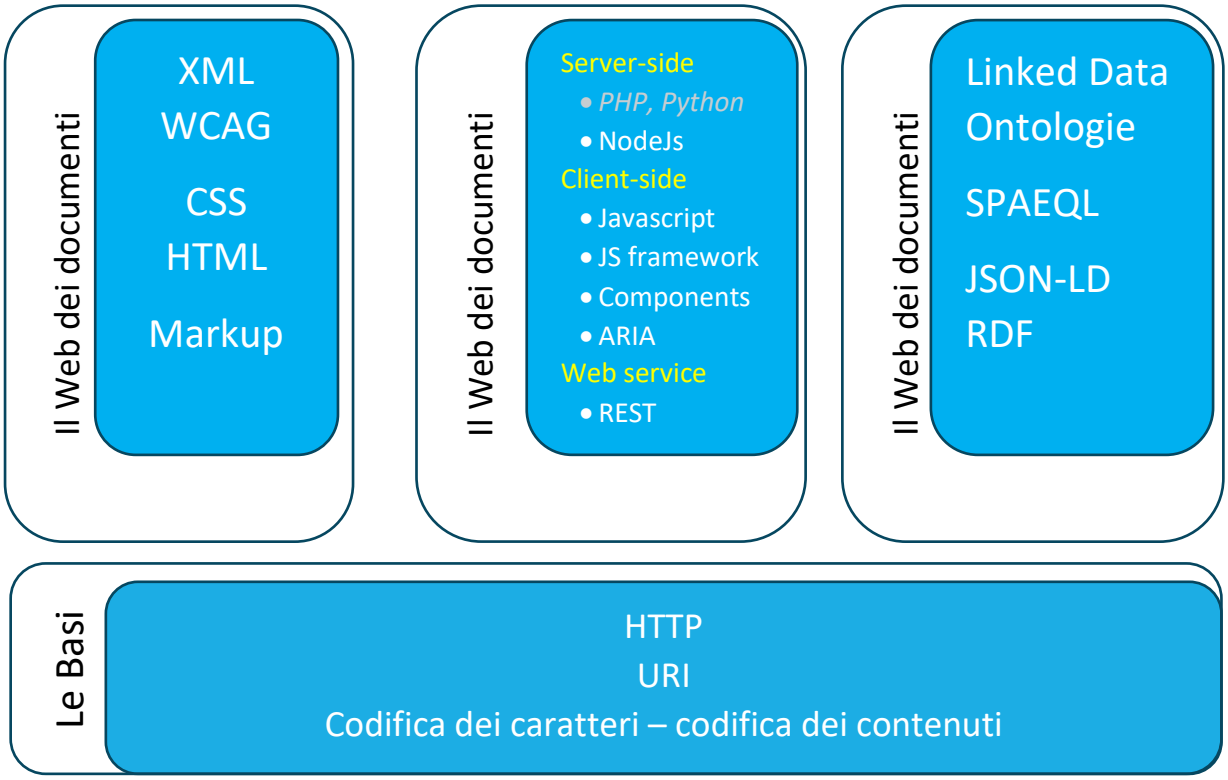
Tecnologie Web

88566 - 9CFU

0 SOMMARIO

1 INTRODUZIONE

1.1.1 La forchetta di internet:



1.1.2 Strumenti per lo sviluppatore web

1.1.2.1 *Framework*

I framework sono librerie per gli scopi più disparati, facilmente mescolabili tra loro

1.1.2.2 *API*

Application Programming Interfaces
Sono meccanismi di manipolazione delle strutture dati fondamentali e accesso ad algoritmi più sofisticati per applicazioni sviluppate dai clienti

- API REST
  restful API, sfruttano la natura HTTP e degli URI per fornire servizi
  Una API RESTful fornisce:
  - URI base per ottenere servizi
  - sintassi URI
  - media type per ottenere/dare dati da utilizzare (XML, JSON, etc…)
- API servizi locali
  API che accede a servizi speciali dei device che ne sono forniti, come telecamere, microfoni, suoni, vibrazioni, GPS, etc…

  Ci sono 80 API diverse utilizzabili sulla maggior parte dei browser

Questi strumenti permettono la sofisticazione, velocità di sviluppo e componibilità che sarebbero umanamente impossibili.

1.2 World Wide Web

Il world wide web è un sistema per la presentazione a schermo di documenti per la navigazione

È basato sui concetti di:

- Client
  - Strumento di visualizzazione dei documenti
- Plug-ins e un linguaggio di programmazione interno (JavaScript)
  Permettono di visualizzare formati speciali e di realizzare applicazioni autosufficienti
- Servers
  Meccanismi di accesso a risorse locali che trasmettono documenti
- Applicazioni server-side
  Rendono il browser l'interfaccia dell'applicazione

I protocolli alla base del www sono

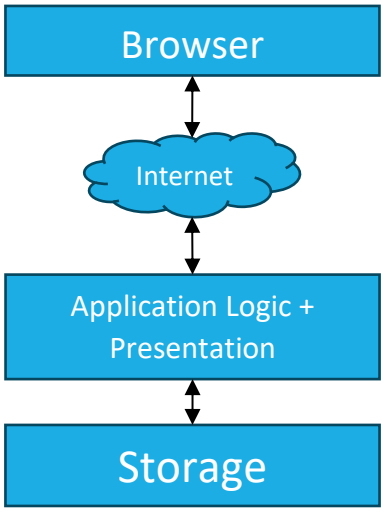
- standard (identifica risorse di rete generali)
- protocollo di comunicazione state-less e client server (accesso a risorse ipertestuali, HTTP)
- linguaggio per la realizzazione di documenti ipertestuali (HTML, XHTML)

1.3 Architettura del WWW

Il sito web **statico** è un semplice server contenente file fisici che risponde a richieste di visualizzazione da parte di client, nulla di più, nulla di meno.

Il sito web **dinamico** è strutturato su più livelli e comprende un template e una logica

1.3.1 Modelli a tre livelli



- In parte statica, la parte dinamica è generata in output dall'applicazione server-side.
- Raccoglie dati su un DBMS (DataBase Management System) che elabora e spedisce come risposta al browser
- L'application logic si basa sull'usufruire della velocità e funzionalità del DataBase, riguarda l'embedded code e la full application

Embedded code:

- Il sito è un'applicazione, e c'è un file per ogni funzionalità
- I file HTML contengono commenti e codice di qualche linguaggio di programmazione

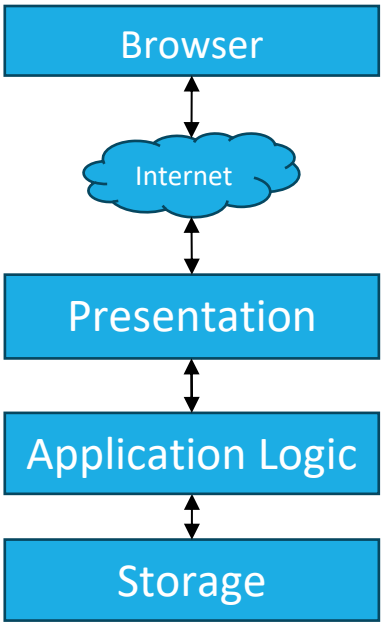
```
<HTML>
    <head><title>PHP Test</title></head>
    <body>
        <p>Stai usando il browser
            <!-- php echo $_SERVER['HTTP_USER_AGENT']; -->
        </p>
    </body>
</HTML>
```

Full application:

- Forte separazione tra logica e presentazione dell'app
- Alla fine viene comunque generato un solo output in HTML

1.3.2 Modelli a quattro livelli

Il livello logico genera un output privo di presentazioni (es in XML) e lo passa al livello superiore. La presentazione modifica completamente il documento creando un template. Template diversi possono applicarti allo stesso output (in base al device).



1.3.3 Ajax e il web 2.0

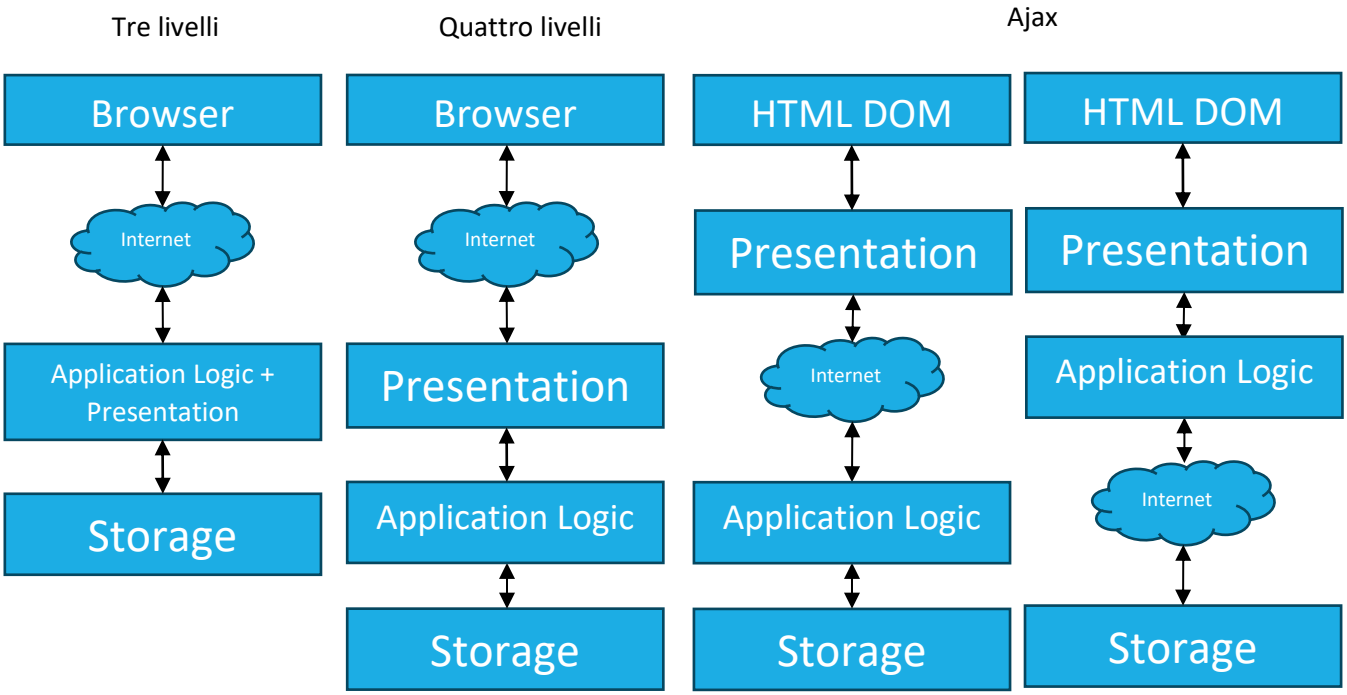
Le librerie Ajax hanno lo scopo di fornire una lista di funzioni comuni e indipendenti dal sistema operativo e dal browser, e a realizzare app client side che siano semplici.
Le librerie vengono caricate all'esecuzione del servizio, su una pagina HTML che contiene codice JS, quando il programma parte effettua una XMLHttpRequest (una chiamata di libreria) chiedendo al server i dati indipendenti da presentazione e li converte in HTML da visualizzare
Sul server esiste solo il minimo per ottenere risposte alle query, il cliente ha si *presentation* che *application*.

I **framework** sono librerie che rendono più ricco, sofisticato e semplice l'uso di una tecnologia (nel nostro caso un linguaggio server/client side o delle specifiche grafiche)

I framework hanno semplificato drasticamente la programmazione a tre livelli

Le **Single-page web applications** sono siti web composti da un unico complesso documento in HTML, reso possibile grazie alla velocità di rete, librerie di Ajax e i framework.

Le **Progressive web applications** sono applicazioni basate su tecnologie web ma che sembrino native sul device installato



2 CODIFICA DEI CARATTERI

Digitalizzare un dato significa associarvi un numero per identificarlo e rappresentarne le caratteristiche più importanti.
Per digitalizzare un dato complesso (un testo) possiamo identificarne e digitalizzarne separatamente ogni componente scendendo fino all'entità atomica (il singolo carattere per il testo, o il pixel per l'immagine).
Assegniamo a ogni carattere un numero univoco, e la digitalizzazione dell'oggetto complessivo sarà data dalla giustapposizione dei valori associati alle varie lettere.

Problema delle lingue, lingue diverse = (a volte) alfabeti diversi quindi un gran casino per riuscire a gestirli tutti.

2.1.1 Lo spazio di rappresentazione

Regole importanti della codifica:

- Ordine
  i valori seguono l'ordine alfabetico conosciuto
- Contiguità
  non ci sono intervalli non utilizzati di codifiche (se uso da `000` a `111` ogni combinazione nel mezzo è utilizzata)
- Raggruppamento
  si raggruppa logicamente tramite considerazioni numeriche (ad esempio i caratteri maiuscoli hanno la forma `10xxxxxx` in ASCII)

uno **shift** è un codice riservato che cambia mappa

codici liberi, codici non associati a nessun carattere, la loro presenza è probabilmente un errore di trasmissione

i **codici di controllo** sono associati alla trasmissione e non al messaggio

2.1.2 Notazioni binarie ed esadecimali

La rappresentazione binaria ed esadecimale si basano su un sistema numerale posizionale
La stessa matematica si applica indipendentemente dal sistema numerale, cambia solo la rappresentazione.

Decimal	Binary	Hex
14 + 25 = 39	0b001110 + 0b011001 = 0b00111	0x0E + 0x19 = 0x27
14 = (1*10^1 + 4*10^0) +	(0*2^5+0*2^4+1*2^3+1*2^2+1*2^1+0*2^0) +	0x0E = (0*16^1 + 14*16^0) +
25 = (2*10^1 + 5*10^0) =	(0*2^5+1*2^4+1*2^3+0*2^2+0*2^1+1*2^0) =	0x19 = (1*16^1 + 9*16^0) =
39 = (3*10^1 + 9*10^0)	**(0*2^5+0*2^4+1*2^3+1*2^2+1*2^1+0*2^0)**	**0x27 = (2*16^1 + 7*16^0)**

Poiché 16 = 2^4, c'è una connessione diretta tra rappresentazione binaria e esadecimale dei numeri secondo cui ogni blocco di 4 cifre binarie (bit) corrisponde ad una cifra esadecimale.
Quindi un byte composto da 8 cifre binarie è composto da 2 cifre esadecimali:

- Per esempio, 0b11001110 può essere diviso in 1100-1110,
  e poiché 0b1100 = 0xC e 0b1110 = 0xE ⇒ 0b11001110 = 0xCE

Recap matematica binaria

- 1 bit: 2^1 combinazioni: 2 valori
- 2 bit: 2^2 combinazioni: 4 valori
- 4 bit: 2^4 combinazioni: 16 valori
- 5 bit: 2^5 combinazioni: 32 valori
- 6 bit: 2^6 combinazioni: 64 valori
- 7 bit: 2^7 combinazioni: 128 valori
- 8 bit: 2^8 combinazioni: 256 valori
- 16 bit: 2^{16} combinazioni: 65536 valori o 65k valori
- 32 bit: 2^{32} combinazioni: 4.294.967.296 = 4G valori

2.2 UNICODE E ISO/IEC 10646

Esistono dozzine di codifiche a 8 bit per alfabeti non latini e alcune codifiche a 16 bit per linguaggi orientali. A seconda della codifica usata, posso avere dozzine di interpretazioni diverse per lo stesso codice numerico. Quindi devo ricorrere a meccanismi indipendenti dal flusso per specificare il tipo di codifica usata come:

- dichiarazioni esterne
- intestazioni interne
- interpretazione di default delle applicazioni usate

Ancora più difficile è il caso di flussi misti (tipo un testo italo-arabo) perché è necessario adottare meccanismi di shift da una codifica all'altra che devono dipendere dall'applicazione usata.

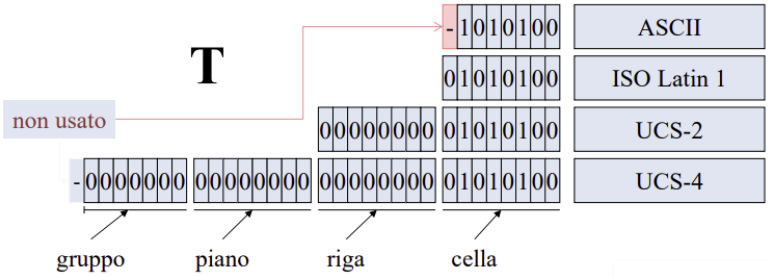
Unicode è la risposta all'esigenza della codifica universale e rispetta alcuni principi

- *Repertorio universale*
  → tutti i caratteri di tutti gli alfabeti
- *Efficienza*
  → minimo uso di memoria, massima velocità di parsing
- *Caratteri, non glifi*
  → i font sono esclusi dalla specifica del codice
- *Semantica*
  → ogni carattere ha il suo significato preciso, la ß tedesca è diversa dalla β greca
- *Testo semplice*
  → non ci sono descrizioni grafiche o tipografiche
- *Ordine logico*
- *Unificazione*
  → caratteri comuni vengono unificati in un singolo codice
- *Composizione dinamica*
  → ideogrammi o caratteri composti sono rappresentati in frammenti indipendenti e vengono ricreati per composizione
- *Stabilità*
  → una volta aggiunti, i codici sono immutabili
- *Convertibilità*
  → esiste un meccanismo di conversione per comodità

2.2.1 UCS-2 e UCS-4

ISO 10646 è composto di due schemi di codifica.

