



Tecweb - Riassunto del corso: concetti principali da sapere

Tecnologie web (Università di Bologna)



Scansiona per aprire su Studocu

INTRODUZIONE E BREVE STORIA DEL WORLD WIDE WEB

Che cos'è il WWW?

Il web e internet sono molto collegati fra loro ma sono a due livelli diversi dell'architettura.

Internet → insieme dei calcolatori che sono interconnessi tra loro. Una parte di internet è una parte della rete internet.

Il web è qualcosa che viene costruito sull'architettura internet sfruttando i protocolli http/https. Quindi abbiamo un sistema distribuito che ha bisogno di un protocollo che ci mette in collegamento fra di loro. Il web per definizione è un ipertesto (insieme di documenti di testo collegati tra loro attraverso link).

Quindi:

il World Wide Web è un sistema ipertestuale di documenti multimediali, distribuito e scalato su internet, con contenuti testuali e non (sempre più dati) pensati per essere consumati da esseri umani (utente) che da applicazioni (robot, intelligenza artificiale...). Una knowledge base su cui costruire servizi sofisticati per gli utenti finale.

Architettura client server

Usa un architettura client server ovvero abbiamo da un lato un client che chiede delle risorse a un server che le eroga da qualche altra parte. Abbiamo un modello a sportello in cui qualcuno fa una richiesta (con il protocollo http), che può essere passata anche a qualcun altro che la manipola per produrre contenuti, poi questi contenuti tornano direttamente al client. Il client può essere di diversi tipi:

- browser: client web in grado di visualizzare documenti e interagire con l'utente durante la navigazione
- Interfacce diverse: ad esempio screen reader o interfacce vocali
- Applicazioni (quindi niente interfaccia utente): che fanno richiesta ai server per poter caricare questi altri contenuti.

Il web sta diventando uno spazio di pubblicazione di contenuti che, una volta identificati in modo univoco tramite URL, possono essere direttamente consumati da applicazioni diverse.

Protocollo e standard fondamentali

Alla base del WWW ci sono tre protocolli/linguaggi:

- URI → standard per identificare in modo generale risorse di rete e per poterle specificare all'interno di documenti ipertestuali.
- HTTP → protocollo di comunicazione stateless e client server per l'accesso a risorse ipertestuali
- HTML → linguaggio per la marcatura di documenti ipertestuali basato su SGML (e XML) che permette di descrivere la struttura di un documento e le sue componenti, inclusi oggetti multimediali e link, e di visualizzare contenuti sul browser.

La nascita del World Wide Web

Il WWW nasce nel 1989 su proposta di un gruppo di fisici del CERN, che iniziarono ad usarla come piattaforma per lo scambio di documenti. L'obiettivo era di costruire una piattaforma per la diffusione e la

ricerca di articoli e informazioni tra ricercatori. Tim Barners Lee e gli altri identificarono tre pilastri su cui fondare la loro architettura:

- Internet → rete di calcolatori collegati fra loro
- Ipertesti → documenti collegati fra loro
- SGML → metalinguaggio per descrivere formati di interscambio di dati e documenti. Derivato dalla proposta GML di IBM

Il WWW su larga scala e la guerra dei browser

Partiamo dal 1989 dove il focus era sui contenuti, quindi si voleva creare un sistema dove si potessero leggere articoli scientifici. Abbiamo un linguaggio pensato per dei contenuti e da qui HTML. Però poi avevo bisogno di uno strumento per guardare questi contenuti, il primo che si è affermato è stato Mosaic e poi Netscape, e a questo punto è intervenuto Microsoft. Hanno introdotto i browser Explorer. In questo periodo c'erano quindi due browser che permettevano di andare direttamente sul web e quello che hanno fatto è stato di proporre di volta in volta dei miglioramenti a HTML (ovvero caratteristiche solo di presentazione che rendevano più accattivanti i contenuti che venivano visti sul browser). Questo periodo è stato chiamato come "la guerra dei browser". L'aspetto fondamentale è che si è partiti da un linguaggio per marcare i contenuti fino all'introduzione di tanti piccoli miglioramenti a livello di presentazione.

Con Explorer 3.0 e poi 4.0, Microsoft vince definitivamente e dominerà la scena per i successivi cinque anni. Sarà anche condannata per questa operazione commerciale e abuso di posizione dominante. Da Netscape nascerà Firefox e da lì la comunità Mozilla.

World Wide Web Consortium

Durante la guerra dei browser si era iniziato a guardare il web come uno spazio per le guerre commerciali. Per questo Tim Berners Lee crea il W3C (World Wide Web Consortium) ovvero un consorzio (insieme di aziende, enti di ricerca, università, istituzioni) che decidono quali sono gli standard del web e come devono essere fatti per garantire il più possibile l'accesso al web da parte di tutti e la libertà di espressione. In questo consorzio intervengono attori di tipo diverso e cercano di costruire un tavolo comune di discussione. Questo è un punto critico perché sul web molto deriva dallo strapotere di alcuni attori, in modo da fare gli interessi di una platea più alta possibile di fruitori di questi contenuti. Il W3C diventerà uno degli attori principali nello sviluppo dell'XML (linguaggio da cui sono derivate alcune versioni di HTML) ma anche di CSS, WAI (standard accessibilità)... Gli standard non derivano dalle esigenze di un'azienda ma in una situazione di consenso e condivisione.

Standard W3C e consenso

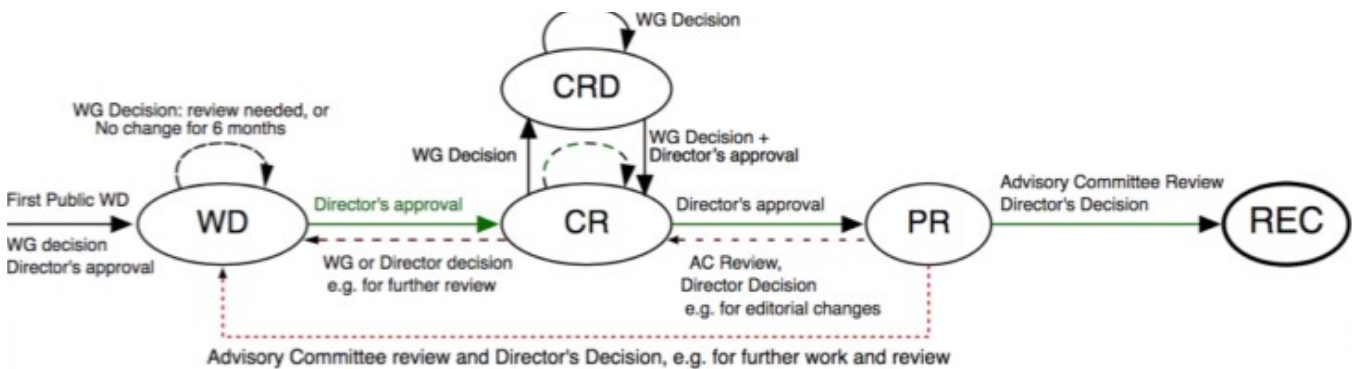
Il W3C promuove processi rigorosi e democratici di standardizzazione che puntano ad un largo consenso da parte della comunità. Partendo dall'interesse della comunità su un dato argomento, il W3C forma gruppi di lavoro di diverso tipo:

- Community Group e Interest Group → per lo scambio di idee e proposte
- Working Group → produrre documentazione e standard.

Questi gruppi lavorano alla redazione di documenti di standard, attraverso fasi ben definite, fino alla loro approvazione con Candidate Recommendation.

Standardizzazione W3C

Viene fatto poi un processo iterativo fin quando non si arriva alla Candidate Recommendation che è una descrizione dello standard che è pronta per essere approvata. La Candidate Recommendation deve essere pubblica per un certo periodo di tempo e si lascia spazio ai commenti. Il working group recepisce i commenti che vengono poi inglobati, viene creata una nuova Candidate Recommendation che viene di nuovo resa pubblica fino a quando non si riesce a convergere.



Il processo è un po' macchinoso. Processo altamente democratico e rigoroso.

Le specifiche per essere approvate devono essere supportate da almeno due implementazioni software (almeno due software che dimostrano di supportare quel requisito)

W3C e HTML

Un processo così rigoroso va di pari passo con lo sviluppo di standard rigorosi. La prima versione di HTML è la 4, ovvero quella subito dopo la prima guerra dei browser. Nel 2001 viene proposto XHTML 1.0 che è la versione di HTML che usa le regole di XML (regole gerarchiche rigorose) che usa regole di buona forma e validazione senza introdurre nuovi elementi. Nel 2002 nasce XHTML 2.0 che introduce nuovi elementi (ad esempio SECTION per gestire correttamente le gerarchie, che mi permette di creare una sezione che contiene una sottosezione) ed è progettato da esperti di linguaggi di marcatura ma senza il supporto di produttori di browser che invece spingono per maggiore interattività e programmabilità (introduzione di API che mi permettono di accedere il più possibile al browser). A questo punto il W3C perde il ruolo di centralità nello sviluppo di HTML e il suo ruolo cambia, così come cambia il ruolo stesso di HTML.

WHATWG e HTML 5

Ci sono due forze contrastanti:

- Il W3C che spinge per specifiche rigorose e basate su XML
- I produttori di browser (che creano un nuovo gruppo chiamato Web Hypertext Application Technology Working Group) che spingono per lo sviluppo di nuove versioni di HTML, con enfasi su aspetti di interattività e programmazione.

Il WHATWG è stato fondato da Apple, Mozilla Foundation e Opera Group. Oggi è guidato da Apple, Google, Mozilla Foundation e Microsoft.

HTML living standard

HTML deve diventare un linguaggio fortemente interattivo con all'interno componenti legati all'interattività e alla programmazione. Da qui nasce il nuovo HTML, che viene definito living standard (standard in cui i produttori di browser aggiungono di volta in volta funzionalità che vanno poi nel linguaggio, ma appena le si hanno a disposizione vengono pubblicate). Non è più un processo rigoroso, di standardizzazione, di approvazione e di consenso ma uno standard in cui i produttori di browser possono iniettare tutte le loro caratteristiche.

Due versioni di HTML ?

HTML 5 è fatto in modo da poter visualizzare qualsiasi file anche se dimentichiamo tag di apertura/chiusura, se annidiamo gli elementi in modo sbagliato o dimentichiamo degli attributi. Si è passati da un'idea di creare i documenti fatti bene così possiamo processarli e manipolarli a creare un ambiente di visualizzazione e interazione che ha un forte motore client side in grado di macinare tutto ciò che arriva, rappresentarlo all'interno del browser e permettere all'utente di interagire.

Nel 2007 il W3C deve ammettere che le modifiche avevano un impatto innegabile e riapri il working group con tutti i membri del WHAT per creare una nuova versione di HTML

Nel 2011 nel working group convivono le due anime ma con scarsi risultati i gruppi si dividono. Lo sviluppo va avanti in parallelo: il W3C continua a standardizzare snapshot di HTML living standard e a dargli un'approvazione formale. Nel 2019 W3C e WHATCG raggiungono un accordo, esiste una versione unica con una componente core di HTML legata al markup e una serie satellite API che descrivono una serie di operazioni che sono possibili all'interno della pagina.

Cos'è il Web 2.0?

Non è diverso dal web normale ovvero che gli standard sono rimasti gli stessi.

Non c'è una nuova versione HTML o HTTP.

Riguarda più che altro il modo con cui questi standard sono utilizzati e soprattutto il fatto che ci sia una forte dinamicità client side. Non un cambio di tecnologie ma un uso diverso di quelle esistenti. Vi è l'affermazione di strumenti di partecipazione attiva alla costruzione di contenuti per il web (blog, wiki, podcast, RSS, folksonomie...). Il web diventa una piattaforma fortemente orientata al mashup di dati da sorgenti diverse, su cui pubblicare contenuti con facilità ed elaborare i dati. Le applicazioni web diventano molto più veloci, complesse ed interattive. Gli utenti partecipano in modo attivo nella creazione di contenuti.

Web 2.0, REST e AJAX

Sviluppo di paradigmi di costruzione applicazioni per maggiore interattività con il client. Creare applicazioni che caricano dei dati e che manipolano client side questi dati tramite librerie javascript.

Alcuni fattori sono stati determinanti in questo cambiamento:

- Enorme diffusione dei dispositivi mobili

- Avanzamento tecnologico di questi dispositivi che ormai hanno elevate capacità di memoria ed elaborazione
- Accesso a Internet diffuso e (ultra) veloce

Per quanto riguarda le tecnologie Web, due aspetti ortogonali:

- + Affermazione del paradigma REST per sfruttare le caratteristiche di HTTP e creare applicazioni solide e scalabili
- + Sviluppo dei linguaggi di programmazione client-side e di AJAX, che permette richieste asincrone al server e pre-caricamento di dati

Rich Client e API

Le pagine web diventano la base per applicazioni che aggregano contenuti, anche da sorgenti diverse e in modo sofisticato (Rich Client, che sono delle applicazioni web che sono fortemente orientate a leggere dati che si trovano su un backend e manipolare questi dati direttamente sul client e reagire ad eventi che avvengono direttamente sul client e modificare l'interfaccia interagendo con l'utente ed eventualmente modificando dei dati che si trovano su backend). Sono numerosi i servizi che producono solo dati; è il client ad occuparsi della loro elaborazione. Diventa sempre più importante fornire i propri dati in modo chiaro attraverso un API (application programming interface).

Ajax è un modo di usare javascript per fare delle richieste asincrone e caricare direttamente dei dati.

API web

Così come per i linguaggi di programmazione, un'API Web definisce le modalità per interagire con un'applicazione. In pratica, elenca le possibili richieste e risposte che l'applicazione è in grado di gestire. Le applicazioni, sia server-side che client side, possono quindi raccogliere i dati per poi elaborarli invocando una o più API:

- applicazioni diverse costruite sugli stessi dati
- applicazioni di mash-up che combinano dati in formati diversi e da sorgenti diverse

Dal web of content al web of data

È l'ultimo passo in cui si trova ora il web che sta diventando una piattaforma non tanto pensata per la lettura di contenuti pensata per gli esseri umani ma per agenti di software. Creare uno spazio di dati interconnessi fra loro. Questo era possibile anche in passato ma molto più complicato, bisognava estrarre informazioni e fare scraping, cosa ancora necessaria oggi in assenza di API. Il Web diventa quindi una piattaforma di dati intercollegati e non solo di contenuti ipertestuali. Da link semplici si passa a link complessi e di diverso tipo in grado quindi di esprimere relazioni tra oggetti e costruire una base di conoscenza distribuita.

Semantic Web

Il Web semantico è una rete di dati. Il Web semantico riguarda due cose. Si tratta di formati comuni per l'integrazione e la combinazione di dati tratti da diversi fonti, dove sul Web originale si concentrava principalmente sul scambio di documenti. Riguarda anche la lingua per la registrazione come i dati si riferiscono agli oggetti del mondo reale. Che permette a una persona, o una macchina, per iniziare in un

database e poi spostarti un insieme infinito di database che sono collegati non da fili ma essendo più o meno la stessa cosa.

Dati e affermazioni

Json è un modello molto semplice, ma esistono anche modelli più strutturati come RDF. L'idea di RDF è che posso esprimere delle triple basate sul concetto di soggetto-predicato-oggetto. Posso costruire delle reti di informazioni molto complesse.

Linked Open Data

Nasce nel 2007. Linked Data riguarda l'uso del Web per creare link tipati tra risorse appartenenti a domini differenti, in modo da esprimere relazioni e proprietà. Linked Open Data (LOD) è un progetto del W3C che si occupa di estendere il Web tradizionale pubblicando dataset liberi e aperti e mettendo in relazione tra loro dati provenienti da sorgenti diverse. Obiettivo: pubblicare dati sul Web in modo che:

- siano machine-readable
- con un significato esplicitamente definito
- abbiano collegamenti verso altri insiemi di dati (dataset) esterni

WikiData

Permette di costruire una base di dati dove ad ogni entità sono collegate delle relazioni. È una knowledge-base collaborativa, aperta e multilingua. Costituita da item (entità) con un codice identificativo univoco e sui quali sono espresse proprietà e relazioni tramite affermazioni. I dati possono essere modificati direttamente e/o importati da altre fonti, inclusa Wikipedia (così come fatto da DBPedia)

Google Knowledge Graph e Schema.org

Anche Google mette a disposizione un'API per accedere al proprio Knowledge Graph, una base di conoscenza usata dal motore di ricerca e che contiene informazioni estratte da Wikipedia e da altre sorgenti (Freebase, CIA World Factbook). Usa classi e proprietà definite schema.org ed espresse in formato RDF e JSON-LD quindi compatibile con tecnologie Linked Data. Schema.org è un progetto inizialmente fondato da Google, Microsoft e Yahoo e ora supportato da una comunità open source per definire vocabolari utili a descrivere Persone, Luoghi, Eventi, etc.

- Classi e proprietà specializzate e condivise da applicazioni diverse

Conclusioni

Il World Wide Web ha una storia relativamente recente ma ha visto diverse evoluzioni ed involuzioni. Da un sistema di contenuti ipertestuali si è arrivati ad un sistema di dati e applicazioni, con modalità di accesso sempre più variegate. Le tecnologie di base tuttavia sono sostanzialmente le stesse ma è cambiato il modo in cui sono sfruttate, il supporto hardware e di comunicazione, l'uso da parte degli utenti. Conoscere questi cambiamenti e il modo in cui le tecnologie sono state sviluppate e influenzate dal contesto aiuta a comprenderle meglio e metterle in relazione

CODIFICA DEI CARATTERI

Codifica di contenuti e dati web

Prima di visualizzare ed elaborare contenuti e dati Web, si pone il problema di rappresentarli all'interno di un calcolatore e trasferirli da un calcolatore all'altro. In particolare, andiamo a parlare della rappresentazione di testi ovvero la sequenza di caratteri che sono poi gli elementi atomici dell'informazione che andiamo a trattare.

Il calcolatore infatti usa una rappresentazione numerica dell'informazione e in particolare la codifica binaria, basata su due simboli 0 e 1 (bit). Per rappresentare un dato non numerico abbiamo quindi la necessità di trasformarlo in una sequenza di numeri memorizzabili nel calcolatore. Questo processo di digitalizzazione (se parliamo di risorse che non sono propriamente digitali) varia a seconda del dato che vogliamo rappresentare: infatti se parliamo di testi è la traduzione di sequenze di caratteri in sequenze numeriche. Qui parliamo di testi, usati per rappresentare indirizzi delle risorse, contenuto dei messaggi HTTP, pagine HTML, CSS, JS, ecc.

Codifica di testi e caratteri

Innanzitutto bisogna mettersi d'accordo sullo schema per la traduzione: perché da una parte dovrò tradurlo per renderlo comprensibile all'elaboratore ma una volta dall'altra parte dovrò fare la traduzione inversa. Gli schemi di codifica non sono altro che delle mappe di traduzione create per rappresentare alfabeti di lingue diverse, con caratteri e simboli diversi, che andiamo a tradurre per fare questa trasmissione. L'obiettivo chiave è fare in modo che questa associazione tra una sequenza di bit e viceversa sia non ambigua (un'applicazione è capace di leggere e decodificare in modo che altre lo possano fare nello stesso modo senza difficoltà per ottenere lo stesso risultato).

Per codificare un testo si traduce ogni carattere in un valore numerico, rappresentato poi in notazione binaria o esadecimale, e si concatenano i valori numerici ottenuti. Si pone il problema di dover rappresentare molte lingue e alfabeti diversi, alcuni con moltissimi caratteri. Soprattutto, deve essere evidente e non ambiguo il criterio di associazione di un blocco di bit ad un carattere dell'alfabeto e il meccanismo di traduzione. Sono stati proposti diversi standard e set di caratteri per gestire le diversità dei vari alfabeti

ASCII (American Standard Code for Information Interchange)

L'idea è che visto che la rappresentazione veniva fatta su 7 bit più dei caratteri di controllo. Io so che ho 8 bit a disposizione che posso rappresentare 2^8 possibili combinazioni quindi ho un set di caratteri di 256 possibili combinazioni (tra valori 0 e 1). In ASCII è stato deciso di usare un sottoinsieme di 7 bit e uno ulteriore che mi serviva solo per fare dei controlli.

Mi permette di codificare l'alfabeto inglese in maiuscolo e minuscolo e alcuni simboli comuni della punteggiatura.

Standard ANSI (X3.4 - 1968) che definisce valori per 128 caratteri, ovvero 7 bit su 8. Nello standard originale il primo bit non è significativo ed è pensato come bit di parità. ASCII possiede 33 caratteri (0-31 e 127) di controllo, inclusi alcuni ormai non più rilevanti e legati all'uso di telescriventi negli anni '60. Codifica anche spazi e "andata a capo" (CR e LF). Gli altri 95 sono caratteri dell'alfabeto inglese, maiuscole e minuscole, numeri e punteggiatura.

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
010	sp	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
111	p	q	r	s	t	u	v	w	x	y	z	{		}	~	canc

c è codificato in ASCII: 1100011

C è codificato in ASCII: 1000011

1

ISO 8859/1 (ISO Latin 1)

Risolve il problema delle lettere accentate. L'idea è prendere una rappresentazione che non utilizza 7 bit ma ne usa 8 (1 byte per rappresentare tutti i caratteri), e a questo punto con il bit aggiuntivo che in ASCII non veniva considerato ho più possibilità. Quindi sfrutto queste possibilità per codificare le lettere accentate e quindi le lettere degli alfabeti latini. Per questo si chiama ISO Latin 1:

- ISO → è lo standard che interviene non solo nella codifica dei testi ma anche in altri contesti di standardizzazione.

Tutti i primi caratteri tra 0 e 128 sono equivalenti ad ASCII, per questo si dice che è retro compatibile.

Diverse estensioni di ASCII sono state proposte per utilizzare il primo bit e accedere a tutti i 256 caratteri. Nessuna di queste è standard tranne ISO Latin 1. ISO 8859/1 (ISO Latin 1) comprende un certo numero di caratteri degli alfabeti europei come accenti, ecc. ISO Latin 1 è usato automaticamente da HTTP e alcuni sistemi operativi. Ovviamente ISO Latin 1 è compatibile all'indietro con ASCII, di cui è un'estensione per i soli caratteri >127.

Windows – 1252

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
80			¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸
90			¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸	¸
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°
B0	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Nota: righe e colonne della tabella sono indicate in notazione esadecimale.

In verde i caratteri aggiunti a windows-1252 rispetto a ISO-Latin1. Lo stesso abbiamo un'estensione a 8 bit quindi abbiamo 1 byte che mi permette di codificare i caratteri.

UCS e UTF

Parallelamente alla codifica dei caratteri latini, ci sono state analoghe codifiche per caratteri del cirillico, del greco, dell'ebraico. Il discorso si complica per i CJKV (giapponesi, cinesi, coreani e vietnamiti) di origine cinese e per cui non bastano 8 bit. Si è cercato di definire a uno standard unico in grado di coprire tutti gli alfabeti che quindi usa un numero maggiore di byte. Anche qui hanno lavorato diversi gruppi che sono riusciti poi a convergere. Due grandi famiglie di schemi:

- a lunghezza fissa: UCS-2 (2 byte) e UCS-4 (4 byte) → in questi decido che i caratteri occupano un certo numero di byte indipendentemente dalla loro posizione.
- a lunghezza variabile: UTF-8, UTF-16 e UTF-32 → quando arriva una sequenza di bit dico che quella sequenza di caratteri occupa tot byte

UTF – 8

UTF nasce dall'osservazione che i testi nella maggior parte dei casi sono scritti in un unico alfabeto o in alfabeti vicini. E' quindi uno spreco usare 2 o 4 byte per ogni carattere, anche quando sarebbe sufficiente usare 1 byte. UTF (Unicode Transformation Format o UCS Transformation Format) è un sistema a lunghezza variabile che permette di accedere a tutti i caratteri definiti di UCS-4:

- + I codici compresi tra 0 - 127 (ASCII a 7 bit) richiedono 1 byte
- + I codici derivati dall'alfabeto latino e tutti gli script non-ideografici richiedono 2 byte
- + I codici ideografici (orientali) richiedono 3 byte
- + Tutti gli altri 4 byte.

Bit iniziali e di controllo permettono di capire come interpretare la restante sequenza di bit (quanti byte considerare).

Differenze tra UTF-8 e ISO Latin -1

Non confondere le codifiche UTF-8 e ISO Latin-1! Per i caratteri appartenenti ad ASCII, le due codifiche sono identiche. Quindi un documento in inglese non avrà differenze nel testo. Viceversa, un testo in italiano, o francese o tedesco risulta quasi corretto, perché non vengono descritte correttamente solo le decorazioni di lettere latine (ad esempio, accenti, umlaut, vocali scandinave ecc.). In questo caso, UTF-8 utilizza 2 byte per questi caratteri, mentre ISO Latin 1 ne usa uno solo.

Conclusioni

Per poter essere elaborati e trasmessi i contenuti e i dati su Web devono essere codificati come valori numerici, espressi in notazione binaria ed esadecimale. Qui abbiamo visto come codificare sequenze di caratteri. Attenzione alla codifica caratteri nei messaggi HTTP, nei documenti e nelle applicazioni. Una buona strategia è usare la codifica UTF-8 in tutte le fasi del processo

Bit iniziali → per la posizione che occupano in quella sequenza

Bit di controllo → perché posso usarli o per capire come interpretare gli altri oppure per verificare che la sequenza di bit che arriva a destinazione da percorsi diversi sia la stessa (nel caso della trasmissione).

URI: UNIFORM RESOURCE IDENTIFIER

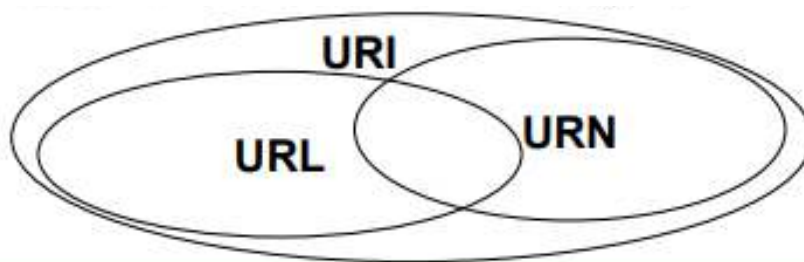
Uniform Resource Identifier

Sono l'elemento chiave che ci permettono di identificare le risorse su web. Sono delle sequenze di caratteri, meccanismi universali, per identificare tutte le risorse che sono disponibili e raggiungibili su web. Nasce dalla necessità di rappresentare queste risorse che possono essere di diverso tipo e che possono essere usate da diversi protocolli. Questo per arrivare ad una sintassi universale che sia facile da memorizzare. L'introduzione dell'URI è precedente al suo utilizzo sul web. Quindi, riepilogando, mi servono per la rappresentazione di risorse sul web che non necessariamente sono file fisici. Sono stati verosimilmente il fattore determinante per il successo del WWW.

