:description**, **dc:publisher**, **dc:date**, **dc:format**, **dc:identifier**, **dc:language**, **dc:rights**
- **Utilizzo**: Fornisce un livello base di **descrizione bibliografica comune** a quasi tutti i dataset

## FOAF (Friend of a Friend)

Un'ontologia per **descrivere le persone**, le loro **attività** e le loro **relazioni sociali**:

- **Classi e Proprietà**: **foaf:Person**, **foaf:name**, **foaf:mbox** (email), **foaf:homepage**, **foaf:knows** (per collegare due persone)
- **Privacy**: L'email è spesso **oscurata** tramite **hash** (**foaf:mbox\_sha1sum**) per **identificare univocamente la persona** senza rivelare l'indirizzo in chiaro

## GoodRelations

Un **vocabolario standard** per l'e-commerce, utile per descrivere prodotti, prezzi, negozi e offerte sul web. Le entità principali sono:

- **Business Entity**: L'**agente** (azienda o persona) che **vende**
- **ProductOrService**: L'**oggetto** della **vendita** (Es.: un monitor)
- **Offering**: L'**offerta commerciale** (prezzo, metodi di pagamento)
- **Location**: Il **luogo fisico** del negozio o magazzino

## DBpedia: Il centro del LOD Cloud

DBpedia è la **versione "Semantic Web"** di Wikipedia. Estraе informazioni strutturate dagli "infobox" (i riquadri riassuntivi) delle pagine di Wikipedia e le rende disponibili come RDF:

- **Mapping**: A ogni pagina di Wikipedia corrisponde una risorsa DBpedia
  - URL Wikipedia: [http://en.wikipedia.org/wiki/The\\_Matrix](http://en.wikipedia.org/wiki/The_Matrix)
  - URI DBpedia: [http://dbpedia.org/resource/The\\_Matrix](http://dbpedia.org/resource/The_Matrix)
- **Multilinguismo**: Sfrutta i **link interlinguistici** di Wikipedia per creare **etichette** (**rdfs:label**) e **abstract** in diverse lingue per la stessa risorsa
- **Categorizzazione**: Utilizza la proprietà **dcterms:subject** per **collegare una risorsa** alle **categorie** di Wikipedia, modellate tramite SKOS (Es.: **skos:broader** per la gerarchia delle categorie)

## Integrazione di Database Relazionali (RDB2RDF)

Poiché gran parte dei dati mondiali risiede in **database relazionali** (SQL), esistono **standard** per **esporli** come **RDF**:

- **Direct Mapping:** Una **mappatura automatica** dove le **tabelle** diventano **classi**, le **colonne** diventano **proprietà** e le **righe** diventano **risorse** (identificate da URI basati sulla Chiave Primaria)
  - Es.: Una riga della tabella People con `ID=7` diventa una risorsa `<People/ID=7>` di tipo `People`
- **R2RML** (RDB to RDF Mapping Language): Un linguaggio che permette una mappatura personalizzata. Consente di definire come le tabelle SQL debbano essere trasformate in specifiche classi e proprietà di un'ontologia target, gestendo trasformazioni complesse (Es.: unire colonne per formare un nome completo)

## Licenze per i Dati (Open Data Licensing)

Perché i dati siano effettivamente "Open" e riutilizzabili, devono essere accompagnati da una licenza chiara

### Creative Commons (CC)

Un set di licenze standardizzate:

- CC-BY (Attribution): Permette tutto (anche uso commerciale), obbligo di citare l'autore
- CC-NC (Non Commercial): Vieta l'uso commerciale
- CC-ND (No Derivative Works): Permette la ridistribuzione ma non la modifica
- CC-SA (Share Alike): Impone di rilasciare opere derivate con la stessa licenza (stile copyleft)
- CC0 (No Rights Reserved): Pubblico dominio

### Italian Open Data License (IODL)

Licenza specifica per la PA italiana, promossa da AgID:

- IODL 1.0: Simile alla Share-Alike (obbligo di mantenere la licenza)
- IODL 2.0: Allineata alla CC-BY 4.0. Richiede solo la citazione della fonte, permettendo libero riutilizzo, anche commerciale