- UCS-2 è uno schema a 2 byte.
  È un'estensione di ISO Latin 1.
- UCS-4 è uno schema a 31 bit in 4 byte
  È un'estensione di UCS-2.
  È diviso in gruppi, piani, righe e celle.



In UCS-4 esistono dunque 32.768 piani di 65.536 caratteri ciascuno.
Il primo piano, o piano 0, è noto come BMP (Basic Multilingual Plane) ed è ovviamente equivalente a UCS-2.

Attualmente sono definiti caratteri solo nei seguenti piani:

- Piano 0 (BMP o Basic Multilingual Plane): alfabeti moderni
- Piano 1 (SMP o Supplementary Multilingual Plane): alfabeti antichi
- Piano 2 (SIP o Supplementary Ideographic Plane): ulteriori caratteri ideografici CJK
- Piano 3 (TIP o Tertiary Ideographic Plane): caratteri cinesi antichi
- Piano 14 (SSP o Supplementary Special-purpose Plane): Caratteri tag, in disuso
- Piano 15 e 16 Private Use Areas

2.2.2 Da UCS a UTF

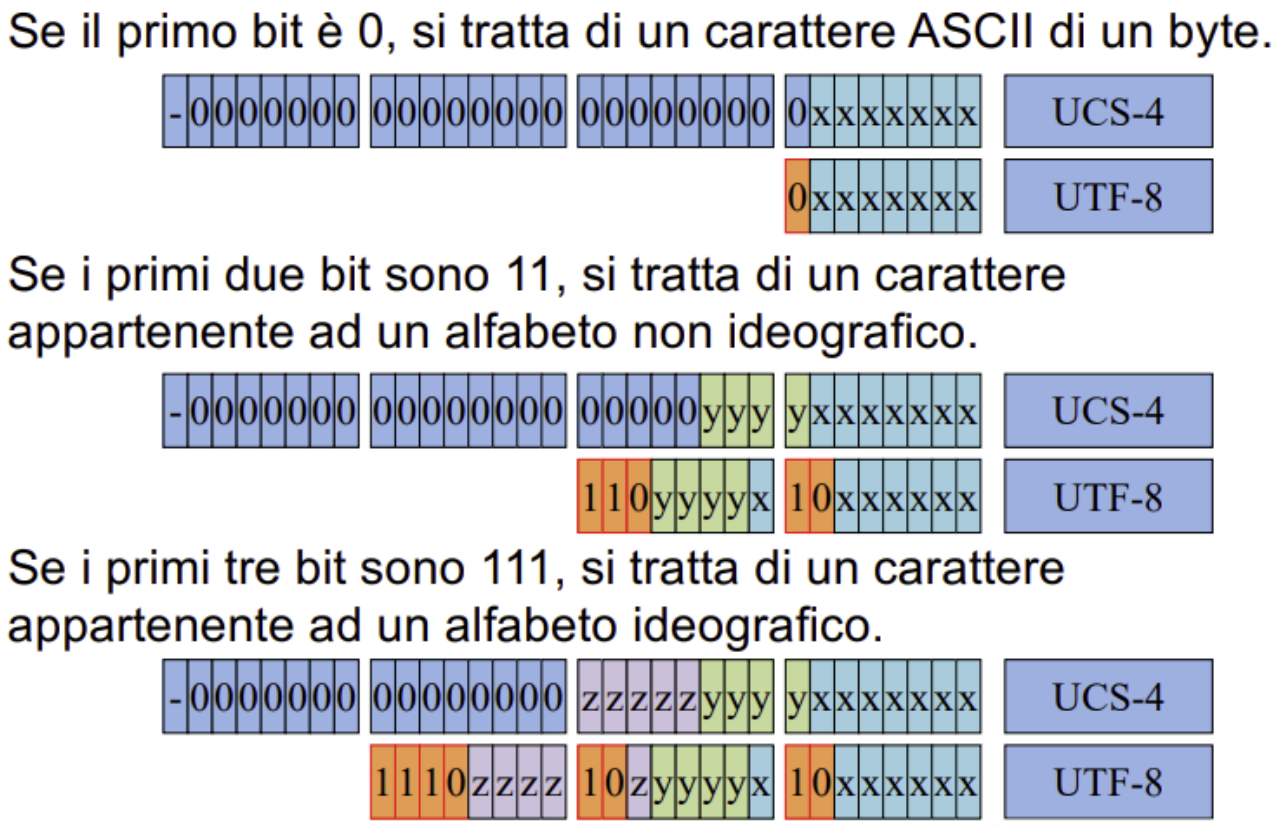
Nella maggior parte dei casi i testi scritti utilizzeranno soltanto uno degli alfabeti del mondo.
Inoltre, la maggior parte degli alfabeti ∈ BMP, e la maggior parte dei documenti sono scritti in ASCII.
Quindi è uno spreco utilizzare 4 byte per ogni carattere in questo caso perché sono necessari solo una minima parte dei caratteri di UCS.

UTF (Unicode Transformation Format o UCS Transformation Format) è un sistema a lunghezza variabile che permette di accedere a tutti i caratteri di UCS in maniera semplificata e più efficiente.

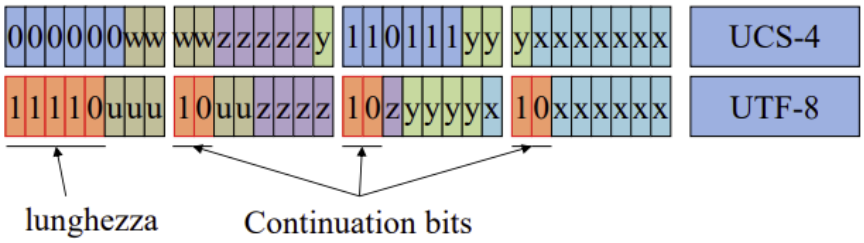
2.2.2.1 UTF-8

UTF-8 permette di accedere a tutti i caratteri definiti da UCS-4, ma utilizza un numero compreso tra 1 e 4 byte per farlo.

- I codici compresi tra 0 - 127 (ASCII a 7 bit) richiedono un byte, in cui ci sia 0 al primo bit
- I codici derivati dall'alfabeto latino e tutti gli script non-ideografici richiedono 2 byte
- I codici ideografici (orientali) richiedono 3 byte
- I codici dei piani alti richiedono 4 byte



Se i primi 4 bit sono 1111, si tratta di un carattere che in UTF-16 utilizza coppie di surrogati (caratteri appartenenti ad un piano ≠ BMP ma già precisato).



Il primo byte contiene tanti 1 quanti sono i byte complessivi per il carattere (*lunghezza*).
Il secondo byte e gli altri contengono la sequenza 10 (*continuation bit*) e 6 bit significativi.

Se il byte inizia per 10, allora è un byte di continuazione e devo ignorarlo fino al primo byte utile.

Quindi:

0xxxxxxx	È un carattere UTF-8 completo da ASCII
10zyyyyx	È un byte di continuazione
110yyyyx	Primo byte di un carattere di uno script alfabetico
1110zzzz	Primo byte di un carattere di uno script ideografico
11110uuu	Primo byte di un carattere che non sta in BMP ma è già noto
111110uu	Non definito. Apparterrebbe ad un piano non ancora occupato

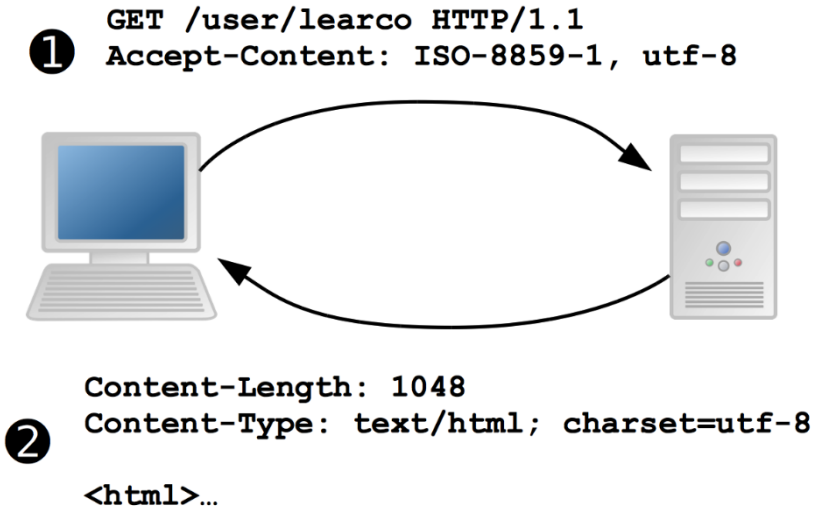
2.2.3 Differenze tra UTF-8 e ISO Latin-1

Attenzione a non confonderle,
Per i caratteri appartenenti ad ASCII le due codifiche sono identiche; quindi un documento in inglese non avrà differenze nel testo
Viceversa, un testo italiano, francese o tedesco risulta quasi corretto, perché non vengono descritte correttamente solo le decorazioni di lettere latine (accenti umlaut, vocali scandinave, …)

In questo caso:

UTF-8 → 2 byte
ISO Latin 1 → 1 byte

2.3 DEFINIRE LA CODIFICA SU HTTP



2.4 CONCLUSIONI

Set di caratteri:

- A lunghezza fissa, 7, 8 bit (ASCII, EBCDIC, ISO Latin 1)
- A lunghezza fissa, 16, 31 bit (UCS-2, UCS-4)
- A lunghezza variabile, 1/4 · 8 bit (UTF-8, UTF-16)

2.5 CONTENT ENCODING

Molti ambienti informatici forniscono restrizioni sulla varietà di caratteri usabili.

- Modelli di rappresentazione dei dati (eg. stringhe nei linguaggi di programmazione, formati dati, linguaggi di markup, …)
  Spesso alcuni caratteri hanno scopi tecnici interni al linguaggio e non è possibile utilizzarli semplicemente come contenuto
- Canali di trasmissione (eg. Protocolli internet): molti di questi canali sono stati creati quando si utilizzava soprattutto ASCII 7 bit e non sono trasparenti all'uso di flussi di dati a 8 bit (8 bit clean).

Termini frequenti:

- Escaping ⟹ il carattere proibitivo viene preceduto o sostituito da una sequenza di caratteri speciali
  - `String c = "Questa stringa \"contiene\" caratteri speciali";`
  - `<p>Questa stringa &quot;contiene&quot; caratteri speciali</p>`
- Encoding ⟹ il carattere proibitivo viene rappresentato numericamente con il suo codice naturale secondo una sintassi speciale
  - `"felicit\u00E0";`
  - `<p>felicit&#x00E0;</p>`
  - `<p>felicit&#224;</p>`

L'origine dei problemi: SMTP

È il protocollo di trasferimento delle mail (vedi appunti Reti d.C.)

Ha dei limiti:
- La lunghezza massima 1Mb
- Accetta solo caratteri ASCII 7 bit
- Ogni messaggio deve contenere una sequenza CRLF ogni 1000 caratteri

Questi limiti impediscono la trasmissione di documenti binari:
- Un file binario usa tutti i 256 tipi di byte
- Un file binario può facilmente essere più lungo di 1Mb
- In un file binario la sequenza CRLF è una sequenza come tutte le altre e introdurla artificialmente può corrompere il file.

MIME bypassa questi limiti di SMTP

Multipurpose Internet Mail Extensions

MIME ridefinisce il formato del corpo di RFC 822 così da permettere:
- Messaggi di testo in altri set di caratteri al posto di US-ASCII
- Un insieme estensibile di formati per messaggi non testuali
- Messaggi multi-parte
- Header con set di caratteri diversi da US-ASCII

Messaggi MIME su canali SMTP

- Al server SMTP il messaggio non compatibile con SMTP viene trasformato in uno o più messaggi SMTP da un preprocessore
- Quello che viaggia sul canale SMTP è un messaggio SMTP puro
- All'arrivo il messaggio SMTP viene decodificato e riaccorpato a formare il messaggio originale

3 URI

3.1 IL CONCETTO DI RISORSA

È un concetto indipendente dal meccanismo di memorizzazione o dal tipo di contenuto.

Una **risorsa** è una qualunque struttura che sia oggetto di scambio tra applicazioni all'interno del WWW.

Molti identificatori fanno riferimento a file memorizzati in un file system gerarchico, questo tipo di risorsa:
- potrebbe essere in un database, e l'URI essere la chiave di ricerca
- potrebbe essere il risultato dell'elaborazione di un'applicazione e l'URI essere i parametri di elaborazione
- potrebbe essere una risorsa non elettronica (libro, persona, …) e l'URI essere il suo nome Uniforme
- potrebbe essere un concetto astratto (la grammatica di un linguaggio)

Per questo si usa il termine Risorsa, invece che File.

Gli URI (Uniform Resource Identifier) sono una sintassi usata nel web per definire nomi e indirizzi di risorse su internet.
risolvono il problema di creare un meccanismo di accesso unificato alle risorse di dati disponibili in rete.

Distinguiamo gli URI in:

- URL (Uniform Resource Locator)
  Sintassi che contiene informazioni immediatamente utilizzabili per accedere alla risorsa
  Sono fragili e soggetti a modifiche del meccanismo di accesso
- URN (Uniform Resource Names)
  Indipendentemente dalle informazioni di accesso, indica l'etichettatura della risorsa
  Sono informazioni certe e affidabili della sua esistenza e accessibilità

3.2 DESIGN DEGLI URI

Gli URI sono progettati per essere:
- ***Trascrivibili***
- ***Fornire identificazione, non interazione***
- ***Fornire spazi di nomi organizzati gerarchicamente***: (": / ? #" sono separatori per ambiti diversi)

URI = `schema : [// authority] path [? query] [# fragment]`
`http://www.ietf.org/rfc/rfc2396.txt` è un esempio

un'**authority** individua un'organizzazione dei nomi a cui sono delegati,
e si divide in authority = `[userinfo @ host [: port] ]`

- La parte `userinfo` è facoltativa (intuitivo)
- L'`host` è un dominio o un indirizzo ip
- La `port` è anch'essa facoltativa, omessa se è una well-known port (per http è 80)

Il **path** è la parte identificativa della risorsa nello spazio di nomi identificato da schema e authority

- Il path è diviso da slash "/", che ne divide le componenti

La **query** è una specifica della risorsa, sono spesso parametri usati per specificare un risultato dinamico

- Ha forma `nome1=valore1&nome2=valore+in+molte+parole`

Un **fragment** individua un pezzo specifico della risorsa (un paragrafo di una pagina)

3.3 CARATTERI AMMESSI NEGLI URI

I caratteri degli URI si dividono in:

- unreserved (non riservati)
  - caratteri alfanumerici e punteggiatura non ambigua
    uppercase, lowercase, digit, punctuation ('-', '_', '!', '~', '*', ''', '(', ')')
- reserved (riservati)
  - hanno funzioni particolari negli schemi
    (';', '/', '?', ':', '@', '&', '=', '+' , '$')
- escaped
  - quando sono parte della stringa identificativa naturale, hanno diverse categorie
    - caratteri non US-ASCII
      ISO Latin-1 > 127
    - caratteri di controllo
      US-ASCII < 32
    - caratteri unwise
      ('{' , '}' , '|' , '\' , '^' , '[' , ']' , '`')
    - delimitatori di spazio
      ('<' , '>' , '#' , '%' , '"')
    - riservati quando usati in un contesto diverso dal loro uso

    I caratteri *escaped* sono scritti come %XX dove XX è il codice HEX del carattere

    `paper@ → paper%40`

3.4 ROUTE

Una **route** è un'associazione della parte path di un URI ad una risorsa gestita o restituita da un server.