URI, URL e URN

Abbiamo la necessità di tradurre un identificativo indipendente dalla rappresentazione di questa risorsa in una fisica che possiamo trovare da qualche parte e a cui possiamo accedere (ad esempio un file sul file system, un dato che si trova nel database, un'immagine che si trova sul server, file di testo, file CSS...). Gli URI si dividono in due categorie:

- + URL (uniform resource locator) → sono i cosiddetti locator ovvero indirizzi che hanno tutte le informazioni necessarie per recuperare quel dato, quel tipo di info. Mi specifica tutto ciò di cui ho bisogno per arrivare a quella risorsa. Una sintassi che contiene informazioni immediatamente utilizzabili per accedere alla risorsa (ad esempio, il suo indirizzo di rete)
- + URN (uniform resource names) → qualcosa di più generico e che mi permette di dare un nome. La N sta per names infatti. Mi permette di dare un nome alla risorsa che è permanente e che non può essere ripudiato, ovvero mi definisce un'etichetta/sequenza di caratteri che rappresentano quella risorsa e se questa ha anche una rappresentazione diversa (esempio è stata salvata su un altro file/database, ha cambiato la sua posizione interna) cambia l'URL da cui vado ad estrarla ma non cambia il suo nome. Necessario quindi un meccanismo di traduzione verso gli URL.



Risorsa vs File

La risorsa è intesa come entità della quale io voglio leggere l'informazione, scrivere l'informazione. È un concetto astratto. Una risorsa non è necessariamente un file presente su un filesystem ma potrebbe essere:

- in un database, e l'URI essere la chiave di ricerca
- il risultato dell'elaborazione di un'applicazione, e l'URI essere i parametri di elaborazione.

– una risorsa non elettronica (un libro, una persona, un pezzo di produzione industriale) e l'URI essere il suo nome, ad esempio nel caso di account Twitter, Instagram, etc.

– un concetto astratto

Per questo si usa il termine Risorsa, invece che File, e si fornisce una sintassi indipendente dal sistema effettivo di memorizzazione.

Il file è inteso come l'oggetto che si trova sul calcolatore/database.

Organizzazione degli URI

Tutti gli URI condividono la stessa struttura. Gli URI sono progettati per fornire spazi di nomi organizzati gerarchicamente:

URI = schema : [// authority] path [? query] [# fragment]

Stiamo costruendo quello che si chiama URL cioè una sequenza di caratteri che può essere interpretata per andare a localizzare quella risorsa. Questa è divisa in diverse parti attraverso alcuni simboli significativi e può essere trasferita in formato testuale.

Degli esempi sono:

- <http://www.ietf.org/rfc/rfc2396.txt>
- <ftp://ftp.is.co.za/rfc/rfc1808.txt>
- <https://purl.oclc.org/OCLC/PURL/FAQ>
- <file:///Documenti/corsi/tw/slides/I1.html>
- [mailto: \[angelo.diiorio@unibo.it\]\(mailto:angelo.diiorio@unibo.it\)](mailto:angelo.diiorio@unibo.it)
- [data:image/
png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAyAAAC
NbybIAAAAEIEQVQI12P4//8/
w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAyAAACNbybIAAAAEIEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==)

Componenti URI

Il primo elemento è lo schema che è identificato da una stringa presso IANA usata come prefisso.

Dopo di lui abbiamo la authority che è uno spazio gerarchico che mi permette di definire, nel caso di http per esempio, qual è l'host su cui andare a recuperare la risorsa. Se sto usando un dominio avrò bisogno di un servizio di traduzione di un dominio che mi dirà che a un certo indirizzo corrisponde un certo computer. Un'authority, o meglio un nome di dominio, che usiamo molto spesso è localhost che rappresenta la macchina su cui sta girando il browser. È suddivisa in:

authority = [userinfo @] host [: port]

La parte userinfo non deve essere presente se lo schema non prevede identificazione personale. La parte host è o un nome di dominio o un indirizzo IP. La port può essere omessa se ci si riferisce ad una well-known port (per http è la porta 80).

Poi la parte successiva è il path che mi permette di identificare una risorsa sul server. E' la parte identificativa della risorsa all'interno dello spazio di nomi identificato dallo schema e (se esistente) dalla authority. La parte path è divisa in blocchi separati da slash "/", ciascuno dei quali è un componente del path organizzato in gerarchia. In questo caso diventano significativi gli pseudo componenti "." e "..".

Poi abbiamo la parte query che mi permette di aggiungere quei parametri che mi servono per identificare e specificare meglio le informazioni utili per recuperare quella risorsa. Ovvero il formato sottostante che useremo per comporre tutte le richieste che facciamo per raccogliere i nostri dati. Individua un'ulteriore specificazione della risorsa all'interno dello spazio di nomi identificato dallo schema e dall'URI precedente. Di solito questi sono parametri passati all'URI (un processo) per specificare un risultato dinamico (es. l'output di una query su un motore di ricerca). Tipicamente ha la forma nome1=valore1&nome2=valore+in+molte+parole

Infine abbiamo i fragment che mi permettono di identificare una risorsa secondaria all'interno della risorsa principale da cui sono partito. Un' esempio sono le sezioni all'interno di una pagina. E' tutta la parte che sta dopo al carattere di hash "#". Usata ad esempio per identificare sezioni all'interno di una pagina HTML

Route

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

Il problema principale che ho a questo punto è di tradurre un path (identificativo risorsa che sto cercando) con la risorsa vera che si trova sul server web. Siamo in una situazione client-server: il client fa una richiesta scrivendo un URI poi il server riceve la richiesta e capisce qual è la risorsa che serve, cercando tra tutte le risorse locali. Questo meccanismo avviene con due specifiche strategie, dette di routing:

- Managed Route → usato per risorse ad esempio express. Il server associa ogni URI ad una risorsa o attraverso il file system locale (risorse statiche) oppure generate attraverso una computazione (risorse dinamiche).
- File – System Route → usato per risorse statiche (come immagini, CSS...). Il server associa la radice della parte path ad una directory del file system locale e ogni filename valido all'interno di quella directory genera un URI corretto e funzionante.

URI reference

Fino ad ora gli URI iniziavano con uno schema, ma non sempre è così. Un URI assoluto contiene tutte le parti predefinite dal suo schema, esplicitamente precisate. Un URI gerarchico può però anche essere relativo, (detto tecnicamente un URI reference) ed in questo caso riportare solo una parte dell'URI assoluto corrispondente "tagliando progressivamente parti da sinistra". Un URI reference fa sempre riferimento ad un URI di base (ad esempio, l'URI assoluto del documento ospitante l'URI reference) rispetto al quale fornisce porzioni differenti.

Risolvere un URI relativo

Risolvere un URI relativo significa identificare l'URI assoluto cercato sulla base dell'URI

	Dato il base URI 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/doc2.html
Se inizia con "/", allora è un path assoluto all'interno della stessa autorità del documento di base, e gli va applicata la stessa parte autorità.	/dir3/doc3.html si risolve come http://www.site.com/dir3/doc3.html

Altrimenti:

	Dato il base URI 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.	doc4.html si risolve come http://www.site.com/dir1/doc4.html dir5/doc5.html si risolve come http://www.site.com/dir1/dir5/doc5.html

Si procede infine a semplificazioni:

"./" (stesso livello di gerarchia): 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

Esercizio

Risolvere i seguenti URI relativi rispetto al seguente URI base:

<http://www.sito.com/data/2021/index.html>

- a) `images/`
- b) `/images/`
- c) `../2020/images/img2.jpg`
- d) `../../images/img1.png`
- e) `./videos/`
- f) `../videos/`
- g) `/videos/images/`
- h) `../index.html#y2021`

- a) <http://www.sito.com/data/2021/images/>
- b) www.sito.com/images/
- c) www.sito.com/data/2020/images/img2.jpg
- d) www.sito.com/images/img1.png
- e) www.sito.com/data/2021/videos/

Alcuni schemi usati negli URI

http è lo schema più usato negli URI. Https prevede la cifratura dei messaggi, in entrambi i versi. Per il resto è identico ad http.

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

Host → è l'indirizzo TCP-IP o DNS della macchina su cui si trova la risorsa

Post → è la porta a cui il server è in ascolto per le connessioni. Per default, la porta è 80 per HTTP e 443 per HTTPS.

Lo schema file (RFC 8089)

Esistono diversi tipi di schema e di solito utilizziamo maggiormente http e https. In alcuni casi, però, uno degli schemi che adottiamo in modo implicito è lo schema file che ci permette, ad esempio al browser, di leggere i dati e i contenuti che si trovano direttamente sul computer su cui sta girando. Se l'url inizia con "file://" significa che non è un percorso che richiede l'intervento di un server web ma un percorso di file direttamente sul file system.

`file://<host>/<path> [#fragment]`

La parte host può essere eliminata, assumendo che sia localhost, che porta alla sintassi più frequente:

`file://localhost/path`
`file:///c:/Users/mario/Pictures/img1.jpg`

MS Windows accetta anche "\", che però non è nello standard approvato!

Lo schema data (RFC 2397)

Uno schema non gerarchico, che non fa riferimento ad una risorsa, ma CONTIENE la risorsa: tutti i dati della risorsa sono inseriti nell'URI vero e proprio. Usato per immagini online su cui non si vuole attivare una connessione HTTP separata. La sintassi è:

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

media type è un media type MIME registrato presso IANA. Base64 è un parametro opzionale per indicare che il dato è codificato in base 64. Data sono i dati codificati con il media-type appena indicato

HYPERTEXT TRANSFER PROTOCOL

E' il protocollo su cui si basa il web che è associato anche ad uno schema all'interno degli URL. Ha tre caratteristiche fondamentali:

- Client – server → il client attiva la connessione e richiede dei servizi. Il server accetta la connessione, nel caso identifica il richiedente, e risponde alla richiesta. Alla fine chiude la connessione.
- Generico → è pensato per funzionare con qualunque tipo di dato, non solo con html. Può funzionare per documenti HTML come per binari, eseguibili, oggetti distribuiti o altre strutture dati più o meno complicate
- Stateless → nella sua versione di base dice che il server non è tenuto a mantenere nessuna informazione su diverse connessione, quindi ognuna è un connessione a sé stante per cui non è necessario mantenere le informazioni sulle richieste precedenti. Il server non deve fare nessuna assunzione sul contesto in cui quella richiesta viene fatta. Il client è tenuto a ricreare da zero il contesto necessario al server per rispondere. Sono stati introdotti diversi meccanismi per mantenere informazioni diverse, tra cui quello dei cookie (informazioni aggiuntive che mi servono per tenere traccia del collegamento fra client e server che sono state aggiunte alla struttura base di http).

Concetto chiave: risorse http

HTTP permette lo scambio di risorse identificate da URI. Separa nettamente le risorse dalla loro rappresentazione e fornisce meccanismi di negoziazione del formato di dati, cioè la possibilità di richiedere e ricevere (la rappresentazione di) una stessa risorsa in formati diversi. HTTP implementa inoltre sofisticate politiche di caching che permettono di memorizzare copie delle risorse sui server (proxy, gateway, etc.) coinvolti nella trasmissione e controllare in modo accurata la validità di queste copie. Un uso corretto di questi meccanismi migliora notevolmente le performance delle applicazioni.

Connessione http

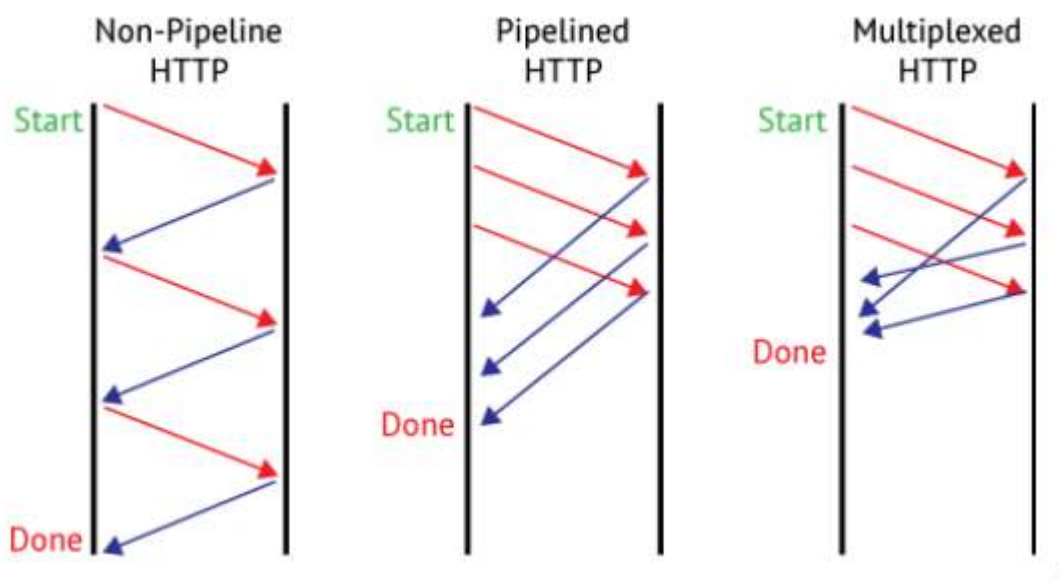
Una connessione HTTP è composta da una serie di richieste ed una serie corrispondente di risposte. Le connessioni sono persistenti con:

- ❖ Pipelining → trasmissione di più richieste senza attendere l'arrivo della risposta alle richieste precedenti. Ma le risposte sono restituite nello stesso ordine delle richieste

- ❖ Multiplexing → nella stessa connessione è possibile avere richieste e risposte multiple, restituite anche in ordine diverso rispetto alle richieste e “ricostruite” nel client

HTTP/2 ha introdotto molte novità per migliorare le performance:

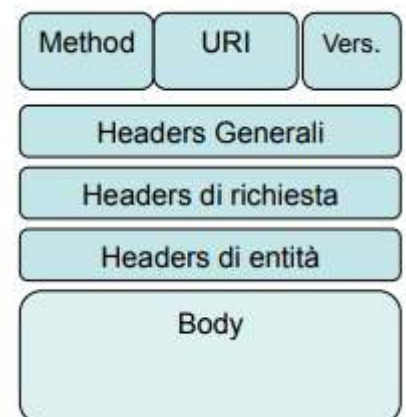
- ✓ Operazioni push → il server anticipa alcune risposte prima di ricevere alcune richieste del client. Questo è permesso anche dal fatto che le pagine web sono formate da risorse collegate fra di loro.
- ✓ Compressione degli header → le informazioni piuttosto che viaggiare in chiaro sono compresse con degli algoritmi di compressione simili a quelli usati per zip, in realtà sono algoritmi specializzati. Ad esempio se abbiamo una sequenza di testo che occupa un certo numero di byte, prima di spedirla viene compressa e quindi spedisco meno informazioni. Altra cosa che fa http 2 è nel caso ho più richieste all'interno della stessa connessione, che hanno degli header (informazioni aggiuntive che ci sono nella richiesta http) in comune, se questi header non sono stati modificati non vengono rispediti con l'obiettivo di ridurre il carico di dati che viaggia dal client al server.



La richiesta HTTP

È una sequenza di caratteri che sono codificati in qualche modo e all'interno della quale ci sono informazioni importanti, fondamentali:

- Method → è l'informazione principale che ci interesserà. Mi dice qual è l'operazione che devo eseguire su questa risorsa.
- URI → è il primo passaggio e identifica qual è la risorsa che sto caricando
- Header → sono le informazioni aggiuntive per soddisfare quella richiesta
- Body → sono i dati che eventualmente, non necessariamente, passo per fare questa richiesta.
- Version → versione http



Un esempio:

```
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
```

Esempi di GET e POST

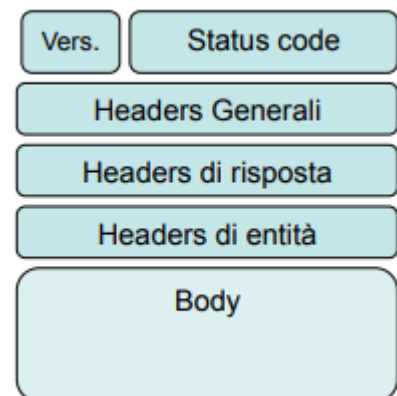
GET è il metodo più frequente, ed è quello che viene attivato facendo click su un link ipertestuale di un documento HTML, o specificando un URL nell'apposito campo di un browser.

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

La risposta HTTP

Nella risposta abbiamo:

- Status code → è la prima riga. Mi dice l'esito della risposta
- Header → mi aggiungono delle informazioni
- Version → versione http
- Body → è il corpo della risposta, ciò che il server mi ha risposto.



```
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>
```

Un esempio:

Status code

Sono dei codici di tre cifre (la prima indica la classe di risposta e le altre due la risposta specifica) che mi dicono l'esito della richiesta. Quando mi arriva la risposta, al suo interno bisogna dire se la richiesta è andata a buon fine, se era sbagliata o se c'erano problemi sul server. Per dirlo http fornisce un meccanismo standardizzato: ha una serie di codici riconosciuti e definiti nel protocollo che mi dicono qual è l'errore o lo stato di successo di quella richiesta. Questi codici vengono messi all'inizio della risposta ma li troviamo anche nel browser. Questi sono:

- ✓ 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 da parte 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

Un esempio:

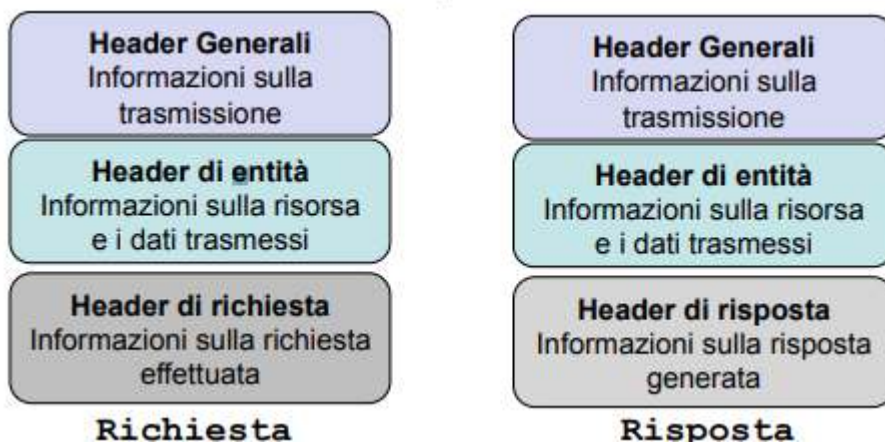
100 Continue (se il client non ha ancora mandato il body)
200 Ok (GET con successo)
201 Created (PUT con successo)
301 Moved permanently (URL non valida, il server conosce la nuova posizione)
400 Bad request (errore sintattico nella richiesta)
401 Unauthorized (manca l'autorizzazione)
403 Forbidden (richiesta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

Utilità dello status code HTTP

L'uso corretto degli status code aiuta a costruire API chiare e più semplici da usare. Il client non ha necessità di leggere il body della risposta ma già dallo status code può capire se la richiesta è andata a buon fine. Permette a tutte le entità coinvolte nella comunicazione di capire meglio cosa succede, e sfruttare i meccanismi di caching e redirezione di http. Migliora uniformità e interoperabilità

Gli header

Gli header sono righe di testo (RFC822) che specificano informazioni aggiuntive. Sono presenti sia nelle richieste che nelle risposte e ne descrivono diversi aspetti.



Header generali

Gli 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. Alcuni esempi:

- Date → data ed ora della trasmissione
- Transfer-Encoding → il tipo di formato di codifica usato per la trasmissione
- Cache-Control → il tipo di meccanismo di caching richiesto o suggerito per la risorsa

- Connection → il tipo di connessione da usare (tenere attiva, chiudere dopo la risposta, ecc.)

Header dell'entità

Gli header dell'entità danno informazioni sul body del messaggio, o, se non vi è body, sulla risorsa specificata. Alcuni esempi:

- Content-Type → il tipo MIME dell'entità nel body. Questo header è obbligatorio in ogni messaggio che abbia un body.
- Content-Length → la lunghezza in byte del body. Obbligatorio, soprattutto se la connessione è persistente.
- Content-Encoding, Content-Language, Content-Location, Content-MD5, Content-Range: la codifica, il linguaggio, l'URL della risorsa specifica, il valore di digest MD5 e il range richiesto della risorsa.

Header della richiesta

Gli header della richiesta sono posti dal client per specificare informazioni sulla richiesta e su se stesso al server. Esempi:

- User-Agent → una stringa che descrive il client che origina la richiesta; tipo, versione e sistema operativo del client, tipicamente.
- Referer → l'URL della pagina mostrata all'utente mentre richiede il nuovo URL
- Host → il nome di dominio e la porta a cui viene fatta la connessione

Altri header della richiesta sono usati per gestire la cache e i meccanismi di autenticazione

Header della risposta

Gli header della risposta sono posti dal server per specificare informazioni sulla risposta e su se stesso al client. Alcuni esempi:

- Server → una stringa che descrive il server: tipo, sistema operativo e versione.
- WWW-Authenticate → challenge utilizzata per i meccanismi di autenticazione

Importanza del content – type

È quell'informazione che ci dice, all'interno di una risposta, qual è il tipo del dato che riceviamo e c'è un altro header content length che ci dice quanti sono i byte che devo andare a processare. Questa è un'informazione importante perchè il client lo utilizzerà per capire che formato di dati che riceve. Queste informazioni permettono al client di processare correttamente la (rappresentazione di una) risorsa. Se viene fornita un'entità in risposta, infatti, gli header Content-type e Content-length sono obbligatori. E' solo grazie al content type che lo user agent sa come visualizzare l'oggetto ricevuto. E' solo grazie al content length che lo user agent sa che ha ricevuto tutto l'oggetto richiesto.

I metodi di HTTP

I metodi ci descrivono l'operazione che stiamo facendo su una risorsa. La progettazione di API passa prima di tutto nella progettazione di metodi. I metodi ci dicono le operazioni che compiamo ad ogni richiesta. I metodi indicano l'azione che il client richiede al server sulla risorsa, o meglio sulla rappresentazione della

risorsa o, ancora meglio, sulla copia della rappresentazione della risorsa. Chiamati anche verbi HTTP per evidenziare l'idea che esprimono azioni da eseguire sulle risorse, identificate a loro volta da nomi (URI). Un uso corretto dei metodi HTTP aiuta a creare applicazioni interoperabili e in grado di sfruttare al meglio i meccanismi di caching di http. I metodi principali:

- + GET
- + HEAD
- + POST
- + PUT
- + DELETE
- + OPTION
- + PATCH

Guardiamo esempi dei più usati, GET e POST, e poi torneremo sui metodi a fine lezione visto la loro importanza per REST

Due proprietà importanti

- Sicurezza → l'idea è che un metodo è sicuro se non genera cambiamenti allo stato interno del server a parte i log. In genere sono metodi lettura piuttosto che metodi di scrittura. Il vantaggio è che può essere eseguito da un intermediario senza che abbia effetti negativi. Niente a che vedere con password, hacking e privacy.
- Idempotenza → vuol dire che un metodo ha lo stesso effetto sul serve indipendentemente che stia facendo una sola richiesta o richiesta multiple. Può essere ripetuto più volte perché mi aspetto che il cambiamento sullo stato del server sia sempre lo stesso e questo mi permette di fare operazioni di controllo e verifica di ciò che è successo.

Le due caratteristiche sono collegate tra loro ma non sono la stessa cosa. Infatti, esistono metodi che sono sicuri ed idempotenti, metodi che sono sicuri ma non idempotenti o viceversa.

Il metodo GET

È il metodo che mi permette di leggere e richiedere le informazioni di una risorsa ovvero la rappresentazione di una risorsa. È sicuro ed idempotente perché metodo di solo lettura quindi vado a richiedere le informazioni non sto cambiando lo stato e dall'altra parte ogni volta che richiedo le informazioni su un qualcosa, i risultati sono gli stessi. questo per quanto riguarda la comunicazione in http. Questo 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.

Il metodo POST

È un metodo che cambia le risorse, lo stato di una risorsa. Ha nel body le informazioni che mi servono per cambiare lo stato di questa risorsa. Non è sicuro e nemmeno idempotente perché passo informazioni diverse che cambiano questo stato. Serve per trasmettere informazioni dal client al server relative alla risorsa identificata nell'URI.

Il metodo POST serve per trasmettere informazioni dal client al server relative alla risorsa identificata nell'URI. Può essere usato anche per creare nuove risorse. Viene usato per esempio per spedire i dati di un form HTML ad un'applicazione server-side.

Il metodo HEAD

Ha lo stesso comportamento del metodo GET con la differenza che una risposta al metodo HEAD è la stessa risposta al GET corrispondente ma senza il body. È un metodo usato solo per fare dei controlli. Trasferisce solo gli header della risposta che mi velocizza la costruzione dell'applicazione.

Il metodo HEAD è simile al metodo GET, ma il server deve rispondere soltanto con gli header relativi, senza il corpo. Viene usato per verificare validità, accessibilità e coerenza in cache di un URI. È sicuro e idempotente.

Il metodo PUT

Mi serve per trasmettere la rappresentazione di una risorsa che sovrascrive una risorsa esistente o crea quella risorsa sul server. È un metodo idempotente perché trasferisco le informazioni che mettono nel body rappresentano quella risorsa che viene creata o sostituita. Non è sicuro.

In generale, l'argomento del metodo PUT è la risorsa che ci si aspetta di ottenere facendo un GET in seguito con lo stesso nome. Non offre garanzie di controllo degli accessi o locking.

