

Capitolo 1

Web

- **Web 1.0**
 - Statico
 - Interazione Unilaterale (Il server presenta, io seleziono)
 - Adatto solo per contenuti che cambiano raramente

- **Web 1.5**
 - Dinamico
 - Script per generazione di contenuti grazie ad accesso a DB relazionali (AJAX)
 - Il server genera, noi consumiamo

- **Web 2.0**
 - L'utente diventa prosumer (consuma e produce contenuti)
 - Divisione logica dal Web 1.5, non **tecnologica**

- **Web Semantico**
 - Le informazioni ora hanno un significato
 - Possono essere comprese anche dalle macchine
 - Anche le relazioni vengono specificate

- **Web 2.5**
 - Parallelo al Web 3.0
 - Concentrato sul **mobile computing**
 - Pagine Web Veloci (leggere) e Progressive

- **Web 3.0**
 - Parallelo al Web 2.0
 - Evoluzione del Web Semantico
 - **Include:**
 - **Web Of Data**
Crea Ontologie (Esistenza mediante linguaggio formale) e ***fa emergere il deep-web***
 - **AI**

Strumento per predizione di interessi e
comprensione linguaggio umano

- **Ubiquity**

Servizio usato da più piattaforme

- **Web 3.0 (alt)**

- Decentralizzazione del computing mediante l'uso delle blockchain

Protocolli Internet

Simile a Stack ISO-OSI

- **Applicativo**

Contiene i servizi internet (http, https, ssh, ecc...)

- **Trasporto**

Servizi di trasporto (TCP, UDP)

- **Internet**

Trasmissione logica (IP e Protocolli IEEE 802)

- **Rete**

Trasmissione fisica con interfacce

- **Fisico**

Trasporto fisico nel mezzo

Capisaldi del Web

- **HTML**

Linguaggio di Markup che include collegamenti Ipertestuali (HyperText Markup Language)

- **URI**

Stringhe Identificative di Risorse (Uniform Resource Identifier)

- **HTTP**

Trasporto e scambio sulla rete (HyperText Transfer Protocol)

W3C

Consorzio per lo sviluppo del Web

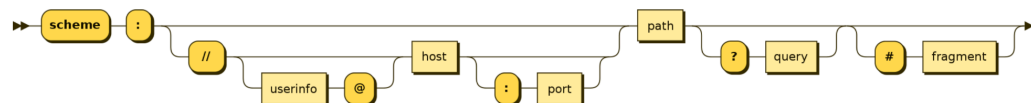
Capitolo 2 (Protocolli URI e HTTP)

Uniform Resource Identifier

- **URI**: può essere utilizzata per **identificare** qualunque cosa in maniera **univoca**.
 - **URL** - **U**niform **R**esource **L**ocator: standard introdotto per la **localizzazione** di risorse Web su una rete di computer.
 - **URN** - **U**niform **R**esource **N**ame: fornisce **identificatori** e nomi globali e **persistenti** a determinate risorse, all'interno di specifici namespace, ma non può essere usato per la loro localizzazione
- **IRI**-Internationalized Resource Identifier: **standard** che espande l'**insieme** di **caratteri** consentiti da URI

Uri in Dettaglio

Schema di un URI:



http in Dettaglio:

Nota, togliere gli spazi

[http://dominio : numero_porta / \[path ? query # fragment \]](http://dominio:numero_porta/path?query#fragment) ripetuti n volte

In caso la porta non sia specificata, verrà usata la porta:

- 80 per **http**
- 443 per **https**

HyperText Transfer Protocol

- **HTTP:**

Protocollo di livello **applicativo** nella suite TCP/IP.che usa TCP come livello di trasporto. Esso regola gli **scambi** dati nel web: permette di **ottenere** risorse presenti su web server e inviare contenuti al web server.La comunicazione avviene mediante il **paradigma richiesta-risposta** (quindi una comunicazione client-server). Trova la sua forza anche nell'essere **human-readable**

- **CLIENT:**

generalmente ricoperto dall'utente, **richiede le risorse** per mezzo del suo **user-agent**, il **browser**. Il client, di regola, inizia la richiesta HTTP.

- **SERVER:**

Accetta connessioni HTTP e genera risposte

- **PROXY:**

Entità posizionate tra client e server che eseguono diverse operazioni (a livello applicativo):

- firewall
- filtraggio delle richieste
- caching: memorizza le risorse
- load balancing: smista le richieste
- autenticazione
- logging

Versioni di HTTP

- **0.9**

Protocollo a una riga. Dava al client il file HTML richiesto

- **1.0**

Libera un po' dalle catene il protocollo 0.9. Permette di inviare altro, oltre ai file .html grazie al nuovo header: **"Content-Type"**:

Fino ad ora le richieste sono tutte stateless. Una risorsa, Una richiesta.

- **1.1**

Standard in uso ancora oggi. Permette Caching, trasferimenti chunked, multi-homing e connessioni persistenti.

Fino ad ora le richieste erano inviate in **plain text**

- **2.0**

Multiplexing e richieste in frames.

Da qui in poi le richieste **sono mandate in binario**

- **3.0**

Basate su QUIC, appoggiato a UDP, ma affidabile come TCP

Richieste HTTP

GET:

- Metodo per la **richiesta** di risorse
- Prevede un **campo Host** e l'**assenza di un corpo**
- Dopo "?" è possibile **mandare** una **query**

POST:

- Metodo per l'**invio** di risorse
- Usato per mandare **blocchi di dati**

HEAD:

- Simile a GET
- **Restituisce** soltanto l'**header** della risorsa
- Serve per **ottenere** informazioni **senza** il bisogno di **scaricare** la risorsa per intero

PUT e DELETE:

- Permettono la **rimozione** di risorse che sono **su server**
- Rispetto alle post sono **idempotenti** (più PUT -> stesso risultato)

Altre:

- **OPTIONS**
- **TRACE**
- **CONNECT**

Proprietà metodi HTTP

SAFE

I metodi sono read-only e non cambiano lo stato del server.

NB.: Talvolta le GET non sono safe (Caso della verifica email). Safe implica Idempotente

IDEMPOTENZA

Questi metodi producono sempre lo stesso risultato sul server anche se eseguiti più volte

CACHEABLE

le risposte si possono memorizzare nella cache dei client

	CORPO REQ	CORPO RES	SICURO	IDEMPOTENTE	CACHEABLE	FORM HTML
--	-----------	-----------	--------	-------------	-----------	-----------

GET	X	✓	✓	✓	✓	✓
POST	✓	✓	X	X	*	✓
HEAD	X	X	✓	✓	✓	X

* Le risposte delle POST possono essere Cacheate solo in caso di informazioni sulla freshness

Risposte HTTP

1xx - [Risposte Provvisorie]:

- **100 - Continue**

2xx - [Richieste ricevute e comprese correttamente]

- **200 - OK** : (dopo GET o POST)
- **201 - Created** : (dopo POST o PUT)
- **204 - No Content** : (dopo POST o PUT)

3xx -

[Richieste ricevute e comprese, ma necessitano ulteriori azioni dell'Agent]

- **301 - Moved Permanently** :(risorsa spostata permanentemente)
- **302 - Found** :(necessaria nuova locazione della risorsa)

4xx - [Richiesta non soddisfatta a causa del client]

- **400 - Bad Request** : (Richiesta formulata male)
- **401 - Unauthorized** : (Non ci si è identificati)
- **403 - Forbidden** : (Non si gode dei privilegi per accedere alla risorsa)
- **404 - Not Found** : (Non trovata)

5xx - [Richiesta non soddisfatta a causa del server]

- **500 - Internal server error** : (di solito un errore script)
- **501 - Not implemented** : (metodo non supportato)

Header HTTP

Header: componenti essenziali di richieste e risposte e che permettono la comunicazione client-server. Sono costituiti da una coppia chiave valore (Es. Content-Type: application/json)

Esistono 4 tipi di header:

- **Generali:**

usati sia in richieste che risposte (Es.: Cache-Control)

- **Richiesta:**

Usati nelle richieste inviate dal client al server per fornire informazioni sulle intenzioni del client o sulle condizioni in cui viene effettuata la richiesta (Es.: User-Agent, che indica il tipo di browser o client che sta inviando la richiesta)

- **Risposta:**

Inviati dal server come parte della risposta a una richiesta per fornire informazioni sullo stato della risposta o per specificare ulteriori istruzioni al client.

- **Entità:**

Usati per fornire informazioni specifiche sul corpo della richiesta o della risposta (Es.: Content-Length, che specifica la dimensione del corpo della richiesta o della risposta in byte)

Content-Type

Grazie a questo attributo, l'header fornisce istruzioni su come decodificare la risorsa inviata e su come interpretarla: "oggetto/formato".

Es. text/html, image/jpeg

Content Negotiation

Permette di servire la migliore rappresentazione di una risorsa avente differenti varianti. La migliore rappresentazione è scelta mediante:

- **Negoziazione server-driven**

il **browser** invia, insieme alla richiesta, una serie di **header** che **descrivono la scelta preferita dall'utente**. Il **server** utilizza un proprio algoritmo per **determinare la migliore rappresentazione**

- **Negoziazione agent-driven**

il **server** invia una **pagina** contenente i **link** alle **risorse alternative disponibili**, così che lo **user-agent** possa **scegliere la migliore**. **Non esiste** un **formato standard** sulla **lista** restituita dal **server**.

Supporto Chunked e Multipart

Chunked

- Dimensione nota solo alla fine del trasferimento

- Dimensione del file stimata => Grande
- Il contenuto **può** essere dinamico
- Si possono aggiungere Header al termine della richiesta
- Connessione Stateful fino al termine del trasferimento

Chunk Header

n_byte (in esadecimale) \r\n

Il chunk finale ha lunghezza 0

Chunk Trailer

\r\n

Header della richiesta

Transfer-Encoding [Obbligatorio]

Codifica il **BODY DI RICHIESTA**

- chunked
- compressed
- deflate
- gzip

Content-Encoding

Codifica la **RISORSA**

Multipart

Essenzialmente il **Content-Type** è **multipart/form-data** e serve per fare una POST da un Form.

Il body avrà un delimitatore di parte definito in **boundary**, che delimita le parti.

Permette il caricamento dei file da Form

Caching

Caching: insieme di meccanismi che permettono alle risposte HTTP (soprattutto GET) di rimanere temporaneamente memorizzate, ha una serie di effetti positivi. Può essere:

- Privato
 - Client-side
- Condiviso
 - Server-side
 - Proxy-side

Cache-Control [Header]

L'header Cache-Control permette di specificare direttive sul caching e può assumere valori come:

- **no-store**
Non viene effettuato Caching
- **no-cache**
Il caching può avvenire, rivalidazione prima di fornire la risorsa
- **public**
La risposta può essere memorizzata da qualunque cache
- **private**
La risposta è pensata per un singolo utente e può essere memorizzata solo da questo
- **max-age**
Permette di specificare (in secondi) il tempo massimo per cui una risorsa è considerata “fresca”
- **must-revalidate**
La cache deve rivalidare lo stato delle risorse scadute prima di usarle
- **expires**
Indica la data di scadenza. È **ignorato** se c'è anche **max-age**

Come avviene la Rivalidazione

Si usa un campo chiamato **E-Tag** che è una sottospecie di **firma** della risorsa. Viene **confrontato** questo per controllare se i file sono uguali.

Talvolta l'**E-Tag** contiene informazioni più importanti (Devono combaciare perfettamente) e parti meno importanti, (Possono anche non combaciare)

Connessioni HTTP 1.1

Richieste Singole

Prima di Http 1.1, le connessioni erano: 1 richiesta - 1 risorsa

In http 1.1 si può usare la stessa metodologia con [**Connection:** close] nelle richieste e risposte.

Richieste Multiple

Consiste nel fare più richieste in parallelo (fino a 6)

- Usa più banda
- Una richiesta, Un socket
- Richieste parallele
- Non è detto che il tempo di caricamento sia più veloce

Connessioni Persistenti [Connection: Keep-Alive]

Ha i seguenti parametri

- **timeout** (secondi) => Quanto posso aspettare senza risposte prima di chiudere la connessione
- **max** => Numero max di richieste dalla connessione

Purtroppo pone il rischio di attacchi DoS

Pipelining

Consiste nel mandare richieste multiple prima di una risposta.