Managed route: il server associa ogni URI a una risorsa:
attraverso il file system locale (risorse statiche)
attraverso una computazione (risorse dinamiche)

```
Var router = require("express").Router();

Function getName(req, res) {
      res.send("<p>Fabio</p>");
}

Function getEmail(req, res) {
      res.send("fabio.vitali@unibo.it");
}

router.get("/name", getName);
router.get("/email", getEmail);
app.use("/css", express.static('css'))
```

URI della richiesta	Risorsa restituita
`http://www.example.com/name`	`<p>Fabio</p>`
`http://www.example.com/email`	`fabio.vitali@unibo.it`
`http://www.example.com/css/style.css`	*contenuto del file* `css/style.css`

File-system route: il server associa la radice della parte *path* ad una directory del file system locale e ogni file al suo interno è valido e funzionante

Organizzazione del file system	URI disponibili
/	-
var/	-
www/	-
index.html	http://www.example.com/
alice/	-
index.html	http://www.example.com/alice/
bruce/	-
index.html	http://www.example.com/bruce/
...	...
fabio/	-
index.html	http://www.example.com/fabio/
pages/	
doc1.html	http://www.example.com/fabio/pages/doc1.html
doc2.html	http://www.example.com/fabio/pages/doc2.html
index.html	http://www.example.com/fabio/pages/
img/	
img1.gif	http://www.example.com/fabio/img/img1.gif
img2.jpg	http://www.example.com/fabio/img/img2.jpg
css/	
style.css	http://www.example.com/fabio/css/style.css

3.5 URI REFERENCES (URIREF)

Un URI *assoluto* contiene tutte le parti predefinite dal suo schema precisate esplicitamente

Un URI *gerarchico* che è anche essere relativo è detto **URI reference**, in questo caso riporta solo una parte dell'URI assoluto corrispondente, tagliando progressivamente cose da sinistra.
Un URI reference fa sempre riferimento ad un URI di base, rispetto al quale fornisce porzioni differenti.

> Es. L'URL reference `pippo.html` posto dentro al documento con URI
> `http://www.sito.com/uno/due/pluto.html`
> fa riferimento al documento il cui URI assoluto è
> `http://www.sito.com/uno/due/pippo.html`

3.5.1 Operazioni su URI

Vediamo quali sono le *operazioni* possibili sugli URI:

- URI resolution
  Genera l'URL assoluto corrispondente all'URI
  Si esegue quando l'URI è un URI reference o un URI a cui non corrisponde una risorsa fisica
- URI dereferencing
  Restituisce una risorsa identificata dall'URI (eg. Il documento cercato)

3.5.1.1 Risolvere un URI reference

Ovvero identificare l'URI assoluto cercato, dato l'URI relativo stesso e (di solito) l'URI di base:

	Dato l'URI base http://www.site.com/dir1/doc1.html
Se inizia con "#", è un frammento interno allo stesso documento di base	#anchor1 si risolve come http://www.site.com/dir1/doc1.html#anchor1
Se inizia con uno schema, è un URI assoluto	http://www.site.com/dir2/doc2.html si risolve come http://www.site.com/dir2/doc3.html
Se inizia con "//", è un URI assoluto con lo stesso schema della base. Serve per creare riferimenti che funzionano sia con http che con HTTPS	//www.site.com/dir5/doc7.hmtl si risolve in http://www.site.com/dir5/doc7.html N.B.: se la base fosse https://... diventerebbe https://www.site.com/dir5/doc7.html
Se inizia con "/", allora è un path assoluto all'interno della stessa autorità della base, e ha la stessa authority	/dir3/doc3.html si risolve come http://www.site.com/dir3/doc3.html
Altrimenti:	Dato l'URI base http://www.site.com/dir1/doc1.html
Altrimenti, si estrae il path assoluto dell'URI di base, meno l'ultimo elemento, e si aggiunge in fondo l'URI relativo. Si procede infine a semplificazioni:	doc4.html si risolve come http://www.site.com/dir1/doc4.html dir5/doc5.html si risolve come http://www.site.com/dir1/dir5/doc4.html
"./" (stesso livello di gerarchica): viene cancellata	./doc6.html si risolve come http://www.site.com/dir1/./doc6.html che è equivalente a http://www.site.com/dir1/doc6.html
"../" (livello superiore di gerarchia): viene eliminato insieme all'elemento precedente.	../doc7.html si risolve come http://www.site.com/dir1/../doc7.html che è equivalente a http://www.site.com/doc7.html

3.6 GLI SCHEMI

- Gli schemi più usati sono **HTTP** e **HTTPS**, l'unica differenza è che HTTPS prevede una crittografazione in entrambi i sensi del contenuto del messaggio.

```
http://[:port]/path[?query][#fragment]
https://[:port]/path[?query][#fragment]
```

- Esiste anche lo schema **FILE** capace di aprire i file presenti sul computer locale

```
File://host/path [#fragment]
```

- Lo schema **DATA** è invece usato per contenere una risorsa, e la sua sintassi è

```
data:[<media type>][;base64],<data>
```

- **FTP** è uno schema strutturato così

```
ftp://[user[:password]@]host[:port]/path
```

la password lo prevede come parte facoltativa ma sconsigliata data la scarsa sicurezza

3.7 APPROFONDIMENTI

Se un URI inizia con // sarà risolto in un URI assoluto utilizzando le parti mancanti dell'URI base, eg.

```
//www.sito.com/dir1/dir2/fig1.gif
```

Sarebbe la pagina HTML ospitante il tag <img>, e quindi in pratica quello significa "*carica il file usando lo stesso protocollo HTML:HTTP se siamo fuori dall'area protetta e HTTPS se siamo dentro*"

3.7.1.1 CDN

Un **Content Delivery Network** è una rete distribuita di server commerciali che lavorano tutti insieme per un uso ottimale

Ad esempio, se diversi siti usano le stesse librerie importanti, se non le prendono da un CDN, un utente deve scaricarle ogni volta, mentre viceversa saranno scaricate e messe in cache una sola volta e utilizzate per più siti differenti.

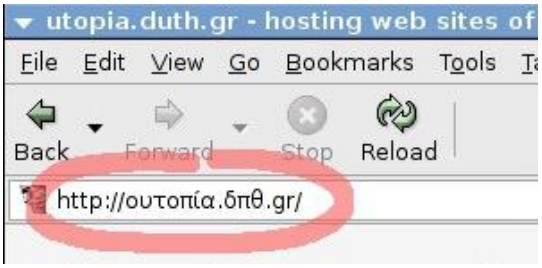
3.7.1.2 IRI, IDN, CURIE

IRI – Internationalized Resource Identifier

Fornisce una sintassi per inserire caratteri USC-4 in un URL e per risolverli

IDN – Internationalized Domine Name

È uno standard IETF per estendere i nomi di dominio a tutti i caratteri non ASCII di Unicode.



Uno dei rischi dell'implementazione di questo standard è l'utilizzo di caratteri omografi per attacchi di phishing, ad esempio i caratteri:

a c e i j o p s x y

Sono in cirillico, ma sono visivamente identici ai caratteri latini.

Eg. il dominio "http://paypal.com" contiene 2 'a' in cirillico, ma è indistinguibile da "http://paypal.com"

CURIE – Compact URI

Sono un modo per risparmiare spazio su disco quando ci sono tanti URI lunghissimi che condividono la prima parte della stringa

4 HTTP - HyperText Transfer Protocol

HTTP è un protocollo *client-server*, *generico* e *stateless* utilizzato in molte applicazioni distribuite, permette lo scambio di risorse identificate da URI

- *Client-server*
  Il client si occupa della comunicazione e richiede servizi
- *Generico*
  Il protocollo è indipendente dal formato di dati con cui vengono trasmesse le risorse e funziona su tutto
- *Stateless*
  Non è tenuto a mantenere informazioni che persistano tra una connessione e la successiva su precedenti richieste

HTTP implementa inoltre sofisticate *politiche di caching* che permettono di memorizzare copie di risorse per ottimizzare la navigazione

4.1 Connessione HTTP

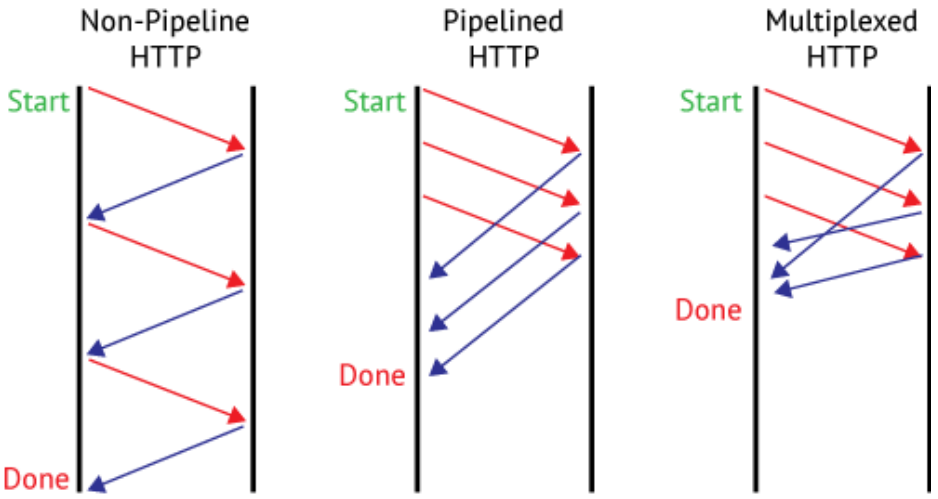
Una connessione HTTP è composta da:

- X richieste
- X risposte

Le connessioni sono **persistenti** con:

- **Pipelining**:
  trasmissione di più richieste senza attendere l'arrivo della risposta alla richiesta precedente
  le risposte sono restituite nello stesso ordine delle richieste
- **Multiplexing**:
  Si possono avere richieste e risposte multiple, restituite anche in ordine diverso di quello di invio

  Introdotto in HTTP/2

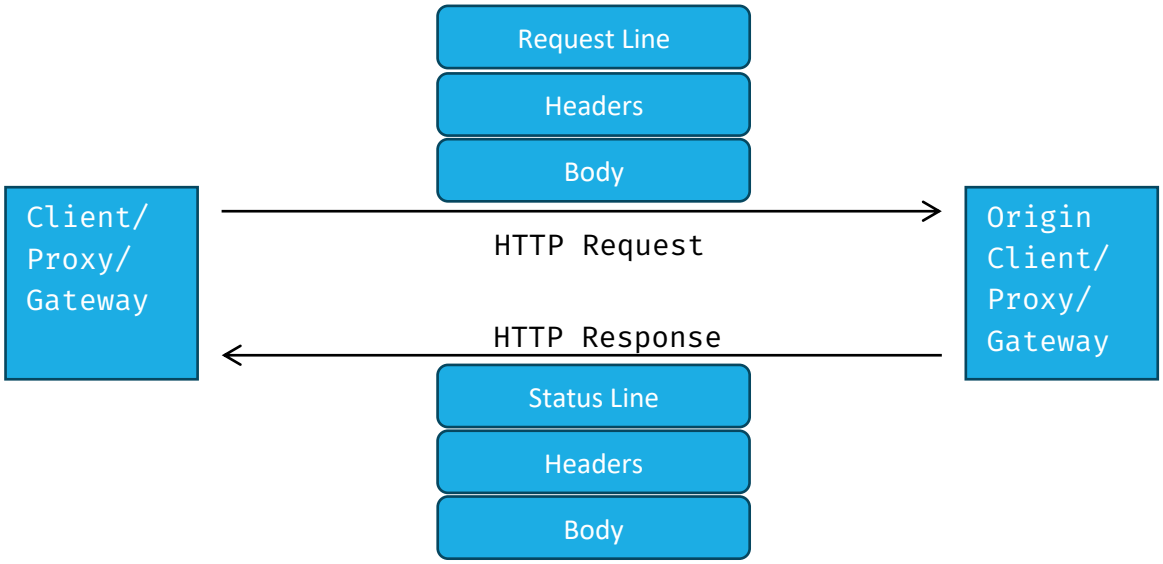


4.1.1.1 HTTP/2

HTTP/2 ha introdotto molte novità per migliorare la performance

- Operazioni PUSH:
  il server anticipa le richieste successive del client
- Compressione degli header:
  i dati non sono plaintext ma complessi

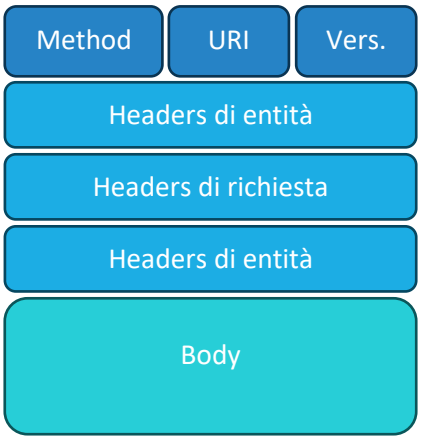
4.2 RICHIESTE E RISPOSTE HTTP



4.2.1 La richiesta http

si compone di:

- **Method:**
  azione del server richiesta dal client
- **URI:**
  identificativo della risorsa locale al server
- **Version:**
  numero di versione di http
- **Header:**
  sono linee RFC822 divise in
  - header generali
  - header di entità
  - header di richiesta
- **Body:**
  è un messaggio MIME



Esempio di richiesta

```
GET /beta.html http/1.1
Referer: http://www.alpha.com/alpha.html
Connection: Keep-Alive
User-Agent: Mozilla/4.61 (Macintosh; I; PPC)
Host: www.alpha.com:80
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

4.2.1.1 I metodi di HTTP

I metodi indicano l'azione che il client richiede al server riguardo alla rappresentazione della copia della risorsa;

I metodi principali: GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH

Esempi di GET e POST

GET è il metodo più frequente, ed è quello che viene attivato facendo click su un link ipertestuale di un documento HTML, o specificando un URL nell'apposito campo di un browser.

```
GET /courses/tw.html
GET /students/123456/exams
```

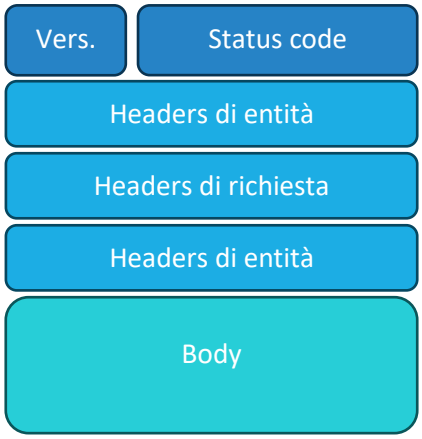
POST serve per trasmettere informazioni dal client al server relative alla risorsa identificata nell'URI

```
POST /courses/1678
{
    "titolo":"Tecnologie Web",
    "descrizione":"Il corso..bla..bla"
}
```

4.2.2 La risposta http

si compone di:

- **Status code:**
  azione del server richiesta dal client
- **Version:**
  numero di versione di http
- **Header:**
  sono linee RFC822 divise in
  - header generali
  - header di entità
  - header di richiesta
- **Body:**
  è un messaggio MIME



Esempio di richiesta

```
GET /index.html HTTP/1.1
Host: www.cs.unibo.it:80
```

```
HTTP/1.1 200 OK

Date: Fri, 26 Nov 2007 11:46:53 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Mon, 12 Jul 2007 12:55:37 GMT
Accept-Ranges: bytes
Content-Length: 3357
Content-Type: text/html