Il metodo DELETE

È l'operazione che mi serve per cancellare una risorsa. Non è sicuro perché cancella i dati ma è idempotente perché se ho una risorsa, la prima volta che la cancello quando chiamo delete successivamente quella risorsa non c'è più e il risultato sul server risulta lo stesso.

Dopo l'esecuzione di un DELETE, la risorsa non esiste più e ogni successiva richiesta di GET risulterà in un errore 404 Not Found. Il metodo DELETE su una risorsa già non esistente è lecito e non genera un errore. È idempotente e non sicuro.

Il metodo PATCH

È un metodo che spesso viene usato per questioni di performance, molto vicino a POST. L'idea è che mi serve per aggiornare parzialmente una risorsa identificata dall'URL. Molto spesso nel body trovo scritte le operazioni di modifica. Non è né sicuro né idempotente.

Usato per indicare quindi modifiche incrementali e non sovrascrivere risorse al server, come ad esempio nel caso di PUT. Indica quindi le modifiche da effettuare su una risorsa esistente.

Il metodo OPTION

Mi permette di fare delle richieste e, come HEAD, di ricevere solo gli header di risposta senza il body. Mi serve quando vado a fare delle operazioni di controllo. Rispetto ad HEAD, che riceve informazioni sulla risorsa, questo riceve informazioni riguardo al server, all'host o altre informazioni di contesto dove questa risorsa si trova. Per questo viene spesso usata per verificare i permessi. L'idea è di sfruttare il canale http per

farsi restituire dal dominio di origine l'elenco di tutti gli altri domini che sono ammessi nelle comunicazioni successive. Queste informazioni le faccio viaggiare negli header (sono il meccanismo che utilizzo per aggiungere informazioni all'interno della comunicazione http).

OPTIONS viene usato per verificare opzioni, requisiti e servizi di un server, senza implicare che seguirà poi una altra richiesta. Usato per il problema del cross-site scripting.

Riassumendo:

HTTP Methods	IDEMPOTENT	SAFE METHOD
GET	YES	YES
HEAD	YES	YES
OPTION	YES	YES
DELETE	YES	NO
PUT	YES	NO
PATCH	NO	NO
POST	NO	NO

API REST E OPENAPI

API Web

Un API è un'interfaccia che un applicazione offre per utilizzare i propri servizi ed pensata per essere usata da un altro blocco della stessa applicazione (caso comune è il form) oppure anche da applicazioni terze. Quindi la possibilità di leggere dati che si trovano altrove e creare visualizzazioni, interazioni, animazioni...

Molti servizi non forniscono solo la propria interfaccia per accedere ai propri dati ma forniscono un API disponibile per gli altri sviluppatori che possono comunicare con questo API utilizzando il linguaggio di http per creare delle nuove applicazioni, facendo delle operazioni di mashup (unione di dati che provengono da soggetti diversi).

Creare un API significa che sto creando un modo per comunicare con la mia applicazione

Queste possono essere:

- Applicazioni automatiche che utilizzano i dati dell'applicazione
- Applicazioni Web che mostrano all'utente un menù di opzioni, magari anche un form, e gli permettono di eseguire un'azione sui dati dell'applicazione

Rest (REpresentational State Transfer)

È un modo per sfruttare le caratteristiche di http e progettare delle API che sono ben fatte secondo i canoni di http e permettono di sfruttare i meccanismi di caching e i meccanismi messi in campo dagli intermediari

della comunicazione. E' il modello architetturale che sta dietro al World Wide Web e in generale dietro alle applicazioni web "ben fatte" secondo i progettisti di HTTP. Applicazioni non REST si basano sulla generazione di un API che specifica le funzioni messe a disposizione dell'applicazione, e alla creazione di un'interfaccia indipendente dal protocollo di trasporto e ad essa completamente nascosta. Viceversa, un'applicazione REST si basa fundamentalmente 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.

Non è un nuovo protocollo ma è il modo di utilizzare http sfruttando al meglio metodi, url e l'idea stessa di risorsa.

Il termine representational si riferisce al fatto che ho una risorsa con alcuni dati e che di questa risorsa trasferisco la rappresentazione dal server al client, o viceversa (operazione di aggiornamento), anche in formati diversi (html, json...) attraverso interfacce generiche.

Modello CRUD

È il modello che usiamo in REST. L'idea è che tutte le operazioni sui dati sono operazioni di creazione, lettura (individuale o contenitore), aggiornamento e cancellazione. Rest associa ad ogni operazione del modello CRUD un metodo di http e di usare sempre quel metodo per eseguire un'operazione su quella risorsa, distinguendo le risorse che rappresentano un'entità singola all'interno della mia applicazioni da quelle che rappresentano collezioni di entità. È un pattern tipico delle applicazioni di trattamento dei dati.

REpresentational State Transfer

L'architettura REST si basa su quattro punti:

1. Definire tutti i concetti che sono rilevanti nella nostra applicazione
2. Associare ad ognuno di questi concetti una risorsa (associargli un URI come l'identificatore e selettore primario).
3. Usare i verbi http per esprimere operazioni dell'applicazione secondo il modello CRUD
4. Una risorsa può avere rappresentazioni diverse che possono essere espresse tramite il content type

Individui e collezioni

L'idea alla base è che su ognuno di queste categorie posso esprimere tutte le operazioni di creazione, lettura, update e delete. Quando ho un URI, questo deve indentificare la risorsa e non l'operazione che devo fare su di essa. Quando eseguo un'operazione all'interno del body passo tutte le informazioni che sono necessarie per aggiornare quella risorsa o per sovrascriverla o solo per leggerla, sto passando tutto quello che mi serve per portare a termine queste operazioni.

REST identifica due concetti fondamentali: individui e collezioni e fornisce URI ad entrambi. Ogni operazione avviene su uno e uno solo di questi concetti. Su entrambi si possono eseguire operazioni CRUD. A seconda della combinazione di verbi e risorse otteniamo l'insieme delle operazioni possibili. Ciò che passa come corpo di una richiesta e/o risposta NON E' la risorsa, ma una rappresentazione della risorsa, di cui gli attori si servono per portare a termine l'operazione.

Gerarchie

La caratteristica principale di collezioni o delle individual rest sono le organizzazioni in gerarchie ossia di URI che mantengono o rappresentano le relazioni gerarchiche (vuol dire di contenimento o di applicazione fra le entità che mi interessano) all'interno del mio dominio. E' consigliabile strutturare gli URI in modo gerarchico, per esplicitare queste relazioni. API più leggibile e routing semplificato in molti framework di sviluppo.

Linee guida degli URI in REST

Individuare sia le collezioni che le entità singole con dei nomi: singolare per gli individui e plurale per le collezioni che è seguito da un identificatore che identifica un oggetto all'interno di quella collezione. Se ho collezioni che contengono altre collezioni sfrutto l'annidamento gerarchico.

Se l'URI termina con una barra significa che sto indicando una collezione di risorse mentre se non c'è la barra mi riferisco ad un solo individuo. Ad 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,ecc.)
/customers/abc123/addresses/1	primo indirizzo del cliente con id=abc123

Filtri e search negli URI REST

Un altro passo che facciamo nella costruzione della nostra API è la costruzione di filtri che mi permettono di selezionare una parte delle mie risorse passando dei parametri. Un filtro genera un sottoinsieme specificato attraverso una regola di qualche tipo. La gerarchia permette di specificare i tipi più frequenti e rilevanti di filtro. Come faccio a mettere i parametri? Attraverso le query string. Li esprimo dopo la parte parametrica, quindi dopo il punto interrogativo.

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 clienti che hanno comprato più di 100.000€

Semantica del POST

POST manda le informazioni per poter modificare, è uno metodi più controversi e non sempre viene usato con le stesse regole. Usarlo su una collezione per aggiornarla aggiungendo nuovi individui o su un individuo per modificare le informazioni di quell'individuo e di trasferire informazioni necessarie per il funzionamento dell'individuo stesso all'interno dell'applicazione. Il POST può essere usato in una moltitudine di situazioni secondo una semantica decisa localmente, purché non sovrapposta a quella degli altri verbi.

Altri consigli e linee guida

Adottare una convenzione di denominazione coerente e chiara negli URI. Usare gerarchie ma valutare i livelli necessari (chiarezza vs. carico sul server). Evitare di creare API che rispecchiano semplicemente la struttura interna di un database. Fornire meccanismi – parametri nelle query – per filtrare e paginare le risposte. Supportare richieste asincrone e in questo caso restituire codice HTTP 202 (Accettato ma non completato) e informazioni (URL) per accedere allo stato della risorsa.

Descrivere una RESTful API

Le API si possono descrivere in modo testuale oppure attraverso lo standard OpenAPI per dire quali sono le richieste, i parametri, quali sono i codici di errore, come è fatto il body... è un modo standardizzato e il più utilizzato.

Una API è RESTful se utilizza i principi REST nel fornire accesso ai servizi che offre. Per documentare un API è necessario definire:

- Endpoint → devo descrivere tutti i possibili URI dell'applicazione che posso utilizzare per interagire con essa.
- Metodi http di accesso → decidere quali operazioni rendere possibili.
- Rappresentazione input e dell'output → questi passano dal body. Le nostre richieste http hanno header e body, e all'interno di quest'ultimo metto i dati (di input nella richiesta, di output nella risposta) e possono essere di formato diverso e devono descrivere quali input sono validi e quali output mi aspetto di ricevere dalla mia applicazione. Di solito non si usa un linguaggio di schema, ma un esempio fittizio e sufficientemente complesso
- Condizione di errore e messaggi che restituisce in questi casi

Questo è quello che devo andare a descrivere .

Swagger e OpenAPI

OpenAPI nasce da un sistema proprietario che si chiama Swagger (che è un insieme di tool per la descrizione di API, soprattutto in ambito REST). Ha creato all'inizio un linguaggio per la documentazione delle API REST che veniva usato solo da lui ma che adesso si usa anche al di fuori di esso (OpenAPI).

È uno strumento che mi permette, automaticamente, di creare la documentazione, i modelli ma anche il codice. Può essere inizializzato serializzato in due linguaggi: JSON e YAML.

Apriamo una parentesi: YAML

È una sintassi derivata da Python per descrivere dei dati strutturati usando l'indentazione per annidare le informazioni. Simile a JSON (in realtà un superset). Supporto di tipi scalari (stringhe, interi, float), liste (array) e array associativi (coppie :)

Sezione PATH

La parte centrale di un'API descrive i percorsi (URL) corrispondenti alle operazioni possibili sull'API. Seguono la struttura:

<host>/<basePath>/<path>

Per ogni percorso (path o endpoint) si definiscono tutte le possibili operazioni che, secondo i principi REST, sono identificate dal metodo HTTP corrispondente. Per ogni path quindi ci sono tante sottosezioni quante sono le operazioni e per ognuna:

- Informazioni generali
- Parametri di input e di output

Parametri Input

I parametri in input sono descritti nella sezione parameters e per ogni parametro è possibile definire:

- ❖ tipo del parametro: keyword in che può assumere valori path, query o body
- ❖ nome (name) e descrizione (description)
- ❖ se è opzionale o obbligatorio (required)
- ❖ formato del/i valore/i che il dato può assumere (schema) → Il tipo può essere scalare (interi, stringhe, date, ecc.), o un oggetto o un vettore di valori scalari o oggetti

Oggetti e definizioni

La sezione definitions permette di definire i tipi degli oggetti, le loro proprietà e possibili valori.

Questi tipi possono essere referenziati (tramite schema -> \$ref) sia delle richieste che delle risposte.

Output

L'output (dati e codici e messaggi di errore) sono definiti attraverso la keyword responses. Si specifica il tipo di output atteso nel body della risposta. Inoltre ogni risposta ha un id numerico univoco, associato al codice HTTP corrispondente:

- ✓ 200 → viene usato per indicare che non c'è stato alcun errore
- ✓ 400 in su → vengono in genere usati per indicare messaggi di errore

Conclusioni

REST considera ogni applicazione come un ambiente di cui si cambia lo stato attraverso un insieme limitato di comandi (i metodi HTTP) applicati alle risorse (esprese attraverso URI) e visualizzate attraverso una esplicita richiesta di rappresentazione (attraverso un content Type MIME).

REST ha il pregio di sfruttare completamente ed esattamente tutti gli artifici del web, ed in particolare caching, proxying, sicurezza, ecc.

Inoltre l'aprirsi all'uso sistematico di URI permette ad applicazioni sofisticate basate su logica ed inferenza di sfruttare le tecniche del Semantic Web per creare funzionalità ancora più sofisticate e intelligenti con applicazioni create su architettura REST.

MARKUP: HTML, SGML, XML

HTML

È il linguaggio con cui sono scritte le pagine web. La M del nome significa markup e indica il linguaggio con cui marchiamo tutti i componenti di una pagina. È un linguaggio che mi permette di definire qual è il ruolo strutturale di ognuno degli elementi anche se storicamente nasce per descrivere contenuti che devono essere collegati fra di loro. Abbiamo due fasi di utilizzo:

- Per descrivere il sorgente di una pagina → ci serve un editor, o che mi permette di vedere un risultato, interfaccia grafica o editor testuale, che produce un file di testo
- Viene processato dal browser viene tradotto in una visualizzazione.

Spesso html lo usiamo anche come base per costruire le nostre applicazioni.

Il sorgente delle pagine Web è in HTML (HyperText Markup Language), un linguaggio che “marca”:

- + Struttura del documento
- + Informazioni di presentazione
- + Collegamenti ipertestuali (e URL della destinazione)
- + Risorse multimediali (e URL)

Il browser fa due cose:

- Parla http per recuperare i contenuti
- Quando i contenuti sono in html ce li fa vedere attraverso delle regole che ogni browser decide

Il browser è in grado di interpretare queste informazioni e visualizzare la pagina finale

Un passo indietro: il markup

Il linguaggio di markup non è altro che qualunque mezzo che usiamo per rendere espliciti l'interpretazione di un testo: ogni qual volta che abbiamo una sequenza di caratteri, quindi un testo che vogliamo processare, abbiamo bisogno di istruzioni che ci dicono cosa fare con quel testo. Queste istruzioni sono il markup. È stato creato prima dell'informatica e non con html. Quindi marchiamo le informazioni in modo che possano essere consumate o da umani o da macchine. Esempi di linguaggio di markup sono:

- ✓ PDF
- ✓ LaTeX
- ✓ Markdown
- ✓ XML
- ✓ HTML

Oltre a rendere il testo più leggibile, il markup permette anche di specificare ulteriori usi del testo. Con il markup per sistemi informatici (il nostro caso), specifichiamo le modalità esatte di utilizzo del testo nel sistema stesso. Esistono diversi tipi di markup tra cui presentazionale, procedurale, descrittivo e metalinguaggi per definire linguaggi di markup

Meta-linguaggio di markup

In html abbiamo gli elementi per marcare i paragrafi, le liste, le tabelle, i contenitori.... Quindi abbiamo bisogno di definire quali sono gli elementi del mio linguaggio. Per farlo usiamo il meta-linguaggio di markup. Questo è un linguaggio usato per definire dei linguaggi specifici di markup da utilizzare in un determinato contesto. Non hanno una conoscenza specifica del linguaggio ma forniscono grammatiche che ci permettono di definirli. Un esempio è:

- + SGML
- + XML

HTML nasce come linguaggio di markup definito usando la grammatica SGML

Markup SGML

Abbiamo un meta-linguaggio che mi permette di definire linguaggi che hanno due caratteristiche fondamentali:

- Strutturati → possono definire una serie di regole per cui un contenuto è corretto, a differenza di altri tipi di markup. Ho il focus sul contenuto della struttura della marcatura che stiamo usando.
- Gerarchico → perché in html, e tutti i linguaggi derivati da SGML, abbiamo la possibilità di creare dei contenitori che contengono altro contenuto e che contengono a loro volta altro contenuto e così via. Questa struttura la useremo pesantemente in due modi:
 - In CSS con le regole di stile
 - Per identificare elementi del linguaggio delle pagine che a loro volta ne contengono altri e che possono essere manipolati come entità a sé stanti.

Un documento con markup di derivazione SGML - inclusi HTML e XML – contiene:

- + Elementi, attributi e testo
- + Entità
- + Commenti e Processing Instructions

Elementi e attributi e testo

In SGML, e poi in XML, abbiamo l'utilizzo di:

- Elementi → è una parte di documento che ha un proprio significato, una propria struttura, che noi andiamo a identificare attraverso dei tag (ovvero pezzo di testo che mi permette di identificare l'inizio e la fine di un documento)

Apertura tag: < >

Chiusura tag: </ >

La parte di documento compresa tra tag di apertura e chiusura si chiama section.

- Attributi → mi permettono di aggiungere informazioni ai nostri elementi. Vanno nel tag iniziale. La sintassi è: nome = "valore". L'ordine degli attributi non è rilevante e html fornisce una serie di attributi.

- Testo → sono il contenuto che viene visualizzato. Viene anche detto, in XML , PCDATA perché vengono processati per sostituire le entità.

Entità

Mi servono tutte le volte che ho un pezzo di testo che voglio riutilizzare quindi lo specifico in un punto e negli altri lo vado a riutilizzare. Viene usato soprattutto per le lettere accentate e tutte quelle che non fanno parte dell'alfabeto ascii.

Le entità sono frammenti di documento memorizzati separatamente e richiamabili all'interno del documento. Sono usate sia nei linguaggi che nei metalinguaggi che li definiscono.

Commenti e processing instruction

I documenti di markup possono contenere commenti, non fanno parte del contenuto, ignorati dalle applicazioni di rendering. Le processing instructions (PI) sono elementi particolari usati per dare indicazioni alle applicazioni. Molto usate in XML, poco in HTML.

XML

Un altro meta-linguaggio è XML, molto simile a SGML ma con struttura molto più rigorosa. Quello che è stato fatto con XML è stato di definire i documenti validi in XML solo se rispettano alcune regole. Introduce due concetti:

- La buona forma → un documento è sufficientemente ben strutturato e rispetta alcune regole sintattiche per essere processato.
- Validazione → un documento segue uno schema quindi usa gli elementi e gli attributi definiti in quello schema.

Documenti XML ben formati

In XML c'è una netta distinzione tra queste due parti. Un documento XML si dice ben strutturato se rispetta alcune regole sintattiche:

- ❖ Tutti i tag di apertura e chiusura corrispondono e sono ben annidati
- ❖ Esiste un elemento radice che contiene tutti gli altri
- ❖ Gli elementi vuoti (senza contenuto) utilizzano un simbolo speciale di fine tag:
- ❖ Tutti gli attributi sono sempre racchiusi tra virgolette
- ❖ Tutte le entità sono definite

Se un documento non rispetta queste regole NON può essere processato da un'applicazione XML. Questa rigidità garantisce maggiore controllo e interoperabilità tra le applicazioni

E la buona forma in HTML?

È cambiata la logica con cui interpreto html. Le versioni di HTML hanno adottato in modo molto diverso le regole di buona forma e validazione, sotto le diverse spinte del W3C e del WHATCG e dei produttori di browser:

- HTML 4.x è un linguaggio SGML, ha molta flessibilità e alcune regole meno rigide
- XHTML1.0 e 2.0 derivano da XML e insistono sulle regole di buona forma

- HTML Living Standard (poi solo HTML) rilassa questi vincoli e definisce un algoritmo per fare il parsing di qualunque documento HTML

Da HTML a "tag soup"

I browser in realtà si sono preoccupati poco della correttezza dei documenti HTML e molte pagine sono diventate "tag soup". Le pagine anche se non hanno una struttura ben formata sono comunque processabili dal browser che riesce a visualizzarle. Esistono differenze anche sensibili tra un documento HTML corretto e uno "interpretato e visualizzato" da un browser. Per gestire sia le pagine conformi allo standard che quelle non compatibili, infatti, sono stati introdotti due modelli di rendering:

- Quirks mode → più permissivo
- Strict mode → compatibile con le specifiche ufficiali

Se non specificato, il browser adotta il modo compatibile. E questa (in)compatibilità è ormai lo standard con HTML Living Standard...

"Buona forma" in HTML 5

"HTML Living Standard" definisce un algoritmo per fare il parsing di qualunque documento HTML, anche dei documenti mal formati, sulla base di ciò che i browser già facevano (in quirks mode). In realtà dalla prospettiva WHATCG questi documenti non sono propriamente "mal formati" ma semplicemente "non strict". Sono validi a tutti gli effetti, tanto quanto i documenti XHTML! Pragmaticamente potremmo dire: "l'importante è arrivare ad una struttura dati in memoria unica su cui costruire applicazioni". È a questo scopo che la vera attenzione da parte del WHATWG è la costruzione di una struttura dati chiamata Document Object Model (DOM), a cui sia possibile arrivare a partire dalla stringa HTML e da cui si possa generare nuovamente una altra stringa HTML.

Document type

Nella sintassi SGML e XML, i documenti iniziano con dichiarazione di tipo (DocType) che serve a specificare le regole che permettono di verificare la correttezza strutturale. Esistono quindi diversi DocType per HTML:

```
<?DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 2.0//EN" "http://
www.w3.org/MarkUp/DTD/xhtml2.dtd">
```

HTML(5) semplifica e usa un unico DocType, da indicare all'inizio delle pagine prima dell'elemento radice:

```
<?DOCTYPE html>
```

Conclusione: un problema risolto? NO!

Aver uniformato l'algoritmo di parsing non è un deterrente per creare pagine "ben formate", al contrario lascia maggiore libertà e margine di errore agli sviluppatori. Più complesso estrarre dati e implementare manipolazioni automatiche dei contenuti, tranne ovviamente il caso in cui il sistema espone un'API di accesso ai dati. La cosa si complica ancora di più con i sistemi a componenti (es. AngularJS o React): lo stesso concetto di markup perderà valore a vantaggio di sintassi miste Javascript/Markup/CSS/whatever create ad hoc e mai standardizzate.

HTML: HYPERTEXT MARKUP LANGUAGE (PARTE I)

Hello World HTML

```

      Dichiarazione del tipo del documento
      (versione HTML e schema di "validazione")
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

HEAD
Informazioni sul documento

BODY
Contenuto vero e proprio

Ciò che è dentro a body è quello che noi visualizziamo nella finestra del browser mentre quello che è nell'head sono le informazioni aggiuntive che mi servono per migliorare la visualizzazione, aggiungere comportamenti dinamici, rendere visibile la pagina ai motori di ricerca, definire l'encoding

Premesse sintattiche

- Maiuscolo/minuscolo → HTML non è sensibile al maiuscolo/minuscolo (XHTML lo è e vuole tutto in minuscolo).
- Whitespace → HTML collassa tutti i caratteri di whitespace (SPACE, TAB, CR, LF) in un unico spazio. Questo permette di organizzare il sorgente in modalità leggibile senza influenzare la visualizzazione su browser. L'entità ` ` permette di inserire spazi non collassabili.
- Parsing e (non) buona forma → HTML rilassa diversi vincoli sulla buona forma, che invece erano imposti in XHTML

Entità in HTML

HTML definisce un certo numero di entità per quei caratteri che sono:

- proibiti perché usati in HTML (, & , " , ecc.)
- proibiti perché non presenti nell'ASCII a 7 bit.

Ad esempio:

- amp	&	quot	"
- lt (less than)	<	gt	>
- reg	®	nbsp (non-breaking space)	
- Aelig	Æ	Aacute	Á
- Agrave	À	Auml	Ä
- aelig	æ	aacute	á
- agrave	à	auml	ä
- ccedil	ç	ntilde	ñ
- ecc.			

Attributi globali: coreattrs

In HTML ci sono degli attributi core ovvero che sono disponibili su tutti gli elementi del linguaggio. I coreattrs costituiscono attributi di qualificazione e associazione globale degli elementi. Per lo più per CSS e link ipertestuali.

- ✓ Id → per indicare un pezzo specifico del contenuto. Un identificativo unico (su tutto il documento)
- ✓ Style → un breve stile CSS associato al singolo elemento
- ✓ Class → per assegnare una categoria. Una lista (separata da spazi) di nomi di classe (per attribuzione semantica e di stile CSS)
- ✓ Title → un testo secondario associato all'elemento (per accessibilità e informazioni aggiuntive)

Attributi globali: i18n e eventi

Gli attributi i18n (internationalization) garantiscono l'internazionalizzazione del linguaggio e la coesistenza di script diversi:

- Lang → una codifica dei linguaggi umani (stringa a due caratteri: it, en, fr, etc. da RFC1766)
- Dir → uno dei due valori ltr (left-to-right) o rtl (right-to-left) per indicare la direzione di flusso secondario del testo.

Gli attributi di evento permettono di associare script a particolari azioni sul documento e sui suoi elementi:

- Onclick
- Ondoubleclick
- Onmouseclick
- Onkeypress
- Ecc.....

Gli elementi di HTML

Gli elementi di HTML sono organizzati secondo alcune categorie:

Struttura complessiva

- + HTML
- + HEAD
- + BODY

Struttura del contenuto		<ul style="list-style-type: none"> + SECTION + NAV + FOOTER + Ecc...
Elementi di blocco e liste	La visualizzazione di default interrompe il flusso orizzontale	<ul style="list-style-type: none"> + P + H1 + H2 + DIV + UL + OL + Ecc..
Elementi inline	Mi permettono di marcare dei pezzi di testo e senza andare a capo	<ul style="list-style-type: none"> + B + I + SPAN + Ecc..
Elementi speciali		<ul style="list-style-type: none"> + A + IMG + HR + BR
Tabelle		<ul style="list-style-type: none"> + TABLE + TR + TD + TH
Form		<ul style="list-style-type: none"> + FORM + SELECT + OPTION + INPUT

Elementi inline

Gli elementi inline (o di carattere) non spezzano il blocco (non vanno a capo) e si includono liberamente l'uno dentro all'altro. Non esistono vincoli di contenimento. Si dividono in:

- Tag fontstyle → forniscono informazioni specifiche di rendering. Molti sono deprecati e si suggerisce comunque sempre di usare gli stili CSS. Ad esempio:

- + TT (TeleType, font monospaziato, ad es. Courier)
 - + I (corsivo)
 - + B grassetto
 - + U (sottolineato - deprecato)
 - + S e STRIKE (testo barrato - deprecato)
 - + BIG, SMALL (testo più grande e più piccolo)
- Tag phrase (di fraseazione o idiomatici) → aggiungono significato a parti di un paragrafo. Ad esempio:
- + EM (enfasi)
 - + STRONG (enfasi maggiore)
 - + DFN (definizione)
 - + CODE (frammento di programma)
 - + SAMP (output d'esempio)
 - + KBD (testo inserito dall'utente)
 - + VAR (variabile di programma)
 - + CITE (breve citazione)
 - + Q (citazione lunga),
 - + ABBR e ACRONYM (abbreviazioni ed acronimi)
 - + SUP e SUB (testo in apice e in pedice)
 - + BDO (bidirectional override)
 - + SPAN (generico elemento inline)

Elementi di blocco

I tag di blocco definiscono l'esistenza di blocchi di testo che contengono elementi inline. Elementi base:

- P (paragrafo)
- DIV (generico blocco)
- PRE (blocco preformattato)
- ADDRESS (indicazioni sull'autore della pagina)
- BLOCKQUOTE (citazione lunga)

Blocchi con ruolo strutturale:

- H1, H2, H3, H4, H5, H6 (intestazione di blocco)

Elementi per liste

Le liste di elementi sono contenitori di elementi omogenei per tipo. Sono:

- UL → Lista a pallini di ; Attributo type (disc, square, circle)
- OL → lista a numeri o lettere di ; attributi start (valore iniziale) e type (1, a, A, i, I).
- DIR, MENU → liste compatte, poco usate
- DL → lista di definizioni <DT> (definition term) e <DD> (definition data)

Elementi generici

<div> e sono cosiddetti elementi generici: privi di caratteristiche semantiche o presentazionali predefinite, assumono quelle desiderate con l'aiuto dei loro attributi (style, class, lang).

<div> mantiene la natura di elemento blocco, e > la natura di elemento inline, ma ogni altra caratteristica è neutra.