- Solo per metodi idempotenti
- Risposte eseguite in ordine [Head of Blocking]
- Fallimento- i => Fallimento- $j \quad \forall j > i$
- Può essere anche parallelizzato
- Fa aspettare meno

Socket

Coppia indirizzo porta

Virtual Hosting

Tecnica che permette ad una singola macchina di servire risorse diverse

Virtual Hosting basato su IP

Stessa macchina, più IP

- IPv4 pubblici finiti
- Necessità di più interfacce fisiche di rete
- Usabile con http 1.0

Virtual Hosting basato su nome

Stessa macchina, più domini.

Reso possibile grazie all'introduzione dell'header **Host**

- Domini "illimitati"
- Usabile con http 1.1 e successive

Cookies

Serve per mantenere lo "stato" nelle connessioni e possiede questi determinati campi:

- **Name = Value**
- **Set-Cookie:**
 - **SameSite=Strict:** utilizzo first-party, con richieste iniziate dallo stesso sito
 - **SameSite=Lax:** utilizzo first-party, inviato anche verso siti dall'esterno
 - **SameSite=None:** utilizzo anche in third-party, con richieste XOrigin (utilizzo con **Secure**)
- **Max-Age:** definisce un cookie permanente che scade dopo tot tempo
- **Expires:** definisce un cookie permanente che scade alla data prefissata
- **Domain:** si imposta l'host di origine, per farlo valere in tutti i siti "come quello" (poliba.it -> *poliba.it)

CSFR [Si pronuncia Sea-Surf]

Tipologia di attacchi basati sull'utilizzo di cookies, consigliato:

- GET idempotenti
- Token CSFR da inviare con le POST
- SameSite=Strict per evitare XS
- Usare HTTPOnly per evitare che js acceda ai cookie

HTTPS [HttpSecure]

Crypta i dati in maniera trasparente per l'utente.
Per il trasporto ci si affida a TLS.

Esempio di connessione sicura attraverso scambi di messaggi

[Handshake avvenuto]

Client

Server

ClientHello

(Mando la suite di cifrature che supporto)

ServerHello

(Rispondo con il mio certificato)

ServerHello Done

Creo la mia chiave simmetrica

Cifro la mia chiave con la chiave pubblica del server

Invio la chiave criptata al server

Decripto la chiave **client** con la mia chiave privata

Da ora in poi comunico solo con la chiave **creata dal client**

Chain-of-Trust

Alice

Charlie

Bob

Alice conosce Bob

Charlie conosce Bob

Bob conosce Charlie e Alice

Charlie passa la sua chiave pubblica e la passa a Bob

Bob firma la chiave di Charlie con la propria chiave privata

Charlie passa la chiave firmata da Bob ad Alice

Alice, che conosce la chiave pubblica di Bob, risale alla chiave pubblica di Charlie.

Ora Alice comunica con Charlie verificando che la chiave sia giusta

Riparte la connessione sicura

Ma come avviene nella realtà?

All'inizio della catena, ci sono degli **organi** che sono **fidati** a prescindere. Da loro parte tutta la chain of trust.

Inoltre, una volta verificata l'attendibilità del sito, il sito diventa fidato per il client

Http 2.0

Rispetto a http 1.1, permette un vero multiplexing su singola connessione.

Ecco qui uno schema dei cambiamenti:

- Multiplexing su una connessione
- Trasmissione di dati compressa
- Supporto alle priorità
- Supporto al server push

Iniziare una connessione Http 2.0

Dal momento che non tutti i client lo supportano (si intende comprese anche le versioni vecchie), tutte le connessioni partono come Http 1.1 e viene negoziato un upgrade (**Upgrade:** h2c/h2 [http/https], **Connection:** Upgrade, HTTP2-Settings).

[Queste sono ancora risposte http 1.1]

Se il server accetta, manderà **101 Switching Protocols**

Se il server rifiuta, manderà **200 OK**

Connessioni e Stream

Per quanto ci sia una singola connessione, per ogni connessione possono esserci più stream.

Ogni stream ha un ID Univoco e porta i propri messages che hanno all'interno i frame di header e data.

I frame possono essere inviati in qualsiasi ordine, perché tanto vengono riassemblati dal client grazie agli ID degli Stream. Inoltre, nessun messaggio blocca un altro, avendo più stream.

Preferenze del client sul caricamento

Sono solo delle preferenze, e in quanto tali, non sono soggette ad essere onorate dal server. Tuttavia, lato server, indica le priorità di allocazione delle risorse.

Server Push

Sapendo che il client chiederà lo stesso delle richieste, il server manda un frame di tipo PUSH_PROMISE che indica al client che si manderanno risorse aggiuntive. Poi il client può accettare o meno (si rifiuta con RST_Stream). Solitamente i client tendono a rifiutare e Chrome sta pensando di rimuovere la feature. NginX ha inoltre rimosso il supporto a tale pratica.

Esercizi su HTTP

Consigli su come svolgerli:

- **Non persistente** → 2RTT per **ogni** richiesta di ulteriori file
- **Persistente** → 2RTT **solo** per l'**apertura** della connessione, RTT per richieste ulteriori
- **Senza pipelining** → Una richiesta soddisfatta dopo l'altra, il tempo è $\cdot n$
- **Con pipelining** → Richieste soddisfatte contemporaneamente, il tempo è $/n$

1) Connessione HTTP non persistente senza possibilità di utilizzare connessioni parallele

$$L_p = 500 \text{ kbit}$$

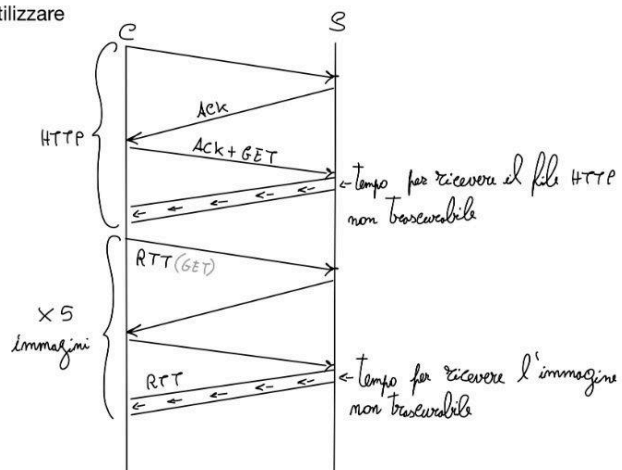
$$L_i = 100 \text{ kbit (5 immagini)}$$

$$RTT = 250 \text{ ms}$$

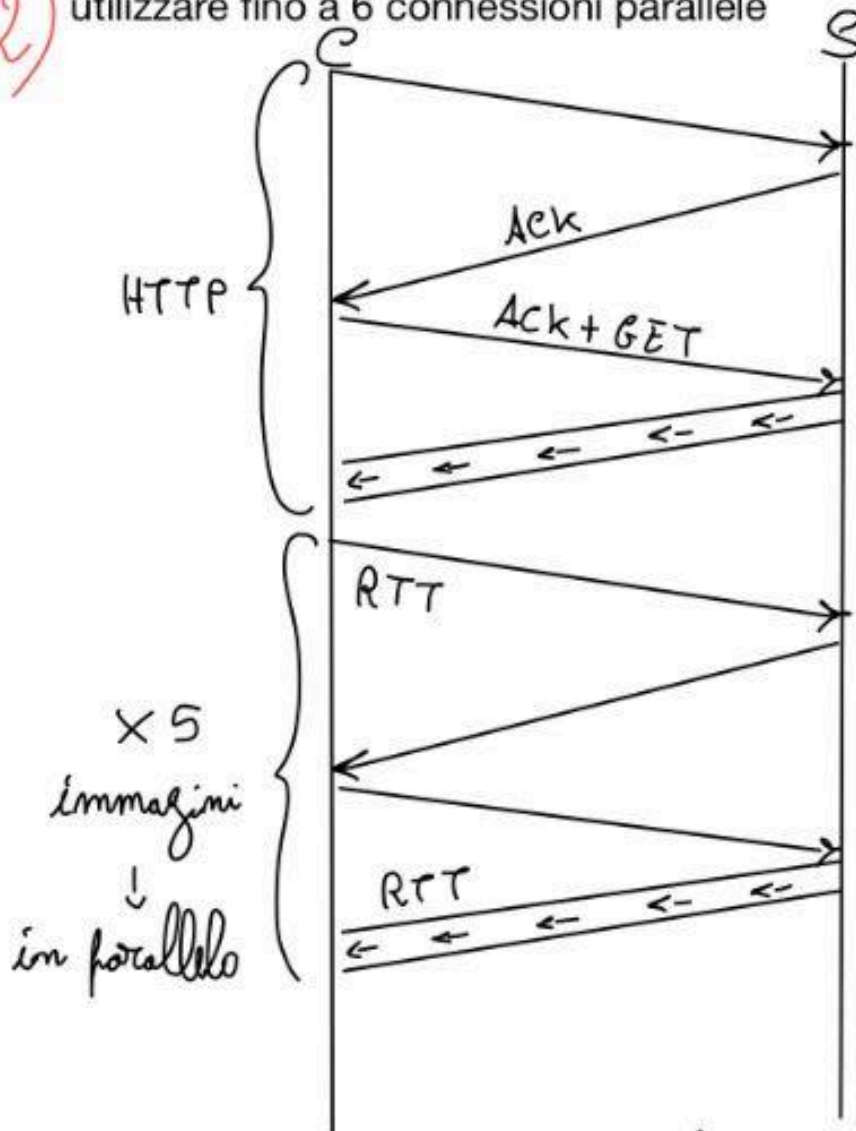
$$C = 100 \text{ Mbps}$$

$$RTT + RTT + \underbrace{T_{\text{ricezione HTML}}}_{\text{Tempo ricezione HTML}} + 5 \cdot \left(2RTT + \frac{L_i}{C} \right)$$

$$\frac{L_p}{C} = \frac{500 \text{ kb}}{100 \text{ Mbps}} = \frac{500 \cdot 10^3 \text{ b}}{100 \cdot 10^6 \text{ bps}}$$

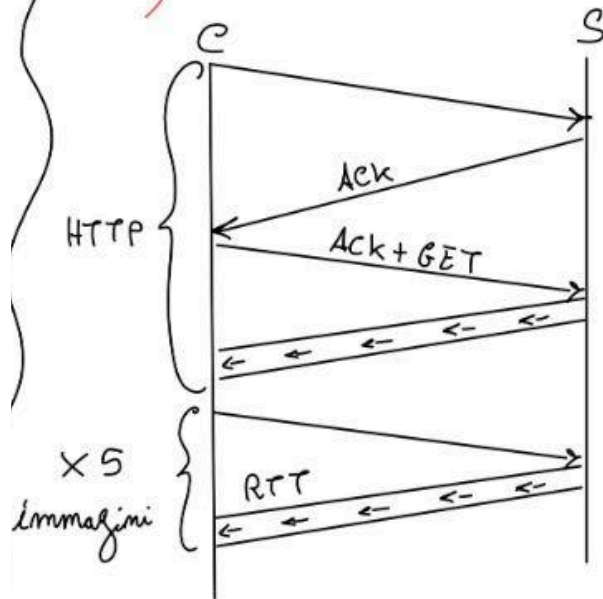


2) Connessione HTTP non persistente con possibilità di utilizzare fino a 6 connessioni parallele



$$2 \cdot RTT + T_R + T_{ML} + 2 \cdot RTT + \frac{L_i}{\frac{C}{5}}$$

3) Connessione HTTP persistente senza possibilità di utilizzare il pipelining

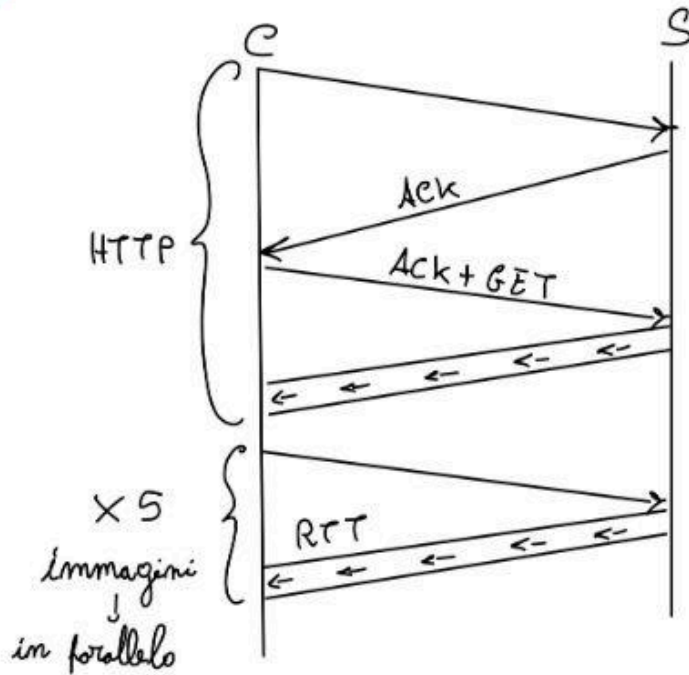


non serve richiedere
la connessione al server

$$RTT + RTT + T_{R\text{HTML}} + 5 \left(RTT + \frac{L_i}{c} \right)$$

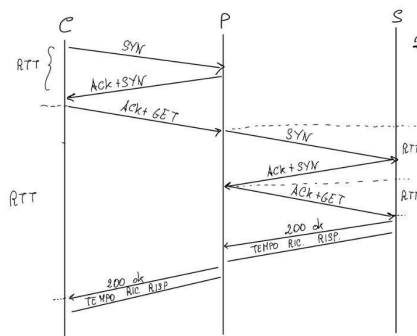
4)