<HTML> …. </HTML>
```

4.2.3 Status code

Lo status code è un numero di tre cifre nel quale

- La prima indica la classe della risposta
- La seconda indica la risposta specifica

Esistono le seguenti classi:

1xx: Informational	Una risposta temporanea alla richiesta durante il suo svolgimento
2xx: Successful	Il server ha ricevuto, capito e accettato la richiesta
3xx: Redirection	La richiesta è corretta, ma sono necessarie altre azioni da parte del client per portare a termine la richiesta.
4xx: Client error	La richiesta del client non può essere soddisfatta per un errore del client (errore sintattico o richiesta non autorizzata)
5xx: Server error	La richiesta può anche essere corretta, ma il server non riesce a soddisfarla per un problema interno

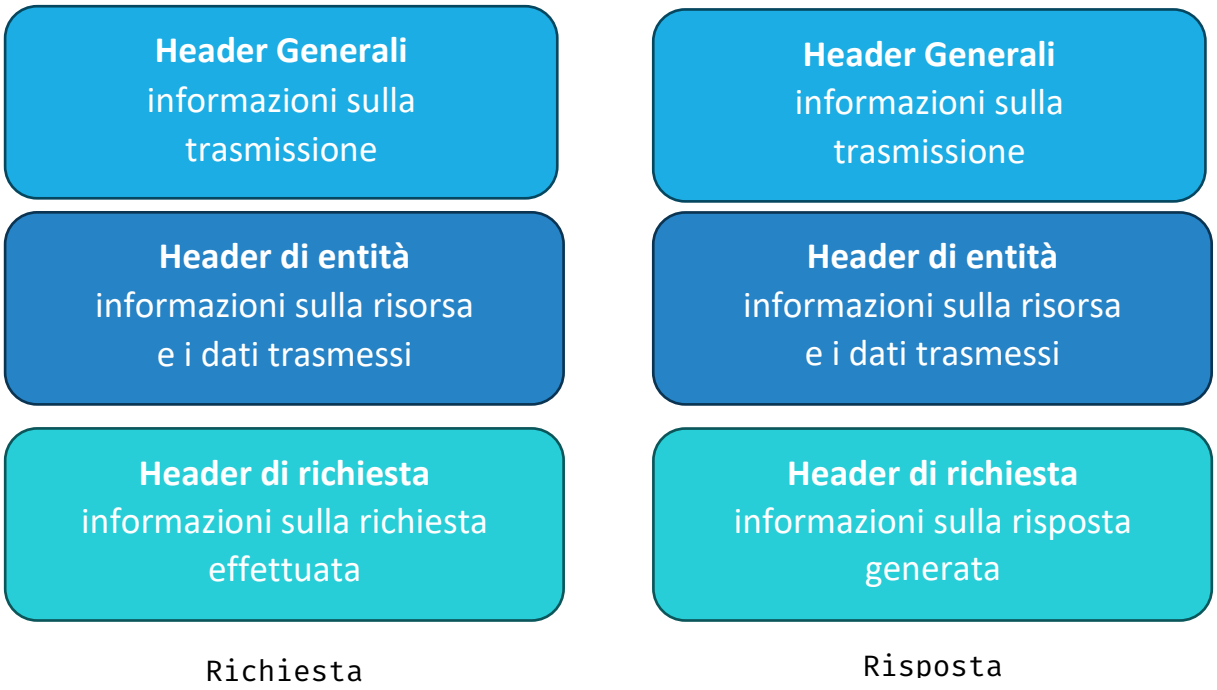
Esempi di status code

100 **Continue** (se il client non ha ancora mandato il body)
200 **ok** (GET con successo)
201 **Created** (PUT con successo)
301 **Moved permanently** (URL non valida, il sever conosce la nuova posizione)
400 **Bad request** (errore sintattico nella richiesta)
401 **Unauthorized** (manca l'autorizzazione)
403 **Forbidden** (richiesta non autorizzabile)
404 **Not found** (URL errato)
500 **Inernal server error** (tipicamente un errore nel codice server-side)

L'uso corretto dello status code aiuta a costruire API chiare e semplici da usare

4.2.4 Gli header

Gli **headers** sono righe di testo che specificano informazioni addizionali, sono presenti sia nelle richieste che nelle riposte e ne descrivono diversi aspetti



Header generali
Si applicano solo al messaggio trasmesso e si applicano sia ad una richiesta che ad una risposta, ma non necessariamente alla risorsa trasmessa

Header di entità
Danno informazioni sul body del messaggio oppure, se non c'è body, sulla risorsa specificata

Header della richiesta
Vengono posti dal client per specificare informazioni sulla richiesta e su se stesso al server

Header della risposta
Sono posti dal client per specificare informazioni sulla risposta e su se stesso al client

4.2.5 I metodi HTTP

I metodi indicano l'azione che il client richiede al server sulla rappresentazione della copi della risorsa
Vengono anche chiamati **verbi HTTP** per evidenziare l'idea che esprimono azioni da eseguire sulle risorse.

Proprietà dei metodi

- *Sicurezza*
  Un metodo è sicuro se non genera cambiamenti allo stato interno del server
  un metodo sicuro può anche essere eseguito da un nodo intermedio (tipo una cache) senza effetti negativi
- Idempotenza
  Un metodo è idempotente se a uguale richiesta corrisponde un solo effetto
  Un metodo idempotente può essere ri-eseguito da più agenti o in più tempi diversi senza effetti negativi

4.2.5.1 GET

Sicuro e idempotente
Richiede una risorsa ad un server
il più comune e il più importante, si attiva al click di un link ipertestuale

```
GET /courses/tw.html
GET /students/123456
GET /students/123456/exams
```

4.2.5.2 HEAD

Sicuro e idempotente
è un GET ma che risponde solo con header

```
HEAD /courses/tw.html
```

4.2.5.3 POST

Non Sicuro e non idempotente
può essere usato per creare nuove risorse, viene usato per spedire i dati di un form su HTML ad un'app server side

```
POST /courses/tw.html
{
  "titolo":"Tecnologie Web",
  "descrizione":"Il corso..bla.."
}
```

4.2.5.4 PUT

Idempotente e non sicuro
serve per trasmettere delle informazioni dal client al server, creando o sostituendo la risorsa specificata

```
PUT /courses/tw.html
{
  "id":1678,
  "titolo":"Tecnologie Web",
}
```

4.2.5.5 DELETE

Idempotente e non sicuro
rimuove informazioni connesse con una risorsa (se effettuiamo un GET dopo un DELETE, error 404)

```
DELETE /courses/tw.html
```

4.2.5.6 PATCH

Non sicuro e non idempotente
aggiorna parzialmente la risorsa identificata dall'URI
a differenza di PUT sovrascrive solo le informazioni indicate

```
PATCH /courses/1678
{
  "op" : "update"
  "cfu":"6"
}
```

4.2.5.7 OPTIONS

Sicuro e idempotente
viene usato per verificare opzioni, requisiti e servizi di un server

Metodi HTTP	Idempotente	Sicuro
GET	YES	YES
HEAD	YES	YES
OPTION	YES	YES
DELETE	YES	NO
PUT	YES	NO
PATCH	NO	NO
POST	NO	NO

5 API REST

Un'**API Web** descrive un'interfaccia http che permette ad applicazioni remote di utilizzare i servizi dell'app. App che possono essere:
- Applicazioni automatiche che usano i dati dell'app
- Applicazioni Web che mostrano all'utente un menù di opzioni e gli permettono di eseguire un'azione sui dati dell'applicazione

REST (REpresentational State Transfer) è il modello architetturale che sta dietro al WWW e dietro alle web app "ben fatte"

Un'applicazione REST si basa sull'uso dei protocolli di trasporto (HTTP) e di naming (URI) per generare interfacce generiche di interazione con l'applicazione, e fortemente connesse con l'ambiente d'uso.

5.1.1 CRUD

È un pattern tipico delle applicazioni di trattamento dei dati

- **Create** (inserimento nel database di un nuovo record)
- **Read** (accesso in lettura al database)
- **Update** (aggiornamento di una o più informazioni)
- **Delete** (rimozione dal database di un record)

5.1.2 4 Punti dell'architettura REST

1. Definire risorsa
2. Associargli un URI come identificatore
3. Usare i verbi http per esprimere operazioni
4. Esprimere in maniera parametrica ogni rappresentazione dello stato interno della risorsa (attraverso un content type preciso)

5.1.3 Presupposti del web

L'architettura si basa sui 3 presupposti del web:

- *Nome*
  Che ogni entità sia definita come risorsa e che sia associata ad un URI
- *Verbo*
  Che ogni interazione deve essere esprimibile tramite metodo HTTP
- *Formato*
  Che ogni risorsa abbia un content type

5.2 INDIVIDUI E COLLEZIONI

REST identifica due concetti: **individui** e **collezioni**, per esempio:

- Un cliente vs. l'insieme di tutti i clienti
- Un esame vs. l'insieme di tutti gli esami superati

Le collezioni possono contenere individui o altre collezioni in una gerarchia, per esempio:

URI	Rappresentazione
`/customers/`	collezione dei clienti
`/customers/abc123`	cliente con id=abc123
`/customers/abc123/`	collezione delle sotto-risorse del cliente con id=abc123 (es. indirizzi, telefoni, …)
`/customers/abc123/addresses/1`	primo indirizzo del cliente con id=abc123

Tutti i clienti
`/clients/`

Il cliente 1234
`/clients/1234`

Tutti gli ordini del cliente 1234
`/clients/1234/orders/`

5.2.1 Filtri

Un **Filtro** genera un sottoinsieme specificato attraverso una regola di qualche tipo. Vengono usate solitamente le parti query di un URI per costruirlo

URI	Rappresentazione
`/regions/ER/customers/`	collezione dei clienti dell'Emilia-Romagna
`/status/active/customers/`	collezione dei clienti attivi
`/customers/?tel=0511234567` oppure `/customers/?search=tel&value=0511234567`	collezione dei clienti che hanno telefono = 051 1234567
`/customers/?search=sales&value=100000&op=gt`	collezione dei client che hanno comprato più di 100.000€

5.3 OpenAPI

Una API si dice RESTful se utilizza i principi REST nel fornire accesso ai servizi che offre

per documentare un API è necessario definire:

- **end-point** (URI / route)
  separando collezioni ed elementi singoli
- **metodi**
  Coda succede con un GET, un PUT, un POST, un DELETE, …
- **rappresentazioni in I/O**
- **condizioni di errore e i messaggi di errore**

Swagger è un ecosistema di tool per la creazione completa e accesso ad API in abito REST, che può essere serializzato sia in YAML che in JSON.

Swagger è lo standard industriale per API REST

5.3.1 YAML

YAML è una linearizzazione di strutture dati con sintassi ispirata a python, con indentazione come modello di annidamento e supporto di tipi scalari, liste e array associativi.

```yaml
nome: Angelo

cognome: Di Iorio

ufficio:
  città: Bologna
  civico: 14
  via: Ranzani

corsi:
  - Programmazione
  - Tecnologie Web
```

```yaml
name: Sagre
news:
  - id: 1
    titolo: Sagra del ...
    articolo: Lo stand ...
    immagine: sagra.jpg
  - id: 2
    titolo: Tortellini per tutti
    articolo: Bologna la patria
...
    immagine: tortellini.jpg
```

5.3.2 path

i path sono i percorsi (RL) corrispondenti alle operazioni possibili sull'API, la loro struttura è:

<host>/<basePath>/<path>

Per ogni path si definiscono tutte le operazioni possibili che contengono informazioni e parametri I/O, e ogni operazione conduce ad una sottoselezione

```yaml
/pet/{petID}:   # Risorsa
    get:        # Metodo HTTP
        summary: "Find pet by ID"
        description: "Returns a single pet"
        operationID: "getPetById"
        produces:
        - "application/xml"   # Formati in Output
        - "application/json"  # Formati in Output

        parameters:           # Parametri in Input
          ...
          ...

    post:      # Metodo HTTP
        summary: "Updates a pet in the store with form data"
        description ""
        operationID: "updatePetWithForm"
        consumes:
        - "application/x-www-form-urlencoded"
        produces:
        - "application/xml"
        - "application/json"
        parameters:
          ...
          ...
```

5.3.3 Parametri in input

I parametri in input sono descritti nella sezione corrispettiva, che li definisce tutti con questa struttura:

- Tipo
  keyword `in` che può assumere valori `path`, `query` o `body`
- Nome e descrizione
  `name` e `description`
- Opzionale o obbligatorio
  `required`
- Formato dei valori che può assumere
  `type` o `schema`

```yaml
/pet/{petID}:   # Risorsa
    get:        # Metodo HTTP
      summary: "Find pet by ID"
      description: "Returns a single pet"
      operationID: "getPetById"
      parameters:   # Parametri in Input
      - name: "petID"
        in: "path"
        description: "ID of pet to return"
        required: true
        type: "integer"
        format: "int64"
```

5.3.4 Oggetti e definizioni

Il body contiene un oggetto di tipo user, viene infatti passata un'intera risorsa come parametro

La sezione definitions permette di definire interamente gli oggetti, e possono essere referenziati (tramite schema -> $ref)

Esempi di schemi

```yaml
User:
  type: object
  properties:
    id:
      type: integer
      format: int64
    username:
      type: string
    firstName:
      type: string
    lastName:
      type: string
    email:
      type: string
    password:
      type: string
    phone:
      type: string
    userStatus:
      type: integer
      format: int32
      description: User Status
```

```yaml
Order:
  type: object
  properties:
    id:
      type: integer
      format: int64
    petId:
      type: integer
      format: int64
    quantity:
      type: integer
      format: int32
    shipDate:
      type: string
      format: date-time
    status:
      type: string
      description: Order Status
      enum:
        - placed
        - approved
        - delivered
    complete:
      type: boolean
      default: false
```

5.3.5 Output

I possibili output sono definiti attraverso la keyword responses

Si specifica il tipo di output (dati, codici, messaggi di errore) atteso nel body delle risposte

Ogni risposta ha un id numerico

- 200 non c'è stato alcun errore
- 400 errore

6 MARKUP

Il **markup** è qualunque mezzo utilizzato per rendere esplicita una particolare interpretazione di un testo

Il markup può assolvere a diversi ruoli a seconda del sistema di elaborazione del testo:

- Puntazionale
  È il markup che fornisce informazioni sintattiche sul testo, quindi la punteggiatura
- Presentazionale
  Consiste in effetti grafici (et simila) per rendere più presentabile un testo, come la divisione in paragrafi, interlinea, …
- Procedurale
  Utilizza le potenzialità di un sistema automatico per specificare qualche effetto o procedura attivate nella visualizzazione del contenuto
- Descrittivo
  Ormai si dice "semantico", ovvero da un significato a ogni parte del testo specificando di cosa si tratti (titolo, sottotitolo, box, paragrafo, …)
- Referenziale
  Fa riferimento ad entità esterne al documento per fornire significato o effetti grafici a elementi del testo (tipo le note)
- Metamarkup
  Ovvero la definizione di regole di interpretazione del markup, per estenderne il significato

6.1 SGML

Lo **Standard Generalized Markup Language** non è un linguaggio di markup, ma uno standard per definire linguaggi di markup.

I linguaggi SGML devono possedere certe caratteristiche:

- Leggibile
  In SGML il markup è posto in maniera leggibile a fianco degli elementi del testo a cui si riferiscono
- Descrittivo
  Il markup in SGML non è pensato unicamente per la stampa su carta. È possibile combinare markup utile per scopi o applicazioni diverse, e in ogni contesto considerare o ignorare di volta in volta i markup non rilevanti.
- Strutturato
  Cioè, è possibile definire una serie di regole affinché il testo sia considerabile strutturalmente corretto.
- Gerarchico
  È possibile stabilire una gerarchia nel testo (capitoli, paragrafi, capoverso, …)

Un documento SGML è sempre composto da tre parti:

dichiarazione SGML	(specifica valori fondamentali come il set di caratteri usati, se omessa viene usata la dichiarazione di default)
dichiarazione di documento	(DOCTYPE, definisce le caratteristiche del documento)
istanza del documento	(il documento vero e proprio)

6.2 COMPONENTI DEL MARKUP

Un documento SGML contiene i seguenti componenti:

- Elementi
  Sono le parti del documento dotate di senso proprio, (eg. titolo, autore, paragrafo, …)
- Attributi
  Sono informazioni aggiuntive che non fanno parte del documento (eg. i tag HTML)
- Entità
  Sono frammenti di documento memorizzati separatamente e richiamabili all'interno del documento. Permettono di riutilizzare lo stesso frammento in molte posizioni garantendo sempre l'esatta corrispondenza dei dati e permettendo una loro modifica semplificata.
- Testo
  Contenuto del documento
- Commenti
  Sono parti ignorate dal parser, che vengono usate per annotare il documento e esplicitare informazioni agli autori.
- Processing Instructions
  Sono istruzioni particolari per dare ulteriori indicazioni su come gestire il documento

6.3 XML

XML è un linguaggio definito come sottoinsieme di SGML
XML accetta due tipi di documenti: i documenti validi e i documenti ben formati.

Un documento si dice "ben formato" se:

- Tutti i tag sono ben annidati
- Esiste un tag radice che contiene tutti gli altri
- Gli elementi vuoti usano un simbolo speciale di fine tag
- Tutti gli attributi sono sempre racchiusi tra virgolette
- Tutte le entità sono definite

Un documento è valido poi se è fornito anche di un DTD

7 HTML

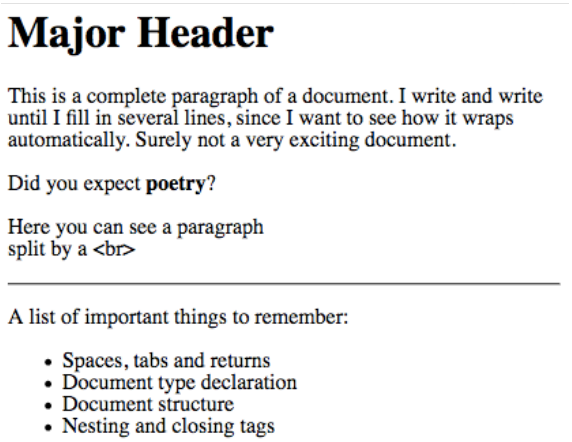
La struttura di un documento HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Document title</title>
  </head>
  <body>
    <p>Text of a paragraph</p>
  </body>
</html>
```

<!DOCTYPE html>: segnala il tipo di markup usato nel documento e la sua versione, con molte complessità e variazioni.
Case-insensitive, ma se si usa XHTML e XHTML 5, allora DOCTYPE è maiuscolo e tutti i tag sono minuscoli.