```
<div id="introduction">
  <p>
    <span class="name">Angelo Di Iorio</span> works
    at <span class="place">Bologna</span>
  </p>
  ...
</div>
```

Link ipertestuali (anchors) (1)

I link sono definiti con elementi <a> (àncore nel documento). <a> è sintatticamente un elemento inline. L'unica limitazione è che gli elementi <a> non possono annidarsi. Attributi:

- Href → specifica l'URI della destinazione. Quindi è l'ancora di partenza di un link.
- Name → specifica un nome che può essere usato come ancora di destinazione di un link. Quindi è l'ancora finale di un link.

Immagini

Le immagini inline sono definite attraverso l'elemento IMG. Formati tipici: JPEG, GIF, PNG.

```

```

Alcuni attributi:

- ✓ SRC (obbligatorio) → l'URL (assoluto o relativo) del file contenente l'immagine
- ✓ ALT → testo alternativo in caso di mancata visualizzazione dell'immagine
- ✓ NAME → un nome usabile per riferirsi all'immagine

- ✓ WIDTH → forza una larghezza dell'immagine.
- ✓ HEIGHT → forza una altezza dell'immagine.
- ✓ ALIGN, BORDER, VSPACE, HSPACE → deprecati, specificano il rendering.

Elemento figure

HTML5 ha introdotto anche un elemento specifico per aggiungere un'immagine ad un documento, corredata da una didascalia, in un unico blocco semanticamente rilevante. Diverso da IMG che è un elemento inline.

```
<figure>
  
  <figcaption>La Primavera di Botticelli
</figcaption>
</figure>
```

Elementi di struttura

HTML 4 fornisce pochi elementi per strutturare i contenuti in sezioni e sottosezioni: il contenitore gerarchico <div> (cui associare un ruolo tramite l'attributo @class) e gli elementi <h1>, ..., <h6> (che però non sono annidati). HTML 5 (ma la maggior parte erano già presenti in XHTML 2.0) introduce nuovi elementi con una semantica precisa, per organizzare il documento in contenitori:

- ❖ <section> → un contenitore generico annidabile
- ❖ <article> → una parte del documento self-contained e pensata per essere ri-usata e distribuita come unità atomica (post di un blog, news, feed, etc.)
- ❖ <aside> → una sezione collegata al testo ma separata dal flusso principale (note a margine, sidebar, incisi, pubblicità, etc.)
- ❖ <header> e <footer> → elementi iniziali e finali di un documento
- ❖ <nav> → liste di navigazione

Header e footer

<header> contiene l'intestazione della sezione corrente o dell'intera pagina ed è solitamente usato per tabelle di contenuti, indici, form di ricerca, intestazioni. Può essere anche usato come contenitore degli headings H1, .., H6.

<footer> contiene la "parte conclusiva" della sezione corrente o dell'intera pagina. E' usato principalmente per mostrare informazioni (testuali, non metadati) sugli autori della pagina, copyright, produzione, licenze. Non deve essere visualizzato necessariamente a fondo pagina. Può essere usato anche per contenere intere sezioni come appendici, allegati tecnici, colophon, etc.

Liste di navigazione

HTML 5 riprende da XHTML 2.0 anche le “liste di navigazione” ossia particolari sezioni dedicate a raggruppare link alla pagina corrente (o a sezioni di) o ad altre pagine. Si usa l’elemento <nav> molto spesso in combinazione con <header> e <footer>. Le sezioni <nav> sono molto utili per l’accessibilità, in quanto possono essere più facilmente identificate e accedute anche da utenti disabili (tramite screenreaders o altri ausili). Possono essere usate anche per attivare/disattivare le funzionalità di navigazione in base allo user-agent (browser) che sta accedendo alla pagina

Tabelle

Le tabelle vengono specificate riga per riga. Di ogni riga si possono precisare gli elementi, che sono o intestazioni o celle normali. Una tabella può anche avere una didascalia, un’intestazione ed una sezione conclusiva. E’ possibile descrivere insieme le caratteristiche visive delle colonne. Le celle possono occupare più righe o più colonne. Si usano gli attributi @rowspan e @colspan per indicare il numero di righe o colonne occupate dalla cella.

Elementi e attributi per tabelle

- TABLE → tabella. @border, @cellpadding e @cellspacing per indicare bordi e spazi tra le celle e nelle celle (deprecati, meglio usare CSS)
- TR → riga
- TD e TH → cella semplice o di intestazione. @rowspan e @colspan per indicare celle che occupa più righe o colonne
- THEAD, TBODY, TFOOT → raggruppano righe rispettivamente in intestazione, body e footer della tabella.
- CAPTION → didascalia
- COLGROUP, COL → specificano proprietà di gruppi colonne

CASCADING STYLE SHEETS

HTML e presentazione

Nella versione iniziale, il Web e HTML ponevano poca enfasi sugli aspetti presentazionali. Con la guerra dei browser, sono stati aggiunti diversi elementi e attributi per esprimere caratteristiche tipografiche e di presentazione, fanno parte di HTML, non sono in una specifica separata.

CSS è un linguaggio che ci permette di esprimere degli aspetti presentazionali. HTML all’inizio poneva l’accento solo su aspetti strutturali poi sono state introdotte alcune caratteristiche di presentazione. Lo sforzo che è stato fatto con CSS è stato di estrarre queste informazioni, quindi è un linguaggio dedicato alla gestione e descrizione degli stili.

Un aspetto interessante: il primo prototipo di Berners-Lee e i primi browser, permettevano ai lettori di definire personalmente come presentare i documenti HTML (es. dimensioni e font)

Stili a cascata

Bert Bos (belga) e Håkon Lie (danese) sono tra i tanti proponenti di un linguaggio di stylesheet per pagine HTML: il Cascading Style Sheet (CSS). La parola chiave è cascading: è prevista ed incoraggiata la presenza di fogli di stile multipli, che agiscono uno dopo l'altro, in cascata, per indicare le caratteristiche tipografiche e di layout di un documento HTML

Molteplici fogli di stile destinati anche a dispositivi diversi. La forza di CSS è che il contenuto possa essere presentato sulle diverse piattaforme. Questo riprende un concetto iniziale di Burners-Lee in cui si permetteva ai lettori di aggiungere uno stile personalizzato sui contenuti. Da questo derivano anche che possono esistere una serie di contenuti privi di presentazione, possono essere in HTML ma anche in XML, posso applicare fogli di stile a cascata.

Caratteristiche:

- Controllo sia dell'autore sia del lettore di un documento HTML
- Indipendente dalla specifica collezione di elementi ed attributi HTML (si può usare anche su XML)

Come e a chi assegnare gli stili

HTML prevede l'uso di stili CSS in tre modi diversi:

- ✓ Posizionato direttamente sull'elemento (da usare poco o mai!)
- ✓ Posizionato nell'elemento `<style>` → `<style>` attributo core di HTML e che può essere aggiunto a qualunque elemento e mi permette di definire le proprietà CSS su quell'elemento specifico.
- ✓ Indicato dell'elemento `<link>` → `<link>` ha un attributo href che mi permette di indicare l'URL del CSS.

Una volta definito il CSS vuol dire che ho applicato una serie di regole al documento corrente.

Selettori → componenti delle regole che mi permettono di capire qual è il pezzo di sorgente di HTML o XML su cui vado ad applicare queste regole. Questi sono:

- Tipo → mi permettono di associare delle regole a tutti gli elementi che si chiamano in un determinato modo
- Categoria → mi permettono di associare una regola a tutti gli elementi che hanno una determinata categoria. Tutti gli elementi che hanno l'attributo `@class` che contiene un determinato valore. `@class` assume un valore qualunque:
 - + più elementi possono condividere lo stesso valore, in modo da assegnare gli elementi a diverse categorie che si riferiscono a differenti semantiche
 - + si possono specificare più classi per uno stesso elemento, separandole attraverso uno spazio
- Id → mi permettono di specificare un elemento specifico. Assume un valore univoco su tutto il documento, in modo da identificare quello specifico elemento tra tutti gli altri

La cascata

Regole a cascata e ereditarietà sono i due meccanismi che mi permettono di avere il risultato finale. Se ho più regole il risultato finale deriva dalla somma di tutte queste regole. Posso avere più regole che si

applicano allo stesso elemento e la soluzione finale è il risultato finale di queste regole che si svolgono a cascata. Il conflitto viene gestito da un algoritmo predefinito del browser che tiene conto dell'ordine in cui applico i vari stili, della specificità e infine l'ordine.

Gli attributi di un elemento vengono presi non da uno (il primo, l'ultimo, ecc.) dei fogli di stile, ma composti dinamicamente sulla base del contributo di tutti, in cascata

Ereditarietà

Gli elementi HTML (e i contenitori CSS) sono organizzati in una struttura gerarchica. A parte alcune eccezioni, le proprietà CSS degli elementi sono ereditate, ossia assumono lo stesso valore che hanno nel contenitore dell'elemento a cui si riferisce la proprietà. La proprietà display (che serve per specificare il flusso del testo, es. blocchi vs. inline) NON è ereditata

Computed style (1)

Gli stili vengono così calcolati da questo algoritmo che decide qual è la regola che vado ad applicare. Lo stile finale di un elemento viene quindi "calcolato" combinando le diverse sorgenti e regole. Viene applicato un algoritmo di ordinamento sulle dichiarazioni secondo alcuni principi:

1. Tipo di dispositivo (print, screen, speech, ecc.), ne parleremo
2. Importanza di una dichiarazione, esprimibile tramite il simbolo "!" prima di una regola
3. Origine della dichiarazione (utente > autore > user agent)
4. Specificità della dichiarazione (es. specifica su un elemento > ereditata)
5. Ordine in cui si trovano le dichiarazioni (si applica l'ultima)

Proprietà e statement

Tutte le regole CSS hanno due parti:

- Foglio di stile
- Insieme delle regole → non usano la sintassi HTML ma una separata. Questa ha un insieme di regole ognuna delle quali è identificata da un selettore che mi dice quali sono gli elementi su cui applicare la regola, e un insieme di proprietà (statement) espresse con chiave nome-valore.

Una proprietà è una caratteristica di stile assegnabile ad un elemento, CSS1 prevede 53 proprietà diverse, CSS2 ben 121, CSS3 abbiamo perso il conto.

Tipi di dato

I valori delle proprietà possono essere di tipo diverso (ci torneremo quando parleremo delle varie proprietà CSS):

- Interi e reali → rappresentano numeri assoluti, e sono usati ad esempio per indicare l'indice in caso di elementi sovrapposti (proprietà z-index)
- URI → indica un URI, ed è usato per caricare risorse esterne, ad esempio importare altri fogli di stile o immagini di background
- Stringhe → una stringa posta tra virgolette semplici o doppie. Si usa ad esempio per indicare contenuto generato automaticamente e aggiunto alla pagina (proprietà content) o il nome di un font

- Dimensioni → valori numerici rispetto ad un'unità di misura assoluta (px, pt, in, cm) o relativa (vh, vw, fr, ecc.) o percentuali

Colori

I colori sono indicati in tre modi:

- Nome → come definiti in HTML. Sintassi HTML (#XXXXXX)
- Codice RGB → sintassi rgb(X,X,X) o rgba(X,X,X,O), dove X è un numero tra 0 e 255 mentre O (opacità) è un numero tra 0 e 1

Selettori e regole

Un selettore permette di specificare un elemento o una classe di elementi dell'albero HTML (o XML) al fine di associarvi caratteristiche CSS.

Una regola è un blocco di statement associati ad un elemento attraverso l'uso di un selettore

Selettori principali

pattern	significato	descrizione	esempio
*	qualunque elemento	Selettore universale	*
E	un elemento di tipo E	Selettore di tipo	h1
E.nomeclasse	un elemento di tipo E e di classe nomeclasse	Selettore di classe	p.codice
E#ilmioid	un elemento di tipo E e con id ilmioid	Selettore di id	p#abc1

Altri selettori

pattern	significato	descrizione	esempio
E::first-line E::first-letter	la prima riga (lettera) formattata dell'elemento E	pseudo-elemento	p::first-line { text-transform: capitalize; }
E::before E::after	contenuto generato prima (dopo) dell'elemento E	pseudo-elemento	q::before { content:"<<"; }
E F	elemento F discendente di un elemento E	combinatore di prossimità	table th
E > F	elemento F figlio di un elemento E	combinatore di prossimità	tr > th
E[foo="bar"]	un elemento E con un attributo foo con valore uguale a "bar"	attributi	table[border="1"]
E:hover	un elemento E ha il puntatore sopra di esso	pseudo-classe	a:hover { cursor:pointer; }

pattern	significato	descrizione	esempio
<code>E:nth-child(n)</code>	un elemento <i>E</i> che è l' <i>n</i> -simo figlio di suo padre	<code>p:nth-child(odd)</code>	<code>E:nth-child(n)</code>
<code>E:first-child</code>	un elemento <i>E</i> che è il primo figlio di suo padre	<code>h1:first-child</code>	<code>E:first-child</code>
<code>E:visited</code>	un elemento <i>E</i> che è un link già visitato	Link	<code>a:visited{color:gray;}</code>

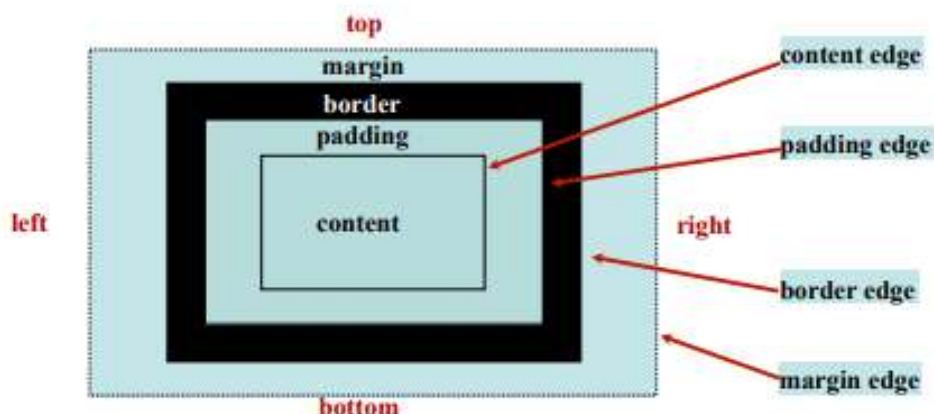
La scatola CSS (box)

La visualizzazione di un documento con CSS avviene identificando lo spazio di visualizzazione di ciascun elemento. Le scatole possono occupare diverse posizioni nello spazio di visualizzazione

CSS organizza tutti i contenuti in box, scatole rettangolari, che sono dislocate nello spazio. Si basa a tutti gli elementi della pagina. Questa box ha delle proprietà:

- Margine
- Bordo
- Padding
- Contenuto

Elementi della scatola



- Margine → la regione che separa una scatola dall'altra, sempre trasparente (ovvero ha il colore di sfondo della scatola contenitore)
- Bordo → la regione ove visualizzare un bordo per la scatola
- Padding → la regione di respiro tra il bordo della scatola ed il contenuto. Ha sempre il colore di sfondo del contenuto

- Contenuto → la regione dove sta il contenuto dell'elemento

Proprietà della scatola

Esistono quindi le corrispondenti proprietà per controllare questi elementi. Inoltre su una scatola CSS è possibile definire colore del testo (ereditato dagli elementi che contiene) e background (non ereditato). A proposito del bordo: la proprietà border-radius permette di creare bordi arrotondati.

Forme abbreviate

Nella definizione delle proprietà di una box - ma è possibile anche con altre proprietà - si usano spesso forme abbreviate. CSS permette di riassumere in un'unica proprietà i valori di molte proprietà logicamente connesse. Si usa una sequenza separata da spazi di valori, secondo un ordine prestabilito (senso orario per le box). Se si mette un valore solo esso viene assunto da tutte le proprietà individuali.

Layout e posizione delle box

Le scatole sono in relazione alle altre e la loro visualizzazione nello spazio è legata a due aspetti:

- Flusso → indica il modo in cui una scatola è posta rispetto alle altre e all'interno della scatola che la contiene
- Posizione → indica la posizione della scatola rispetto al flusso e alle altre scatole

CSS definisce diverse proprietà per controllare questi aspetti e creare layout sofisticati e responsive.

Flusso

I flussi sono controllati dalla proprietà display. Ogni elemento di HTML ha un valore di default per questa proprietà e quindi un comportamento di default rispetto al flusso. Flussi di base:

- Flusso blocco: le scatole sono poste l'una sopra l'altra in successione verticale (come paragrafi)
- Flusso inline: le scatole sono poste l'una accanto all'altra in successione orizzontale (come parole della stessa riga)
- Flusso float: le scatole sono poste all'interno del contenitore e poi spostate all'estrema sinistra o destra della scatola, lasciando ruotare le altre intorno

CSS e testo

CSS definisce molte proprietà per il controllo del testo. Riassumiamo un po' di terminologia:

- Font → collezione di forme di caratteri integrate armoniosamente per le necessità di un documento in stampa. Tipicamente contiene forme per lettere alfabetiche, numeri, punteggiatura e altre caratteri standard nello scritto
- font-family (o type face) → stile unico che raggruppa molti font di dimensione, peso e stili diversi
- peso → spessore dei caratteri
- stile → effetto sul testo, normale, corsivo o obliquo

Esistono diverse classificazioni dei font, proposte negli anni, e ovviamente moltissimi file di font utilizzabili in CSS

Proprietà del testo

font-family: il/i nomi del/dei font (es: "Times New Roman", Georgia, Serif)

– l'ordine indica una preferenza, importante indicare sempre il fallback

font-style: (normal | italic | oblique)

font-variant: (normal | small-caps) *

font-weight: (normal | bold | bolder | lighter | 100<-> 900),

font-stretch: (normal | wider | narrower | etc.): caratteristiche del font

text-indent, text-align: indentazione, allineamento e interlinea delle righe della scatola

text-decoration (none | underline | overline | line-through | blink),

text-shadow: ulteriori stili applicabili al testo

text-transform (capitalize | uppercase | lowercase | none):
trasformazione della forma delle lettere

letter-spacing e word-spacing: spaziatura tra lettere e tra parole

white-space (normal | pre | nowrap): specifica la gestione dei ritorni a capo e del collassamento dei *whitespace* all'interno di un elemento

Font con grazie (serif) o senza grazie (sans serif)

Una importante classificazione distingue i font con o senza grazie, ossia allungamenti alle estremità dei caratteri.

DOCUMENT OBJECT MODEL

DOM

È il modo con cui interagiamo con una pagina html del nostro browser piuttosto che una pagina XML della quale abbiamo fatto il parsing. Il DOM è la rappresentazione in memoria ed è stata standardizzata. Il document object model è un'interfaccia di programmazione quindi un insieme di metodi e proprietà che caratterizzano la rappresentazione ad albero di un documento XML o HTML. Che caratterizzano due cose:

- da un lato la struttura → ci permette di definire i componenti logici della pagina XML
- dall'altro, essendo un API, le operazioni che sono possibili su questa struttura dati. Quindi il modo in cui lavoriamo per manipolare questa struttura dati

Il passaggio successivo sarà di manipolare la struttura di dati in memoria sul browser nel momento della visualizzazione.

Il DOM non è un API specifico solo per javascript ma è standardizzato, cioè esistono diverse versioni per manipolare documenti XML sfruttando le stesse operazioni. Così ci permette di gestire la struttura ad albero del documento in maniera corretta.

Definisce la struttura logica dei documenti ed il modo in cui si accede e si manipola un documento. Utilizzando DOM i programmatori possono costruire documenti, navigare attraverso la loro struttura, e aggiungere, modificare o cancellare elementi. Ogni componente di un documento HTML o XML può essere letto, modificato, cancellato o aggiunto utilizzando il Document Object Model.

Oggetti del DOM

L'oggetto principale è il DOMnode, che è un'istanza che viene poi usata per creare delle istanze specifiche di tutti gli oggetti che abbiamo nella pagina. Il core del DOM definisce alcune classi fondamentali per i documenti HTML e XML, e ne specifica proprietà e metodi.

- DOMdocument → è il documento di cui si sta parlando. Bisogna tenere presente che è il documento completo quindi dobbiamo distinguere DOMdocument dalla radice del nostro documento perché il documento completo potrebbe contenere delle intestazioni.
-
- DOMElement → ogni singolo elemento del documento
-
- DOMAttr → ogni singolo attributo del documento
-
- DOMText → ogni singolo attributo del documento
-
- DOMComment
-
- DOMProcessingInstruction
-
- DOMDataSection
-
- DOMDocumentType
-
- Ecc....

Per ognuna di questi oggetti, o meglio, di queste classi definite all'interno dello standard DOM esistono dei metodi che mi descrivono le operazioni possibili. Questi sono i metodi che andremo ad utilizzare per manipolare correttamente le strutture dati.

DOM node

DOMNode specifica i metodi per accedere a tutti gli elementi di un nodo di un documento, inclusi il nodo radice, il nodo documento, i nodi elemento, i nodi attributo, i nodi testo, ecc. Semplificando:

membri	metodi
- nodeName	insertBefore()
- nodeValue	replaceChild()
- nodeType	removeChild()
- parentNode	appendChild()
- childNodes	hasChildNodes()
- attributes	hasAttributes()

DOM document

DOMDocument specifica i metodi per accedere al documento principale, tutto compreso. È equivalente alla radice dell'albero (non all'elemento radice!!!). Semplificando:

membri	metodi
- docType	createElement()
- documentElement	createAttribute()
	createTextNode()
	getElementsByTagName()
	getElementById()

Operazioni di creazione → sono operazioni che mi permettono di creare degli oggetti che sono associati al documento ma non occupano nessuna posizione specifica.

Operazioni di get → mi permettono di cercare degli elementi all'interno della pagina.

DOM element

Mi permette di fare le manipolazioni. DOMElement specifica i metodi e i membri per accedere a qualunque elemento del documento. Semplificando:

membri	metodi
- tagName	getAttribute()
	setAttribute()
	removeAttribute()
	getElementsByTagName()
	getElementById()

... e analogamente per le altre classi ed interfaccia del DOM.

Altro esempio di navigazione DOM

Particolarmente usate le funzioni DOM getElementsByTagName() e getElementById().

Restituiscono una lista di nodi e un singolo nodo, indipendentemente dalla loro posizione nell'albero

HTML: HYPERTEXT MARKUP LANGUAGE

Form

Questi sono lo strumento di html che ci permette di inserire all'utente dei valori che sono spediti direttamente sul server per poter eseguire delle operazioni su questi dati. È un oggetto specifico del

browser ma le nostre applicazioni devono essere pensate in maniera indipendente a che vengano usate attraverso un form.

Con i FORM si utilizzano le pagine HTML per inserire valori che vengono poi elaborati sul server. I FORM sono legati ad applicazioni server-side

Oggetto che lavora client side, si occupa di reagire agli eventi che l'utente fa, impacchettare i dati che l'utente crea/inserisce all'interno del form, costruisce una risposta http che viene spedita direttamente al server che rielabora.

In javascript è possibile controllare il comportamento di un form, quindi scrivere del codice che a sua volta richiama http. Quindi attraverso il form è il browser che fa direttamente la richiesta.

L'attributo principale dell'elemento form è ACTION che ci specifica l'azione che deve essere eseguita su quella risorsa. Una volta costruito il form, ACTION e METHOD ci permettono di dire dove dovranno essere spediti i dati. Sono parametri globali di form e poi è costituito da tanti elementi quanti sono i singoli modi che l'utente ha per aggiungere questi contenuti.

Il browser raccoglie dati dall'utente con un form. Crea una connessione HTTP con il server, specificando una ACTION (cioè un'applicazione che funga da destinatario) a cui fare arrivare i dati. Il destinatario riceve i dati, li elabora e genera un documento di risposta, che viene spedito, tramite il server HTTP, al browser.

I controlli tipati e nominati vengono usati per l'inserimento dei dati nei form: campi di inserimento dati, pulsanti, bottoni radio, checkbox, liste a scomparsa, ecc.

Gli elementi di un form sono:

- `<form>`: il contenitore dei widget del form
 - **Attributi:**
 - `method`: il metodo HTTP da usare (GET, POST in HTML)
 - `action`: l'URI dell'applicazione server-side da chiamare
- `<input>`, `<select>`, `<textarea>`: i widget del form.
 - **Attributi:**
 - `name`: il nome del widget usato dall'applicazione server-side per determinare l'identità del dato
 - `type`: il tipo di widget (input, checkbox, radio, submit, cancel, etc.)
 - **N.B.:** Tutte le checkbox e tutti i radio buttons dello stesso gruppo condividono lo stesso nome.
- `<button>`: un bottone cliccabile (diverso dal submit)
- `<label>`: la parte visibile del widget.

Un passo indietro: caratteri ammessi negli URI

Nel momento in cui il form è compilato, il browser fa richiesta http e per fare questo deve tradurre questi dati che sono nel form in qualcosa che viaggia attraverso il protocollo. C'è una codifica specifica che si chiama code encoded che passa le informazioni dei form attraverso un URI che viene specificato nell'URI della risorsa, quindi come parametro della richiesta che viene fatta, oppure la stessa codifica può essere fatta anche all'interno del body. Questi dati vengono presi e trasferiti all'interno dell'URI che hanno come prima parte i contenuti di ACTION e come seconda parte, la parte query, dove ognuno dei parametri è seguito dal proprio valore. I caratteri degli URI possono essere:

- + Unreserved → utilizzabili liberamente, sono alfanumerici caratteri alfanumerici (lettere maiuscole e minuscole, cifre decimali) e alcuni caratteri di punteggiatura non ambigui `-_!~*()'`
- + Reserved → che hanno delle funzioni particolari in uno o più schemi di URI. In questo caso vanno usati direttamente quando assolvono alle loro funzioni e devono essere "escaped" per essere usati come parte della stringa identificativa naturale `;/?:@&=+$`

- + Escaped → caratteri riservati di cui si fa escaping per usarli nelle stringhe identificative (attenzione alle successive elaborazioni degli URI e dei caratteri riservati)

Caratteri escaped

Tutti quei caratteri che non sono ascii. I caratteri escaped fanno riferimento alle seguenti tipologie di caratteri:

- I caratteri non US-ASCII (cioè ISO Latin-1 > 127)
- I caratteri di controllo: US-ASCII < 32
- I caratteri unwise: { } | \ ^ [] `
- I delimitatori: spazio <> # % "
- I caratteri riservati quando usati in contesto diverso dal loro uso riservato

In questo caso i caratteri vanno posti in maniera escaped, secondo la seguente sintassi:

%XX, dove XX è il codice esadecimale del carattere

La codifica application/x-www-form-urlencoded

E' usata per spedire coppie in operazioni di POST da un form HTML. Si può usare sia nell'URL che nel body della richiesta.