Connessione HTTP persistente senza possibilità di utilizzare il pipelining



$$2 \cdot RTT + T_R + T_{ML} + RTT + \frac{L_i}{\frac{C}{5}}$$

$L_g = L_s = 100 \text{ bit}$
 $L_p = 100 \text{ kbit}$
 $RTT = 250 \text{ ms}$
 $P = 0.4$
 $C_{cp} = 1 \text{ Gbps}$
 $C_{ps} =$



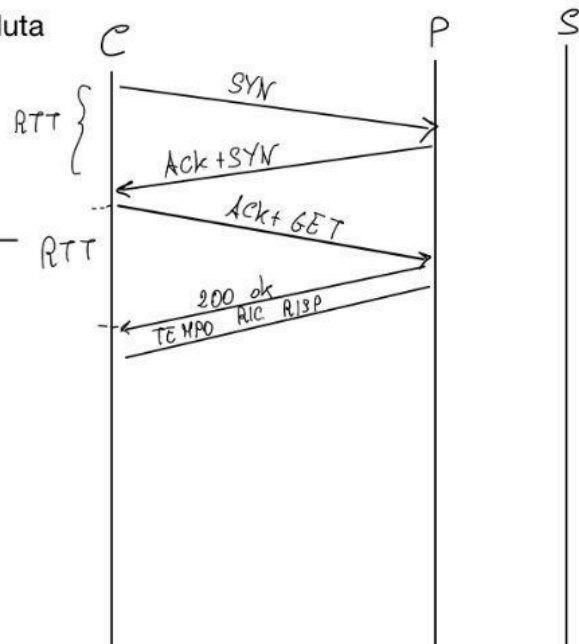
$$2 \cdot RTT + \frac{L_p}{C_{cp}} + \frac{L_g}{C_{cp}} + 2RTT + \frac{L_p}{C_{ps}} + \frac{L_g}{C_{ps}} = 4RTT + \frac{L_p}{C_{cp}} + \frac{L_p}{C_{ps}} + \frac{L_g}{C_{cp}} + \frac{L_g}{C_{ps}} = T_{16} \text{ cache}$$

la richiesta non è nel proxy
 caso generale

- 1) La risorsa contiene un header Cache-Control: must-revalidate e non è scaduta

$$T_1 = 2RTT + \frac{L_p}{C_{cp}} + \frac{L_g}{C_{cp}}$$

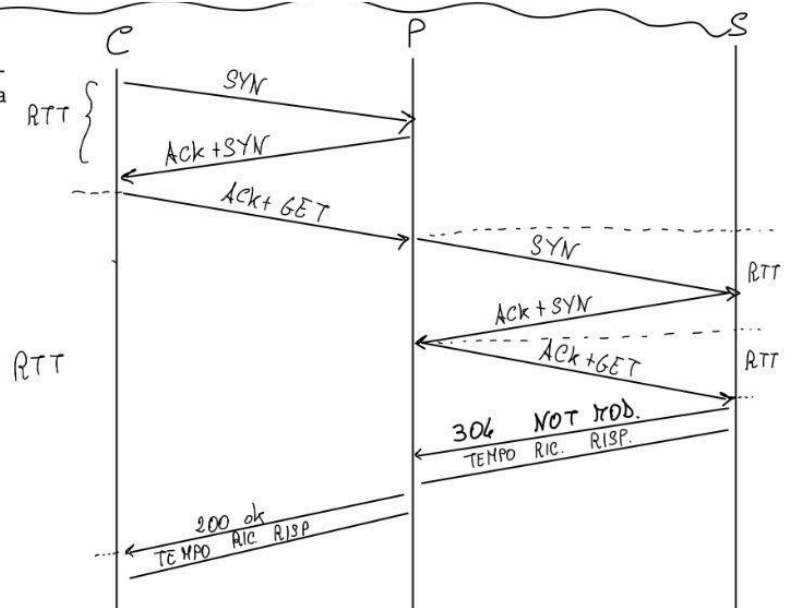
$$T_{\text{medio}} \Rightarrow \bar{T}_1 = 0.4 * T_1 + 0.6 T_{\text{Vocache}}$$



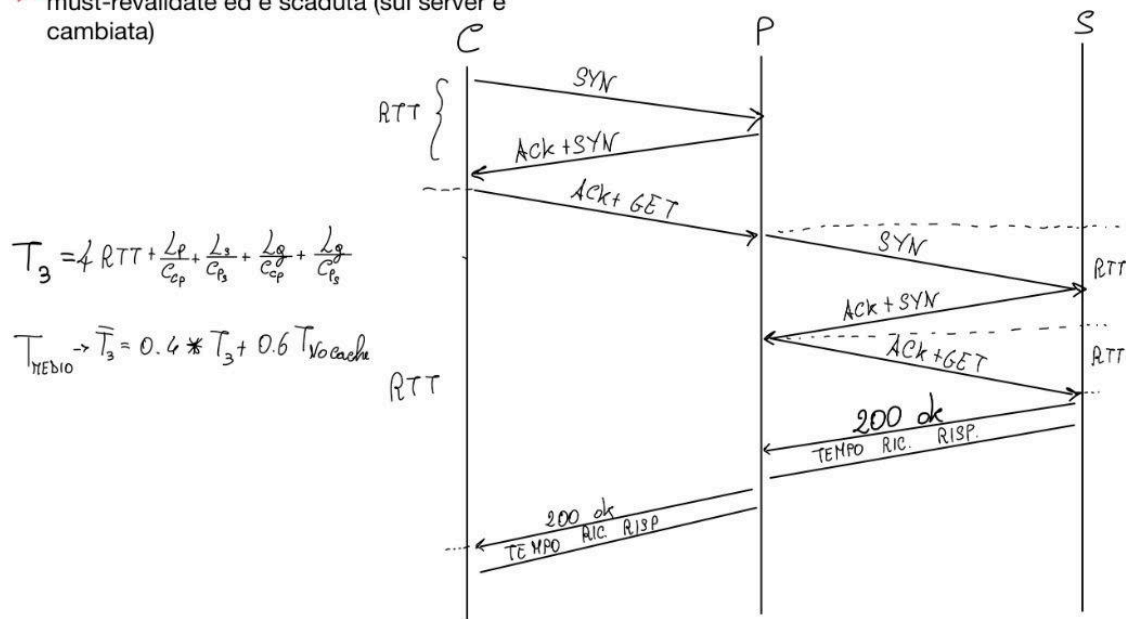
- 2) La risorsa contiene un header Cache-Control: must-revalidate ed è scaduta (sul server non è cambiata)

$$T_2 = 4RTT + \frac{L_p}{C_{cp}} + \frac{L_s}{C_{ps}} + \frac{L_g}{C_{cp}} + \frac{L_g}{C_{gs}}$$

$$T_{\text{medio}} \Rightarrow \bar{T}_2 = 0.4 * T_2 + 0.6 T_{\text{Vocache}}$$



- 3) La risorsa contiene un header Cache-Control: must-revalidate ed è scaduta (sul server è cambiata)



Capitolo 3 (HTML)

SGML

Dichiarazione SGML

Indica i caratteri consentiti e i delimitatori del linguaggio

DTD

Definisce i tag e le regole di markup. Permette la validazione dei documenti

Istanza del Documento

Contiene i dati del documento e usa il markup definito nel DTD

Elementi

Questo tag indica solo e soltanto come è composto l'elemento, ma non i suoi attributi!!!

Per ogni elemento devono essere specificati **nome**, informazioni su **STAG** e **ETAG** e **struttura del contenuto**.

[STAG => Start Tag |

ETAG => Ending tag]

- STAG E ETAG:

- obbligatorio (-)
- facoltativo (0)
- Content model:
 - ANY
 - EMPTY
 - #PCDATA
 - Altro...
- Quantificatori:
 - ? (0-1)
 - * (0 o più)
 - + (1 o più)
- Elementi logici:
 - , (AND ordinato)
 - | (OR)

Modello di esempio

```
<! Element nome_elemento - - (content_model)>
```

[\[Preso direttamente da w3.org\]](http://Preso direttamente da w3.org) //Questi sono tag reali

```
<!ELEMENT IMG - O EMPTY>
```

Attributi

Questo tag indica solo e soltanto gli attributi di un elemento, ma non come è composto!!!

Si dichiarano con ATTLIST e permettono di caratterizzare degli elementi. Composti da 3 campi:

- **Proprietà (Ovvero il nome dell'attributo)**
- **Tipo di dato**
 - CDATA
 - NUMBER
 - ID
 - *Liste*
 - *Entità*
- **Valore default**
 - #REQUIRED (Il valore deve essere specificato)
 - #FIXED *valore* (Il valore è quello provvisto e non può essere cambiato)
 - #IMPLIED (Il valore è ricavabile e non è necessario specificarlo)
 - *valore* (Valore di default che può essere cambiato)

Modello di esempio

```
<!ATTLIST TABLE
  width    CDATA    #IMPLIED
  cols     NUMBER   #IMPLIED
  align    %Talign; #IMPLIED
```

```
>
```

[\[Preso direttamente da w3.orgl](http://Preso%20direttamente%20da%20w3.orgl) //Questi sono tag reali

```
<!ATTLIST P
  id          ID          #IMPLIED  -- document-wide unique id --
  class       CDATA       #IMPLIED  -- comma list of class values
--
  style       CDATA       #IMPLIED  -- associated style info --
  title       CDATA       #IMPLIED  -- advisory title/amplification
--
  lang        NAME        #IMPLIED  -- [RFC1766] language value --
  dir         (ltr|rtl)    #IMPLIED  -- direction for weak/neutral
text --
  align       (left|center|right|justify) #IMPLIED
>
```

Entità

Questo tag non definisce né elementi, né attributi, ma delle macro (sintassi più semplice che sostituisce tutta la definizione lunga)!!!

[Basta immaginare quando chiamate una funzione chiamando il nome e non riscrivendo il corpo]

Il tag è composto da 2 entità:

- nome_entità
- valore che racchiude (vedi tipi di dato qui sopra)

Modello di esempio

```
<! ENTITY % nome_entità "valori">
```

[\[Preso direttamente da w3.orgl](http://Preso%20direttamente%20da%20w3.orgl)

//Questi sono tag reali

```
<!ENTITY % inline "#PCDATA | %font | %phrase | %special |
%formctrl">
```

[Quando voi inserirete %inline, inserirà al suo posto tutta la pappardella scritta tra le virgolette]

XML

Anch'esso **metalinguaggio di markup**, sottoinsieme **più rigido** di SGML. La creazione di linguaggi XML è basata su:

- **DTD (Document Type Definition)**
 - **Simile** al DTD di SGML, ma **più restrittivo**
 - **Più semplice da scrivere rispetto al DTD di SGML**

- impossibile definire strutture con tag omessi o violazione di di vincoli di nesting quindi, non richiede di specificare informazioni su STAG e ETAG
- Difficile la creazione di vincoli complessi
- Impossibile definire vincoli semantici
- **XML Schema**
 - Non ha corrispondenti in SGML
 - Tipizzazione forte e sintassi XML
 - Supporta i namespaces
 - Supporta datatype semplici e complessi
 - Semplici: vincoli aggiuntivi sui built-in
 - Complessi: composti ricorsivamente da semplici e complesso

Un documento si dice **ben formato** se:

- Include una **dichiarazione** del documento
- Soddisfa l'**annidamento**
- **Non include** entità **undefined**

Un documento ben formato è **valido** se:

- Soddisfa i vincoli del **DTD**

Evoluzione di HTML

2004-2008: Nasce un gruppo di lavoro alternativo a W3C, formato da Apple, Mozilla, Opera e Google. Si passa, quindi, da XHTML 1.1 ad **HTML 5** che rappresenta l'unificazione di HTML e XHTML.