html: La radice dell'albero.
head: Informazioni globali sul documento
body: Il contenuto del documento

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
  <head>
    <title> Document title </title>
  </head>
  <body>
    <h1> Major Header </h1>
    <p>This is a complete paragraph of a document. I
write and write until I fill in several lines, since I
want to see how it wraps automatically. Surely not a
very exciting document.</p>
    <p>Did you expect <b>poetry</b>?</p>
    <p>Here you can see a paragraph <br>
       split by a &lt;br&gt;</p>
    <hr>
    <p> A list of important things to remember: </p>
    <ul>
      <li>Spaces, tabs and returns</li>
      <li>Document type declaration</li>
      <li>Document structure</li>
      <li>Nesting and closing tags</li>
    </ul>
  </body>
</html>
```



7.1.1 Elementi inline

Sono elementi che non spezzano il blocco, si dividono in tag fontstyle e tag phrase, i primi forniscono informazioni di rendering del testo e sono sconsigliati (meglio usare il CSS), i secondi definiscono informazioni semantiche.

7.1.2 Elementi di blocco

I tag di blocco definiscono l'esistenza di blocchi che contengono elementi inline. I tag di blocco fondamentali sono:

- <p>: paragrafo
- : blocco generico
- <pre>: blocco preformattato
- <address>: l'autore della pagina
- <blockquote>: citazione lunga

I tag di blocco che hanno un ruolo strutturale (o meglio gerarchico) sono:
h1, h2, h3, h4, h5 e h6

7.1.3 Elementi di lista

Le liste contengono elementi omogenei, i tag principali sono:

- `<ul>`: unordered list (lista con pallini o altri elementi grafici)
- `<ol>`: ordered list (lista numerata o con lettere alfabetiche=
  - `<li>`: (list item, usato con `ul` o `ol`)
- `<dl>`: (lista di definizioni)
  - `<dt>`: (definition term)
  - `<dd>`: (definition data)

7.1.4 Elementi generici

I due elementi generici sono `<div>` e `<span>`. Non hanno caratteristiche semantiche, quindi è consigliato usarli se non ci sono altri tag migliori aggiungendo significato semantico (tramite le classi)
la differenza tra i due è che i `<div>` sono elementi di blocco (i contenuti prima e dopo vanno a capo), mente lo `<span>` è di tipo inline (quindi il testo prosegue senza spezzarsi in blocchi diversi).

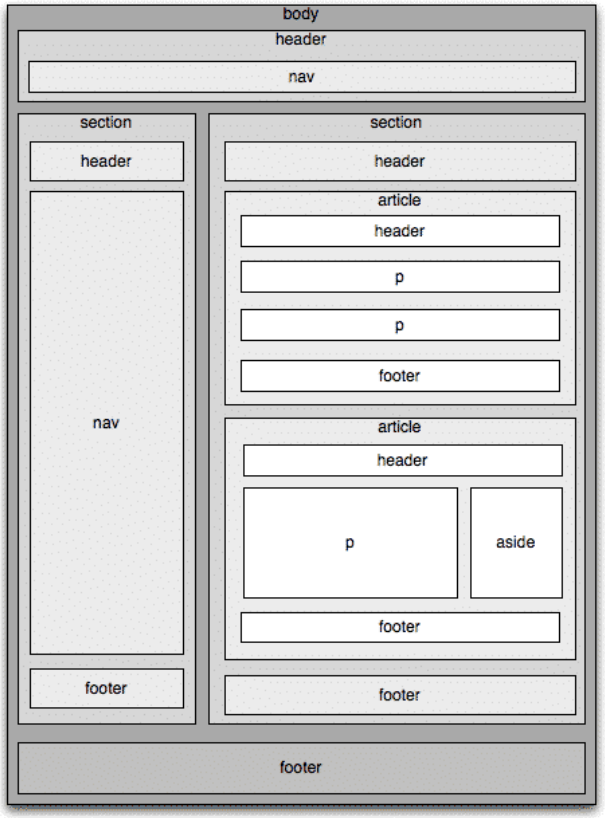
7.1.5 Piccoli effetti grafici

I tag `<hr>` (horizontal rule) e `<br>` (break) mandano a capo il testo, la differenza è che `<hr>` aggiunge una linea orizzontale (della quale è possibile modificarne l'aspetto), mente `<br>` manda solo a capo e non rompe logicamente il paragrafo

7.1.6 Elementi di struttura

In HTML5 ci sono vari tag utili per strutturare il documento:

- `main`: indica la parte princilae della pagina, al cui interno possiamo trovare altri tag;
  - `section`: è un generico contenitore annidabile
  - `article`: una parte del documento self-contained pensata per essere riutilizzata
- `aside`: indica la parte che solitamente sta di fianco al main
- `header`: elemento iniziale del documento
- `footer`: elemento finale del documento
- `nav`: barra di navigazione



7.1.7 Link ipertestuali

I link si mettono con il tag `a` (come anchor):

```
<a href="url-del-collegamento">Clicca qui</a>
```

L'attributo `href` definisce la destinazione di un link.

7.1.8 Immagini

Per aggiungere un'immagine (formati disponibili `jpg`, `png` e `gif`) si usa il tag : `img`

```
<img src="url-dell'immagine" alt="descrizione">
```

Gli attributi sono:

- `src`: l'URL della risorsa che contiene l'immagine
- `alt`: alternative text, descrive l'immagine (uso tipico: reader per non vedenti)
- `name`: un nome per riferirsi all'immagine
- `width` e `height`: per ridimensionare l'immagine (bad practice, meglio farlo da CSS).

Usando i tag figure si può creare un'immagine con descrizione:

```
<figure>
    <img src="images/prova.jpg" alt="Una immagine">
    <figcaption>Un'immagine</figcaption>
</figure>
```

7.1.9 Tabelle

Un gran casino e quasi nessuno le usa

7.1.10 Form

I form sono usati per inviare dati ai server per essere elaborati. La sintassi è:

```
<form>
    <label>
        Inserisci il tuo nome:
        <input type="text" name="Nome">
    </label>
    <button type="submit">Submit</submit>
</form>
```

Un form può essere formato da:

- `input`: per inserire i dati, possono essere di vari tipi
  - `type="text"`: per inserire un breve testo
  - `type="radio"`: per opzioni a scelta singola
  - `type="checkbox"`: per opzioni a scelta multipla
  - …
- `button`: si possono aggiungere pulsanti nei form, se non viene specificata la funzione col tag `submit` un bottone viene considerato di default come un tipo submit (ovvero invia i dati al server quando cliccato)
- `label`: permette di etichettare i campi del form per aiutare l'utente

7.1.11 Attributi globali

Oltre agli attributi specifici di ogni tag (per i quali è sempre meglio guardare MDN), ci sono degli attributi universali per tutti i tag:

- `id`: è l'identificativo di un certo tag html, idealmente ogni id compare una sola volta in tutta la pagina web, anche se non si rompono i siti se la condizione non viene rispettata
  Esempio di uso di `id`: do al tag `nav` l'id `'navigazione'`, nessun altro elemento della pagina avrà lo stesso id.
- `class`: anche questo identifica un certo tag, con la differenza che può (e in certi casi deve) essere usato su più elementi nella stessa pagina
- `style`: attributo per assegnare uno stile inline ad un elemento html (vedi CSS)
- `title`: attributo per dare un testo secondario all'elemento, utile per questioni di accessibilità e semantica.
- `data-`: usando il prefisso `data-` seguito da una qualsiasi parola io voglia posso creare attributi personalizzati (torna comodo in combo col CSS)

7.1.12 head

Il tag `head` si trova prima del body e contiene meta-informazioni su tutto il documento

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
        <link rel="stylesheet" type="text/css" href="style.css">
        <script src="script.js"></script>
</head>
```

Dove:

- `title` indica il titolo del documento
- `link` permette di linkare documenti a tutto il documento
- `script` permette di caricare script
- `meta` fornisce meta-informazioni sul documento (eg. la codifica usata)

7.1.13 Tag oggetto

Introdotti da HTML5, permettono di integrare più facilmente i contenuti multimediali nelle pagine, sono:

- `canvas`: definisce un'area rettangolare su cui disegnare immagini 2D e modificarle con il JS
- `video` e `audio`: permettono di inserire contenuti vide/audio in una pagina senza bisogno di plugin esterni, purtroppo non esistono codifiche universali quindi tramite il tag `src` bisogna specificare più file in vari formati.

7.1.14 Il DOM

Il **Documetn Object Model** (DOM) è un concetto chiave soprattutto con Javascript.

In pratica un file HTML una volta visualizzato in un browser può essere "letto" tramite gli strumenti per sviluppatori (ctrl + shift + i). Quello che vediamo nell'inspector però non è più il file HTML ma il DOM, ovvero il codice HTML renderizzato.

La differenza sostanziale è che l'HTML è statico, una volta salvato il file, quello è e quello rimane, mentre il DOM può diventare diverso dell'HTML da cui viene generato, ad esempio del codice Javascrip potrebbe far comparire un form.

8 CSS E TIPOGRAFIA

8.1 TIPOGRAFIA

Il **graphic design** è l'arte e il mestiere di creare lo stile e la presentazione visuale di un testo o documento multimediale.

Noi parleremo in particolare della *tipografia*, ovvero la disposizione dei tipi (forme di caratteri) per creare testo leggibile e piacevole alla vista.

8.1.1 Font

Un **font** è una collezione di forme di caratteri integrate armoniosamente per le necessità di un documento in stampa.

Un type face (o *font-family*) è uno stile unico di caratteri che viene espanso in diversi font di dimensione, peso e stili diversi (bold, italic, …)

TERMINOLOGIA

Nell'immagine, in ordine, abbiamo:

- Caratteri tondi (o romani)
- Caratteri italici
- Caratteri gotici (o blackletter)
- Caratteri corsivi (o script)
- Caratteri monospaziati

8.2 CSS

CSS può essere usato assieme a HTML in tre modi

- Inline (posizionando nel tag a cui si riferisce)
- Dentro al tag `style`.
- In un file esterno indicato dal tag `link`.

La cosa migliore è usare fogli di stile a parte e collegarli col tag `link`, in modo da mantenere il massimo dell'elasticità e riusabilità del codice.

8.2.1 Sintassi CSS

CSS di compone di selettori e proprietà. I selettori possono riferirsi a tag specifici o a id o a classi:

```
selettore {
    attributo: valore;
}
```

Come detto, i selettori possono selezionare generalmente un tag, per esempio:

```
h1 {
    color: rgb(255,0,0);
}
```

Seleziona tutti i tag `h1` nella pagina e ne colora il testo di rosso. Similmente si può selezionare qualsiasi tag: `div`, `a`, `body`, `main`, `section`, …

Per selezionare gli id si usa il carattere #, seguito dall'id vero e proprio

```
<h1 id='titolo'>Titolo</h1>

#id {
    font-family: 'Times New Roman';
}
```

In questo caso l'header con id titolo sarà scritto in Times new Roman, tutti gli altri resteranno invariati.

Allo stesso modo le classi vengono selezionate anteponendo il . davanti al nome della classe:

```
.classe-prova {
    text-align: center;
}
```

In questo caso tutti gli elementi con la classe-prova vengono allineati al centro della pagina (solo se sono elementi testuali).

Infine, è possibile combinare i selettori per aumentarne la specificità:

```
<h1 class="prova">Titolo di prova</h1>
<p class="prova">Blablablabla</p>

p.prova {
    font-size: 40px;
}
```

Nonostante sia l'h1 che il p hanno la classe prova, in questo caso solo l'elemento p verrà modificato per avere una dimensione del testo di 40px.

ATTENZIONE: scrivere p.prova o p .prova NON è la stessa cosa!
Nel primo caso selezioniamo tutti i tag pragrafo che hanno **anche** la classe prova,
nel secondo caso selezioniamo tutti i tag paragrafo **e** tutti i tag che hanno la classe prova.

8.2.2 Unità di misura

Le unità di misura che si possono usare in CSS sono tantissime e in continuo aumento; segnaliamo solo le più usate (e che verosimilmente saranno le uniche che userete nelle vostre vite).

- %: esprimi la dimensione in pixel
- px: esprimi la dimensione come frazione dell'elemento contenitore

```
<div>
    <button>Click Me!</button>
</div>
```

```
div {
    width: 400px;
    height: 100px;
}
button {
    width: 80%;
    height: 100%;
}
```

In questo caso il bottone sarà largo 320px e alto 100px.

- em: è un'unità di misura relativa molto usata, è simile al %, fa riferimento alla dimensione definita nell'elemento padre;

```
div {
    font-size: 16px;
}
```

```
h1 {
    font-size: 2em;
}
```

- rem: diminutivo di relative em, è simile all' però le dimensioni non sono relative all'elemento padre ma all'elemento radici, solitamente il body (ovvero lo stile globale del documento);

```
/* usando lo stesso html di prima */
body {
    font-size: 12px;
}
```

```
div {
    font-size: 16px;
}
```

```
h1 {
    font-size: 2rem;
}
```

In questo caso il titolo avrà dimensione , poiché il rem fa riferimento alle dimensioni definite dalla radice del documento.

Di solito si usa rem piuttosto che em.

- vh, vw: vertical height e vertical width sono unità di misura che fanno riferimento alla dimensione della finestra in percentuale;

```
div {
    width: 100vw;
    height: 50vh;
}
```

In questo caso i div saranno larghi quanto tutta la lunghezza della finestra e alti quanto metà dell'altezza della finestra, se la finestra viene ridimensionata i vengono ridimensionati di conseguenza. Non è obbligatorio usare vw per le larghezze e vh per le altezze.

- cm, mm: ogni tanto vengono usati ma di rado, definiscono le dimensioni in centimetri e millimetri.

8.2.3 Colori

Per definire i colori in CSS ci sono vari modi, nessuno particolarmente migliore quindi si va a sentimento:

- **Hex code:** si indica il valore rgb con numeri esadecimali, dove i pimi due caratteri indicano il valore del rosso, i due centrali de verde e gli ultimi due del blu;

```
div {
    background-color: #FF5500;
}
```

Questo hex code indica un colore sull'arancione.

- rgb, rgba; definisce il colore con valori rgb, nel caso di rgba definisce anche l'opacità del colore (dove 100 indica il colore pieno, come se si usasse rgb mentre 0 indica il colore trasparente).

```
div {
    background-color: rgb(0, 128, 128);
}
```

```
div.prova {
    background-color: rgba(255, 0, 255, 50);
}
```

In questo codice abbiamo uno sfondo turchese spento con sopra un elemento dallo sfondo viola/fucsia che però farà trasparire parte dello sfondo sottostante.

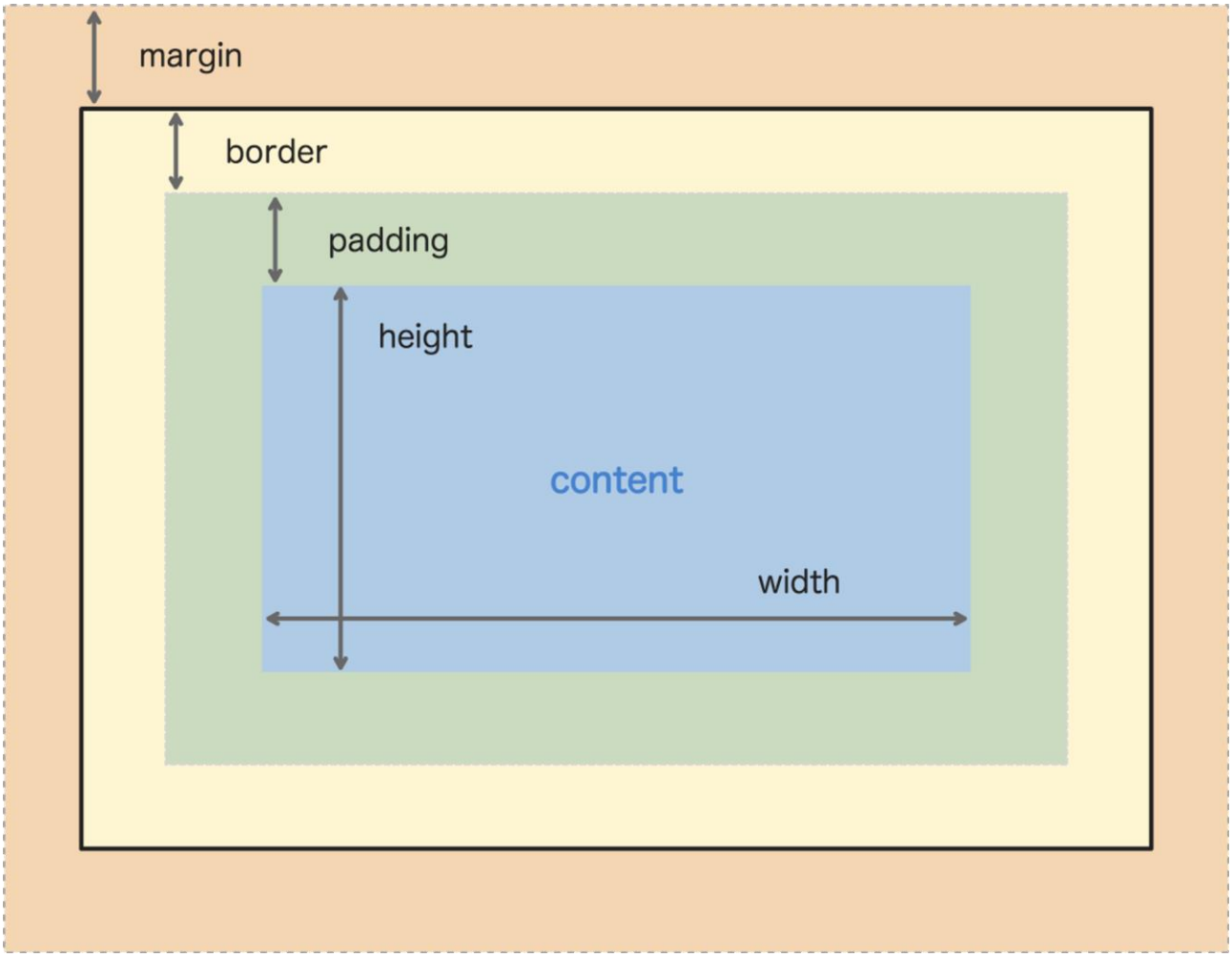
- hsl: definisce il colore con i parametri di hue (tonalità), saturation, e lightness

```
div {
    background-color: hsl(202, 50, 100);
}
```

In questo caso abbiamo un azzurro delicato.