```
http://www.site.com/serverside.py?name=John&surname=Smith&gender=m&likes=art&likes=lit&nationality=italian
```

- i codici non alfanumerici sono sostituiti da '%HH' (HH: codice esadecimale del carattere),
- gli spazi sono sostituiti da '+',
- i nomi dei controlli sono separati da '&',
- il valore è separato dal nome da '='

Un'altra parentesi: MIME

MIME ridefinisce il formato del corpo di messaggio SMTP per 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

Qui ci interessa una caratteristica specifica di MIME: un messaggio può contenere parti di tipo diverso (es. un messaggio di tipo testo e un attachment binario). In questo caso si creano dei sottomessaggi MIME per ciascuna parte e il messaggio MIME complessivo diventa "multi-parte", qualificando e codificando in maniera diversa ciascuna sottoparte.

La codifica multipart/form-data

I dati spediti via form possono essere codificati anche con un messaggio multipart, usando la codifica multipart/form-data. Si può usare per qualunque campo del form ma si usa principalmente per spedire file. Il messaggio è diviso in blocchi di dati, delimitati da boundary, e ognuna può avere Content-type diverso

Interactive content: form

HTML 5 introduce molte novità per velocizzare, semplificare e controllare l'inserimento dei dati da parte dell'utente. L'oggetto input è arricchito con due attributi che permettono di controllare il focus e aggiungere suggerimenti agli utenti:

- Placeholder → contiene una stringa che sarà visualizzata in un campo di testo se il focus non è su quel campo
- Autofocus → indica il campo sul quale posizionare il focus al caricamento del form.

In HTML 5 sono introdotti molti nuovi tipi per l'oggetto input. I browser forniscono widget diversi nell'interfaccia in base al tipo del campo, indicato nell'attributo @type

Alcuni nuovi tipi di input

- **email**: in fase di validazione il browser verifica se il testo inserito contiene il simbolo '@' e il dominio è (sintatticamente) corretto
- **url**: si verifica che il testo inserito segue le specifiche degli URL
- **number**: il browser visualizza bottoni per incrementare/decrementare il valore. E' possibile specificare minimo, massimo e unità di modifica in altri attributi
- **range**: il browser visualizza uno slider per incrementare o decrementare un valore numerico. E' possibile specificare il valore minimo, massimo e l'unità di modifica.
- **date**: richiede al browser la visualizzazione di un calendario tra cui selezionare una data. Esistono vari tipi collegati come **month**, **week**, **time**
- **search**: testo renderizzato in modo diverso su alcuni browser (iPhone)
- **color**: usato per mostrare una tavolozza di colori, da cui selezionare un codice RGB

Validazione automatica

HTML 5 prevede anche la possibilità (anzi, è il default) di validare un form client-side, senza ricorrere a funzioni Javascript aggiuntive.

Dopo l'evento submit i dati inseriti nel form sono verificati in base al tipo di ogni campo (email, URL, etc.). Si può catturare l'evento invalid e implementare comportamenti specifici.

possibile inoltre indicare i campi obbligatori, tramite l'attributo **required** dell'oggetto input. L'attributo **novalidate** può essere invece usato per indicare di non validare l'intero form o uno specifico campo.

```
<input name="mail" type="email" required>
<input name="url" type="url" novalidate>
```

<head>

L'elemento contiene delle informazioni che sono rilevanti per tutto il documento. Esse sono:

- **<title>**: il titolo del documento
- **<link>**: link di documenti a tutto il documento
- **<script>**: librerie di script
- **<style>**: librerie di stili
- **<meta>**: meta-informazioni sul documento
- **<base>**: l'URL da usare come base per gli URL relativi

<title>: titolo del documento

Contiene semplice testo (non elementi HTML) che definisce il titolo del documento.

- titolo della finestra (o tab) del browser
- nome illustrativo della pagina nei bookmark
- nome illustrativo della pagina nei motori di ricerca. Inoltre i motori di ricerca trattano con più importanza il contenuto dell'elemento title rispetto al resto del contenuto della pagina.

<link>, <script> e <style>

Gli elementi SCRIPT e STYLE si possono definire, rispettivamente, blocchi di funzioni di un linguaggio di script e blocchi di stili di un linguaggio di stylesheet.

```
<style type="text/css">
  p {color:red;}
</style>
```

A volte può esser utile mettere esternamente queste specifiche, e riferirvi esplicitamente. In questo caso si usa LINK, che permette di creare un link esplicito e tipato al documento esterno.

```
<link rel="stylesheet" type="text/css"
href="style1.css"/>
```

<meta>: meta informazioni

E' un meccanismo generale ovvero esistono una serie di attributi che hanno un valore specifico che siamo abituati ad utilizzare. Due attributi principali: @name e @content

```
<html>
<head>
  <meta charset="utf-8">
  <title>My document</title>
  <meta http-equiv="Content-Language" content="en-US"/>
  <meta name="keywords" content="HTML, Web, ">
  <meta name="author" content="Angelo Di Iorio">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

Tre tipi di meta-informazioni definibili:

- ✓ La codifica caratteri utilizzata nel file:
`<meta charset="utf-8">`
- ✓ Intestazioni HTTP: la comunicazione HTTP fornisce informazioni sul documento trasmesso, ma il suo controllo richiede accesso al server HTTP. Con il tag META si può invece fornire informazione "stile-HTTP" senza modifiche al server.
`<meta http-equiv="expires" content="Sat, 23 Mar 2019 14:25:27 GMT">`
- ✓ Altre meta-informazioni: i motori di ricerca usano le metainformazioni (ad esempio "Keyword") per organizzare al meglio i documenti indicizzati

<base>: URL relativi ed assoluti

Ogni documento HTML visualizzato in un browser ha associato un URL ed è possibile esprimere le risorse collegati tramite URL assoluti o relativi. L'element BASE permette di indicare l'URL di base da usare per risolvere URI relativi, nell'attributo @href. Se non specificato si usa l'URL del documento corrente ma è possibile esplicitarlo.

`<base href="www.cs.unibo.it/corsi/tw/">`

Embedded content

HTML5 estende notevolmente le possibilità di includere contenuti multimediali nelle pagine e permette di interagire con questi elementi in modo sofisticato. Alcuni elementi:

- `<canvas>`: immagini bidimensionali
- `<audio>`: file audio
- `<video>`: file video
- `<math>`: frammenti MathML per espressioni matematiche

Il modello ad eventi di DOM è esteso con eventi specifici che permettono la manipolazione degli oggetti. Noi non guardiamo in dettaglio questi elementi ed API.

Canvas

L'elemento definisce un'area rettangolare in cui disegnare direttamente immagini bidimensionali e modificarle in relazione a eventi, tramite funzioni Javascript. La CanvasAPI descrive questi metodi di manipolazione. La larghezza e l'altezza del canvas sono specificati tramite gli attributi `width` e `height` dell'elemento. Le coordinate (0,0) corrispondono all'angolo in alto a sinistra. Gli oggetti non sono disegnati direttamente sul canvas ma all'interno del contesto, recuperato tramite un metodo Javascript dell'elemento chiamato `getContext()`.

Video e audio

L'elemento specifica un meccanismo generico per il caricamento di file e stream video, più alcune proprietà DOM per controllarne l'esecuzione. Ogni elemento in realtà può contenere diversi elementi che specificano diversi file, tra i quali il browser sceglie quello da eseguire. L'elemento è usato allo stesso modo per i contenuti sonori. Non esiste tuttavia una codifica universalmente accettata ma è necessario codificare il video (o audio) in più formati, per renderlo realmente cross-browser.

Altri API

HTML5 include diverse API per manipolare i contenuti e gestire eventi specializzati, sempre nell'ottica di creare applicazioni sofisticate nel browser. Diventa centrale il ruolo di Javascript e di oggetti disponibili client-side per interagire con:

- Browser, finestre e tab
- Navigazione utente
- Storage nel browser
- Computazione nel browser
- Ecc...

Interagire con il browser

Le specifiche HTML 5 definiscono i meccanismi di caricamento e navigazione, nella sezione "Loading Web Pages". Alcuni oggetti:

- **Window**: la finestra principale del browser, con i meccanismi di caching, scrolling, gestione degli eventi. Esiste in Javascript ma è molto più dettagliata e inclusa direttamente nelle specifiche HTML (ancora una volta, interattività e dinamicità)
- **Browsing context**: ambiente in cui i contenuti sono presentati all'utente. Può essere la finestra del browser, ma anche un tab o un *iframe*
- **Session History**: la sequenza di documenti mostrati in un browsing context.
- Interfaccia **History** che definisce i metodi per navigare la "cronologia", memorizzare e aggiornare i bookmark.

Web storage

WebStorage permette di memorizzare dati nel browser come coppie <nome-valore>:

- ❖ Informazioni memorizzate per domain (tutte le pagine di uno stesso dominio posso scrivere ed accedere gli stessi dati)
- ❖ due oggetti per memorizzare dati che scadono alla fine della sessione (window.sessionStorage) o sono permanenti, se non cancellati esplicitamente (window.LocalStorage)

INTRODUZIONE A JAVASCRIPT

ECMAScript

Questo in realtà è il vero nome di JavaScript. Nasce come Javascript negli anni 90 come un linguaggio per arricchire le pagine web di micro controlli. Questo prima di mandare i dati al server controlla che siano giusti, popolati eccc... e solo se questi controlli sono passati manda i dati al server per fare ciò che deve fare, inclusa una verifica più seria della coerenza e correttezza dei dati. Nel frattempo Microsoft fa un suo linguaggio e ne viene creato un altro ancora ma non ci sono similarità con Javascript. Nasce l'esigenza di creare un gruppo di lavoro per uniformare questa situazione.

Per fare questo vanno da ECMA, che è un piccola agenzia di standard europeo che di suo fa tutt'altro. nel 97 nasce il linguaggio uniformato per tutti chiamato ECMAScript che è diviso in due parti:

- da una parte il nucleo fondamentale del linguaggio, che è di fatto quello che vediamo oggi cioè la grammatica
- ammissione di oggetti predefiniti, cioè strutture dati già preorganizzate immediatamente a disposizione dei programmatori che forniscono una serie di funzionalità ulteriori e sofisticate

Si dice che la grammatica del linguaggio è ECMAScript e che JavaScript è quella parte del linguaggio che faccio eseguire client side perché ho a disposizione una serie di oggetti predefiniti che mi ritrovo nel browser.

Server side c'è stato per molto tempo una versione di JavaScript chiamata ASP poi nel 2014/15 è uscita la prima versione di NodeJS che è interprete di JavaScript server side. Da questo momento vi è l'abitudine di creare solo interamente in JavaScript con la parte client side sul browser e quella server side via Node.

Nel 1997 la prima pubblicazione ufficiale del linguaggio standardizzato non succede niente di evolutivo dopodichè WHATWG prende possesso della politica di sviluppo del linguaggio prima con HTML, poi con CSS e infine con ECMAScript.

ES.Next è un nome dinamico che fa riferimento a feature già implementate mentre vengono discusse, e prima o poi rilasciate in una versione ufficiale di EcmaScript.

Come eseguire uno script Javascript

Ambiente client side e server side non possono essere più diversi e distanti.

Client side → vivo all'interno di un ambiente che chiamerò browser, il quale mi fornisce l'interfaccia con il sistema operativo, l'interfaccia con l'ambiente di presentazione sullo schermo, mi gestisce gli input a livello del sistema operativo... come se fosse un'estensione di una specifica funzionalità di un'applicazione del sistema operativo che è il browser. Lavoro direttamente con un umano, reagisco alle richieste e operazioni di un essere umano. Ogni elemento del documento ha alcuni eventi associati (click, mouseover, doppio click, tasto di tastiera, ecc), e degli attributi on+evento associati. Inserendo istruzioni JavaScript (o chiamata a funzione) nel valore dell'attributo si crea una chiamata callback. Ci sono due eventi particolari da aggiungere: load (della window) e ready (del documento).

Server side → non ho niente di tutto questo. Mi pongo dietro a un server http il quale è in idolo la maggior parte del tempo. Ogni tanto riceve una richiesta (che hanno una forma molto chiara e specifica, sono URL), decide quale logica di programmazione deve eseguire, la esegue, genera una risposta (tipicamente una struttura dati o frammento HTML o formato JSON) e la comunicazione si interrompe fino alla prossima richiesta. I servizi server-side sono associati a URI. Il modo più antico e semplice è creare servizi diversi e inserirli in file separati, ciascuno con un URI proprio. Aprendo una connessione all'URI, viene invocato lo script ed eseguito il servizio. NodeJs (Express.js, in realtà) fornisce un meccanismo per associare una funzione javascript (callback) ad ogni tipo di URI. Aprendo una connessione all'URI, lo script centrale esegue la funzione corrispondente.

Quindi il primo si basa sul concetto di eventi. Le funzioni sono svolte, in JavaScript, in maniera bloccante sia dal browser sia dall'applicazione server side ovvero c'è un unico thread (flusso di esecuzione) il quale durante l'esecuzione dello script fa solo quello.

Come eseguire uno script sul browser

Asincronia → un disaccoppiamento temporale tra qualcosa e qualcos'altro. Assume forme diverse a seconda della situazione. Associando il codice ad un evento sul documento (e.g., il click su un bottone): event-oriented processing

Sincrono → appena il browser si accorge di qualcosa lo esegue e poi si rimette in idolo. In maniera sincrona, appena lo script viene letto, in un tag <script> o in un file, adatto per inizializzare oggetti e variabili da usare più tardi.

Qualunque operazione in JavaScript avviene in modo bloccante: il browser fa quello e nient'altro. Quindi è importante che queste esecuzioni siano belle frammentate e vengano eseguite nella maniera più rapida e corta possibile da lasciare all'utente la sensazione di interattività e controllo.

Tutto ciò che è in Javascript nella pagina HTML viene eseguito sincronamente.

In maniera asincrona mi preparo la riorganizzazione di tutta la mia pagina quindi identifico quali sono le aree attive, i widget, e gli assegno le funzioni di callback. Altre operazioni che si svolgono in questo modo sono quelle di rete: il modello di AJAX prevede che la pagina HTML sia sempre lei, non ci sia più navigazione, ma carico una volta la pagina, ci carico sopra un'applicazione JavaScript, dopodiché quando l'utente fa un certo numero di azioni che richiedono dati che sul browser non ci sono, senza che l'utente se ne accorga, viene creata una connessione http asincrona che richiede e riceve dati e l'applicazione JavaScript client side può andare e cambiare delle parti della pagina. Infine posso creare dei meccanismi che dopo un tot di millisecondi oppure ogni tot di millisecondi fanno un qualche cosa. Per fare ciò io richiamo una funzione di timeout.

Come mandare in output gli script

HTML visualizza l'output degli script in 4 modi diversi:

- scrivendo direttamente nella finestra del browser:

```
document.write(string) ;
```


- scrivendo sulla console:

```
console.log(string) ;
```

- scrivendo in una finestra di alert:

```
alert(string) ;
```

- modificando il DOM del documento visualizzato:

```
document.getElementById(id).innerHTML = string ;
```

Come attivare gli script

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

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

JAVASCRIPT BASE

JS: Tipi di dato

JavaScript è un linguaggio tipato ma non tipa le variabili ma i valori. Ogni valore ha associato un tipo. La variabile in sé può ricevere valori di tutti i tipi. Javascript è minimale e flessibile per quel che riguarda i tipi di dati. Ci sono quattro importanti tipi di dati atomici built-in:

- ✓ booleani
- ✓ numeri (sia interi, sia floating point)
- ✓ stringhe

Inoltre vanno considerati come tipi di dati anche:

- ✓ null
- ✓ undefined

C'è poi un unico tipo di dato strutturato, object, di cui fanno parte anche gli array.

JS: Variabili

I dati in Javascript sono tipati, ma le variabili no. Due modi per definire variabili:

- + 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.

JS: Operatori

Numeri

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

Stringhe

+	composizione	<code>g = "Hel"+"lo"</code>	g vale "hello"
+	composizione + casting	<code>h = "5" + 7</code>	h vale "57"

Confronto e booleani

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

JS: Strutture di controllo condizionali (1)

- Blocco if

```

if (a==5)
    istruzione_singola ;

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

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

if (a==5) {
    istruzione_1;
    istruzione_2;
    ...
} else {
    istruzione_3;
    istruzione_4;
    ...
}

```

JS: Strutture di controllo condizionali (2)

- Operatore ternario

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

- Blocco switch

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



JS: Strutture di controllo cicli (1)

- Blocco for

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

i è l'indice di controllo del loop. E' un intero

- Blocco for ... in

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

j è l'indice di controllo del loop. E' una stringa

JS: Strutture di controllo cicli (2)

- Blocco while

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

- Blocco do ... while

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

JS: Strutture di controllo eccezioni

- Blocco try ... catch

```
var x=prompt("Enter a number between 0 and 9:","");

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

JS: funzioni

Le funzioni in Javascript sono blocchi di istruzioni dotati di un nome e facoltativamente di parametri. Possono ma non sono obbligate a restituire un valore di ritorno. Le funzioni non sono tipate, i valori di ritorno sì (come al solito). Se manca un parametro, non restituisce errore ma assume che il parametro sia undefined.

JS: tipi di dati strutturati

Javascript ha un unico tipo di dato strutturato, chiamato object. Anche gli array sono un tipo speciale di object. Gli object sono liste non ordinate di proprietà, coppie nome-valore.

Il valore di una proprietà può essere esso stesso un object. E' un (frequente) errore la virgola dopo l'ultimo elemento di un blocco. Alcuni interpreti la ignorano, altri no.

Ci sono due sintassi per accedere alle proprietà di un object:

- ❖ Dot syntax (ispirazione dai linguaggi Object Oriented)

```
alert(persona.nome + ' ' + persona.cognome)
```

- ❖ Square bracket syntax (ispirazione dagli array associativi)

```
alert(persona['nome'] + ' ' + persona['cognome'])
```

il nome della proprietà in questo caso è una normalissima stringa:

```
var n1 = 'nome' ;  
alert(persona[n1]);
```

posso anche fare elaborazioni sulle stringhe:

```
var n2 = 'cog'+ n1;  
alert(persona[n1]+' '+persona[n2]);
```

Uso entrambe le sintassi per leggere e per scrivere le proprietà dell'object:

```
persona.cognome = 'Verdi' ;  
persona['nome'] = 'Antonio' ;  
  
var n1 = 'nome' ;  
persona[n1] = 'Andrea' ;
```

Moltiplico i punti o aggiungo square bracket per proprietà in oggetti annidati:

```
persona.indirizzo.via.numero = '36' ;  
persona['indirizzo']['via']['numero'] = '15/a' ;
```

JS: Array

Un array è un object in cui le chiavi sono numeri interi assegnati automaticamente. Per distinguerlo da un oggetto normale usa la parentesi quadra invece che la graffa.

```
var nomi = ['Andrea', 'Beatrice', 'Carlo'] ;
```

La dot syntax non può essere usata, ma solo quella bracket. Il primo numero è lo zero.

```
alert(nomi[0] + ' ' + nomi[1]) ;
```

Posso normalmente leggere e scrivere elementi dell'array:

```
nomi[0] = 'Adriano';
```

Un array è un object con alcuni metodi e proprietà molto utili:

- **length**: lunghezza dell'array
`var n = nomi.length;` ← *n vale 3*
- **indexOf(item)**: la posizione di *item* nell'array
`var k = nomi.indexOf('Beatrice');` ← *k vale 1*
- **push(item)**: aggiunge *item* in fondo all'array
`nomi.push('Davide');` ← *nomi ha un elemento in più*
- **pop()**: toglie un valore in fondo all'array e lo restituisce
`var d = nomi.pop();` ← *d vale 'Davide' e nomi ha un elemento in meno*
- **shift()**: toglie un valore in cima all'array e lo restituisce
`var a = nomi.shift();` ← *d vale 'Andrea' e nomi ha un elemento in meno*
- **unshift(item)**: aggiunge *item* in cima all'array e restituisce la nuova lunghezza
`var d = nomi.unshift('Antonio');`
← *nomi ha un elemento in più*

Altri metodi utili:

- **slice(start,end)**: restituisce un array da start a end (escluso):
`var b = nomi.slice(0,1);` ← *b = ['Antonio']*
- **splice(pos,rimuovi,inserisci)**: inserisce e rimuove elementi
`var pers = ["Andrea", "Barbara", "Carlo", "Elena"];`
`pers.splice(2, 1, "Claudio");`
← *Rimosso 1 item dalla pos. 2*
Aggiunto un nuovo item
persone = ["Andrea", "Barbara", "Claudio", "Elena"]
- **join(sep)**: crea una stringa usando *sep* come separatore.
`var p = pers.join(", ");` ← *p vale "Andrea, Barbara, Carlo, Elena"*

Oggetti e array possono contenersi liberamente. Attenzione ad usare parentesi quadre per gli array e graffe per gli oggetti.

```
var persona = {  
  nome: ['Giuseppe', 'Andrea', 'Federico'],  
  cognome: 'Rossi',  
  altezza: 180,  
  nascita: new Date(1995,3,12),  
  indirizzo: {  
    via: {  
      strada: 'Via Indipendenza',  
      numero: '15'  
    },  
    città: 'Bologna',  
    nazione: 'Italia'  
  },  
  telefono: [  
    { tipo: 'casa', numero: '051 123456'},  
    { tipo: 'cell', numero: '335 987654'}  
  ]  
}
```

JS: oggetti predefiniti

Javascript predefinisce alcuni oggetti utili per raccogliere insieme i metodi più appropriati per certi tipi di dati.

Oggetti multipli:

- Object
- Array
- String
- Date
- Number
- RegExp
- ...

Oggetti singolo:

- Math
- JSON
- ...

JS: Stringhe

L'oggetto String contiene metodi disponibili per tutti i valori di tipo stringa:

```
var str = 'Precipitevolissimevolmente';
```

- **length**: lunghezza della stringa
`var s = str.length;` ← s vale 26
- **indexOf(sub)**: la posizione di *sub* nella stringa
`var t = str.indexOf('ss');` ← t vale 13
- **substring(start, end)**: restituisce la sottostringa da start ad end
`var x = str.substring(3,8)` ← x vale 'cipit'
- **substr(start, length)**: restituisce la sottostringa da start per length caratteri
`var y = str.substr(3,8)` ← y vale "cipitevo"
- **split(sep)**: separa una stringa in array di utilizzando sep come separatore.
`var w = "130.136.1.110" ;`
`var z = w.split(".");` ← z vale ['130', '136', '1', '110']

JS: JSON

JSON (JavaScript Object Notation) è un formato dati derivato dalla notazione usata da JS per gli oggetti.

```
{
  "nome": ["Giuseppe", "Andrea", "Federico"],
  "cognome": "Rossi",
  "altezza": 180,
  "nascita": "1995-04-11T22:00:00.000Z",
  "indirizzo": {
    "via": {
      "strada": "via Indipendenza",
      "numero": "15"
    },
    "citta": "Bologna",
    "nazione": "Italia"
  },
  "telefono": [
    { "tipo": "casa", "numero": "051 123456"},
    { "tipo": "cell", "numero": "335 987654"}
  ]
}
```

Rispetto alla notazione usata nei programmi, bisogna ricordare solo:

- + Solo valori string, number, boolean, array o object

- + Anche i nomi delle proprietà sono tra virgolette
- + Si usano solo le virgolette doppie e non le semplici
- + Non si possono inserire commenti di nessun tipo

Tutti i linguaggi di programmazione più importanti oggi accettano e scrivono dati in JSON.

JSON è un singoletto in JavaScript che supporta due soli metodi.

```
var j = {
  nome: 'Fabio',
  voto: 10
};
var s = JSON.stringify(j);
var t = JSON.stringify(j, null, 2);

var k = '{
  "cognome": "Rossi",
  "amici": ["Andrea", "Lucia"]
}';
var u = JSON.parse(k);
```

s vale la stringa: '{ "nome": "Fabio", "voto": 10 }'

*t vale la stringa '{
 "nome": "Fabio",
 "voto": 10
}'*

*u vale l'oggetto {
 cognome: "Rossi",
 amici: ["Andrea", "Lucia"]
}*

JS: Date

Una data è un oggetto che esprime un giorno e un orario rappresentandolo come il numero di millisecondi trascorsi dalla mezzanotte del 1 gennaio 1970 e la data in questione. Poiché in realtà è un numero, questo permette di fare operazioni aritmetiche e confronti numerici.