Caratteristiche di HTML5:

- I documenti devono essere documenti XML ben formati
- La sintassi può essere HTML (quindi senza DTD) o XHTML
- Gestione degli errori come XHTML rimane molto rigorosa
- Gestione degli errori come HTML è definita da regole più precise
- Supporto a contenuti multimediali
- Introduzione di API

Versioni di HTML

- **1.0**
Prima proposta
- **2.0**
I moduli di rendering dei browser ora sono associati con ogni elemento HTML
- **3.2**
Inclusione di tabelle, applet, ecc...
- **4.0**
Supporto a fogli di stile, scripting, supporto per altre lingue

- **4.01**

Standard approvato con elementi ora deprecati

- **XHTML 1.0**

Case sensitive, meno tollerante e impone la tolleranza 0 sui browser

- **XHTML 1.1**

Abbandonato in favore di HTML

- **HTML 5.0...**

Nasce un gruppo di lavoro alternativo a W3C, formato da Apple, Mozilla, Opera e Google. Si passa, quindi, da XHTML 1.1 ad **HTML 5** che rappresenta l'unificazione di HTML e XHTML.

Aggiunge supporto a contenuti multimediali, Introduce API per sviluppo web app complesse

Documento HTML

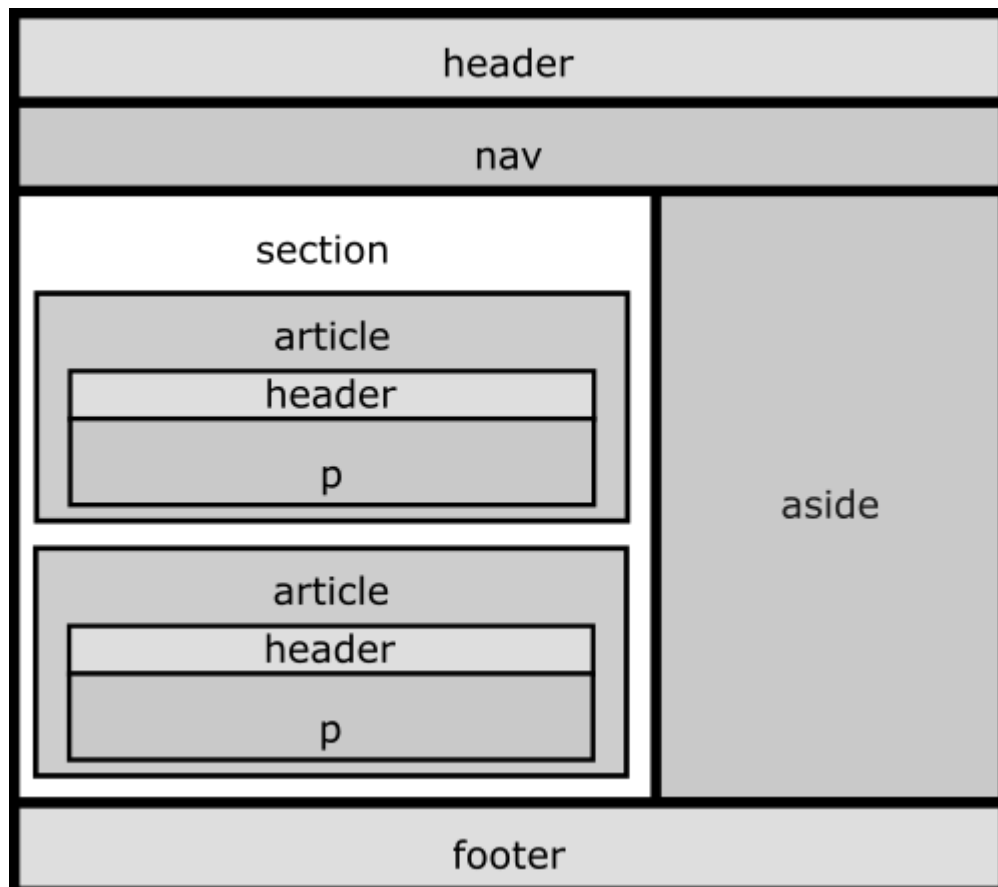
Descrizioni di contenuti interattivi, indipendenti dal modo in cui sono rappresentati.

Bisogna **slegare** la logica **semantica** da quella di **presentazione**, altrimenti:

- **Accessibilità ridotta** su dispositivi differenti
- Codice **difficile** da **manutenere**
- **Documenti** di **dimensione maggiore**
- Presenti anche alcuni tag (b, i, hr, s, small, u) con effetti di presentazione

Lo sviluppo del web ci ha portati alla creazione di un **web semantico**, in cui il **nome** di ogni elemento è **funzionale al suo scopo**. E' preferibile utilizzare gli elementi per lo scopo per cui sono stati creati, senza complicarsi la vita

Esempio di documento



Head di un documento

L'elemento **head** di un documento HTML non contiene contenuto visualizzato nel browser ma **informazioni** come titolo della pagina, icona, link a risorse esterne,...

I tag più importanti:

- <title>: specifica il **titolo** del documento
- <base>: elemento **vuoto** che permette di stabilire il **document** base URL
- <link>: specifica **relazioni** tra il **documento** e una **risorsa esterna** (JS, CSS)
- <style>: permette di **incorporare** codice CSS nel documento
- <script>: **esegue** codice **esterno**
- <meta>: permette l'inclusione di **metadati**:
 - charset: set di **caratteri** utilizzato nel documento ("utf-8")
 - Open Graph Data: **protocollo**, inventato da Facebook, che permette alle pagine di diventare rich object all'interno del grafo sociale
 - viewport: utile per la **responsività** della pagina sui diversi dispositivi

Corpo di un documento

All'interno del tag **body** vi è la parte visibile della pagina web


Tag	HTML
Titoli	<code><h1>, <h2>, <h3>, ...</code>
Paragrafo	<code><p></code>
Semantici	<code>, </code>
Lista	<code>, </code> <code></code>
Collegamento ipertestuale	<code></code>
Strutturali	<code><header>, <section>, <main>, <nav>, <footer>, <aside>, <div></code>
Form	<code><form action=" " method=" "></code> action= richiesta da eseguire method= GET, POST
Input	<code><input type=" "></code> type= text, password, hidden, checkbox, radio, file, submit, button, textarea
Oggetti multimediali	<code>, <audio src=" ">, <video src=" "</code> <code>></code>
Tabelle	<code><table></code> <code><tr>, <td>, <th></code>

Capitolo 4 (CSS)

I fogli **CSS** regolano **stile** e **layout** delle **pagine**, si occupano della parte di **presentazione**. E' possibile associare codice CSS a HTML in 3 modi:

- **Collegamento** tramite *stylesheet* (`<link rel="stylesheet" href="styles.css">`)
- Aggiungere l'attributo **style** nell'**head**
- Attributo **style** nell'**HTML**

Selettori

Selettori	
Universale	 { }

di Tipo	h1{ }
di Classe	.class { }
di ID	#id { }
di Attributo	a [title] { }
di Pseudoclasse	a:hover { }
di Pseudoelementi	a::part-of-element { }
di Discendenti	article p { }
di Figli	article > p { }
di Fratelli adiacenti	article + p { }
di Fratelli	article ~ p { }

Per eventuali **conflitti** nati dall'utilizzo di **selettori**, CSS usa l'algoritmo a **cascata**, per cui, in ordine di importanza:

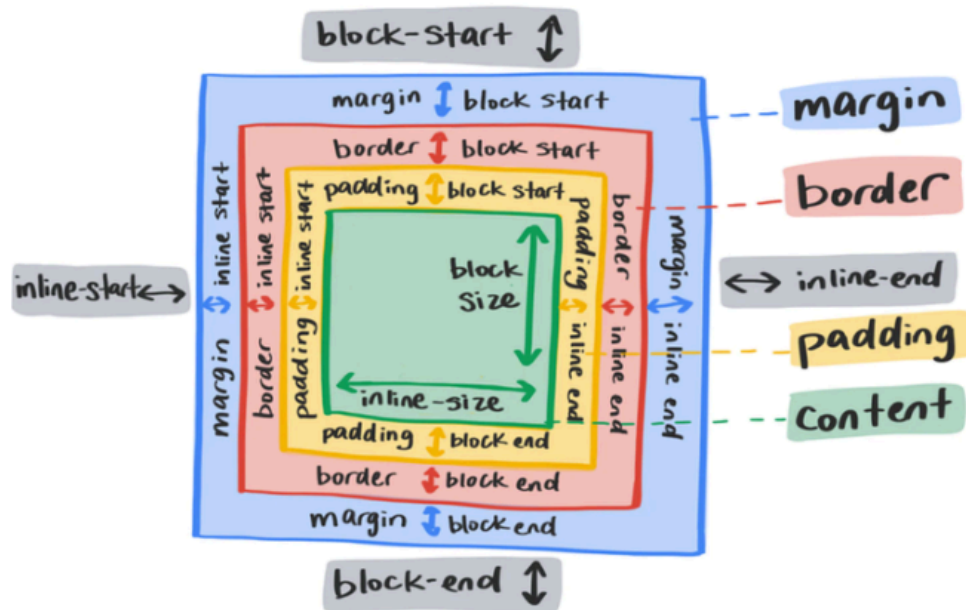
- **Regole inline:** Massima priorità
 - **Posizione e ordine:** le ultime regole hanno la priorità
 - **Specificità:** la regola più specifica ha la priorità
 - **Origine** della regola: la provenienza è prioritaria
 - Authored CSS (CSS sito)
 - User-Style (Stile OS)
 - User-Agent (Stile User-Agent)
- Nota: Se marchiato con !important, l'ordine è invertito**
- **Importanza:** se è etichettata come importante ha priorità

Ci sono delle proprietà ereditabili che vengono ereditate da tutti gli elementi annidati che possono ereditarla, ci sono 3 istruzioni per poter modificare il comportamento:

- **inherit:** eredita il comportamento dal genitore più vicino
- **initial:** comportamento standard
- **unset** si comporta come:
 - **inherit** per **proprietà ereditabile**
 - **initial** per **proprietà non ereditabile**

Box model

Ciascun elemento all'interno della pagina è considerato un box in CSS



Ciascun box nella pagina possiede due **tipi di visualizzazione**:

- **esterna**, per le interazioni con gli altri elementi
 - di tipo **block**
 - di tipo **inline**
- **interna**, per le interazioni tra elementi all'interno del blocco

Mediante la proprietà **display** è possibile alterare la visualizzazione esterna ed interna di un box

Pseudo-elementi e Pseudo-classi

Pseudo-classi interattive	
:hover	Cursore su un elemento (link)
:active	Dopo il click su un elemento
:focus	Click o comandi da tastiera
Pseudo-classi storiche	
:link	Al primo click
:visited	Ricorda che sei passato per un elemento (spesso link)
Pseudo-classi di un form	
Pseudo-classi per la selezione di elementi	

<code>:first-child</code>	
<code>:last-child</code>	
<code>:only-child</code>	
<code>:nth-child(N/even/odd/An+B)</code>	
<code>:nth-last-child</code>	
Pseudo-classi per la selezione multipla	
<code>:is(...)</code>	
<code>:not(...)</code>	

Costruzione di un layout

Di default, il browser utilizza il **flusso normale** per posizionare gli elementi HTML, ma è possibile alterare questo comportamento con la proprietà **display**.