8.2.4 Il Box Model

In HTML/CSS il concetto fondante è quello del box model: qualsiasi elemento della pagina (da un pulsante a un testo) è contenuto in un box che ha determinate proprietà uguali per tutti gli elementi:



Il margine definisce una sorta di zona cuscinetto attorno all'elemento che rimarrà libera, viene utilizzato per distanziare gli elementi della pagina fra loto.

Il bordo può essere customizzato tramite CSS ed è per l'appunto il bordo del box che contiene l'elemento.

Il padding è simile al margine ma sta all'interno del bordo e "estende" il contenuto del box, per intenderci la differenza principale è che se l'elemento ha un `background-color`, il padding sarà uno spazio intorno all'elemento che avrà comunque lo stesso colore di sfondo, mentre il margin sarà trasparente.

Inoltre se ci sono due elementi con margine uno di fianco all'altro i margini vengono sovrapposti, quindi se entrambi hanno `margin: 10px`, tra di essi non ci saranno `20px` di distanza ma `10`.

Queste sono le principali proprietà di CSS, per impararlo la cosa migliore è fare pratica e consultare spesso dei cheat-sheet come quello di W3School, eventualmente tutti questi tag diventeranno familiari. Consiglio fortemente anche le guide (guide non i tutorial) di CSS-triks.

9 JAVASCRIPT

9.1 SINTASSI BASE

ECMAScript è il nome originale di Javascript

Gli script javascript si eseguono client-side tramite **eventi** (click, mouseover, tasti di tastiera, …) e server-side tramite **routing** (aprendo connessioni a URI)

Possono essere eseguiti:

- In maniera *sincrona*, appena lo script viene letto, in un tag `<script>` o in un file
- In maniera *asincrona*, associando il codice ad un evento sul documento (click, …)
  - event-oriented processing
  - associando codice al completamento di un'operazione, o a un timeout

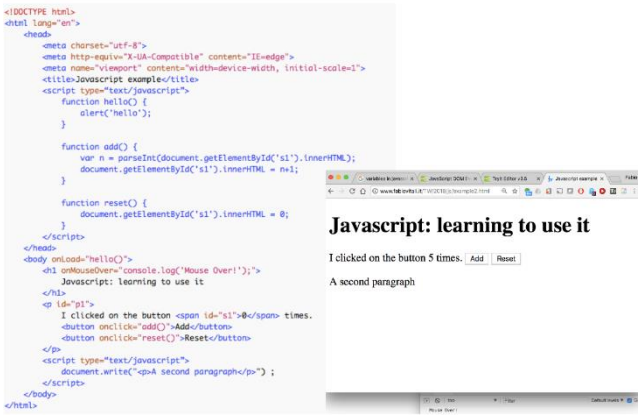
Output:

è possibile usare `document.write(string);` per scrivere direttamente sulla pagina
`console.log(string);` per scrivere sulla console
`alert(string);` in una finestra di alert
`document.getElementById(id).innerHTML = string;` modifica il DOM

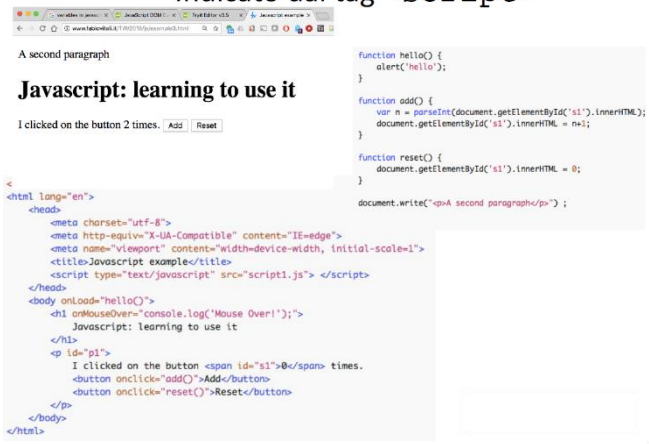
HTML prevede l'uso di script in 3 modi diversi:

- posizionato dentro l'attributo di un evento
- posizionato nel tag `<script>`
- indicato in un file puntato dal tag `<script>`

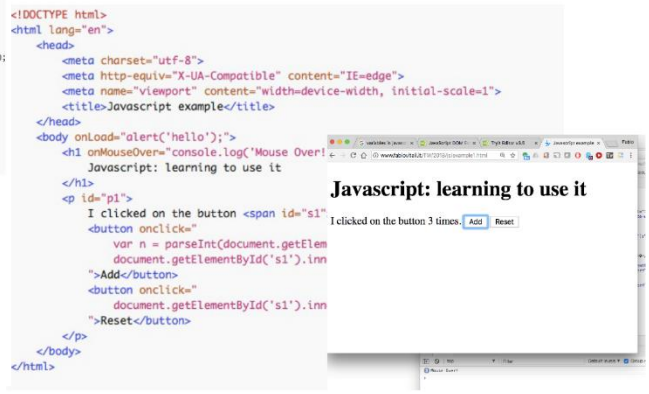
Posizionato nel tag <script>



Indicato dal tag <script>



Posizionato dentro all'attributo di un evento



9.2 JAVASCRIPT BASE

9.2.1 Tipi di dato

- booleani
- numeri (int, float)
- stringhe
- null
- undefined
- object (comprendono gli array e sono strutturati)

9.2.2 Operatori

9.2.3 Variabili

Le variabili non sono tipate in JS

```
var pippo ;
pippo = "ciao" ;
pippo = 15;
pippo = [1, 2, 3] ;
```

```
var pippo='ciao' ;
//definisce una
variabile nello
scope della funzione
o del file in cui
si trova.
```

```
let pippo='ciao';
// definisce una
variabile nello
scope del blocco
parentetico o della
riga in cui si trova.
```

```
const pippo='ciao';
//definisce una variabile non
//ulteriormente modificabile.
```

NUMERI

Operatore	Descrizione	Esempio	Commento
+	Somma	let a = 5 + 7	a vale 12
-	Sottrazione	let b = 17 - 2	b vale 15
*	Moltiplicazione	let c = 5 * 4	c vale 20
/	Divisione	let d = 28 / 4	d vale 7
%	Modulo	let e = 15 % 6	e vale 3 (15/6 = 2 resto 3)
**	Esponente	let f = 3**2	f vale 9 (cioè 3^2)
++	Incremento	e++	e vale 4 (3 + 1)
--	Decremento	f--	f vale 8 (9 – 1)

STRINGHE

+	Composizione	g = “Hel” + “lo”	g vale “hello”
+	Composizione + casting	h = “5” + 7	h vale “57”

CONFRONTO E BOOLEANI

Operatore	Descrizione	Esempio	Commento
==	Uguaglianza	let i = b==c let j = 5==’5’	i è falso (b vale 15 e c vale 20) j è vero (con casting di 5 in ‘5’)
<	Minore	let k = b < c	k è vero
>	Maggiore	let l = b > c	l è falso
<=	Minore o uguale	let m = b <= c	m è vero
>=	Maggiore o uguale	let n = b >= c	n è falso
!=	Disuguaglianza	let o = b != c let p = 5!=’5’	o è vero p è falso (con casting di 5 in ‘5’)
===	Uguaglianza senza casting	let q = 5===’5’	q è falso (non c’è casting di 5 in ‘5’)
!==	Disuguaglianza senza casting	let r = 5!==’5’	r è varo (non c’è casting di 5 in ‘5’)
&&	AND	let s = i && j	s è falso
||	OR	let t = i || j	t è vero
!	NOT	let u = !t	u è falso

9.2.4 Strutture di controllo

9.2.4.1 *Condizionali*

- Blocco if

```
if (a==5)
    istruzione_singola;

if (a==5) {
    istruzione_1;
    ...
}

if (a==5) {
    istruzione_1;
    ...
} else {
    istruzione_2;
    ...
}
```

- Blocco switch

```
switch (a) {
    case 'a':
        istruzione_1; istruzione_2;
        ...;
        break;
    case 'b':
        istruzione_3; istruzione_4;
        ...;
        break;
    ...
    default:
        istruzione_n;
istruzione_npiu1;
        ...;
}
```

- Operatore ternario

```
let x = (b==5 ? 'pippo' : 'paperino')
```

9.2.4.2 *Cicli*

- Blocco for

```
for (let i=0; i<k; i++) {
    istruzione_1;
    istruzione_2;
    alert(i);
    ...
}
```

- Blocco for … in

```
for (j in obj) {
    istruzione_1;
    istruzione_2;
    alert(obj[j]);
    ...
}
```

- Blocco while

```
while (k < 5) {
    istruzione_1;
    istruzione_2;
    ...
}
```

- Blocco do … while

```
do {
    istruzione_1;
    istruzione_2;
    ...
} while (k < 5)
```

9.2.4.3 *Eccezioni*

- Blocco try … catch

```
let x = prompt("Enter a number between 1 and 9; ", "");

try {
    let el = document.getElementById("menu"+x)
    let address = el.attributes["href"].value
    return address;
} catch (e) {
    return "input value out of bounds";
}
```

9.2.5 Funzioni

Sono blocchi di istruzioni con un nome e facoltativamente dei parametri.
Possono restituire un valore di ritorno, le funzioni non sono tipate ma i valori di ritorno sì.

```
function double(n) {
    let m = n + n;
    return m;
}
let a = double(4);
let b = double('test');
```

Se manca un parametro non restituisce errore ma assume che sia undefined.

```
function double(n) {
    if (typeof n!= undefined) {
        let m = n + n;
    } else {
        let m = 0;
    }
    return m;
}
let c = double();
```

9.2.6 Tipi di dati strutturati

In javascript c'è un solo tipo di dati strutturati: gli oggetti.

E ti dirò di più, gli array sono un sotto-insieme degli oggetti.

E ti dirò di più ancora, in javascript tutto è un oggetto!

Infatti se si prova ad aprire uno script nel browser e si stampa nella console un oggetto o un array o altri tipi di dati (ci torniamo), si vedrà come ogni dato è accompagnato da un campo prototipo chiamato `<prototype>`, esso contiene le proprietà che il nostro oggetto eredita dal suo super oggetto chiamato `__proto__`; questo meccanismo fa si che gli array, le stringe, gli oggetti e praticamente qualsiasi tipo di dato nasce con dei metodi predefiniti che possono essere usati con esso (ad esempio l'array ha il metodo `.length` che restituisce la lunghezza dell'array, mentre le stringhe hanno il metodo `.toUpperCase` che rende la stringa tutta in maiuscolo).

Tornando agli oggetti, un oggetto in js è un tipo di dato strutturato nella forma `{campo:contenuto}`, il suo contenuto è definito in piena libertà e i campi sono separati da virgola

```
const foo = {
    nome: 'Giovanni',
    anni: 20,
    handler: (x) => {console.log(x)},
    vivo: true,
    sport: {
        terra: 'calcio',
        acqua: 'nuoto'
    }
}
```

Un oggetto contiene dati eterogenei, incluse funzioni e altri oggetti. L'ultimo campo non termina con una virgola (alcuni editor non lo segnalano come errore ma il browser potrebbe intendere che ci sia un ulteriore campo contenente `undefined` che potrebbe dare comportamenti inaspettati nell'utilizzo).

9.2.6.1 *Accesso ai campi*

Per accedere ai dati contenuti in un oggetto si può usare:

- la dotted-notation `oggetto.campo`
- la square-braket-notation `oggetto['campo']`

9.2.6.2 *Array*

Abbiamo detto che gli array sono un tipo speciale di oggetto: possono essere dichiarati usando la parentesi quadra invece della graffa e come campi utilizzano l'indice del contenuto che è assegnato automaticamente.

Anche nel caso degli array questi possono contenere qualsiasi tipo di dato; funzioni, altri array e oggetti:

```
const arr = [
    'Giovanni',
    22,
    {
        residenza: 'Bologna'
    }
]
```

Per i metodi che possono essere usati sugli array rimando alla lista infinita su MDN.

9.2.7 JSON

JSON (JavaScript Object Notation) nasce come derivazione della notazione javascript degli oggetti.

Il suo funzionamento è in tutto e per tutto uguale a quello degli oggetto a parte che, per motivi di sicurezza, JSON non contiene codice eseguibile; quindi, se si aggiunge una funzione in un JSON questa verrà salvata come stringa e non sarà possibile chiamarla per eseguirne i contenuti.

L'altra differenza è che, mentre in js i nomi dei campi possono essere o meno fra virgolette, in JSON le virgolette sono obbligatorie

9.2.8 Altri oggetti

Per una lista degli oggetti built-in di js (compresi Math, Data, …) rimando alla documentazione su MDN.

9.3 MODELLO OGGETTI DEL BROWSER

Oltre a quanto visto finora javascript possiede anche delle funzionalità specifiche per interagire con il browser e la finestra visualizzata (e manipolare quindi il DOM)

9.3.1 Oggetti principali

- `window`: è l'oggetto top-level della finestra del browser, è dotato di proprietà e metodi
  - posizione `moveBy(x,y), moveTo(x,y)`
  - dimensioni `resizeBy(x,y), resizeTo(x,y)`
  - altre finestre `open("URLname","Windowname",["opt"])`
- `navigator`: è l'oggetto con le proprietà del client come nome, numero di versione, plug-in installati, supporto per i cookie, etc.
- `document`: rappresenta il contenuto del documento, le proprietà per manipolarlo sono
  - `document.title`
  - `document.forms[0]`
  - `document.querySelector`
  - `...`

9.3.2 Metodi per manipolare il DOM

Javascript include metodi standard per manipolare il DOM. Data la mole di metodi forniti rimando alla documentazione di MDN:

- Introduzione al DOM;
- HTML DOM API;
- Manipulating Documents.

9.3.3 I selettori

Una parte dei metodi per manipolare il DOM sono i selettori, che si usano per riferirsi a parti del DOM e manipolarle tramite javascript. I selettori sono:

- `getElementById`: ritorna il primo elemento corrispondente id, undefined se non esiste
- `getElementsByName`: ritorna un array con tutti gli elementi con l'attributo name corrispondente
- `getElementsByTagName`: ritorna un array con tutti i tag con nome specificato
- `getElementsByClassName`: ritorna un array con tutti gli elementi con la classe corrispondente
- `querySelector`: ritorna il primo elemento che corrisponde alla query passata (eg. per cercare una classe come argomento si passa `querySelector`)
- `querySelectorAll`: ritorna una lista con tutti gli elementi che corrispondono alla query passata

Volendo uno può campare usando solo i due `querySelector`.