```
- Costruttore:
  var d = new Date();
  var XMas2018 = new Date(2018, 11, 25);

- getDay(): Il giorno della settimana della data (0-Domenica fino a 6-Sabato)
  var w = XMas2018.getDay();

- toString(): converte il numero in una stringa leggibile
  var a = XMas2018.toLocaleDateString();
  var b = XMas2018.toString();

- Operazioni
  var msInADay = 1000*60*60*24;
  var msSinceXMas = d - XMas2018;
  var daysSinceXMas = Math.round(msSinceXMas / msInADay);

- Confronti
  var inThePast = d > XMas2018;
```

d contiene la data e l'ora di adesso

I mesi sono contati da 0

w vale 2 (Martedì)

"25/12/2018"

"Tue Dec 25 2018"

daysSinceXMas vale 70

inThePast è vero

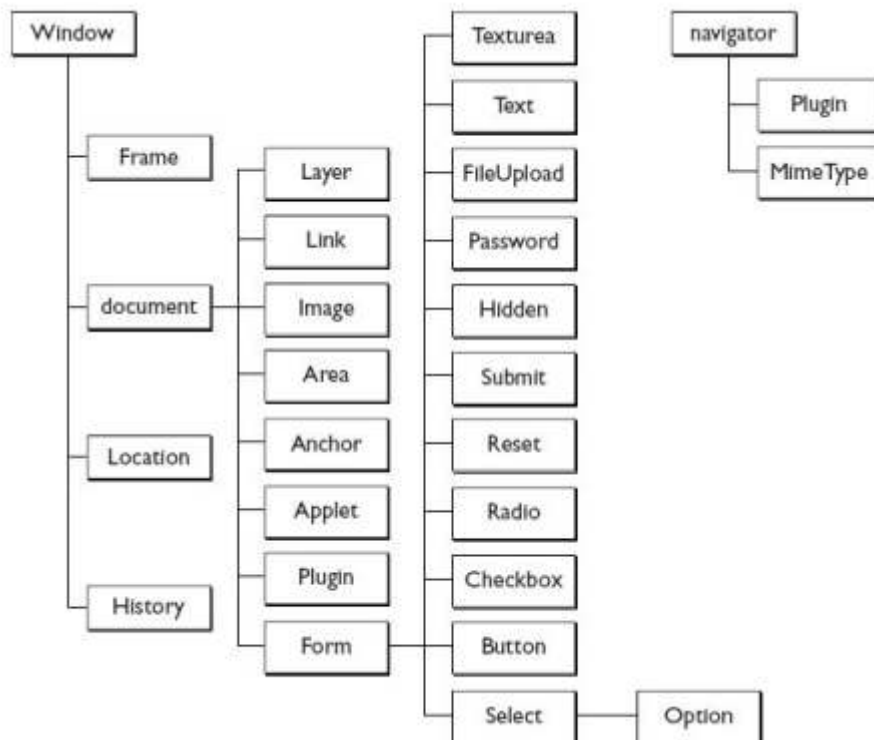
Altri oggetti

- Math:
 - Singoletto che raccoglie funzioni e costanti matematiche utili
 - Math.PI, Math.abs(), Math.sin(), Math.cos(), Math.round(), Math.sqrt(), Math.random, ecc.
- RegExp:
 - Classe delle espressioni regolari, da usare per fare match su stringhe. Le RegExp usano '/' come delimitatore.
 - var str = 'Precipitevolissimamente';
 - var re = /pi(.)/;
 - var x = str.match(re);
 - var y = re.exec(str);

sia x, sia y sono array con due match: ['pit' e 't']

Il browser è un'applicazione desktop che gira sul computer che è diretto e continuo contatto con l'essere umano e controlla il sistema operativo in cui sta.

Gli oggetti predefiniti del browser



Frame → meno si usa meglio è. Per separare in settori diversi e indipendenti i documenti

Gli oggetti principali: window e navigator

Dentro il browser JavaScript è basato su due oggetti di primo livello chiamati **Window** (spazio dei nomi fondamentale, principale client side e ha un certo numero di elementi predefiniti al suo interno direttamente accessibili. È possibile creare nuovi elementi) e **Navigator**.

Window → è l'oggetto top-level con le proprietà e i metodi della finestra principale:

- posizione: `moveBy(x,y)`, `moveTo(x,y)`, etc.
- dimensioni: `resizeBy(x,y)`, `resizeTo(x,y)`, etc.
- 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.

Gli oggetti principali: location e history

Location → rappresenta le informazioni relative alla location della pagina che stiamo guardando (Uri, protocollo...). È una proprietà sia leggibile (scopre in che pagina siamo) sia scrivibile (quando cambio location il browser farà un'attività di navigazione verso la nuova pagina). L'URL del documento corrente. Modificando questa proprietà il client accede a un nuovo URL (redirect):

- `window.location = "http://www.cs.unibo.it/";`
- `window.location.href = "http://www.cs.unibo.it/";`

History → rappresenta la history della singola pagina. Il browser fa di tutto per impedire lo scambio fra applicazioni di pagine diverse. 'array degli URL acceduti durante la navigazione. Possibile creare applicazioni client-side dinamiche che 'navigano la cronologia':

- Proprietà: length, current, next
- Metodi: back(), forward(), go(int)

Gli oggetti principali: document

Document → rappresenta l'elenco delle strutture dati particolarmente importanti del documento. Rappresenta il contenuto del documento, ed ha proprietà e metodi per accedere ad ogni elemento nella gerarchia:

- document.title: titolo del documento
- document.forms[0]: il primo form
- document.forms[0].checkbox[0]: la prima checkbox del primo form
- document.forms[0].check1: l'oggetto con nome "check1" nel primo form (non per forza una checkbox!)
- document.myform: l'oggetto "myform"
- document.images[0]: la prima immagine

Modello di documento

Ogni oggetto nella gerarchia è caratterizzato da un insieme di proprietà, metodi ed eventi che permettono di accedervi, controllarlo, modificarlo.

```
function Verify() {  
    if (document.forms[0].elements[0].value == ""){  
        alert("Il nome è obbligatorio!")  
        document.forms[0].elements[0].focus();  
        return false;  
    }  
    return true;  
}  
  
<FORM ACTION= "..." onSubmit="Verify()">  
<P>Name: <INPUT TYPE="text" NAME="nome" ...> </P>
```

JavaScript e DOM

Javascript implementa i metodi standard per accedere al DOM del documento.

```
var c = document.getElementById('c35');  
  
c.setAttribute('class', 'prova1');  
c.removeAttribute('align') ;  
  
var newP = document.createElement('p');  
  
var text = document.createTextNode('Ciao Mamma.');
```

```
newP.appendChild(text);  
  
c.appendChild(newP);
```

```
// Creazione elementi singoli
olist = document.createElement("ol");
voce1 = document.createElement("li");
voce2 = document.createElement("li");
testo1 = document.createTextNode("un po' di testo");
testo2 = document.createTextNode("altro testo - item 2");

// Creazione lista completa
voce1.appendChild(testo1);
voce2.appendChild(testo2);
olist.appendChild(voce1);
olist.appendChild(voce2);

// Inserimenti lista in una data posizione
div = document.getElementById("lista");
body = document.getElementsByTagName("body").item(0);
body.insertBefore(olist,div);
```

JavaScript ed eventi DOM

JavaScript permette di associare callback di eventi ad oggetti (dichiarazione locale o globale).

```
<script language="JavaScript">
window.onkeypress= pressed;
window.document.onClick = clicked;

function pressed(e) { alert("key pressed: " + e.which);}
function clicked() { alert("Mouse Click! "); }

</script>
</head>
<body>
  <p id="mypara">Puoi
    <a href="test.htm" onClick="alert('Link!');">
      cliccare qui
    </a>
    oppure qui.
  </p>
</body>
```

InnerHTML e OuterHTML

JavaScript (non DOM!) permette di leggere/scrivere interi elementi, trattandoli come stringhe:

- InnerHTML → sto chiedendo la linearizzazione di un sottoalbero che viene rapidamente riformata sottoforma di stringa, dopodiché posso modificarla come voglio e a questo punto andando a modificare nell'HTML riattivarla. Legge/scrive il contenuto di un sottoalbero (escluso il tag dell'elemento radice)
- OuterHTML → legge/scrive il contenuto di un elemento (incluso il tag dell'elemento radice)

```
// HTML: <div id="d"><p>Paragrafo!</p></div>
d = document.getElementById("d");

alert(d.innerHTML); // <p>Paragrafo!</p>
alert(d.outerHTML); // <div id="d"><p>Paragrafo!</p></div>
d.innerHTML = "<ul><li>Lista!</li></ul>";
```


Selettori in DOM

I metodi standard in DOM per accedere ai nodi di un documento sono essenzialmente:

- getElementById → solo ovviamente se l'elemento ha un id
- getElementsByName → se l'elemento ha un attributo name
- getElementsByTagName → tutti gli elementi con nome specificato

Il successo di JQuery ha portato nel tempo a proporre ed implementare in DOM HTML anche due nuovi selettori:

- getElementByClassName → cerca tutti gli elementi di classe specificata
- querySelector → accetta un qualunque selettore CSS e restituisce il primo elemento trovato - del tutto equivalente a `$()[0]` in JQuery
- querySelectorAll → accetta un qualunque selettore CSS e restituisce tutti gli elementi trovati - del tutto equivalente a `$()` in JQuery

AJAX: Introduzione

AJAX (Asynchronous JavaScript And XML) è una tecnica per la creazione di applicazioni Web interattive. Permette l'aggiornamento asincrono di porzioni di pagine HTML. Utilizzato per incrementare:

- ✓ l'interattività
- ✓ la velocità
- ✓ l'usabilità

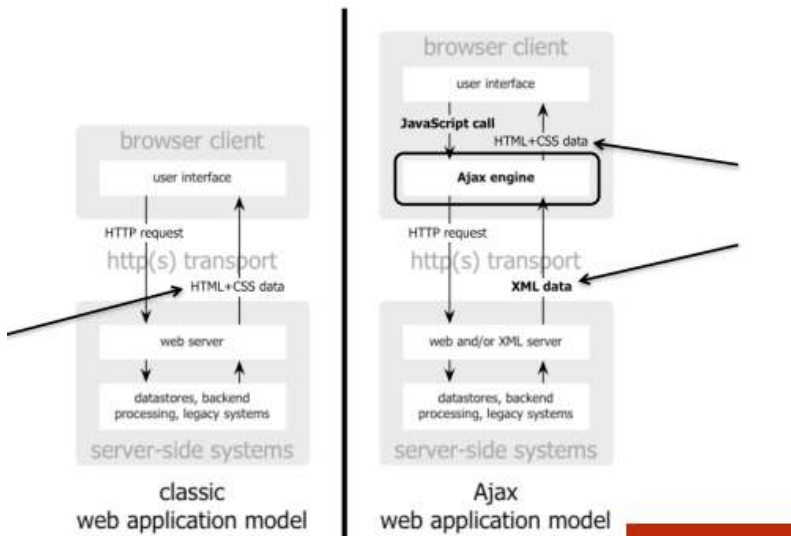
AJAX: Discussione

Non è un linguaggio di programmazione o una tecnologia specifica E' un termine che indica l'utilizzo di una combinazione di tecnologie comunemente utilizzate sul Web:

- XHTML e CSS
- DOM modificato attraverso JavaScript per la manipolazione dinamica dei contenuti e dell'aspetto
- XMLHttpRequest (XHR) per lo scambio di messaggi asincroni fra browser e web server
- XML o JSON come meta-linguaggi dei dati scambiati

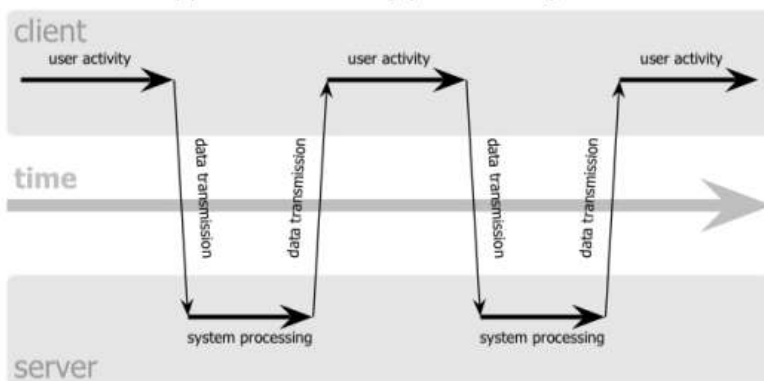
Asincrono → la richiesta di dati non avviene in un momento di navigazione ma quando la mia applicazione decide di voler attivare una richiesta, la quale non cambia la finestra ma trasmette alla mia applicazione web dei dati. Un ulteriore elemento di asincronicità è che il momento di creazione della richiesta e di ricezione della risposta sono separati concettuale e teoricamente nella mia applicazione perché non sono in grado di prevedere quanto ci impieghi il server a produrre la risposta e perché l'esecuzione di JavaScript è bloccante.

AJAX: Architettura 1



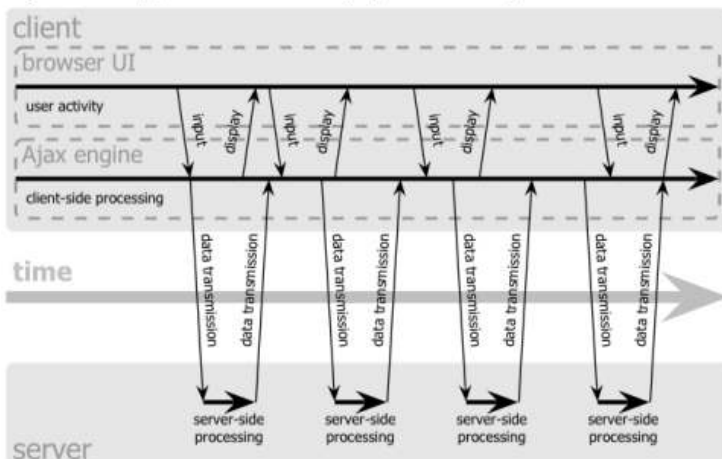
AJAX: Architettura 2

classic web application model (synchronous)



AJAX: Architettura 3

Ajax web application model (asynchronous)



AJAX: un po' di storia

Inizialmente sviluppato da Microsoft (XMLHttpRequest) come oggetto ActiveX In seguito implementato in tutti i principali browser ad iniziare da Mozilla 1.0 sebbene con alcune differenze Il termine Ajax è comparso per la prima volta nel 2005 in un articolo di Jesse James Garrett.

AJAX: 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 (non più clickand-wait)
- Velocità
 - Minore quantità di dati scambiati (non è necessario richiedere intere pagine)
 - Una parte della computazione è spostata sul client
- Portabilità
 - Supportato dai maggiori browser
 - Se correttamente utilizzato è platform-independent
 - Non richiede plug-in

AJAX: difetti

- Usabilità
 - Non c'è navigazione: il pulsante "back" non funziona
 - Non c'è navigazione: l'inserimento di segnalibri non funziona
 - Poiché i contenuti sono dinamici non sono correttamente indicizzati dai motori di ricerca
- Accessibilità
 - Non supportato da browser non-visuali
 - Non supportato da browser non-visuali
- Configurazione
 - È necessario aver abilitato Javascript
 - in Internet Explorer è necessario anche aver abilitato gli oggetti ActiveX
- Compatibilità
 - È necessario un test sistematico sui diversi browser per evitare problemi dovuti alle differenze fra i vari browser
 - Richiede funzionalità alternative per i browser che non supportano Javascript

Creare un'applicazione AJAX

Creazione dell'oggetto XMLHttpRequest

```
if (window.XMLHttpRequest) { // Mozilla, Safari,...
    http_request = new XMLHttpRequest();
} else if (window.ActiveXObject) { // Internet Explorer
    try {
        http_request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
        try {
            http_request = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {
            ...
        }
    }
}
```

Inizializzazione della richiesta

Prima di inviare la richiesta è necessario specificare la funzione che si occuperà di gestire la risposta e aprire la connessione con il server.

```
http_request.onreadystatechange = nameOfTheFunction;
http_request.open('GET', 'http://www.example.org/some.file', true);
```

I parametri della 'open' specificano:

- il metodo HTTP della richiesta
- l'URL a cui inviare la richiesta
- un booleano che indica se la richiesta è asincrona
- due parametri opzionali che specificano nome utente e password

Invio della richiesta

La richiesta viene inviata per mezzo di una 'send':

```
http_request.send(null);
```

Il parametro della 'send' contiene il body della risorsa da inviare al server:

- per una POST ha la forma di una query-string
`name=value&anothername=othervalue&so=on`
- per un GET ha valore "null" (in questo caso i parametri sono passati tramite l'URL indicato della precedente "open")
- può anche essere un qualsiasi altro tipo di dati; in questo caso è necessario specificare il tipo MIME dei dati inviati:

```
http_request.setRequestHeader('Content-Type', 'mime/type');
```

Gestione della risposta

La funzione incaricata di gestire la risposta deve controllare lo stato della richiesta

```
function nameOfTheFunction() {
    if (http_request.readyState == 4) {
        // risposta ricevuta
    } else {
        // risposta non ricevuta ancora
    }
    ...
}
```

I valori per 'readyState' possono essere:

0 = uninitialized

1 = loading

2 = loaded

3 = interactive

4 = complete

E' poi necessario controllare lo status code della risposta HTTP:

```
if (http_request.status == 200) {
    // perfetto!
}
else {
    // c'è stato un problema con la richiesta,
    // per esempio un 404 (File Not Found)
    // oppure 500 (Internal Server Error)
}
```

Infine è possibile leggere la risposta inviata dal server utilizzando:

- http_request.responseText che restituisce la risposta come testo semplice
- http_request.responseXML che restituisce la risposta come XMLDocument

Leggere e visualizzare dati (modalità sincrona)

```
function getData(){
    // load the xml file
    myXMLHttpRequest = new XMLHttpRequest();
    myXMLHttpRequest.open("GET", "example1.xml", false);
    myXMLHttpRequest.send(null);

    //legge la risposta
    xmlDoc = myXMLHttpRequest.responseXML;

    //recupera il frammento
    fragment = xmlDoc.getElementById("content").item(0);

    // modifica il documento corrente
    document.getElementById("area1").appendChild(fragment);
}
```

Leggere e visualizzare dati (modalità asincrona)

```

function getData(){
    // load the xml file
    myXMLHttpRequest = new XMLHttpRequest();
    myXMLHttpRequest.onreadystatechange = showData;
    myXMLHttpRequest.open("GET", "example1.xml", true);
    myXMLHttpRequest.send(null);
}

function showData() {
    if (myXMLHttpRequest.readyState == 4) {
        if (myXMLHttpRequest.status == 200) {
            //legge la risposta
            xmlDoc = myXMLHttpRequest.responseXML;
            //recupera il frammento
            fragment = xmlDoc.getElementById("content").item(0);
            // modifica il documento corrente
            document.getElementById("area1").appendChild(fragment);
        }
    }
}
}

```

framework AJAX

Sono librerie Javascript che semplificano la vita nella creazione di applicazioni Ajax anche complesse. Hanno tre scopi fondamentali:

- Astazione → gestiscono le differenze tra un browser e l'altro e forniscono un modello di programmazione unico (o quasi) che funziona MOLTO PROBABILMENTE su tutti o molti browser.
- Struttura dell'applicazione → forniscono un modello di progetto dell'applicazione omogeneo, indicando con esattezza e 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
 - Sono frameworks che vengono usati direttamente dentro alla pagina HTML con programmi scritti in Javascript
 - Framework esterni
 - Sono frameworks usati all'interno di un processo di sviluppo client e server e sono disponibili in un linguaggio di programmazione indipendente da Javascript (ad esempio in Java).
- **Ricchezza funzionale**
 - Librerie di supporto JavaScript
 - Prototype, jQuery e MooTools sono semplicemente livelli di astrazione cross-browser per task di basso livello DOM-oriented, ad esempio per arricchire graficamente un sito web tradizionale.
 - Framework RIA (Rich Internet Application)
 - Ext, GWT, YUI, Dojo e qooxdoo sono framework ricchi per la creazione di applicazioni complete, e includono una ricca collezione di widget, modelli di comunicazione client e server, funzionalità grafiche e interattive, e spesso anche strumenti di sviluppo.

INTRODUZIONE A NODE.JS

Node.js

Ci permette di utilizzare lo stesso linguaggio che abbiamo visto client side anche nella parte server side, quindi se oggi parliamo di un'applicazione che usa Node sul server e java nell'anti framework client side, parliamo di java full stack perché tutti i passi della costruzione utilizziamo lo stesso linguaggio.

Node non è altro che un ambiente di esecuzione di Javascript che viene usato per eseguire computazioni lato server. Il progetto nasce nel 2009, per un progetto individuale. E' alla base di un vasto ecosistema di moduli software e supportato da una vasta comunità open-source . Questo ecosistema è gestito tramite npm che è il gestore dei pacchetti al suo interno. La caratteristica principale di Node è che le applicazioni sono modulari, quindi il numero di moduli che vengono aggiunti è molto alto perché ogni operazione è svolta da un modulo separato.

Include un motore Javascript, lo stesso usato da Google Chrome.

Aspetti fondamentali

- Modello esecuzione → permette di eseguire gli script in modo efficiente. Si basa su operazioni non bloccanti quindi con la possibilità di avere operazioni asincrone.
- Moduli → per estendere l'applicazione
- Estensioni e gestione dei pacchetti via npm

Single thread e event loop

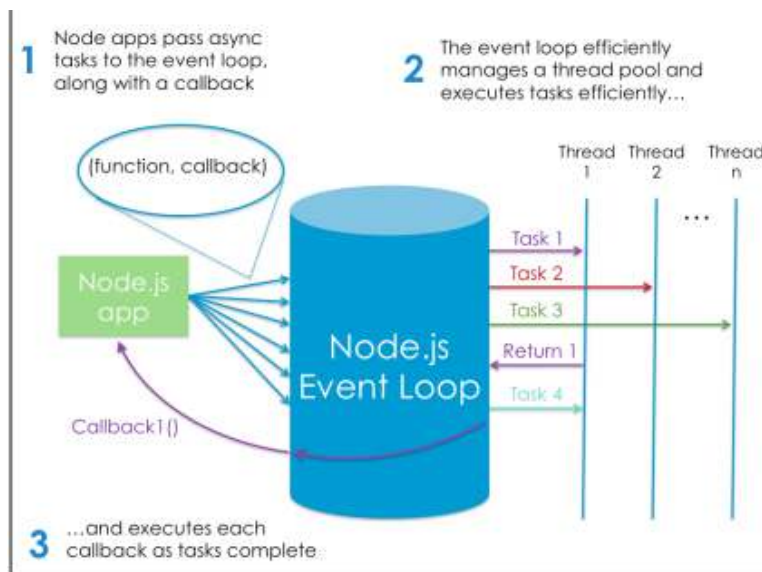
Node è un'applicazione single thread ovvero che ogni richiesta che arriva ha un unico thread di esecuzione, con un notevole guadagno in termini di efficienza. Per questo:

- Ridotto il tempo di context-switch da una richiesta all'altra
- Si usa meno memoria, si possono ricevere molte più richieste in parallelo

Per ottenere questo risultato al suo interno c'è un meccanismo, chiamato event loop che permette di gestire tutte le richieste, ogni volta che arriva una richiesta questa viene passata ad un worker che la esegue e continua l'event loop e a ricevere tutte le altre richieste. Quindi può eseguire in parallelo tutte le richieste che sono arrivate. Questo meccanismo riduce la necessità di fare content switch. Quindi i vari passi sono:

- Tutte le richieste sono gestite da un solo processo
- Questo processo passa i task da eseguire (work items) ad altri worker che lavorano in background attraverso la registrazione di funzioni di callback: registrata una callback quindi l'esecuzione continua e viene processata la richiesta successiva
- quando un worker termina la sua esecuzione il thread principale viene notificato (tramite eventi) e la callback invocata

Event loop



Event Loop non è una libreria esterna ma un elemento core del motore Node.js. Node.js entra nell'Event Loop quando viene eseguito lo script ed esce quando non ci sono più funzioni di callback pendenti da invocare. Internamente usa la libreria libuv for operazioni di I/O asincrone, che a sua volta mantiene un pool di thread. La comunicazione avviene tramite emissione di eventi. Gli oggetti che possono generare eventi (Event Emitter) espongono una funzione che permette di associare altre funzioni ad eventi che quell'oggetto genererà

`eventEmitter.on(eventName, listener)`

Al verificarsi dell'evento queste funzioni saranno invocate in modo sincrono

Funzioni asincrone e callback

Node.js è quasi esclusivamente basato su funzioni asincrone e callback. La convenzione suggerisce di creare funzioni che accettano una funzione callback asincrona come ultimo parametro. La funzione callback per convenzione prende in input come primo parametro un oggetto che contiene l'errore (o meglio un oggetto in cui la funzione che chiamerà la callback avrà memorizzato l'errore). Resta il problema di callback hell

Callback hell → perché se ho tante operazioni che sono in sequenza, ognuna devo indicarla dentro alla callback dell'operazione precedente

Siamo in JS e possiamo usare le tecniche per gestire operazioni asincrone: promesse, `async/await`, ecc.

Moduli Node.JS

I moduli in Node.js permettono di includere altri file JS nelle applicazioni e di riusare librerie esistenti. Favoriscono l'organizzazione del codice in parti indipendenti e riutilizzabili. L'enorme quantità di moduli disponibili gratuitamente ha contribuito al successo di Node.js. Node.js ha un sistema di caricamento dei moduli semplice ma potente:

- un modulo è un file Javascript
- quando si include un modulo, questo viene cercato localmente o globalmente

Per includere un modulo si usa la keyword `require(<modulo>)`

Caricare moduli

Node moduls, che è locale all'applicazione che sto costruendo, contiene tutte le librerie installate solo per questa applicazione. Esistono tre tipi di moduli:

- core → built-in nel sistema, non è necessario installarli separatamente
- dipendenze locali → installati per l'applicazione corrente nella directory./node_modules/
- dipendenze globali → disponibili per tutte le applicazioni e installati nelle directory globali specificate nella variabile d'ambiente NODE_PATH

Il fatto che installo librerie localmente mi permette di creare degli oggetti autocontenuti quindi degli oggetti che contiene tutte le dipendenze di questa applicazione. Le dipendenze che posso poi includere all'interno dell'applicazione stessa come avrei fatto con i file jar oppure dichiarare all'interno di un manifest (file che mi elenca tutte le dipendenze dell'applicazione) e caricarle nel momento in cui vado ad utilizzarle.

Un modulo può essere caricato specificando il percorso o il nome:

- ✓ `foo = require('./lib/bar.js');`
- ✓ `foo = require('bar')`

L'interprete cerca il modulo tra quelli core, poi tra le dipendenze locali e poi globali. Le dipendenze locali e globali si installano via npm

Per gestire le dipendenze uso il comando npm con cui posso installare dei nuovi moduli, prendendoli dai repository che ho a disposizione, o localmente o globalmente. Una volta installati, i moduli, li posso caricare.

Riusare moduli

I moduli sono eseguiti in uno scope indipendente. Questo permette di evitare conflitti e di creare librerie facilmente riutilizzabili. I moduli inoltre possono essere assegnati a variabili sulle quali invocare i metodi che il modulo espone.

Creare moduli

Abbiamo bisogno di un meccanismo per rendere gli oggetti visibili dall'esterno del file (lo scope è interno al file/modulo). Si usa un oggetto speciale module che rappresenta il modulo corrente, in particolare l'oggetto module.exports contiene tutto ciò che il modulo espone pubblicamente. Aggiungere una funzione (o altri oggetti) a module.exports vuol dire quindi renderlo pubblico e accessibile dall'esterno. Questo stesso meccanismo è usato, insieme a IIFE, dai moduli che installiamo via npm.

Npm

I moduli di node.js vengono distribuiti ed installati con npm (node package manager). Npm viene eseguito via command-line e interagisce con il registro npm (meccanismo robusto per gestire dipendenze e versioni dei pacchetti (moduli), semplice processo di pubblicazione di pacchetti e condividerli con altri utenti). semplice processo di pubblicazione di pacchetti e condividerli con altri utenti.

Creare un pacchetto per npm

Un pacchetto npm è un insieme di file Node.js, tra cui il file manifest package.json che specifica alcuni metadati del pacchetto, tra cui nome, autore, dipendenze, ecc. Il seguente comando eseguito nella directory che contiene gli script permette di creare il manifest attraverso un'interfaccia testuale interattiva:

- `npm init`

Installare un pacchetto

Una volta creato un package si possono installare le dipendenze locali:

- `npm install express`

Il comando modifica il file `package.json`. Di default i pacchetti sono installati nella directory locale:

- `./node_modules/`

Note:

– nelle versioni precedenti alla 5.0 era necessario specificare il parametro `–save` per installare i pacchetti in questa directory, ora non è più necessario

– l'opzione `–g` permette invece di installarli globalmente (ovviamente ha bisogno dei permessi)

Alcuni comandi per gestire i pacchetti

- `npm list`
- `npm config list`
- `npm search express`
- `npm install express`
- `npm install express –global`
- `npm install express@1.0.2`
- `npm uninstall express@1.0.3`
- `npm update express`
- `npm init`
- ..

INTRODUZIONE A EXPRESS.JS

Applicazioni server-side

La parte server-side di un'applicazione Web può essere scritta in diversi linguaggi di programmazione e diversi framework, che ne semplificano scrittura, manutenzione e deploy. Ogni soluzione ha le sue specificità ma tutte le applicazioni server-side svolgono alcune operazioni:

1. Lettura richiesta HTTP, parametri e query string
2. Autenticazione (se richiesta)
3. Persistenza dati
4. Processing dati
5. Generazione risposta http:
 - + File statici: immagini, video, audio, fogli di stile, ecc. già disponibili sul server
 - + HTML, XML: contenuti pronti per la visualizzazione
 - + JSON, XML, CSV, ecc.: dati da processare e visualizzare client-side

Express.js

Un framework server- side è una libreria che mi serve per semplificare alcune operazioni nell'interazione con il client. In particolare, il più usato nelle operazione Node è un open source e largamente usato, e meccanismo di estensione.

Il core di Express è semplice, si basa sull'idea di aggiungere dei middleware, cioè strati intermedi che ricevono la richiesta, fanno una operazione e la passano al middleware successivo. L'architettura del sistema è fatto in modo che quando creo un'applicazione dichiaro ad ogni passo qual è il framework che voglio utilizzare, questo riceve la richiesta, la processa e la passa al successivo.

Implementa le funzioni principali di un framework serverside e gestisce richieste e risposte HTTP:

- Routing → associa la richiesta alla funzione che la gestisce
- lettura richiesta (parametri, query string, header, ecc.)
- sessioni e autenticazione utenti (tramite middleware)
- costruzione della risposta (status code, body, headers, ecc.)

Routing

Express fornisce una semplice interfaccia per fare routing:

```
app.method(path, function(request, response) { ... })
```

- method → è uno dei metodi HTTP (GET, POST, PUT, ecc.)
- path → è il local path dell'URI richiesto al server
- function(req, res) → è un handler da eseguire quando viene richiesto il path. I due parametri contengono gli object della richiesta HTTP (uri, intestazioni, parametri, dati, ecc.) e della risposta HTTP in via di restituzione (intestazioni, dati, ecc.)

Ogni route può gestire diversi handler sullo stesso path.

```
app.get('/', function (req, res) {  
    res.send('Richiesta GET');  
});
```

Route path

I route paths possono essere stringhe o espressioni regolari. Express.js controlla l'URL della richiesta e se individua un "match" invoca l'handler opportuno

Route parameters

I route paths possono contenere parametri, ossia frammenti del path a cui è associato un nome che può essere usato per recuperare il valore corrispondente.

```
Route path: /clients/:cId/products/:pId
Request URL: /clients/34/products/12
req.params conterrà { "cId": "34", "pId": "12" }
req.params.cId conterrà 34
req.params.pId conterrà 12
```

Oggetti request e response

Le classi Request e Response rappresentano le richieste e le risposte HTTP ed espongono i metodi per accedere a tutte le loro informazioni

- Per le richieste:
 - body, query string, parametri, cookie, header, ecc.
 - <https://expressjs.com/en/4x/api.html#res>
- Per le risposte:
 - Metodi per spedire lo status code e la risposta in diversi formati
 - Metodi per aggiungere gli header
 - <https://expressjs.com/en/4x/api.html#req>

Spedire una risposta

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

Middleware in Express

Express ha pochissime funzionalità proprie (routing), ma utilizza un grande numero di librerie middleware personalizzabili in uno stack di servizi progressivamente più complessi. Per aggiungere un middleware allo stack:

```
app.use(<middleware>)
```

Un'applicazione Express è allora essenzialmente una sequenza di chiamate a funzioni di middleware tra la richiesta e la risposta. Il middleware può:

- accedere agli oggetti di richiesta e risposta
- cambiarli ed eseguire del codice di modifica
- chiamare la prossima funzione del middleware (next())
- uscire dal ciclo e mandare la risposta.

File statici in Express.js

Express.js può essere usato anche per restituire file statici memorizzati sul server. Questa funzionalità si realizza aggiungendo un apposito middleware, che associa ad un path la directory in cui recuperare i file. E' possibile specificare path e directory diverse

Accedere ai dati di un POST

Appositi middleware sono usati anche per processare dati spediti via POST. La libreria bodyparser, ad esempio, permette di accedere al corpo dei dati spediti dal POST nel body della richiesta. Questi middleware si occupano di recuperare e elaborare i dati e li convertono in oggetti accedibili dall'applicazione.

I dati del POST di un form si accedono come:

```
<request>.body.<post_variable>
```

Come negli altri casi, è necessario include il modulo bodyparser e aggiungere il middleware:

```
bodyparser = require('body-parser');  
app.use(bodyparser.urlencoded({ extended: true }));
```

Lo stesso middleware può essere istanziato per processare dati in JSON:

```
app.use(bodyparser.json())
```

Routing modulare

La class express.Router permette di creare route handlers modulari e che possono essere combinati. Anche in questo caso si aggiunge un middleware responsabile di gestire il routing per un insieme di path

ECMASCRIPT 2015

EcmaScript 2015

EcmaScript è la standardizzazione presso ECMA di Javascript. Nata come necessità per permettere un po' di codice comune a Javascript di Netscape, Jscript di Microsoft, and VBA Javascript di Microsoft. La prima versione è del 1997, poi la versione 3 (1999) ha aggiunto qualche struttura di controllo (come try... catch, errors, etc.) e la versione 5 (2009) ha aggiunto il supporto per JSON. La versione 6 (aka EcmaScript 2015) è una major release, con molte nuove feature. Non tutto è stato inserito negli interpreti attuali.

JavaScript lo possiamo definire come il nucleo cuore condiviso EcmaScript più gli specifici oggetti predefiniti che io trovo o nel server o nel client. Definisce un numero estremamente limitato di oggetti predefiniti.

Nel 2014 dopo essersi interessato a html e anche di css, iniziò ad interessarsi anche di JavaScript.

Due sono gli scopi fondamentali:

- l'idea che la programmazione web non è fatta a compartimenti stagni, separati e indipendenti
- linguaggio java era punto di contatto/collante fra il markup e tutti i servizi interattivi e operativi che ogni singolo device mette a disposizione

Es 2015: funzioni freccia

Per definire funzioni inline:

Sintassi tradizionale

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

Nuova sintassi

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

Semplici funzioni di callback:

Sintassi tradizionale

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

Nuova sintassi

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

Es 2015: template literals

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

New way

```
var firstName = 'Jane';  
var x = `Hello ${firstName}!  
How are you  
today?`;
```

Tre elementi fondamentali:

- + Backticks come delimitatori → `
- + I new line fanno parte della stringa
- + Interpolatori → \${varName}

Si tratta sostanzialmente di zucchero sintattico, ma molto utile per sbarazzarsi dell'incubo delle virgole annidate.

Es 2015: definizioni di classe

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

Sintassi tradizionale

```
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;  
};
```

Nuova sintassi

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

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

Es 2015: ereditarietà

La keyword `extends` permette di esprimere anche ereditarietà in modo più semplice. Anche in questo caso è zucchero sintattico, gli oggetti continuano ad essere basati su prototipi

```
class Persona{
    constructor(nome) {
        this.nome= nome;
    }
    get nome() {
        return this.nome;
    }
}

class Studente extends Persona{
    constructor(nome, matricola) {
        super(nome);
        this.matricola = matricola;
    }
    getMatricola() {
        return this.matricola;
    }
}
```

PROGRAMMAZIONE ASINCRONA IN JAVASCRIPT

Programmazione asincrona

Ogni volta che io non sono in grado di creare una sequenza temporale controllata e univoca tra cose diverse.

La caratteristica più peculiare e tipica di Javascript, evidente immediatamente, 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 alla 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.

Abbiamo esigenze di asincronicità ogni volta che abbiamo l'esigenza di chiamare un servizio sulla cui disponibilità o sui cui tempi di esecuzione non abbiamo controllo.

```
var database = remoteService.setDatabaseAccessData() ;
var result = database.query("SELECT * FROM hugetable");
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 percepisce un'esecuzione a scatti, non fluida, non responsive.

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 0 → codice bloccante e amen

```
function searchProducts(query) {
  var async = false;
  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(outnnt);
}
```

➤ 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 non ho nessun particolare vantaggio da Ajax. Inoltre distribuisco la logica dell'applicazione in due luoghi, con evidente complessità della gestione:

- Chi si occupa della gestione delle eccezioni e degli errori?
- Chi si occupa del filtro delle opinioni negative o false?

```
function searchProducts(query) {
  var DB = services.setDBAccess('all') ;
  var allData = DB.search(query);
  var output = prepareDOM(allData.products,
                           allData.opinions,
                           allData.tweets);
  document.getElementById("display").appendChild(output);
}
```

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

```
function searchProducts(query) {
  var productDB = services.setDBAccess('products') ;
  productDB.search(query, function(products) {
    var out = prepareDOM(products);
    document.getElementById("products").appendChild(out);
  });
}
```


Le callback:

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

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, e hanno un ambiente indipendente. 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:

```
function searchProducts(query) {
  var productDB = services.setDBAccess('products') ;
  productDB.search(query, function(products) {
    var output = prepareDOM(products);
    document.getElementById("products").appendChild(output);
    var opinionDB = services.setDBAccess('opinions') ;
    opinionDB.search(products, function(opinions) {
      var output = prepareDOM(opinions);
      document.getElementById("opinions").appendChild(output);
      var twitter = services.setDBAccess('twitter') ;
      twitter.search(opinions, function(tweets) {
        var output = prepareDOM(tweets);
        document.getElementById("tweets").appendChild(output);
      });
    });
  });
}
```

- Soluzione 3 → le promesse. Una promessa è un oggetto che, si promette, tra un po' conterrà un valore. Per operazioni asincrone, quindi, non si restituisce direttamente il risultato ma si promette che l'oggetto restituito conterrà il risultato. La promessa viene:
- creata dalla funzione chiamante, che consumerà i dati prodotti se e quando saranno pronti
 - mantenuta dalla funzione chiamata, che produrrà i dati (executor/ producer)

Al momento della creazione di una promessa si specifica la funzione da eseguire per produrre i dati.

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

```
function search(data, query) {
  var productDB = services.setDBAccess('products') ;
  var opinionDB = services.setDBAccess('opinions') ;
  var twitter = services.setDBAccess('twitter') ;
  productDB.search(query)
    .then(function(data) {
      return opinionDB.search(data)
    }).then(function(data) {
      return twitter.search(data)
    }).then(function(data) {
      var output = prepareDOM(data);
      document.getElementById("display").appendChild(output);
      return output ;
    });
};
```

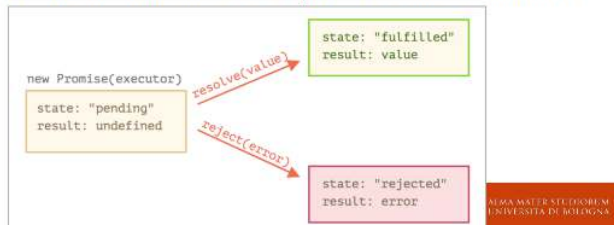


Struttura e stato di una promise

```
var promise = new Promise( function(resolve, reject) {  
  // executor ("producing" code)  
  // eseguito al momento della creazione di una promessa  
  ...  
  resolve(value) // passa il parametro value alla callback  
  ...  
  reject(error) // passa il parametro error alla callback  
} );
```

resolve: callback da invocare se la promessa è mantenuta

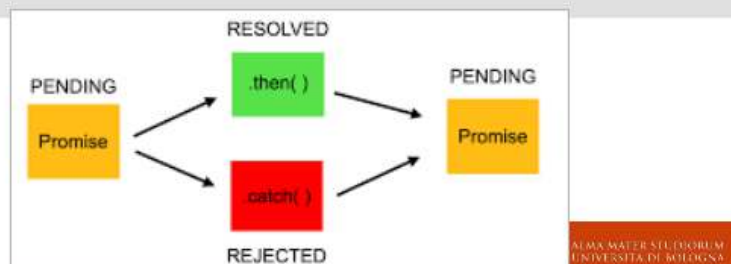
reject: callback da invocare se la promessa non è mantenuta



Promessa mantenuta?

```
promise.then(  
  (res) => { ... }, // funzione eseguita se la promessa è stata  
                    // mantenuta, riceve res dal Producer  
  (err) => { ... } // funzione eseguita se la promessa non è  
                  // stata mantenuta, riceve err dal Producer  
)
```

```
promise.then(  
  (res) => { ... })  
  .catch(  
    (err) => { ... }  
  )
```



Promise changing

Il metodo `.then` restituisce a sua volta una Promise, su cui si può invocare di nuovo `.then` e concatenare promesse

Il metodo `.catch` può essere invocato alla fine della catena per gestire in un unico punto gli errori

```
promise.then(  
  (res) => {return res + "!!!"} ,  
  ).then(  
    (res) => {console.log(res)}  
  ).catch(  
    (err) => { ... }  
  )
```

- Soluzione 4 → generator/yield. Alcune librerie (iniziando con Q) avevano introdotto il concetto di generatore e di yield. Questa è stata poi standardizzata in ES 2016. Il generatore è una metafunzione (una funzione che restituisce una funzione che può essere chiamata ripetutamente ed interrotta fino a che ne hai nuovamente bisogno). Ha una sintassi particolare con * dopo function. Il comando yield mette in attesa la 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.

```
function *main(query) {  
  var products = yield productDB.search(query) ;  
  var opinions = yield opinionDB.search(products);  
  var tweets = yield twitter.search(opinions);  
  var output = prepareDOM(products, opinions, tweets);  
  document.getElementById("display").appendChild(output);  
}  
var m = main();  
m.next();
```

- Soluzione 5 → async/await. La keyword async trasforma una funzione in una promessa: non restituisce un valore ma "promette" di restituirlo. Si può usare con tutte le sintassi viste per dichiarare funzioni

```
async function hello(){console.log("Hello")}  
var hello = async function(){console.log("Hello")}  
var hello = async () => {console.log("Hello")}
```

La keyword await può essere usata prima di qualunque funzione che restituisce una promessa e sospende l'esecuzione fino a quando la promessa non è mantenuta. In caso di errore viene sollevata un'eccezione, come se ci fosse un'istruzione throw. Si può usare SOLO all'interno di una funzione asincrona (async)

```
async function search(data,query) {  
  var productDB = services.setDBAccess('products') ;  
  var opinionDB = services.setDBAccess('opinions') ;  
  var twitter = services.setDBAccess('twitter') ;  
  var products = await productDB.search(query) ;  
  var opinions = await opinionDB.search(products) ;  
  var tweets = await twitter.search(opinions) ;  
  var output = prepareDOM(products, opinions, tweets);  
  
  document.getElementById("display").appendChild(output);  
  return output ;  
}
```

E' un misto di sincronia e asincronia: l'esecuzione di questa funzione è bloccata finché ogni await risponde, ma non prosegue al successivo await finché non risponde il precedente.

Promise.all

Se ho molte chiamate asincrone indipendenti (cioé in cui non debbo aspettare il risultato di una per richiedere la seconda) posso usare Promise.all().

APPROFONDIMENTI SU http: COOKIE, AUTENTICAZIONE, CORS

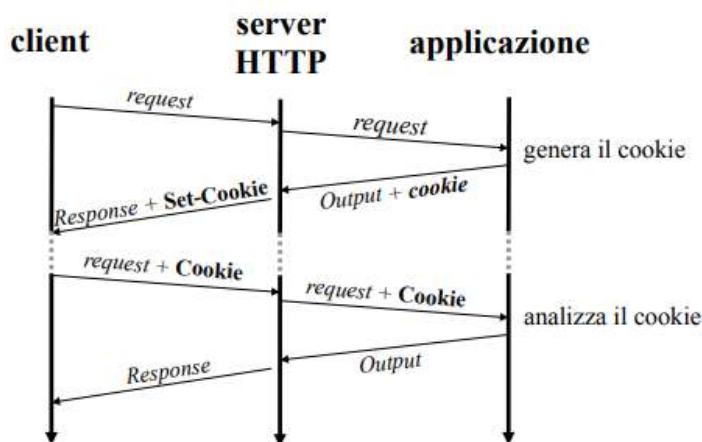
I cookies

È un'estensione dell'HTTP ma non fa parte del protocollo. I cookie non sono una parte delle specifiche di HTTP ma un meccanismo che è stato aggiunto. Serve per superare la caratteristica di HTTP di essere stateless. HTTP è stateless: non esiste nessuna struttura ulteriore alla connessione e il server non è tenuto a mantenere informazioni su connessioni precedenti.

I cookie sono un meccanismo generico che poi è stato istanziato per risolvere problemi specifici, in particolare non sono altro che un'informazione breve e di cui non è definito necessariamente il significato che viene spedita in connessioni multiple per poter mantenere un riferimento che ci permetterà di accumulare queste connessioni. Un cookie (non in HTTP, è un'estensione di Netscape, proposta nell'RFC 2109 e poi ancora RFC 2965) è una breve informazione scambiata tra il server ed il client. Il client mantiene qualche informazione sulle connessioni precedenti, e le manda al server di pertinenza ogni volta che richiede una risorsa.

In generale, l'idea è di avere un blocco di dati che il richiedente riutilizza ogni volta che accede al server e che precedentemente il server ha assegnato. Il termine cookie (anche magic cookie) indica un blocco di dati opaco (i.e.: non interpretabile) lasciato in consegna ad un richiedente per poter ristabilire in seguito il suo diritto alla risorsa richiesta (come il tagliando di una lavanderia).

Architettura dei cookies



Un cookie nasce sul server e viene poi spedito al client, che lo potrà utilizzare per essere autenticato nelle richieste successive. Alla prima richiesta di uno user-agent, il server fornisce la risposta ed un header aggiuntivo, il cookie, con dati arbitrari, e con la specifica di usarlo per ogni successiva richiesta. Il server associa a questi dati ad informazioni sulla transazione. Ogni volta che lo user-agent accederà a questo sito, rifornirà i dati opachi del cookie che permettono al server di ri-identificare il richiedente.

Ci sono due header in HTTP per gestire i cookie, uno per la risposta e uno per le richieste successive:

- **Set cookie** → che mi permette di trasferire il cookie dal server al client. È il primo che entra in gioco quando il client si autentica verso un server e il server spedisce la risposta che è corredata di questo header con le informazioni relative al cookie.

- Cookie → permette al client di rispeditore i cookie per la richiesta successiva. Il client decide se spedirlo sulla base del nome del documento, dell'indirizzo IP del server, e dell'età del cookie

Headers e struttura dei cookie

I cookies contengono dati arbitrari in formato testuale:

```
set-cookie:AWSALB=oJW4vtjxMz8WLY3jSrCUNekkG1aLZQHmb1SvExmv
R6agqb+f+fC4RQWCb+gLVijpRakI8RrnfrXqiDmQ9KwqA8LiVMdhkBRUCctO
gwx5JBxKLMIBQ7gnbIFIzo+; Expires=Wed, 01 May 2000 11:07:46
GMT; Path=/
```

Oltre a nome e valore, includono informazioni che il client usa quando ri-spedisce il cookie:

- ✓ Domain → il dominio per cui il cookie è valido
- ✓ Path → l'URI per cui il cookie è valido
- ✓ Max-Age/Expire → La durata in secondi del cookie
- ✓ Secure → la richiesta che il client contatti il server usando soltanto un meccanismo sicuro (es. HTTPS) per spedirlo
- ✓ Version → La versione della specifica a cui il cookie aderisce.

Uso dei cookie

Esistono vari tipi di cookie usati per scopi diversi, tra cui:

- Cookie persistenti → hanno una validità temporale lunga, o addirittura non scadono mai, e sono usati per mantenere informazioni (semi)permanenti ad esempio informazioni su login, preferenze degli utenti, ecc.
- Cookie di sessione → hanno una durata breve e sono usati per raggruppare operazioni in sessioni di lavoro; contengono un identificativo della sessione le cui informazioni sono sul server; solitamente sono cancellati alla chiusura del browser
- Cookie di terze parti → appartengono ad un dominio diverso rispetto a quello della pagina in cui sono caricati; usati ad esempio per banner pubblicitari, permettono di ricostruire la navigazione degli utenti e possono essere usati in modo improprio; le impostazioni dei browser permettono infatti di disabilitarli

Autenticazione

Vogliamo costruire dei meccanismi per accedere a delle risorse solo se siamo autorizzati all'accesso. Uno dei campi di maggiore applicazione dei cookie è l'autenticazione

Molto spesso infatti l'accesso a risorse (o servizi) Web è ristretto ad uno o più utenti che devono quindi essere riconosciuti. Esiste la differenza fra autorizzazione e autenticazione:

- + Autenticazione → devo riconoscere l'identità dell'utente che si sta collegando

- + Autorizzazione → devo essere sicuro che questa persona, o client, possa svolgere queste operazioni.

Si vuole fare in modo che, una volta autenticato l'utente non debba ripetere questa operazione per le successive richieste (fino alla scadenza della validità)

Autenticazione: sessioni o token

Per aggregare e collegare più richieste, previa autenticazione, si usano due approcci principali:

- Session-based → il server memorizza un ID di sessione e informazioni sull'utente verificate ad ogni richiesta
- Token-based → il client memorizza un token, ricevuto dal server, che spedisce ad ogni richiesta ed è usato per la verifica

Entrambi si possono realizzare sfruttando specifici header HTTP e i cookie.

Header WWW-Authenticate

Il primo header ad entrare in gioco è il www-authenticate, che interviene nel momento in cui si prova ad accedere ad una risorsa a cui non si è autorizzati. Il server risponde con l'errore 401, cioè che c'è bisogno di autenticazione e all'interno dell'header può spedire l'elenco degli schemi che possono essere usati per autenticarsi successivamente. Il meccanismo è generico e permette di usare criteri diversi per spedire i dati di autenticazione, anche personalizzati (ad esempio ne esiste uno specifico per AWS, Amazon Web Services).

Http ha principalmente tre schemi:

- Basic
- Digest
- Bearer → che è quello usato da meccanismi token based.