Flexible box layout (display: flex)

Tramite questo layout è possibile modificare i figli all'interno di un box a proprio piacimento, tramite delle proprietà:

Proprietà	
<code>display:</code>	<code>flex</code> <code>inline-flex</code>
<code>flex-direction:</code>	<code>row</code> <code>row-reverse</code> <code>column</code> <code>column-reverse</code>
<code>flex-wrap:</code>	<code>nowrap</code> <code>wrap</code> <code>wrap-reverse</code>
<code>align-items:</code>	<code>flex-start</code> <code>flex-end</code> <code>center</code> <code>stretch</code> <code>baseline</code>
<code>align-self:</code>	<code>auto</code> <code>flex-start</code> <code>flex-end</code> <code>center</code> <code>stretch</code> <code>baseline</code>
<code>gap:</code> <code>row-gap:</code> <code>column-gap:</code>	<code>...px</code>
<code>align-content:</code>	<code>flex-start</code> <code>flex-end</code> <code>center</code> <code>stretch</code> <code>space-between</code> <code>space-around</code>

justify-content:	flex-start flex-end center space-between space-around space-evenly
------------------	--

Capitolo 5 (WebServer)

E' possibile definirlo come:

- **Hardware: computer** che esegue un software di web server e **conserva i file** dell'app, supporta lo **scambio di dati**
- **Software:** permette la **comunicazione** tramite HTTP a uno o più siti WEB. Deve **comprendere** le richieste e mandare risposte

Server HTTP

Insieme di **processi** e **thread** alcuni dei quali in ascolto su delle **porte**, altri dedicati alla **gestione** delle richieste. Possibile gestire i processi:

- **Generazione** di processi dopo nuove richieste
- Numero **fisso** di processi

E' inoltre possibile stabilire:

- **Massimo numero** di richieste per processo
- **Tempo massimo** di attesa per richieste

Appena stabilita una connessione persistente le risposte vengono mandate seguendo un algoritmo FIFO:

- Servono una **coda** di richieste in **input** e una **coda** di richieste in **output**
- Appena una richiesta sta per essere **elaborata** si **toglie** dalla coda in **input** e si **mette** nella coda in **output**
- Appena si **completa l'elaborazione** della richiesta viene **non** viene **rilasciata**, ma si **controlla** che tutte quelle **prima** di lei siano **uscite** dalla coda di **output**

Messaggi di richiesta

Un web server si occupa di:

- **Leggere e interpretare** un messaggio HTTP
- **Verificare la sintassi**
- **Identificare** Header HTTP conosciuti

Oltre a questo si occupa anche di:

- **Normalizzare:** percorso pulito e standardizzato (tolta /)
- **Mappare:** analisi URL per capire a cosa si riferisce
- **Tradurre:** dal percorso dell'URL occorre creare un percorso nel FS

Capitolo 6 (JavaScript)

È un linguaggio di scripting **client-side** (React) e **server-side** (Node.js ed Express) introdotto per rendere le pagine Web “attive” e dinamiche:

- permette di interagire con le pagine senza ricaricarle ad ogni azione
- fornisce interattività

ECMAScript 3 è stata la versione più utilizzata, nel 2015 sono arrivati la maggior parte dei miglioramenti con ECMAScript 6.

Attualmente è l'**unico** linguaggio di programmazione che i browser possono eseguire **nativamente**.

È utilizzato anche nella gestione di **database** come **MongoDB** che lo usano come linguaggio di scripting e query.

Caratteristiche di JS

- Linguaggio **interpretato**: tradotto al momento dell'esecuzione
- Linguaggio **debolmente tipizzato**: alcuni tipi di dato subiscono casting automatici in base alle operazioni effettuate
- Effettua **typing dinamico**: l'interprete assegna il tipo ad una variabile a runtime
- Linguaggio **orientato agli oggetti basato su prototipi**: un prototipo è esso stesso un oggetto che può essere “replicato”, cioè gli oggetti creano altri oggetti

Tipi di dato

Tipi	
Vuoti	undefined : variabile dichiarata ma non assegnata null : assenza di valore e informazione
Numeri	floating points a 64 bit Infinity Nan
Stringhe	ogni carattere è rappresentato da 16 bit in memoria
Booleani	true o false Operatore booleano : <i>sentence ? if true : if false</i>

typeof: operatore utile per scoprire il tipo di un dato

Type coercion

Operando sx	Operatore	Operatore dx	Risultato
-------------	-----------	--------------	-----------

false (booleano)	+	[] (array vuoto)	"false" (stringa)
"123" (stringa)	+	1 (numero)	"1231" (stringa)
"123" (stringa)	-	1 (numero)	122 (numero)
"123" (stringa)	-	"abc" (stringa)	Nan (numero)
[] (array vuoto)	+	[] (array vuoto)	"" (stringa vuota)
[] (array vuoto)	+	{ } (oggetto vuoto)	"[object Object]" (stringa)

Variabili e binding

Per conservare valori e cambiare lo stato di un programma, JS utilizza variabili, anche dette **binding** perché JS collega il valore all'allocazione di memoria. Il **valore** di un binding può essere **sovrascritto** con l'operatore di **assegnazione** (=).

Per avere variabili che non possono essere riassegnate è possibile utilizzare **const** anziché **let**.

!! I binding non "contengono" valori ma puntano a valori nella memoria.

Funzioni e scope

Le **funzioni** sono **parti** di **programma** che permettono di produrre un side effect o restituire valori.

Lo statement **return** stabilisce se e quali valori vengono **restituiti** dalla **funzione** (altrimenti restituiscono **undefined**)

Tipi di funzioni	
Definizione come valore del binding (la funzione di per sé è anonima)	<pre>const square = function (x) { return x * x; }</pre>
Dichiarazione come funzione	<pre>function square(x) { return x * x; }</pre>
Arrow function	<pre>const square = (x) => { return x * x; } oppure const square= x => x * x;</pre>
Dichiarazione anonima	<pre>function (x) { return x * x; }</pre>
Dichiarazione anonima arrow	<pre>(x) => x * x;</pre>

Scope della funzione: parte generalmente **delimitata** dalle **parentesi** graffe

I binding **dichiarati** all'interno di una funzione con `let` e `const` sono **locali** e visibili solo nello **scope** della funzione.

Binding dichiarati **all'esterno** di una funzione sono **globali** e accessibili **dovunque**.

Hoisting di funzioni e variabili: è il **processo** per cui le **dichiarazioni** di variabili e classi e le definizioni di funzioni vengono **spostate all'inizio** del loro scope. È **preferibile evitare** l'hoisting.

Programmazione funzionale

Viene dall'idea che un **programma** possa essere considerato come una **funzione matematica**.

Essa prevede che:

- qualunque cosa all'interno del codice sia una funzione oppure un'espressione
- non vi siano statement
- non si siano stati (variabili, oggetti, ...)

JS non implementa la prog. funzionale ma ne mutua alcune idee: funzioni come oggetti di prima classe, chiusure, funzioni anonime, ...

Le funzioni sono **first class citizen**, cioè oggetti di prima classe: le loro definizioni rappresentano "valori" che possono essere **passati** come **argomenti** ad altre funzioni (**callback**), possono essere **restituiti** come valori da altre funzioni e possono essere **inseriti** in qualunque **struttura**.

Meccanismo di **Closure**: permette ad un **oggetto** funzione di "**racchiudere**" le variabili accessibili nel suo scope quando era stata definita, mantenendone un **riferimento**.

!! Funzioni che **accettano** come argomenti altre **funzioni** sono dette funzioni di **ordine superiore** (es. `forEach()`, `map()`, ...).

Oggetti, prototipi e classi

Gli **oggetti** in JS sono **collezioni** arbitrarie di **proprietà**, più precisamente **array** associativi **mutabili** le cui **chiavi** rappresentano i **nomi** delle proprietà.

I **valori** delle proprietà sono **accessibili** mediante la notazione `obj.prop`.

Gli **oggetti** possono **possedere** dei **metodi** ovvero delle **proprietà** contenenti funzioni. In JS non c'è distinzione tra funzione e metodo.

La parola chiave **this**, se utilizzata all'interno del metodo, viene **associata all'oggetto** in cui essa **compare**.

Il comportamento del **this** all'interno delle funzioni dipende dal modo in cui la funzione è chiamata, ovvero dal suo **contesto di esecuzione**.

Array

Questo **oggetto** permette di **collezionare** una **sequenza** (non associativa) di valori. Si tratta di oggetti **ridimensionabili** e che possono contenere differenti **tipi** di **dati**. Essi sono **indicizzabili** a partire da 0.

Metodi per iterare un array:

- `for () { }`
- `forEach()`
- `map()` che anziché effettuare operazioni per ogni elemento dell'array (come il `forEach()`), ci permette di **ottenere** un **nuovo array** i cui elementi sono il **risultato** della **mappa** applicata su **ogni elemento** dell'array originale.

Altri metodi di ordine superiore di cui dispongono gli array:

- `filter()`, per **ottenere** un **array** che contiene **solo** gli **elementi** che “**passano**” la funzione predicato
- `reduce()`, **combina** tutti gli **elementi** dell'array in un **unico valore**
- `findIndex()`, per **trovare l'indice** degli elementi che **soddisfano** una certa **funzione** predicato

Tutti prendono come argomento **elemento**, **indice**, **array**; tranne `reduce()` che ha in più un riferimento al valore precedente.

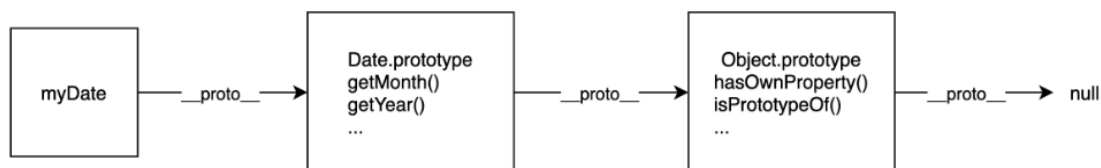
Oggetti vs dati primitivi

Un **dato** primitivo **non può cambiare**, il **contenuto** di un oggetto invece **può cambiare**. È possibile **cambiare** il **valore** del **dato** primitivo a cui è **associato** un **binding**, ma **non** è il **valore** in sé a cambiare.

Tutti i **valori** primitivi in JS hanno un **oggetto equivalente** che svolge il ruolo di **wrapper** per quel valore primitivo. Es le stringhe sono associate all'oggetto `String`.

È possibile **ottenere** il **valore primitivo** di un oggetto wrapper mediante il metodo `valueOf()`.

Ogni oggetto in JS deriva da un proprio **prototipo** da cui “**eredita**” tutte le **proprietà** e i **metodi**. Esiste un **prototipo globale** **Object** che ha **null** come prototipo.



`getPrototypeOf` ci permette di ottenere il prototipo di un oggetto, equivalente a `__proto__`

Metodi per creare oggetti

1. **Creare nuovi oggetti** usando un altro oggetto **come prototipo**, mediante il metodo `Object.create()`
2. **Creare oggetti** utilizzando la **funzione costruttore**, tutte le funzioni possono essere costruttore, **ricevono** infatti la **proprietà** `prototype` che si può **sovrascrivere** o a

cui si possono **aggiungere** altre proprietà. Utilizzando la **keyword** `new` davanti al nome di una funzione, quest'ultima sarà trattata come costruttore.

3. Dal 2015 JS è stato **esteso** con le **classi ES2015** che si basano sempre sul **concetto** di **prototipo**, lo **semplificano** nella notazione e lo estendono nelle potenzialità. Anche parlando di “classi” si tratta sempre di prototipi e funzioni

Esempi:

Usando le classi ES2015 il costruttore sarà tipo:

```
class Speaker {
  constructor(type) {
    this.type = type;
    this.speak = this.speak.bind(this);
  }

  speak() {
    console.log(`Sono ${this.type}!`);
  }
}
```

Altri metodi utili

Overriding: Aggiungendo una **proprietà** ad un oggetto, che sia essa presente o meno nel prototipo, la proprietà viene aggiunta al solo oggetto e “scollegata” da quella del prototipo es.

`Speaker.prototype.country = “Italy”` => aggiunge una proprietà al prototipo, quindi a tutte le istanze

`humanSpeaker.country = “France”` => modifica solo alcune istanze del prototipo

Ereditarietà: ciascun oggetto **deriva** le **proprietà** dal suo **prototipo**, se una proprietà non esiste in un oggetto viene cercata nel suo prototipo, **ereditarietà prototipale**)

Getter, setter e metodi statici

- **Getter:** metodi che permettono di **ottenere risultati** di una computazione come se essi fossero una proprietà dell'oggetto.
- **Setter:** metodi che permettono di utilizzare un metodo **impostando** un **valore** come se esso fosse una proprietà dell'oggetto.
- **Statici:** metodi che permettono di **realizzare** metodi utilizzabili **solo** sulla classe **stessa**, non su una sua istanza.