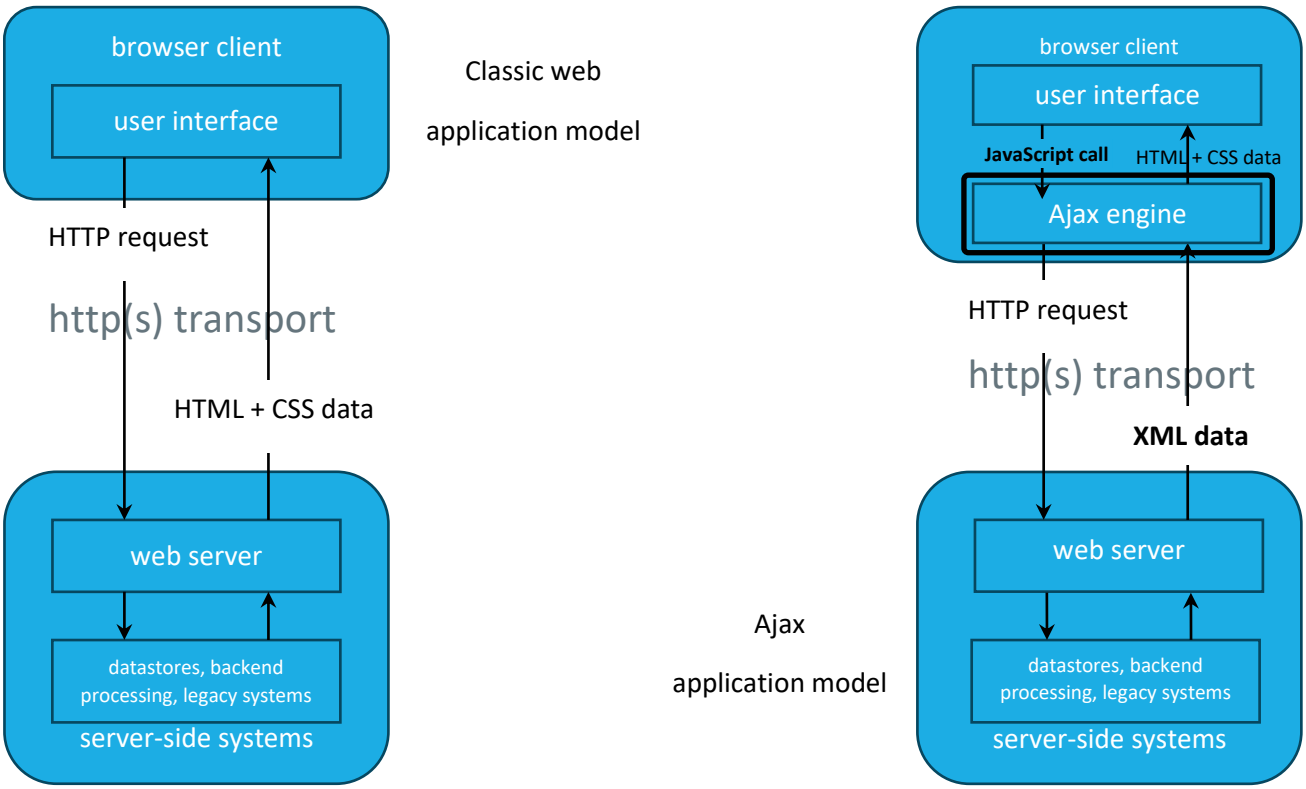
9.4 AJAX

AJAX (**A**synchronous **J**avaScript **A**nd **X**ML) è una tecnica per la creazione di applicazioni Web interattive che permette l'aggiornamento **asincrono** di **porzioni** di pagine HTML

Viene utilizzato per incrementare l'**interattività**, la **velocità** e l'**usabilità**.

NON è un linguaggio di programmazione o una tecnologia specifica

Indica l'utilizzo di una combinazione di tecnologie



Pregi

- Usabilità
  - Interattività (Improve user experience)
  - Non costringe l'utente all'attesa di fronte ad una pagina bianca durante la richiesta e l'elaborazione delle pagine.
- Velocità
  - Minore quantità di dati scambiati (non è necessario richiedere pagine intere)
  - Una parte della computazione è spostata sul client
- Portabilità
  - Supportato dai maggiori browser
  - Se correttamente utilizzato è platform-independent
  - Non richiede plug-in

Difetti

- Usabilità
  - Non c'è navigazione ⇒ il pulsante "back" non funziona
  - Non c'è navigazione ⇒ l'inserimento dei segnalibri non funziona
  - Essendo i contenuti dinamici non vengono indicizzati correttamente dai motori di ricerca
- Accessibilità
  - Non supportato da browser non-visuali
  - Richiede meccanismo di accesso alternativi
- Configurazione:
  - È necessario aver abilitato Javascript
- Compatibilità
  - È necessario un test sistematico sui diversi browser per evitare problemi dovuti alle differenze fra i browser
  - Richiede funzionalità alternative per i browser che non supportano JS

9.4.1 Creare un'applicazione AJAX

Un'app AJAX è divisa in momenti chiave:

1. Creazione e configurazione delle richieste per il server
   - Usando XMLHTTPRequest
2. Attivazione della richiesta HTTP …
3. ***… passa del tempo …***
4. … ricezione della risposta HTTP e analisi dei dati (o dell'errore)
5. Modifiche al DOM della pagina

9.4.2 I framework Ajax

Sono librerie Javascript che semplificano la vita nella creazione di applicazioni Ajax anche complesse.

Hanno tre scopi fondamentali:

- ***Astrazione***: gestiscono le differenze tra un browser e l'altro e forniscono un modello di programmazione unico (o quasi) che funziona su tutti o molti browser.
- ***Struttura dell'applicazione***: forniscono un modello di progetto dell'applicazione omogeno, indicando con esattezza come e dove fornire le caratteristiche individuali dell'applicazione
- ***Libreria di widget***: forniscono una (più o meno) ricca collezione di elementi di interfaccia liberamente assemblabili per creare velocemente interfacce sofisticate e modulari

Categorie di framework

- Modello applicativo
  - Framework interni
    Usati direttamente nella pagina HTML
  - Framework esterni
    Usati nel processo di sviluppo client-server
- Ricchezza funzionale
  - Librerie di supporto JavaScript
    Prototype, jQuery, MooTools
  - Framework RIA (Rich Internet Application)
    Ext, GWT, YUI, Dojo

9.5 JAVASCRIPT AVANZATO

9.5.1 Valori falsy e truthy

Javascript definisce come *falsy* quei valori che in caso di casting booleano diventano falsi:

- *false*
- *0*
- *null*
- *undefined*
- *“”*
- *NaN*

Tutto il resto è *truthy*, (ovvero cast a true), inclusi:

- *“any non-empty string”, 3.14, Infinity*
- *“0”, “undefined”, “null”*
- *{}*
- *[]*

Normalmente (linguaggi C, Java, …) per verificare se un valore è falso o vero scriverà una cosa del tipo

```
if (value != null && value.lenght > 0) {
    // ok agisci
}
```

In Javascript c’è casting automatico, che permette una scrittura più semplice e veloce:

```
if (value) {
    // ok agisci
}
```

Però, se arrivano parametri da fuori bisogna inizializzarli a un valore decente.
Supponiamo che `param` sia spesso un numero ma non sempre:

```
if (param == null || param == 0) {
    misura = "12px";
} else {
    misura = param + 'px';
}
```

oppure si può usare un operatore ternario:

```
misura = (param ? param : '12') + 'px';
```

Ma javascript ha un modo ancora più semplice:

```
misura = (param || '12') + 'px';
```

In pratica moltissimi programmatori lo usano come verifica della presenza e istanziazione di una variabile o la disponibilità di una libreria o un servizio.

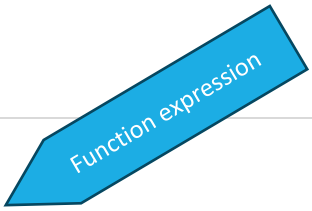
9.5.2 Funzioni come entità di prima classe

In Javascript le funzioni sono oggetti quasi come tutti gli altri, e possono essere:

- Essere assegnate a variabili
- Essere passate come parametri di funzione
- Essere restituite da una funzione
- Essere elementi di un object
- Essere elementi di un array

In più:

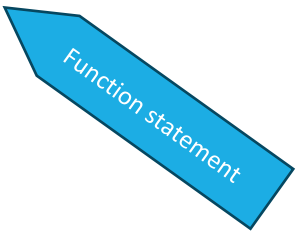
- Essere invocate con l’operatore ( )



Si può assegnarle a una variabile:

```
var potenza = function(a, b) {
    return Math.pow(a, b);
}
```

```
function potenza(a, b) {
    return Math.pow(a, b)
}
```



Si può assegnare una funzione come metodo di un oggetto

```
var e = {p:3, q:5, r:7}
e.sum = function() {
    return this.p + this.q + this.r
}
```

Solo col nome semplice è una variabile, se si aggiungono le parentesi diventa un'invocazione e la funzione viene eseguita:

```
if (e.sum) {   // controllo l'esistenza: variabile
    var c = e.sum() // invoco ed eseguo: funzione
}
```

Si può assegnarle come proprietà di un oggetto o di un array:

```
var persona = {
    nome: 'Giuseppe',
    cognome: 'Rossi',
    altezza: 180,
    nascita: new Date(1995, 3, 12),
    saluta: function(name, id) {
        var saluto = 'Ciao ' + name
        if (id) {
            document.getElementById(id).innerHTML = saluto;
        } else {
            alert(saluto);
        }
    }
}
```

Si possono restituire funzioni come risultato di altre funzioni:

```
var expGenerator = function(e) {
    return function(b) {
        return Math.pow(b, e)
    }
}
```

Posso creare delle funzioni in serie chiamando il generatore:

```
var quadrato = expGenerator(2);
var cubo = expGenerator(3);
```

E queste sono vere funzioni:

```
var c = quadrato(5);  // c vale 25
var d = cubo(5);      // d vale 125
```

Posso passare funzioni anonime come parametri di funzione:

```
var msg = document.getElementById("msg");
msg.innerHTML = '<p>via!<p>' ;
window.setTimeout(function() {
    msg.innerHTML += '<p>1 secondo è passato!<p>' ;
}, 1000);
window.setTimeout(function() {
    msg.innerHTML += '<p>2 secondi sono passati!<p>' ;
}, 2000);
window.setTimeout(function() {
    msg.innerHTML += '<p>3 secondi sono passati!<p>' ;
}, 3000);
```

Nota: `setTimeout(f,n)` esegue la funzione `f` dopo n millisecondi dalla invocazione.

La funzione `bind(obj,args)` permette di associare parametri a funzioni anonime o chiamate indirettamente:

```
var msg = document.getElementById("msg");
msg.innerHTML = '<p>via!<p>' ;
for (var i=0; i<=3; i++) {
  window.setTimeout(
    function(n) {
        this.innerHTML += '<p>' + n +' secondi ono passati';
    }.bind(msg, i),
    i*1000
  );
}
```

Nella chiamata `bind(obj,args)`,`obj` rappresenta l'oggetto a cui verrà associata la funzione (cosa trovo dentro alla variabile predefinita `this`) mentre `args` sono gli argomenti di ciò che voglio passare alla funzione.

9.5.3 Funzioni filtro su array

Gli array di javascript hanno tantissimi metodi che accettano una funzione come parametro. Permettono di fare specifiche operazioni sugli elementi dell'array in maniera veloce e sistematica. Per esempio:

```
let salespeople = [
    { name: 'Alice', cognome: 'Bruni'   , sales:  78500 },
    { name: 'Bruno', cognome: 'Verdi'   , sales: 135000 },
    { name: 'Carla', cognome: 'Rossi'   , sales: 251200 },
    { name: 'Dario', cognome: 'Bianchi' , sales:   7500 }
]
let byCognome = function(i,j) {return i.cognome > j.cognome ? 1 : -1}
let largerthan100 = function(i) {return i.sales >= 100000}
let best = function(i) { i.best = true }

let sorted = salespeople.sort(byCognome);

salespeople.filter(largerthan100).forEach(best);
```

sorted contiene gli elementi di salespeople ordinati per cognome

Ho selezionato solo gli elementi di salespeople con vendite >= 100000, poi ho assegnato loro il campo best a true. Ora salespeople è:

```
[ { name: 'Alice', cognome: 'Bruni'   , sales:  78500             },
  { name: 'Bruno', cognome: 'Verdi'   , sales: 135000, best: true },
  { name: 'Carla', cognome: 'Rossi'   , sales: 251200, best: true },
  { name: 'Dario', cognome: 'Bianchi' , sales:   7500             } ]
```

- `array.sort(f)`
  restituisce un array ordinato sulla base della funzione `f` che deve avere due parametri e restituire un valore 1 (stesso ordine) o -1 (inverti l'ordine)
- `array.filter(f)`
  restituisce un secondo array che contiene solo gli elementi che soddisfano la funzione booleana `f`.
- `array.some(f), array.every(f)`
  restituisce vero se almeno un elemento (`some()`) o tutti gli elementi (`every()`) soddisfano la funzione booleana `f`
- `array.find(f)`
  restituisce il primo elemento che soddisfa la funzione booleana `f`
- `array.foEach(f)`
  esegue sull'array la funzione `f` permettendo di modificare l'array direttamente
- `array.map(f)`
  crea un nuovo array in cui ogni elemento viene modificato dalla funzione `f`
- `array.reduce(f)`
  esegue su ogni elemento dell'array la funzione `f` passando il risultato dell'esecuzione precedente.
  Ottimo per fare totali.

9.5.4 Classi e prototipi

Javascript è object-oriented anche se non tipato come java.
In un linguaggio object oriented tradizionale, la classe è un template sulla base del quale vengono istanziati gli oggetti del programma, specificando **membri** e **metodi**.

Javascript ha oggetti ma non sono basati sul concetto di classi, piuttosto su quello di **prototipo**.
Poiché le funzioni sono oggetti di promo livello, ogni oggetto può contenere al suo interno delle funzioni, senza ricorrere alla classe.

Le classi non sono entità di primo livello, al loro posto si usano degli oggetti che provengono dallo stesso costruttore.
Un costruttore è banalmente una funzione che restituisce un oggetto.

```
function Persona(nome,altezza,nascita) {    // costruttore
    this.nome = nome;
    this.altezza = altezza;
    this.nascita = nascita;
    this.saluta = function() { return 'ciao!' }
}
var mario = new Persona('Mario', 185, new Date(2002,3,14));
            // Creazione di un oggetto
var alice = new Persona('Alice', 168);
            // manca un parametro: non è un problema
            // alice.nascita è undefined
```

Ogni oggetto in JS è autonomo e gli si possono aggiungere tutti i metodi/proprietà che si vuole
Per aggiungere proprietà/metodi si usa l'oggetto prototype.

```
Persona.prototype.welcome = function() {
    alert("Benvenuto " + this.nome+"!");
}   // Aggiunta di una funzione al prototipo Persona

mario.welcome();    // Utilizzo della funzione
```

9.5.5 Closure e IIFE

Javascript non ha protezione dei membri privati di un oggetto, ma sono tutti accessibili e manipolabili.
Per esempio, definiamo la classe Counter con uno stato privato accessibile attraverso l'interfaccia data dalle funzioni `inc()` e `dec()` :

```
counter = function() {
    this.state = 0;             // privata
    this.inc = function() { return this.state++ };
    this.dec = function() { return this.state-- };
}
var c = new counter();
c.inc();     var d = c.inc()    // d = 2
c.state = 7; var e = c.inc()    // possibile, inoltre e = 8
```

Questo è molto grave; significa che l'interfaccia di `inc()` e `dec()` è solo apparente, non posso impedire l'accesso alle variabili privare

In Javascript ci sono 3 scope:

- quello globale
- quello della funzione
- *closure*,
  è lo scope della funzione all'interno della quale viene definita la funzione

Ad esempio, una funzione che restituisce una funzione ha uno scope che è sempre accessibile alla funzione interna, ma non dal mondo esterno.
In questo modo ottengo variabili veramente private.

```
counter = function() {
    var state = 0;             // privata
    return{
        inc: function() { return state++ },
        dec: function() { return state-- }
    }
}
var c = new counter();
c.inc();     var d = c.inc()    // d = 2
c.state = 7; var e = c.inc()    // e = 3, c.state è lecita ma inutile
```

9.5.6 IIFE (Immediatly Invoked Function Expression)

Sono funzioni anonime create e invocate immediatamente, serve per per fare *singleton* (oggetti non ripetibili) dotati di closure

```
var people = (function() {
    var persone [];
    return {
        add:   function(p) { persone.push(p); },
        lista: function() { return persone.join(', '); }
    }
})()
```

L'oggetto people, in questo caso, è un *singleton* con un array come stato interno e due metodi per accedere e modificare i valori.
Per la closure, la variabile persone è accessibile da `add` e `lista` , ma NON dall'esterno.
La coppia di parentesi alla fine invoca la funzione immediatamente.

9.6 ECMAScript 2015

È la standardizzazione presso ECMA di Javascript.