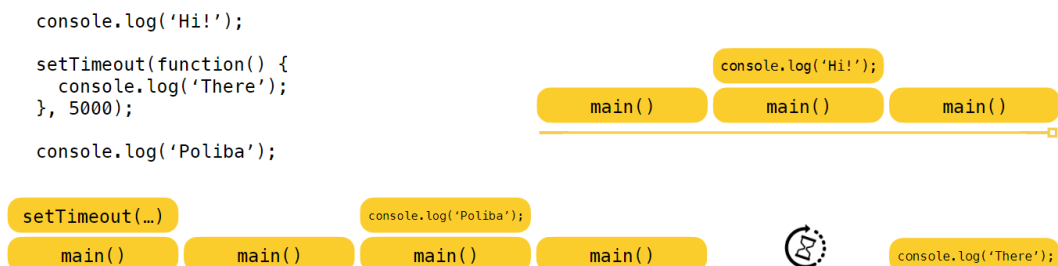
Programmazione asincrona

Un **modello di programmazione asincrona** permette alle **azioni** di avvenire **contemporaneamente**. Quando un'azione è avviata, il programma **continua** la sua **esecuzione** e sarà poi **informato** della **fine** dell'esecuzione dell'azione.

JS è un **linguaggio** di programmazione **single-threaded**, con un **unico** call stack, quindi può fare **solo** una cosa per volta.

Esistono funzioni che hanno bisogno di **tempi** più **lunghi** per produrre un **risultato** (es. richieste al server), queste **bloccherebbero** l'esecuzione dell'intero **programma** finchè non si **conclude** la funzione, ha quindi un comportamento **bloccante** sul programma.

Una **soluzione** è quindi l'**utilizzo** della programmazione **asincrona**, una possibile strategia è quella di chiamare una funzione "lenta" e passarle una **callback** che venga eseguita al termine.



Dove finiscono le funzioni “lente” in esecuzione?

JS è **eseguito all'interno** di un **browser**, il quale è **multi-threaded** e **multi-process** e mette a disposizione nella sua implementazione delle **WebApi** a cui poter fare delle chiamate.

Una volta terminata, la callback viene inserita nella **coda dei task**, intanto JS **continua** la sua **esecuzione** nella call stack, se la call stack è vuota viene eseguito il primo elemento della coda dei task e si ripete.

Promise pattern

Il promise pattern è la **fondazione** della programmazione **asincrona** moderna in JS. Questo **pattern** parte dall'assunto di avere a disposizione degli **oggetti** che possono **rappresentare** il **valore pendente** di un'**operazione** asincrona. Una funzione asincrona quindi **restituisce** una **promise**, cioè un oggetto che rappresenta l'**esito** della sua **esecuzione**, anche se fisicamente ancora non esiste, con la quale è possibile **decidere** cosa farci una volta **ottenuto** l'esito.

Stati di una promise

- **Resolved**: quando il **valore** che rappresenta diviene **disponibile**, cioè quando l'attività asincrona restituisce un valore.

- **Rejected**: quando l'attività asincrona associata **non** restituisce un **valore** o perché si è verificata un'**eccezione** o perché il valore restituito **non** è considerato **valido**.
- **Pending**: quando **non** è né **risolta** né **rigettata**, cioè la richiesta è partita ma non si è ancora ricevuto un risultato.

Creazione di una promise

Il **costruttore** dell'oggetto **Promise** prevede un **parametro** che rappresenta il **promise handler**: è una **funzione** che viene **invocata** immediatamente e che **riceve** a sua volta due **funzioni** da invocare rispettivamente per **risolvere** la promise a un valore o per **rigettarla**.

Uso di promise

Per **lavorare** sul risultato di una promise è possibile **utilizzare** il metodo **then** che prevede due argomenti: il primo è un **resolve handler** che sarà eseguito nel caso di promise risolta, il secondo un **reject handler** (opzionale) che è una funzione che sarà eseguita nel caso di promise rigettata.

```
httpGet('url/to/fetch') PROMISE
  .then(value => console.log('Tutto ok: ' + value), RESOLVE HANDLER
        error => console.log('Si è verificato un errore')) REJECT HANDLER
```

!! then restituisce un'altra **promise**, che **risolve** al valore che la handler function restituisce oppure, se quest'ultima restituisce una promise, aspetta quella promise e poi risolve al suo valore. Questo approccio è chiamato **promise chaining** in quanto realizza una catena di promise.

Mentre il metodo **then** **accetta** un **resolve** handler, il metodo **catch** **accetta** un **reject** handler. La **rejection** di una promise viene **propagata** alla **promise** generata dal **then**, che viene a sua volta **rigettata** e così via, è possibile quindi pensare di **posizionare** un **solo** metodo **catch** **in fondo** alla catena.

È possibile utilizzare il metodo **Promise.all()**, la cui promise è **risolta solo** quando **tutte le promise** all'interno dell'array passatogli come argomento sono **risolte**.

async/await

Queste due parole chiave permettono di semplificare la sintassi del codice asincrono realizzando una struttura tipica del codice sincrono.

- **async** permette di dichiarare una **funzione** come **asincrona**.
- **await** permette di **sospendere** un'esecuzione in attesa che la promise associata ad un'attività asincrona venga risolta o rigettata.

<pre>function getUtente(userId) { httpGet("/utente/" + userId) .then(response => { console.log(response); }).catch(error => console.log("Errore!"))); }</pre>	<pre>async function getUtente(userId) { try { let response = await httpGet("/utente/" + userId); console.log(response); } catch (e) { console.log("Si è verificato un errore!"); } }</pre>
--	--

Capitolo 7(Web API)

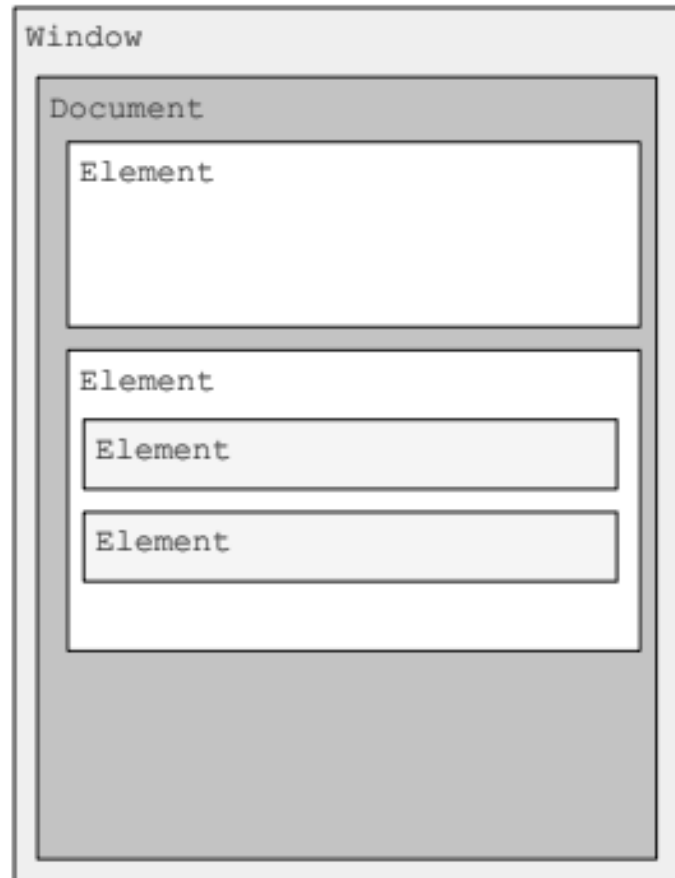
Api

Costrutti che permettono di fare **operazioni complesse** in modo **semplice** e astraendo. Sono presenti molte **funzionalità** non native ma **implementate** da terzi.

- **Browser**: Funzionalità offerte dal browser, implementate a basso livello (l'audio in uscita con un video)
- **Third-party**: Funzionalità di altri servizi (twitter che permette la visualizzazione di alcuni tweet embeddando su un sito)

Browser API	Third-Party API
HTML DOM	Twitter
CSSOM	Youtube
XMLHttpRequest	Maps
Fetch	Flickr
Push	Facebook
Notifications	
WebSocket	
Web Audio	

Il **browser** usa il **DOM (Document Object Model)** per **rappresentare** pagine HTML. Contiene **nodi** di diverso tipo, un nodo **Document** che ne rappresenta la **principale interfaccia** da cui **discendono** tutti gli altri elementi. Con **javascript** possiamo **interrogare** o **modificare** il document



La API HTML DOM permette di:

- Navigare e modificare la gerarchia nodi nel DOM
- Agire su un nodo del DOM
- Agire su attributi di un elemento HTML
- Creare nuovi nodi nel DOM

Proprietà	Cosa fa
<code>Element.innerHTML</code>	Setta o ritorna il contenuto di un elemento
<code>Element.outerHTML</code>	Setta o ritorna il contenuto di un elemento
<code>Element.getAttribute()</code>	Prende il valore dell'attributo di un elemento
<code>Element.setAttribute()</code>	Imposta o cambia il valore di un attributo
Proprietà	Output
<code>getElementsByTagName("")</code>	Vettore con elementi con quel tag
<code>getElementById("")</code>	Unico elemento con quell'ID
<code>getElementsByClassName("")</code>	Vettore con elementi di quella classe
<code>querySelectorAll("")</code>	Query con tutti gli elementi con quei selettori CSS
<code>querySelector("")</code>	Query con il primo elemento con quel selettore CSS
<code>document.createElement("")</code>	Crea un elemento di tipo specificato
<code>document.createAttribute("")</code>	Crea un attributo di tipo specificato
<code>parent.appendChild()</code>	Crea un figlio e lo appende
<code>parent.removeChild()</code>	Toglie il figlio da un elemento
<code>console.log("greve")</code>	Stampa su terminale
<code>alert("greve")</code>	Apre una mini finestra nel browser
<code>confirm("greve")</code>	Apre una mini finestra nel browser con i tasti Ok e Cancel
<code>setTimeout(cb, t)</code>	Esegue la callback dopo t secondi, solo una volta
<code>setInterval(cb, t)</code>	Esegue la callback dopo t secondi, ripetutamente

NB.: Essendo programmazione asincrona non ci sono certezze sui tempi, pertanto le callback invocate con `setTimeout` e `setInterval` potrebbero non rispettare il tempo `t`

Eventi nel Browser

Si **aggancia** ad un oggetto un event Listener (o Handler), del tipo
`button.addEventListener(...)`

All'interno si metterà una arrow function che sarà eseguita non appena l'elemento in questione avrà fatto ciò che è scritto, in questo modo:

Proprietà	Cosa fa
<code>onchange</code>	Al cambiare del valore di un elemento HTML
<code>onclick</code>	Al click su di un elemento HTML
<code>onmouseover</code>	Al mouse sopra un elemento HTML
<code>onmouseout</code>	Al mouse che si sposta da un elemento HTML (e ci era sopra)
<code>onkeydown</code>	Al click di un tasto della tastiera
<code>onload</code>	Appena il browser carica la pagina

È possibile **prevenire** che un'azione sia trattata come farebbe di **default**:

```
event.preventDefault();
```

Si **aggancia** questo metodo ad un **evento** ed **eviterà** il comportamento che avrebbe normalmente

Propagazione degli eventi

Avviene in 3 fasi:

- **Capturing**: viene **recuperato** l'elemento più ancestrale ed eseguito il suo handler, poi si entra **all'interno**, fino al genitore dell'elemento **target**
- **Target**: **invocazione** dell'handler dell'elemento **target**
- **Bubbling**: viene **propagato** l'evento verso l'**esterno**, fino a quando c'è un oggetto con la proprietà **bubbles** a **false**

Ajax

Tecnica per fare **richieste asincrone** HTTP ad un server per **ottenere** nuove risorse.

Permette di:

- **Scaricare** i dati necessari per **aggiornare** una **pagina** web
- **Aggiornare** solo le parti del DOM interessati **senza** ricaricare la pagina

Fondamentalmente con la richiesta HTTP si **scarica** un **oggetto** (.JSON) che contiene tutte le informazioni **aggiornate** su un certo modello.

Fetch API

Interfaccia usata per fare richieste HTTP, alternativa a XMLHttpRequest, fa uso di Promise.

La chiamata restituisce una promise, che risolve ad un oggetto Response, contenente informazioni sulla risposta del server. Il **contenuto** di una risposta può essere trovato mediante il metodo **text()** o **json()**.

Richiesta Get

```
fetch("example/data.txt").then(response => {  
    response.text();  
}).then(text => console.log(text));
```

Richiesta Post

```
fetch(url, {  
    method: "post",  
    headers: new Headers({  
        "Content-Type": "application/json"  
    }),  
    body: JSON.stringify({  
        titolo: "Un articolo",  
        autore: "Mario Rossi"  
    })  
}).then(...)
```

CORS

Per CORS (**C**ross-**O**rigine **R**esource **S**haring) intendiamo un **meccanismo** per cui possiamo **prendere** risorse al di **fuori** del sito stesso. Ci **potrebbero** essere delle **restrizioni** da parte dei browser.

Altre API

WebSocket

Tecnologia che permette di **aprire** una connessione **full-duplex** tra browser e server. Serve a **comunicare senza** effettuare **polling**. Vantaggi:

- Comunicazione a **bassa latenza** (rispetto ad HTTP)
- Lavora su **TCP**

Capitolo 8 (Web application architecture)

Cos'è?

È un **modello architetturale** che **definisce** la **struttura** ad alto livello di una web app. Molto importante perché ci permette di **strutturare un'applicazione** che sia scalabile, affidabile, sicura e performante.

In parole povere ci permette di **capire** come **comporre** la nostra **app** spostando o modificando FE, BE e Database in modo da essere efficiente.

Struttura

- Insieme di componenti
- logica di interazione

Vantaggi di usare un'architettura web (Ricordalo come SSCRIV)

- Sicurezza
- Scalabilità
- Chiarezza
- Riusabilità
- Velocità(Performance)

Prima di affrontare le architetture definisco alcuni concetti:

- Tier: Separazione logica e fisica dei componenti dell'app
- Business logic = Backend

Tipi di architetture

- Single tier

Fe, Be e DB risiedono tutti nella stessa macchina

VANTAGGI

- Dati sempre pronti
- Privacy utente maggiore
- Nessuna latenza di rete

SVANTAGGI

- Calo di performance
- Il produttore non ha il pieno controllo della macchina
- Reverse engineering

- Two tier

Architettura client-server

- Fe Be lato client insieme
- Database lato server

VANTAGGI

- Poche chiamate al server
- Molto economico

SVANTAGGI

- Codice vulnerabile
- **Three tier e N-tier**

Fe, Be e DB separati.

VANTAGGI

- Rispettano i principi di:
 - Single Responsibility
 - Separation of concerns

SVANTAGGI

- Costi più elevati

Le applicazioni **multi-tier** permettono di rispettare i **principi di progettazione**:

Single responsibility:

- ogni componente ha una sua responsabilità. Questo permette di andare a lavorare su un componente senza toccare le funzionalità di un altro componente.

Separation of concerns

- ogni componente deve essere incapsulato per essere isolati l'un l'altro, di modo che i componenti devono poter effettuare implementazioni interne senza interrompere i collaboratori.

REST API (REpresentational State Transfer)

Stile architetturale per sistemi distribuiti (N-tier). Rappresentano un **gateway** per una web app dove i client fanno tutte le richieste, indipendentemente dal tipo di richiesta.

Un'architettura segue il paradigma REST (RESTful) se:

- È client-server
- Richieste stateless
- Possibilità di cachare le risorse
- L'architettura del sistema a strati
- Interfaccia di comunicazione semplice ed uniforme

In un'architettura REST, l'elemento fondamentale è rappresentato dalle **risorse**, tutto può essere identificato mediante una risorsa. Esse possono essere di qualsiasi tipo, es. JSON o XML. L'identificatore di una risorsa deve fornire una maniera **unica, non variabile e inequivocabile** per accedervi (URI).

REST utilizza HTTP per compiere una serie di **azioni**. L'insieme base di azioni che deve mettere a disposizione un sistema resource-oriented sono le azioni **CRUD**:

- Create
- Retrieve
- Update
- Delete

Esse sono implementate attraverso il **mapping** seguente, che è uno standard all'interno della comunità:

- **GET**, accesso alla risorsa in modalità lettura (retrieve)
- **POST**, invio di una nuova risorsa per la sua creazione (create)
- **PUT** (update), **DELETE** (delete), **HEADER**, **OPTIONS**

Operazioni più complesse possono essere mappate mediante variazioni dell'URI.

Load balancing

Permette ad **un'applicazione** di **scalare** bene e rimanere **disponibile anche** in caso di grandi quantità di **traffico**.

I **load balancer** permettono di **distribuire** il traffico fra i **diversi server** di un cluster **mediante** differenti **algoritmi**, al fine di **ottimizzare l'utilizzo** delle risorse. Viene quindi inserito tra client e server (rientra infatti nei proxy).

DNS load balancing

Uno dei **metodi** più **semplici** per suddividere il **carico** fra i diversi data center, Implementato a livello DNS sull'**authoritative server**, permette di **restituire** una **lista di IP** di un particolare dominio. I **client** che **ricevono** la lista **richiedono** le **risorse** al primo della lista e usano gli IP successivi in caso di fallimento delle richieste precedenti.

Il suo **limite** è che non conosce e **non tiene in conto lo stato corrente** di ciascun server di destinazione.

Load balancing hardware

Sono **macchine performanti** posizionate davanti ai server, che **distribuiscono** il carico in base a dei criteri. Queste macchine richiedono **grande e complessa gestione e** manutenzione.

Load balancing software

Questi **software** possono essere **eseguiti** su commodity server o **macchine virtuali**. Essi **permettono** di effettuare **valutazioni avanzate** su un numero elevato di parametri per **distribuire** il carico, permettono anche di effettuare **health check** sui server per mantenere una lista aggiornata dei server attivi.

Architetture monolitiche

In un'architettura monolitica, tutti i servizi di un'applicazione sono **strettamente collegati** in **un'unica** codebase.

Un'applicazione monolitica è semplice da creare, testare e distribuire (inizialmente). Esse risultano utili in contesti estremamente semplici.

Contro:

- difficile continuous deployment
- unico point of failure
- limiti di scalabilità

- difficile uso di diverse tecnologie e linguaggi

Architetture a microservizi

In questa architettura, **feature diverse** di un grande servizio vengono pubblicate **separatamente** come servizi più piccoli debolmente accoppiati, chiamati **microservizi**, i quali lavorano insieme.

Quest'architettura rappresenta a pieno i principi di **singola responsabilità** e **separazione dei concerns**.

Pro:

- facile manutenzione dell'app
- facile sviluppo di nuove features
- semplici test e deployment individuale di moduli
- non vi è un singolo point of failure
- è possibile effettuare continuous deployment

Contro:

- grandi sforzi per garantire la **consistenza** fra i nodi
- logging distribuito
- comunicazione fra servizi:
 - Richieste HTTP, utili per le richieste di dati o effettuare modifiche in maniera sincrona
 - Code di messaggi, utili per azioni asincrone dove per l'utente non conta "immediatamente" se l'azione è andata a buon fine o meno

Tipologie di Web App

Server-side rendering

Un web server si occupa di **realizzare una pagina HTML** ed **inviarla** al **client** che **interpreta** solamente l'HTML che riceve.

Questo meccanismo è utile per **siti web statici** e può funzionare anche se il browser ha disabilitato JS.

Rappresenta un **enorme carico** sul server.

Static-side generation

Questo meccanismo **coinvolge** l'uso di un **generatore** che automatizza la **codifica** di pagine HTML, creandole partendo da un template.

La pagina HTML è **precedentemente generata** e **non** deve essere **rigenerata** ad **ogni richiesta**

Questo approccio è **utile** per **siti web**, ma non applicazioni con contenuto estremamente dinamico, infatti ad ogni nuovo contenuto il sito web deve essere nuovamente **rigenerato**.

Es. Github Pages.

È molto **più veloce** del Server-side.

Single page application

Le **SPA** lavorano **all'interno del browser** e **non** richiedono il **reload** di una **pagina** per mostrare nuovi dati.

Permettono di costruire un'applicazione web **altamente interattiva** mediante API di comunicazione, che possono essere utilizzate anche da altre applicazioni.

Le comunicazioni fra SPA e server avvengono mediante AJAX o WebSocket.

Es. Facebook, Gmail. Twitter.

Micro frontend

I micro frontend rappresentano **componenti debolmente accoppiati del frontend** di un'applicazione.

Ogni team può realizzare i componenti di frontend in maniera separata per poi renderli disponibili per l'integrazione con gli altri.

Una volta realizzati i micro frontend, la loro **integrazione** può avvenire secondo diverse strategie:

- Integrazione client-side
- Integrazione server-side

Progressive Web App

Applicazioni che hanno il **feel di app native** e che possono essere eseguite in browser desktop e mobile.

Con le PWA, le aziende possono offrire applicazioni e l'esperienza di app native eseguite direttamente nel browser che possono **interagire con l'hardware e l'OS del dispositivo**.

Queste app utilizzano l'**Api Service Worker**, che permette di creare un'esperienza online.

Capitolo 9 (Node.js + express)

Node.js

È un

- **runtime JavaScript**: è un programma scritto in C++ che **legge** ed **esegue** del **codice JS** (grazie all'engine V8 senza necessità di un browser) ed effettua **compilazione JIT** (Just In Time)
- **guidato da eventi**: una volta lanciata, l'app rimane in **ascolto** di specifici **eventi** e **reagisce** mediante **callback**
- **asincroni**: Node.js completa i task in maniera asincrona all'interno di un event loop eseguito su un unico thread grazie a primitive di I/O che **evitano** operazioni **bloccanti** passandole al kernel di sistema. Questo permette a Node.js di gestire migliaia di connessioni concorrenti con un singolo thread senza doverne gestire la concorrenza

Include il supporto alle **Node.js API**

Event Loop

Permette a Node.js di effettuare operazioni I/O non bloccanti (nonostante JS sia single thread)

Durante la **fase di poll** se:

- la **coda è vuota** rimane in attesa che venga aggiunta una nuova callback alla coda
- la **coda NON è vuota** l'event loop eseguirà le callback in maniera sincrona fino a svuotarla

Se ci sono istruzioni del tipo:

- `setImmediate()` -> vengono eseguite immediatamente durante la **fase di check**
- `setTimeout()` -> vengono eseguite le callback specificate nei timer
- I **timer** specificano la soglia minima dopo cui una callback deve essere eseguita

Installazione e configurazione

È platform independent ed è possibile gestire più installazioni di Node.js mediante NVM (Node Version Manager)

```
nvm list
nvm ls-remote
nvm install 9.3.0
node -v
```

Con il comando `node` è possibile avvisare la versione REPL (Read-Evaluate-Print-Loop) nella shell prova:

- `.load`
- `.save`
- `.editor`
- `.exit`

Per interpretare un file JS si usa il comando `node file.js` oppure `node file`

Modularità e creazione package

Per la gestione dei moduli Node.js utilizza la strategia di **CommonJS**

Diversi moduli possono essere organizzati in pacchetti con **NPM** (Node Package Manager)

`exports.message` oppure `exports:` definisce un oggetto che può aggiungere proprietà da esportare

Per gestire i pacchetti di default per Node.js si usa `npm` che permette di installare e definire dipendenze

Crea (nella root) un file `packages.json` che specifica tutte le dipendenze e gli script che vengono eseguiti con `npm run <comando>`

```
npm init
npm install <package> [--save]
```

```
npm run <comando>
npm run start
npm start
npm stop
```

Modulo process

Fornisce una serie di metodi e proprietà utili per la gestione del processo in esecuzione, non necessita di essere importato (fa parte del core di Node.js)

`process.exit`: serve a terminare il processo

`process.env.VAR_NAME`: permette di ottenere la variabile d'ambiente **VAR_NAME**

`process.argv`: permette di ottenere in un array gli argomenti passati allo script. I primi due elementi sono:

- il path di Node.js
- il path dello script

Impostare una variabile d'ambiente:

- quando lanciamo l'applicazione:
 - `USER_ID = 239482 USER_KEY = foobar node app.js`
- se le variabili sono impostate in un file `.env` nella root del progetto
 - `require('dotenv').config()`

Web Server

```
const port = 3000; //solitamente utilizzata per i server di sviluppo
const http = require("http"); //http è una libreria standard che introduce un
supporto first-class per il networking
const httpStatus = require("http-status-codes");
const app = http.createServer((req, res) => {
  console.log("Received a request!");
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html"
  });
  res.write("<h1>Hello, World!</h1>");
  res.end();
  console.log("Risposta inviata!");
})
app.listen(port, () => {
  console.log(`The server has started listening on port
${port}`)
});
```

Creiamo un event handler per l'evento request sull'oggetto app

La sintassi per realizzare un event handler in Node.js è `EventEmitter.on("evento", callback)`

Un evento di tipo EventEmitter dispone del metodo *emit* per generare un evento e realizzare quindi il pattern listener/emitter

```
const app = http.createServer();
app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html"
  });
  res.end("<h1>Hello, World!</h1>");
  console.log("Risposta inviata!");
})
app.listen(port, () => {
  console.log(`The server has started listening on port
${port}`)
});
```

L'oggetto di risposta è un'istanza di `http.ServerResponse` i cui metodi sono:

- `writeHead(status, headers)`
- `setHeader('headername', value).write()`
- **per inviare buffered data e** `end([data])`

L'oggetto della richiesta è una istanza di `http.IncomingMessage`.