Funzioni freccia

Porta un nuovo modo per definire funzioni inline:

```
var power = function(a, b) {
    return Math.pow(a, b);
}
var c = power(5, 3)
```

```
var power = (a, b) => {
    return Math.pow(a, b);
}
var c = power(5, 3)
```

Utile per definire semplici funzioni di callback

```
var arr = [1, 2, 3];
var squares = arr.map(function (x) { return x * x; });

var arr = [1, 2, 3];
var squares = arr.map(x => x * x);
```

Template literals

Una nuova sintassi per definire stringhe multi-linea con interpolazione di variabili

```
var firstname = "John";
var x = `Hello ${firstname}!
How are you?`
```

x vale “Hello John!
How are you?

- Backtiks come delimitatori `
- I new line fanno parte della stringa
- Interpolatori: ${firstname}

Definizioni di classe

Un modo semplificato per creare oggetti in maniera retrocompatibile con Java e C++:

```
var shape = function(id, x, y) {
    this.id = id;
    this.x = x;
    this.y = y;
    this.move = null;
};
shape.prototype.move = function(x, y) {
    this.x = x;
    this.y = y;
};
```

```
class shape {
    constructor(id, x, y) {
        this.id = id;
        this.x = x;
        this.y = y;
    }
    move(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

Gli oggetti sono ancora basati sui prototipi ma le definizioni sono più semplici.

9.7 PROGRAMMAZIONE ASINCRONA IN JAVASCRIPT

La caratteristica peculiare e tipica di Javascript è la asincronicità come filosofia di design.

1. Una richiesta Ajax viene eseguita asincronicamente rispetto alla navigazione della pagina HTML
2. La gestione dei dati ricevuti via Ajax viene eseguita asincronicamente rispetto all'emissione della richiesta
3. La gestione degli eventi dell'utente viene eseguita asincronicamente rispetto alla specifica della funzione callback
4. setTimeout() posticipa di n millisecondi l'esecuzione di una funzione
5. …

Per esempio

```
var msg = document.getElementById("msg");
msg.innerHTML = '<p>Via!</p>';
window.setTimeout(function() {
    msg.innerHTML += '<p>1 secondo è passato</p>';
}, 1000);
```

il paragrafo `<p>Via!</p>` compare giustamente prima di quello della funzione.
Ma anche se fosse

```
var msg = document.getElementById("msg");
msg.innerHTML = '<p>Via!</p>';
window.setTimeout(function() {
    msg.innerHTML += '<p>0 secondi sono passati</p>';
}, 0);
```

il paragrafo `<p>Via!</p>` comparirebbe prima lo stesso.
L'interprete Javascript prima esegue completamente lo script in corso e DOPO esegue quelli in callback.

Questa era una domanda di un compito, che richiedeva il contenuto dell'elemento con id test dopo l'esecuzione dello script, assumendo che la connessione Ajax avesse restituito un codice "200"

```
function test() {
  el.innerHTML += "Ho messo ";
  ajax = new AjaxCall('GET', "http://www.mysite.com", true);
  ajax.success = (data) => { el.innerHTML += "l'arrosto "};
  ajax.error = (data) => { el.innerHTML += "la verdura "};
  ajax.send();
  el.innerHTML += "nel forno " ;
}
let el = document.getElementById('test')
el.innerHTML = '';
test();
```

Hanno risposto TUTTI:
"Ho messo l'arrosto nel forno"

Invece la risposta esatta è
"Ho messo nel forno l'arrosto"

Perché la funzione del risultato della chiamata Ajax viene eseguita asincronicamente rispetto allo script di invocazione

L'asincronicità ci serve ogni volta che dobbiamo chiamare un servizio del quale non abbiamo il controllo dei tempi di esecuzione della sua disponibilità.

```
var database = remoteService.setDatabaseAccessData();
var result = database.query("SELECT * FROM table");
var output = prepareDOM(result);
document.getElementById("display").appendChild(output);
```

Non ho controllo sui tempi di esecuzione del comando ***database.query***, che potrebbe metterci molto tempo.

Nel frattempo il processo è bloccato in attesa del ritorno della funzione, e l'utente non è fluido

La situazione potrebbe peggiorare se l'esecuzione avesse necessità, a catena, di tante altre richieste esterne non controllabili.

```
function searchProducts(query) {
    var async = true;
    var productDB = services.setDBAccess('products', async);
    var opinionDB = services.setDBAccess('opinions', async);
    var twitter = services.setDBAccess('twitter', async);

    var products = productDB.search(query);
    var opinions = opinionDB.search(products);
    var tweets = twitter.search(opinions);

    var output = prepareDOM(products, opinions, tweets);
    document.getElementById("display").appendChild(output);
}
```

Soluzione 1 – Logica server-side
Potrei usare un linguaggio multi threaded server-side e fare un'unica richiesta al server che si occupi di tutti i dettagli.

Ho comunque un'attesa e nessun particolare vantaggio da Ajax. Inoltre, distribuisco la logica dell'applicazione in due luoghi, complicando la gestione.

Soluzione 2 – Codice asincrono e callback
Posso passare una funzione callback come argomento di chiamata a funzione, che viene eseguita alla conclusione del servizio.

Però le callback:

- Non possono restituire valori alla funzione chiamante, ma solo eseguire azioni coi dati ottenuti.
- Sono funzioni indipendenti, e vengono eseguire alla fine dell'esecuzione della funzione che le chiama.
- Non hanno accesso alle variabili locali della funzione chiamante ma solo a quelle globali.

L'approccio delle callback è comunissimo ma javascript è *single thread*
Anche quando il servizio è molto veloce, le funzioni callback vengono comunque eseguite alla fine del flusso di esecuzione del thread chiamante.
Quindi i flussi asincroni vengono eseguiti sempre e solo alla fine dell'esecuzione del flusso principale.

Dopo un po' la cosa si complica. Entriamo nel *callback hell*

Soluzione 3 – le promesse
Una promessa è un oggetto che, si promette, che tra un po' conterrà un valore.
La promessa viene creata dalla funzione chiamante e mantenuta dalla funzione chiamata.

La cosa interessante delle promesse è che possiamo evitare il callback hell con una gestione più semplice delle chiamate a catena.

Soluzione 4 - generator/yield
Il generatore è una metafunzione (una funzione che restituisce una funzione che può essere chiamata ripetutamente e interrotta sino a che ne hai nuovamente bisogno).
Il comando yield mette in attesa l'assegnazione di valore fino a che non si chiude l'esecuzione della funzione chiamata.
La funzione next() fa proseguire l'esecuzione della funzione fino al prossimo yield.

Soluzione 5 – async/await

È un misto di sincronia e asincronia: l'esecuzione di questa funzione è bloccata finché ogni `await` risponde, ma non prosegue al successivo finché non risponde la prima.

Soluzione 6 – Promise.all

Se ho molte chiamate asincrone indipendenti (nelle quali non devo aspettare il risultato per chiedere la seconda) posso usare `Promise.all`.

10 SEMANTIC WEB

"Il Web semantico fornisce una struttura comune che consente di condividere e riutilizzare i dati tra applicazioni, aziende e comunità"

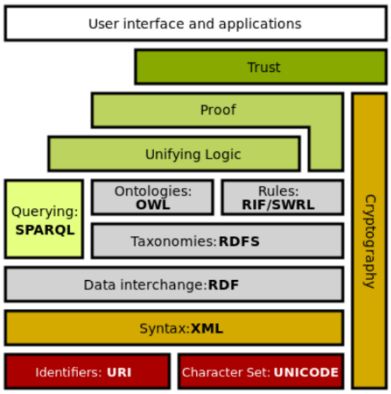
Il SW riguarda:

separare le **informazioni** dalla rappresentazione, per rendere le informazioni **facilmente navigabili** e comprensibili anche dalle macchine.

Come si passa dal vecchio web a quello semantico?

Servono tecnologie che forniscano una **descrizione formale dei concetti**, termini e **relazioni** dentro ad un determinato **dominio di conoscenza**.

L'architettura del SW



Ha diversi livelli:

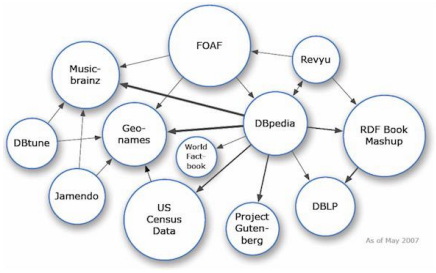
1. Al livello base ci sono le **risorse**
2. I **linguaggi di markup**, per la creazione di documenti composti da dati strutturati.
3. Gli **standard per lo scambio di informazioni**
4. Gli **standard per la creazione di tassonomie**
5. Linguaggi per costruire **ontologie** e standard per definire le **regole**,
6. Sulle regole e ontologie potremmo avere programmi e umani che tentano di **eseguire operazioni logiche** per la **verifica** dei fatti.

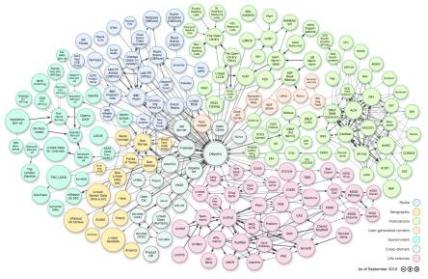
10.1 LD - LINKED DATA

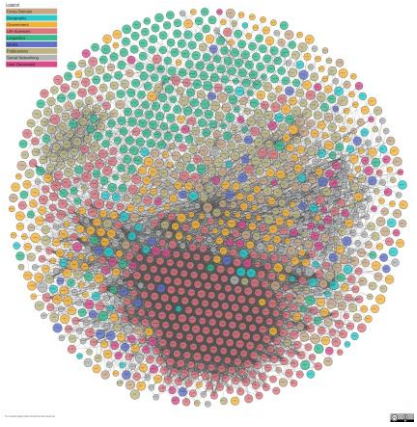
I Linked Data sono **dati strutturati** interconnessi con altri dati.

Linked Data è un'estensione di tecnologie Web standard (come HTTP, RDF e URI) pensata per **condividere le informazioni** in modo che possano essere interpretabili correttamente da **agenti automatici**.

Linked Data è pensato per essere il **database RDF** del SW.
I Linked Data accessibili e interrogabili pubblicamente si chiamano **LOD (Linked Open Data)**



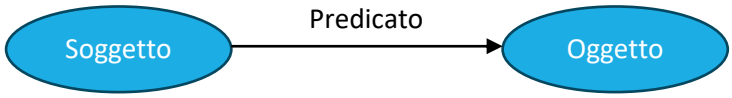




10.2 RDF - Resource Description Framework

RDF si basa sull'idea di fare **affermazioni (statement)** sulle risorse nella forma di **triple** soggetto-predicato-oggetto, in cui:

- **il Soggetto** è una risorsa (ha un URI)
- **l'Oggetto** è una risorsa (ha un URI) o un letterale (stringa, numero, data, …)
- **il Predicato** è una **relazione** tra risorse, una **proprietà**



Esempio: "*Umberto Eco è autore de Il Nome della Rosa*" può essere espresso **informalmente** in **triple** RDF:

Esiste un'entità E
identificato dall'URI <https://dbpedia.org/umberto_eco>
può essere qualunque altro l'indirizzo purché univoco.

1. E è un autore
   ("persona" è il suo tipo).
2. E si chiama "Umberto Eco"
3. E ha scritto B
   Identificato da <http://www.anobii.com/books/Il_nome_della_rosa>
4. B è un libro
   ("libro" è il tipo)
5. B si chiama "Il Nome della Rosa"

- Abbiamo 2 diversi **soggetti**:
  - E, B
- Abbiamo 3 diversi **predicati**
  - è un, si chiama, ha scritto
- Abbiamo 5 diversi **oggetti**:
  - autore, "Umberto Eco", B, libro, "Il Nome della Rosa"

Per un totale di **5 triple** RDF

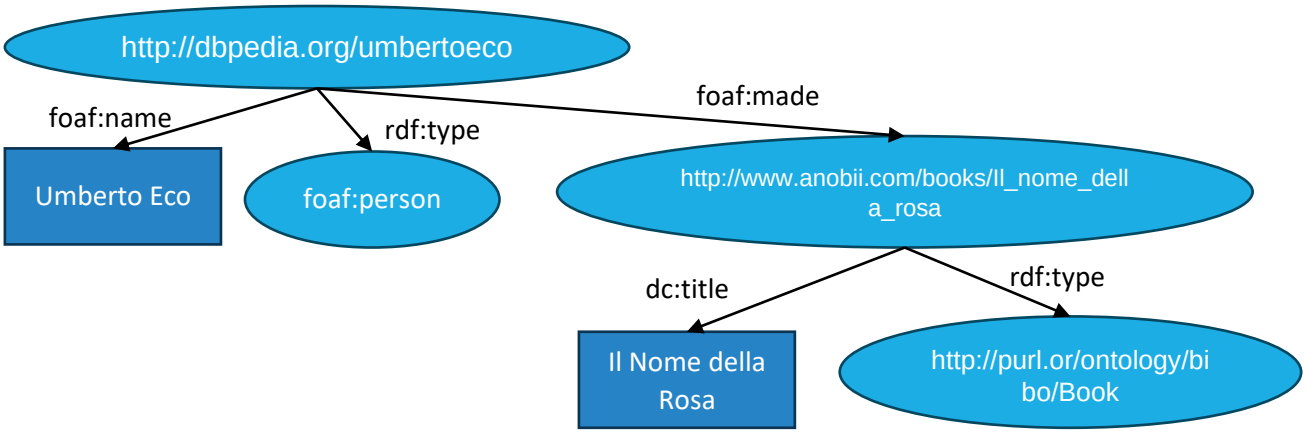
Dall'esempio possiamo notare che:
- tutti i **soggetti** sono URI
- tutti i **predicati** sono verbi
- gli **oggetti** possono essere URI o letterali

10.2.1 RDF – i grafi

Un grafo RDF è un insieme di triple RDF.
Si può rappresentare un grafo in molti modi, ad esempio attraverso una rete semantica tale che:

- Le **risorse** (soggetti e oggetti) sono rappresentate come nodi.
- I **predicati** sono rappresentati come archi.

Questa particolare rappresentazione è facilmente leggibile dagli umani.



Attenzione, stiamo usando diverse **ontologie** per gli URI di **tutti** i predicati e soggetti e degli oggetti che non siano stringhe di letterali ("Umberto Eco" e "Il Nome della Rosa")

Per RDF sono in uso diversi formati di serializzazione comuni:

- Turtle: un formato compatto e human friendly
- JSON-LD: una serializzazione basata su JSON
- RDF/XML: una sintassi basata su XML; il primo formato per la serializzazione di RDF
- ...

Il formato RDF/XML a volte viene chiamato RDF in modo fuorviante. Questo perché RDF/XML è stato introdotto con le specifiche W3C che definiscono RDF ed è stato il primo formato standard si serializzazione

10.2.2 RDF – Pro & Con

PRO

- Il modello a triple è **semplice**
- La struttura dati risultante è un **grafo**
- Il modello RDF è **modulare**, quindi
  - Elaborazione delle informazioni parallelizzata
  - Con *informazioni parziali* l'output è ancora un modello RDF coerente ed elaborabile
- RDF + OWL permette alle AI di ragionare sui dati RDF e di estrarre informazioni **efficientemente**
  - OWL è costruito sulla **Logica della Descrizione (DL)**
  - In DL esistono solo relazioni binarie o unarie ⇒ efficiente il ragionamento

CONS

- Il modello RDF è costituito da dati piccoli e frammentati; quindi, un database razionale di dimensioni medie può corrispondere a un triplestore contenente miliardi di triple
- Limitazioni delle relazioni N-arie.
  Il modello RDF non consente modi semplici per descrivere relazioni N-arie tra i vertici del grafo semantico

10.2.3 Esempi – Turtle & RDF/XML

Turtle: "*Umberto eco è autore de Il Nome della Rosa*"

```
@prefix anobii <http://anobii.com/books/> .
@prefix foaf <http://xmlns.com/foaf/0.1/> .
@prefix dc <http://purl.org/dc/terms/> .
@prefix db <http://dbpedia.org/> .

db:umbertoeco_
    foaf:name "Umberto Eco" ;
    rdf:type foaf:Person ;
    foaf:made anobii:il_nome_della_rosa .

anobii:il_nome_della_rosa
    dc:title "Il nome della rosa" ;
    a <http://purl.org/ontology/bibo/Book> ;
```

RDF/XML: "*Umberto eco è autore de Il Nome della Rosa*"

```
<rdf:RDF
    xmlsn:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dc="http://purl.org/dc/tems/">
    <foaf:Person
rdf:about="http://www.w3.org/People/EM/contact#me">
        <foaf:name>Eric Miller</foaf:name>
        <foaf:made>
            <rdf:Description
rdf:about="http://www.anobii.com/books/Il_nome_della_rosa">
                <dc:title>Il nome della rosa</dc:title>
                <rdf:type
rdf:resource="http://purl.or/ontology/bibo/Book"/>
            </rdf:Description>
        </foaf:made>
    </foaf:Person>
</rdf:RDF
```