Questo oggetto estende la classe `stream.Readable`, ovvero la richiesta HTTP arriva al server come stream di dati

Per leggere il corpo di una richiesta sarà necessario mettersi in ascolto è necessario mettersi in ascolto degli eventi

- `data` per la ricezione di nuovi dati
- `end` per la ricezione del segnale di fine dati

Il corpo di una richiesta POST arriva al server in **chunk** (in pezzi) pertanto può essere gestita così:

```
app.on("request", (req, res) => {
  const body = [];
  req.on("data", bodyData => {
    body.push(bodyData);
  });
  req.on("end", () => {
    body = Buffer.concat(body).toString();
    console.log(body);
  });
});
```

Proprietà:

- `req.method`
- `req.url`

- req.headers

La classe **Buffer** permette di manipolare dati binari come un array di byte ed è utilizzata (per esempio) come risposta di molti metodi di lettura dei file

Routing

Con il termine **routing** si riferisce alla determinazione di come un'applicazione risponde ad una particolare richiesta ad un certo endpoint (URL) e per un certo metodo

Esempio su come **richiamare file .html** in *views/* mediante il loro nome dopo l'hostname

```
const fs = require('fs');
const getViewUrl = (url) => `views${url}.html`;
http.createServer((req, res) => {
  fs.readFile(getViewUrl(req.url), (error, data) => {
    if (error) {
      res.writeHead(404);
    } else {
      res.writeHead(200, {
        "Content-Type": "text/html"
      });
      res.write(data);
    }
    res.end();
  });
});
```

Il modulo **fs** fornisce una serie di funzionalità per accedere e interagire con il **filesystem**

Curiosità: la libreria *bluebird* permette di “promisificare” un oggetto

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));
```

Realizziamo un modulo `router.js` che ci permette di registrare una serie di route e relative azioni all'interno di un dizionario e che si occupi di gestirle

```
routes = {'GET': {}, 'POST': {}};
exports.handle = (req, res) => {
  try {
    if (routes[req.method][req.url]) {
      routes[req.method][req.url](req, res);
    } else {
      res.writeHead(404);
    }
  } catch (ex) {
```

```

        console.log(ex);
    }
}
exports.get = (url, action) =>
    routes['GET'][req.url] = action;
exports.post = (url, action) =>
    routes['POST'][req.url] = action;

```

Framework (express.js)

Sono di alto livello (a differenza delle API che sono di basso livello) e permettono di superare alcune problematiche comuni di sviluppo come metodi e moduli per semplificare la gestione di richieste con diversi metodi HTTP, il serving di contenuti statici e dinamici mediante template, connessione di DB, ecc...

Express.js è il più utilizzato ed offre:

- **Gestori per le richieste:** effettuate con diversi metodi HTTP a differenti URL (routing)
- Integrazione con **rendering engine** per la realizzazione di viste mediante template
- Inserimento di **middleware** per l'elaborazione addizionale di richieste in qualunque momento della pipeline per la loro gestione

Express è minimalista ma gli sviluppatori hanno creato middleware compatibili per la gestione di qualunque problema di sviluppo

Metodi di Express.js

Metodo	Cosa fa
<code>app.get(nome)</code>	Restituisce il valore di <code>nome</code> app setting, dove <code>nome</code> è una delle stringhe nella tabella delle impostazioni dell'app
<code>app.get(path, callback [, callback ...])</code>	Instrada le richieste HTTP GET al percorso specificato con le funzioni di callback specificate.
<code>app.post(path, callback [, callback ...])</code>	Instrada le richieste HTTP POST al percorso specificato con le funzioni di callback specificate.

<code>app.put(path, callback [, callback ...])</code>	Instrada le richieste HTTP PUT al percorso specificato con le funzioni di callback specificate.
<code>app.delete(path, callback [, callback ...])</code>	Instrada le richieste HTTP DELETE al percorso specificato con le funzioni di callback specificate.
<code>app.all(path, callback [, callback ...])</code>	Questo metodo è simile ai metodi <code>app.METHOD()</code> standard, tranne per il fatto che corrisponde a tutti i verbi HTTP.
<code>req.params</code>	Questa proprietà è un oggetto contenente proprietà mappate ai 'parametri' della rotta denominata. Ad esempio, se si dispone della route <code>/user/:name</code> , la proprietà 'name' è disponibile come <code>req.params.name</code> . L'impostazione predefinita di questo oggetto è <code>{}</code> .
<code>req.body</code>	Contiene coppie chiave-valore di dati inviati nel corpo della richiesta. Per impostazione predefinita, non è definito e viene popolato quando si utilizza un middleware di analisi del corpo come <code>express.json()</code> o <code>express.urlencoded()</code> .
<code>req.url</code>	Non è una proprietà Express nativa, è ereditata dal modulo <code>http</code> di Node.
<code>req.originalUrl</code>	Questa proprietà è molto simile a <code>req.url</code> ; tuttavia, conserva l'URL della richiesta originale, consentendo di riscrivere <code>req.url</code> liberamente per scopi di instradamento interno.
<code>req.query</code>	Questa proprietà è un oggetto contenente una proprietà per ogni parametro della stringa di query nella route. Quando il parser di query è impostato su <ul style="list-style-type: none"> - disabilitato, è un oggetto vuoto <code>{}</code>, - altrimenti è il risultato del parser di query configurato.
<code>res.send([body])</code>	Invia la risposta HTTP. Il parametro <code>body</code> può essere un oggetto Buffer, una stringa, un oggetto, un valore booleano o un array.
<code>res.write()</code>	

<code>res.end()</code>	<p>Termina il processo di risposta.</p> <p>Questo metodo in realtà proviene da Node core, in particolare il metodo <code>response.end()</code> di <code>http.ServerResponse</code>.</p> <p>Utilizzato per terminare rapidamente la risposta senza dati.</p> <p>Se devi rispondere con i dati, usa invece metodi come <code>res.send()</code> e <code>res.json()</code>.</p>
------------------------	---

Pattern MVC

È la **strategia** principale da utilizzare all'interno delle web app Express, essa consiste nello spostare le funzioni di callback all'interno di **moduli separati** che riflettono gli scopi di tali funzioni. I moduli si differenziano in 3 moduli principali:

- **Model**
- **View**
- **Controller**

Capitolo 10 (React)

Frontend e React

React **permette** mediante lo pseudolinguaggio JSX la **creazione** di **elementi** React ed è possibile **incorporare** espressioni JS con `{}`

!! Elementi del DOM HTML non sono elementi React !!

- `ReactDOM.render()` permette di inserire il codice JSX del codice da realizzare

React è **eseguibile** nel **browser** grazie al **transpiling** (conversione del codice sorgente di un linguaggio di programmazione ad un altro) di **Babel** ed è preferibile farlo offline usando Node.js

- `npm init -y`
- `npm install babel-cli@6 babel-preset-react-app@3`
- `npx babel -watch src -out-dir . -presets react-app/prod`

Esistono varie **toolchain** (insieme di strumenti di sviluppo software utilizzati contemporaneamente per completare complesse attività di sviluppo software o per fornire un prodotto software) come per esempio:

- Create React App (utilizza **Babel** per il trainspiling e **webpack** per creare un pacchetto di asset utilizzabile nel browser)
 - `npx create-react-app nome-app`
 - `cd nome-app`
 - `npm start`
 - `npm run build` (per il deployment di un'applicazione creata con Create React App)

È possibile **diversificare** ciascuna **istanza** mediante le sue **proprietà** che si possono cambiare mediante l'oggetto **props** che può essere ricevuto come argomento della funzione che renderizza il componente

Liste e chiavi

Se le **informazioni** relative alle info card sono **scaricate** da **API** o sono all'interno di un file **JSON** si può **mappare** l'array con **componenti** di React che ci permette di renderizzare **direttamente** l'array di componenti

Best practice

Associare ogni **elemento**/componente della lista un **attributo key** fra i diversi sibling (con almeno un parente in comune).

Componenti

I **componenti** sono formati da

- dati (possono essere visti come **proprietà**): non possono cambiare
- **stato**: può cambiare

Per tenere conto dello stato attuale si usa una **variabile di stato** (e di un suo **setter**) che:

- Conservi i dati fra due rendering
- Ogni volta che viene settata a un nuovo valore, viene triggerato il re-rendering del componente
-

Hook

Speciali funzioni che permettono di “agganciarsi” a particolari feature di React durante il rendering

La funzione `useState` è un React Hook che permette aggiungere una variabile di stato a un componente

- Valori di ritorno:
 - 1° Stato corrente che nel primo render corrisponde al valore iniziale passato alla funzione
 - 2° Funzione setter che permette di cambiare lo stato corrente

Se gli stati contengono array oppure oggetti bisogna creare un **nuovo** oggetto e **passarlo** alla funzione setter

```
const [position, setPosition] = useState({x: 0, y: 0})
setPosition({ x: 10, y: 0})
```

!! Errore fare `position.x = 10` !!

`setPosition({...position, x: 10})` Per cambiare solo alcune proprietà

`filter`, `map` o `slice` possono essere usati per ritornare gli array, oppure con lo `spread operator`

!!Non usare gli hook in loop o statement condizionali!!

Rendering condizionale

Tecnica molto utilizzata specialmente quando è coinvolto lo stato di un componente

Form

Il **valore** degli elementi del form viene mantenuto all'interno dello **stato** del componente che lo realizza (permette di rispettare il principio di **Single Source Of Truth** (Unica Fonte Attendibile))

Lo **stato** verrà aggiornato ad ogni modifica dei valori del form

Per **inoltrare** un form si può aggiungere l'evento **submit** sul form e agire con un handler asincrono

Problemi:

- **checkbox**: attributo **checked** al posto di **value**
- **radio button**: la proprietà **checked** proviene dal valore dello stato, si può continuare ad utilizzare lo stesso handler

Consiglio per realizzare una UI interattiva

- 1) Identificare gli stati "visivi" del componente
- 2) Determinare cosa può far cambiare gli stati (es: input dell'utente, invio del form, ricezione di una risposta)
- 3) Rappresenta lo stato in memoria mediante useState
- 4) Connetti gli event handler allo stato

Sincronizzazione con gli Effect

Gli **Effect** permettono di eseguire codice subito dopo il primo rendering e dopo il re-rendering relativo ad alcune dipendenze.

Di default, vengono eseguiti dopo ogni rendering

```
function Componente() {  
  useEffect(() => {  
    //Codice  
  });  
  return <div />;  
}
```

La maggior parte degli Effect devono essere eseguiti solo in alcuni casi:

- Esempio:
 - Animazione fade-in (dissolvenza) deve essere eseguita una sola volta
 - Connessione ad una chat deve essere eseguita ogni volta che si cambia "stanza" della chat
- Soluzione:
 - Usare un secondo argomento che specifica le props o le variabili di stato il cui cambiamento deve triggerare l'Effect
 - `useEffect(() => {...}, [state1, state2, props4])`

File redatto da:

Vincenzo Pio Florio

Christian Risi

Marco Roberto

Antonio Sansonne

Michele Scarciglia

Amalia Montemurro