Schema Basic

Il meccanismo basilare, ormai in disuso, prevede la spedizione delle informazioni di autorizzazione in chiaro ad ogni risposta:

1. Il server indica il contesto di sicurezza (detto "Realm") nella prima risposta tramite l'header WWW-Authenticate
2. Il client chiede all'utente le informazioni di autorizzazione, crea una nuova richiesta GET e fornisce le informazioni di autorizzazione codificate in Base64 nell'header Authorization.
3. Il client continua a mandare lo stesso header per le successive richieste allo stesso realm

Molte limitazioni:

- La password passa viaggia in chiaro
- Non è prevista un'operazione di chiusura della sessione di lavoro

- Il form di autenticazione non è personalizzato e integrato nell'applicazione ma nel client

Schema Digest

Per evitare di spedire la password in chiaro, è stato introdotto uno schema Digest. Il client spedisce nell'header Authorization una fingerprint della password, ovvero la password crittografata con il metodo MD5 (RFC 1321). Per evitare replay attack il server spedisce al client anche una stringa causale (nonce) che viene crittografata dal client insieme alla password. Il server decifra i dati ricevuti dal client e se corretti lo autentica. L'operazione è ripetuta ad ogni richiesta

Schema Bearer

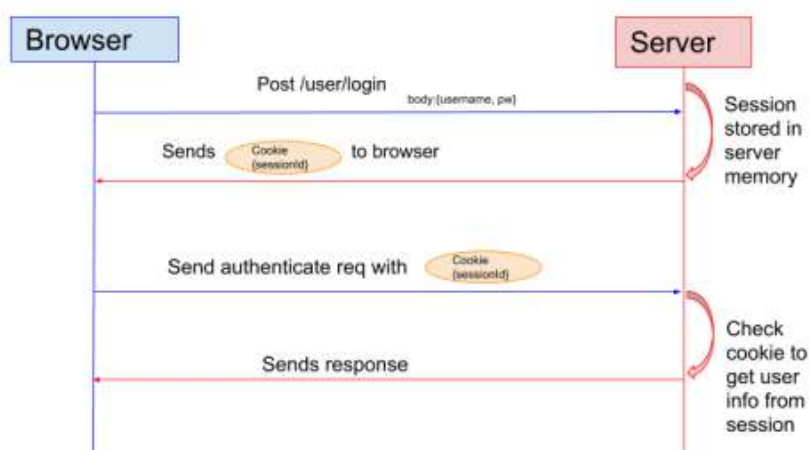
Nello schema Bearer (RFC 6750) il client non spedisce la password, in chiaro o cifrata, ma un token ("bearer token") che gli permette di accedere ad una risorsa. Il token è stato precedentemente generato dal server, controllando l'identità del client, ma il solo possesso è sufficiente per essere autenticati. E' importante quindi fare in modo che questi token non siano falsificati. Il meccanismo è generico e viene usato per l'autenticazione token-based, in contrapposizione a quella session-based.

Session-based authentication

Una sessione è un insieme di azioni "collegate" ed eseguite in un dato intervallo di tempo. Dopo aver verificato l'identità dell'utente, il server:

- genera una sessione a cui associa un ID e le informazioni relative all'utente; memorizza le informazioni sulla sessione
- spedisce al client l'ID della sessione appena avviata

Nelle successive richieste il client spedisce ID della sessione e altre informazioni per autenticarsi. Il server verifica queste informazioni, inclusa la loro validità temporale (una sessione può scadere e richiede quindi di ripetere il processo di autenticazione) e restituisce la risorsa all'utente. Esistono diversi modi per spedire le informazioni di sessione, tra cui i cookie.



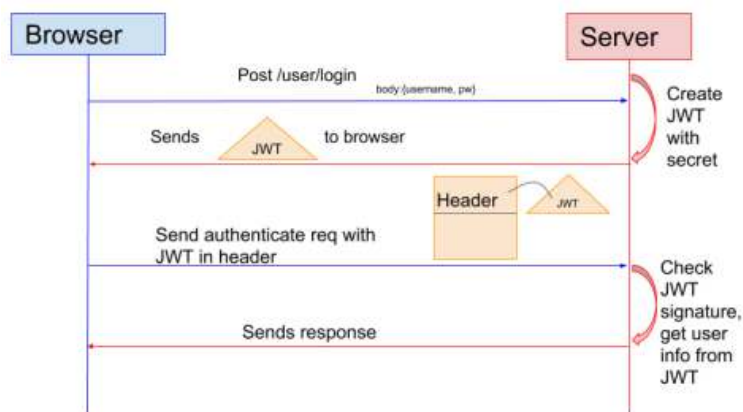
Cookie di sessione

I cookie di sessione contengono un ID univoco di sessione che il server ha memorizzato e riconosce. Il server si occupa anche di gestire la data di expiry (scadenza) del cookie e quindi chiusura della sessione. Tutti i linguaggi di programmazione server-side hanno una gestione automatica dei cookie di sessione e permettono allo sviluppatore di leggere facilmente le informazioni collegate.

Token-based authentication

Un approccio alternativo, e largamente usato oggi perché più scalabile, prevede l'uso di token e non richiede di memorizzare sul server informazioni relative alle sessioni. Dopo aver autenticato l'utente, il server crea un token con un segreto, lo firma con la propria chiave e lo spedisce al client. Il client memorizza il token e lo aggiunge negli header (tipicamente Authorization, con lo schema Bearer) di tutte le successive richieste. Il server verifica la propria firma sul token e le informazioni sull'utente e, se valide, restituisce la risorsa. Lo schema è generico, il token può essere di vari tipi così come il meccanismo usato per firmarlo

Token-based authentication (con JWT)



Sessioni o token?

	Sessioni server-side	Token
Informazioni sulla "sessione"	Server	Client
Riduzione carico e uso di memoria server-side		X
Riduzione dati da memorizzare		X
Maggiore controllo su revoca ed expiry	X	
Meno rischi di contraffazione	X	

JWT – JSON web token

JWT è uno standard (RFC 7519) che definisce un formato JSON per lo scambio di token di autenticazione, in generale di informazioni (detti claims) tra servizi Web. Il meccanismo è generico e permette di:

- Personalizzare i claim
- Usare algoritmi diversi per firmare i messaggi

Si basa a sua volta su altri standard per firmare (JSON Web Signature) e cifrare (JSON Web Encryption) messaggi in formato JSON. Sintassi compatta e URL-safe

Struttura token JWT

Un token JWT è composto da tre parti, separate da un punto e ottenute codificando i dati di input in base64 o firmandoli: header, payload e signature.



Header e Payload

Header: specifica il tipo di token e l'algoritmo di cifratura utilizzato

Payload: informazioni di interscambio organizzate in affermazioni (claims) che possono essere di tre tipi:

- + Registered → claim predefiniti utili per descrivere il token:
 - Entità che ha generato il token (iss, issuer)
 - Timestamp della generazione del token (iat, issue at)
 - Validità del token indicata in secondi (exp, expiration time)
 - Data dopo la quale il token inizia ad essere valido (nbf, not before)
- + Public → arbitrari ma dichiarati nello IANA JSON Web Token Registry per evitare conflitti
- + Private → arbitrari e personalizzabili, utili per scambiare dati tra applicazioni che si accordano sui dati da usare

Signature

Il token (header e payload, in base64) può essere firmato con una chiave segreta lato server in modo da poterlo verificare nelle successive richieste e, se corrotto, non considerarlo valido. Cifratura simmetrica e asimmetrica (nell'esempio è stato usato lo schema HMAC con algoritmo SHA256). Si possono usare diversi algoritmi, da dichiarare nell'header.

Il contenuto del token non è cifrato ma solo codificato in base64!

Si può decodificare facilmente, non bisogna quindi inserire dati sensibili

Express.js e autenticazione

In Express.js l'autenticazione è realizzata con appositi middleware. Come negli altri casi, devono essere installati, inclusi (con require) e aggiunti alla propria applicazione (con use).

Uno dei più usati è Passport.js:

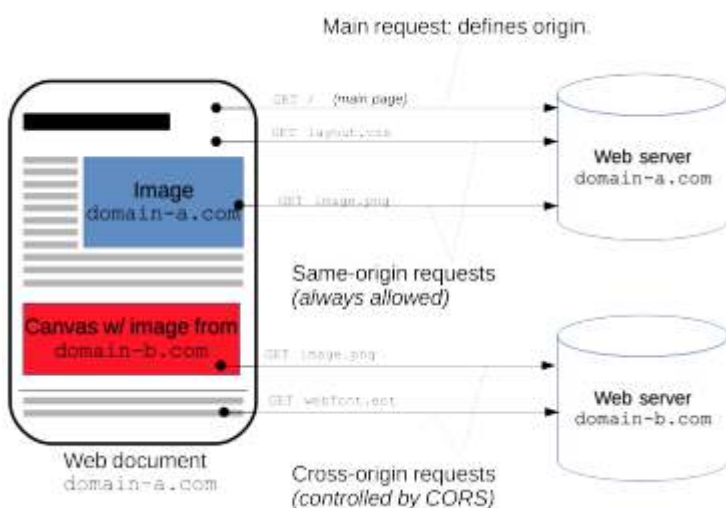
- <http://passportjs.org/>
- Flessibile e modulare
- Supporta diverse strategie di autenticazione tra cui HTTP basic e digest, e JWT

Per aggiungere il supporto a JWT invece si usa il pacchetto express-jwt.

Cross-site vulnerability

Supponiamo che in una pagina appartenente ad un dominio protetto (es. in una sessione autenticata) si riesca ad intrufolare un pezzo di codice malizioso, che raccoglie e trasmette informazioni ad un repository accessibile ai malintenzionati. Si parla di vulnerabilità tra domini o cross-domain. Una politica (molto conservatrice) dei browser è quella di rifiutare qualunque connessione Javascript ad un dominio diverso da quello della pagina ospitante (dominio diverso = schema o dominio o porta diversa). Solo gli script che originano nello stesso dominio della pagina HTML verranno eseguiti. Una delle possibili soluzioni, chiamata CORS, si basa sull'uso del metodo OPTION per indicare i domini ammessi

Cors



Cross-Origin Resource Sharing (CORS)

Una tecnica tutta HTTP introdotta dal W3C (W3C Rec 29/1/2013). Si attiva solo per le connessioni Ajax (non si usa per connessioni HTTP normali) e prevede l'uso di due nuovi header:

- ✓ nella richiesta → Origin, per specificare il dominio su cui si trova il contenuto
- ✓ nella risposta → Access-Control-Allow-Origin per indicare gli altri domini da cui è permesso caricare contenuti

Inoltre si pone l'accento sull'uso di un preflight (verifica preliminare) della possibilità di eseguire un comando cross-scripting, ad esempio usando il metodo OPTIONS di HTTP.

Nel momento in cui il dominio 1 si collega al dominio 2, viene specificato attraverso un header origin l'origine di quella richiesta. Il server che riceve questa richiesta sa che deriva da un

determinato dominio, risponde indicando attraverso un altro header (access control allow origin) tutti gli altri domini verso i quali è possibile fare altre richieste ajax.

Express.js e CORS

In Express.js si usa il middleware cors per aggiungere gli header Origin e Access-Control-Allow-Origin e supportare CORS. Necessario includere il modulo e aggiungere il middleware. Permette di abilitare le successive richieste Ajax su tutti i domini o su domini specifici e di specificare altre opzioni attraverso oggetti JS passati in input

```
cors = require('cors');  
app.use(cors())  
app.options('*', cors())
```

Caching

HTTP offre sofisticati meccanismi di caching, molto utili in applicazioni RESTFUL. Può essere client-side, server-side o su un proxy (intermedia). La cache server-side riduce i tempi di computazione di una risposta, ma non ha effetti sul carico di rete. Le altre riducono il carico di rete. HTTP 1.1 introduce due tipi di meccanismi per controllare la validità dei dati in cache (cache control):

- Server-specified expiration
- Heuristic expiration

Server-specified expiration

Il server indica una scadenza della risorsa con l'header Expires o con la direttiva max-age in Cache-Control. Se la data di scadenza è già passata, la richiesta deve essere rivalidata. Se la richiesta accetta anche risposte scadute, o se l'origin server non può essere raggiunto, la cache può rispondere con la risorsa scaduta ma con il codice 110 (Response is stale). Se Cache-Control specifica la direttiva must-revalidate, la risposta scaduta non può mai essere rispedita. In questo caso la cache deve riprendere la risorsa dall'origin server. Se questo non risponde, la cache manderà un codice 504 (Gateway time-out). Se Cache-Control specifica la direttiva no-cache, la richiesta deve essere fatta sempre all'origin server.

Heuristic expiration

Poiché molte pagine non conterranno valori espliciti di scadenza, la cache stabilisce valori euristici di durata delle risorse, dopo le quali assume che sia scaduta. Queste assunzioni possono a volte essere ottimistiche, e risultare in risposte scorrette. Se non valida con sicurezza una risposta assunta fresca, allora deve fornire un codice 113 (heuristic expiration) alla risposta.

Validazione delle risorse in cache

Anche dopo la scadenza, nella maggior parte dei casi, una risorsa sarà ancora non modificata, e quindi la risorsa in cache valida. Un modo semplice per fare validazione è usare HEAD: il client fa la richiesta, e verifica la data di ultima modifica. Ma questo richiede una richiesta in più sempre. Un modo più corretto è fare una richiesta condizionale (con un apposito header): se la risorsa è stata modificata, viene fornita la nuova risorsa normalmente, altrimenti viene fornita la risposta 304 (not modified) senza body della risposta. Questo riduce il numero di richieste

LIBRERIE FRONT-END – JQUERY

Sviluppo front-end

Queste librerie velocizzano la costruzione di applicazioni. Ognuna di queste librerie si porta dietro il modello che queste librerie adotta, oltre al punto di vista sintattico. Queste librerie in sintesi ci danno due cose:

- Da un lato ci semplificano la vita nella costruzione delle applicazioni
- Dall'altro ci forniscono degli elementi già riutilizzabili nella nostra interfaccia.

Negli ultimi anni si sono ampiamente diffuse librerie che semplificano e velocizzano la produzione di applicazioni Web. Diverse soluzioni sia server-side che client-side, con enfasi sempre maggiore sulla programmazione client-side in linea con gli sviluppi del Web e delle applicazioni mobile. Le tecnologie su cui si basano queste librerie non cambiano: fondamentalmente HTML, CSS e Javascript con alcune eccezioni/estensioni (es. TypeScript usato in Angular). Permettono di lavorare ad un livello di astrazione più alto (le differenze tra i browser sono nascoste agli sviluppatori) e di riutilizzare elementi dell'interfaccia (collezioni più o meno ricche di widget sono già incluse in queste librerie).

Librerie vs Framework

Finora abbiamo parlato di librerie client-side in generale ma è utile fare una distinzione tra:

- collezioni di oggetti e funzioni riutilizzabili → per svolgere un determinato compito in modo semplificato (es. manipolazione del DOM, richieste Ajax, gestione eventi).
 - + l'applicazione decide quando invocare queste funzioni
 - + più "di basso livello", meno potente ma meno vincolante
 - + esempi: JQuery, ExtJS, Prototype
- framework di sviluppo → ossia infrastrutture predisposte per realizzare applicazioni secondo un preciso approccio.
 - + l'applicazione si inserisce in uno schema pre-costruito e implementa un comportamento specifico seguendo questo schema
 - + più potente ma più vincolante
 - + esempi: Vue, React, Angular

Librerie: funzionalità comuni

Ogni libreria client-side "di basso livello" ha le proprie specificità ma diversi aspetti sono in comune e di maggiore interesse per la nostra discussione. In particolare tutte permettono allo sviluppatore di svolgere con facilità:

- Accesso e modifica del DOM → selezionare gli elementi dell'albero, navigarlo e modificarlo, sia nei contenuti che nello stile
- Esecuzione richieste Ajax → fare richieste asincrone, associare funzioni di callback, gestire casi di successo ed errore, convertire le risposte in formati diversi
- Gestione eventi → associare eventi a funzioni di callback, gestire la (non) propagazione degli eventi, sfruttare informazioni dettagliate negli eventi

JQuery

In assoluto la libreria più usata, inclusa ad esempio anche in alcune componenti dei framework CSS Bootstrap o Foundation. Semplice da includere nelle proprie applicazioni, molto leggero e veloce. Nato nel 2006, ha una licenza open source molto aperta. Nato nel 2006, ha una licenza open source molto aperta. Non ha librerie di widget proprie, ma il progetto parallelo jQuery UI fornisce una prima libreria di widget.

jQuery: l'oggetto \$

jQuery ha una peculiarità sintattica, l'uso della keyword \$ all'inizio di ogni comando della libreria:

```
$("#a").click(function() { alert("Hello world!"); })
$("#sayhello").click(...)
$("#hello").text("Benvenuto!")
```

Il carattere \$ in Javascript è usabile negli identificatori. \$() è un alias del costruttore jQuery() che restituisce un oggetto jQuery associato ad un selettore DOM su cui invocare i metodo per svolgere le opportune operazioni. Il codice jQuery è immediatamente identificabile come tale grazie a questo ausilio sintattico

Il ruolo delle callback

jQuery è una libreria Javascript e pertanto usa in modo estensivo le callback. Molti metodi prendono in input funzioni da eseguire al verificarsi di un evento o al completamento di un'operazione (asincrona):

- caricamento della pagina
- caricamento di tutte le risorse esterne
- richieste Ajax asincrone
- click e selezioni sull'interfaccia
-

Le funzioni di callback possono essere definite e passate con le diverse sintassi Javascript.

Caricamento DOM, risorse e script

Gli script vengono eseguiti subito dopo il loro caricamento. Se sono nell'header, ad esempio, sono eseguiti prima del caricamento del contenuto nel body. Ci sono due momenti specifici da tenere in considerazione:

- ❖ document.ready → il DOM è caricato, pronto e inizializzato. Eventuali risorse esterne (come le immagini) possono ancora non essere pronte, ma il loro posto nel DOM è già riservato.
- ❖ window.load → il DOM è caricato e pronto, e tutte le risorse esterne come le immagini sono pronte e caricate.

Eventi ready e load

jQuery mette a disposizione due metodi per associare callback agli eventi di ready e load.

```
$(document).ready(function(){
  // codice da eseguire al ready
})
$(window).load(function(){
  // codice da eseguire al load
})
```

Per il ready si può usare anche la sintassi abbreviata

```
$(function(){
  // codice da eseguire al ready
})
```

Tipicamente si mette tutto il codice dentro a `$(document).ready` a meno di dover leggere informazioni disponibili solo al load

Selettori jQuery

Il parametro del costruttore `$()` è un selettore jQuery. Sono selettori jQuery tutti i selettori CSS 3, più alcuni specifici di jQuery.

- `$("p")` restituisce un array di tutti i p del documento
- `$(".pippo")` restituisce un array di tutti gli elementi di classe 'pippo'
- `$("#pluto")` restituisce un array che contiene l'elemento di id='pluto'.
- `$(":selected")` restituisce un array di tutti gli elementi *selected*
- `$("p[id^='basic']")` restituisce un array di tutti i p il cui id *inizia con* la parola 'basic'
- ...

Selezionato un elemento, o meglio un array di elementi, si può modificare o associarvi eventi e callback

jQuery: modificare il DOM (contenuto, testo e attributi)

jQuery mette a disposizione molte funzioni di modifica del DOM. Esse agiscono o su tutto l'array identificato dal selettore, o sul primo elemento di questo array

- `$("h2").before("<p>Capitolo</p>"), $("h2").after("<hr/>")`: aggiunge contenuto prima o dopo ogni elemento H2
- `$("h2").html(), $("h2").html("Un titolo")`: legge o sostituisce il codice HTML in ogni elemento H2
- `$("h2").text(), $("h2").text("Un nuovo titolo")`: legge o sostituisce il testo in ogni elemento H2
- `$("h2").attr("role"), $("h2").attr("role", "titolone")`: legge o sostituisce il contenuto dell'attributo "role" in ogni elemento H2

jQuery: modificare il DOM (stili)

Diverse funzioni sono specializzate per gestire le proprietà CSS o collegate. Anche queste agiscono o su tutto l'array o sul primo elemento:

- `$("p").addClass("importante")` : aggiunge la classe CSS a ogni elemento dell'array selezionato
- `$("p").removeClass("importante")`: toglie la classe CSS a ogni elemento dell'array selezionato
- `$("h2").css("background-color")`, restituisce il valore della proprietà CSS specificata del primo elemento dell'array selezionato.
- `$("h2").css("background-color", "#FF0000")`: cambia il valore della proprietà

jQuery: effetti e animazioni

E' possibile attivare effetti e animazioni agli elementi del documento, eseguiti subito o dopo un delay e controllabili tramite alcuni parametri. Per attivare gli effetti in seguito ad un evento è necessario specificare l'animazione dentro all'handler dell'evento desiderato (vedi prossime slide).

- `$('#target').hide(); $('#target').show();` nasconde e mostra l'elemento con id='target', assegnando la proprietà CSS specifica (che è diversa nei vari browser ma gestita da jQuery)
- `$('.img').fadeIn('slow'); $('.img').fadeOut('slow');` fa lentamente apparire o sparire ogni elemento dell'array selezionato
- `$('.img').fadeIn(5000)` : stesso effetto di durata 5 secondi

jQuery: eventi

jQuery permette di specificare una funzione di callback da associare agli eventi del DOM.

```
$('#input').mouseover(function() {  
    alert('mouseover');  
})  
  
$('#input').keypress(function(e) {  
    alert(e.which);  
})  
  
$('#a').click( function(e) {  
    document.href="http://www.google.com";  
    e.preventDefault();  
})
```

`e.preventDefault()` blocca l'esecuzione del comportamento di default (in questo caso, la navigazione verso il contenuto originario dell'`href`).

In realtà, `click(...)`, `mouseover(...)`, ecc. derivano da un solo metodo `.on(...)`:

- `$('#input').mouseover(f1)` è equivalente a `$('#input').on('mouseover', f1)`
- `$('#a').click(f3)` è equivalente a `$('#a').on('click', f3)`

Il metodo `.on(...)` ha un secondo parametro facoltativo che indica a quali discendenti del selettore primario associare la callback

```
$('#body').on('click', 'a', f3) equivale (quasi) a $('#a').on('click', f3)
```

Attenzione: il binding della callback all'evento avviene solo su elementi già esistenti nel DOM. Quindi `$('#a').on('click', f3)` si applica solo agli elementi del DOM presenti al momento dell'invocazione.

Se voglio associare un evento a elementi che potrebbero essere creati in seguito al binding, devo delegare un elemento sempre esistente (ad esempio `body`) ad associare il callback all'elemento generato dinamicamente

jQuery: Ajax

Una singola funzione si occupa di tutta la comunicazione asincrona, prendendo in input un oggetto di opzioni:


```
$.ajax( {
  url: 'app.php',
  type: 'GET',
  success: function(data) {
    $('<div>.result</div>').html(data);
    alert('Caricamento effettuato');
  },
  error: function(data) {
    alert('Caricamento impossibile');
  }
})
```

I membri url e success sono obbligatori, success e error sono callback chiamate appena la connessione HTTP si conclude

Si possono usare funzioni equivalenti per maggiore rapidità che esplicitano il metodo HTTP: \$.get(), \$.put() e \$.post() (nota: non esiste uno shortcut per delete)

jQuery e promesse

Una alternativa al call-back è usare le promesse. Il metodo \$.ajax() è già una promessa, per cui si possono creare catene di funzioni then() con due parametri, la funzione da chiamare in caso di successo e quella in caso di insuccesso:

```
var good = function(data) {
  $('<div>.result</div>').html(data);
  alert('Caricamento effettuato');
};
var bad = function(data) {
  alert('Caricamento impossibile');
};
$.get('http://www.server.it/server').then(good, bad);
```

Conclusioni

La semplicità d'uso e la leggerezza rendono jQuery ideale per piccoli compiti di animazione e attivazione di elementi in pagine che sono già ben strutturate e graficamente complete. jQuery non è fatto per progettare interfacce, ma per descrivere comportamenti dinamici su alcuni elementi.

CSS LAYOUT

Layout

Ogni oggetto in CSS è rappresentato da una box e di ognuna di queste box vogliamo decidere i colori ma soprattutto posizionamento e dimensione. Il termine layout, o meglio page layout, indica la disposizione degli oggetti in modo armonioso nella pagina. Richiede quindi di organizzare gli oggetti e in particolare decidere le loro dimensioni e la loro posizione, in relazione agli altri oggetti e alle aree vuote. CSS definisce diverse proprietà per controllare in modo sofisticato il layout. Se il contenuto richiede più spazio di quello disponibile:

- + resizing → si cambiano le dimensioni degli oggetti
- + pagination → i contenuti sono divisi su più pagine
- + scrolling → solo una parte dei contenuti è visibile e si fornisce all'utente un meccanismo per far scorrere i contenuti visibili

Responsive layout

Un layout responsive è ottimizzato per essere letto su tutti i dispositivi. Obiettivo: minimizzare resizing e scrolling da parte dell'utente e facilitare l'esperienza di lettura. La costruzione di layout responsive non richiede nuove tecnologie (e non è di per sé una tecnologia) ma è realizzata direttamente in CSS, specificando il modo in cui il layout si adatta al dispositivo di lettura. Partiamo dai meccanismi CSS per progettare un layout, per poi arrivare alla responsiveness

Viewport

Il canvas è lo spazio (virtualmente infinito) di posizionamento degli elementi via CSS. Il viewport è la parte del canvas attualmente visibile sullo schermo; è possibile posizionare elementi sul canvas anche se non visibili attraverso il viewport. Ovviamente il viewport dipende dalla dimensione dello schermo. I dispositivi di dimensioni ridotte usano un viewport virtuale per calcolare le dimensioni degli oggetti nel layout e successivamente le scalano per visualizzarle nel viewport reale. Questa tecnica è molto utile per layout che non sono progettati come mobile-first.

Se un layout è progettato per dispositivi più piccoli e per modificare il layout in base alle dimensioni (tramite media queries, dettagli nelle prossime slide) questa tecnica non funziona bene. Partendo da una proposta di Apple, è stato introdotto l'uso di un elemento META di HTML per controllare il viewport. Specificando la proprietà "viewport" si possono decidere le dimensioni del viewport e forza il livello di zoom iniziale:

```
<meta name="viewport"
      content="width=device-width, initial-scale=1.0">
```

Suggerimento: aggiungete sempre questa dichiarazione alle pagine e usate initial-scale=1.0

Unità di misura basate su viewport

La più piccola unità indirizzabile sullo schermo è il pixel (px). Le dimensioni dei pixel dipendono fortemente dalla dimensione e dalla risoluzione degli schermi e sono molto diversi tra device e device e non sono un'unità affidabile. L'elemento <meta name="viewport".... è usato per controllare questo fattore di scala. Piuttosto che usare misure in pixel (o in percentuale rispetto al contenitore, poi tradotta in pixel dal browser) si possono usare unità di misura relative al viewport:

- ❖ Viewport width (vw) → una percentuale (1%) della larghezza attuale del viewport. A
- ❖ Viewport height (vh) → una percentuale (1%) della altezza attuale del viewport.

Larghezza e altezza di una box

Con le proprietà width e height si fissano larghezza e altezza delle box CSS, esprimibili con dimensioni assolute o relative. Non usare dimensioni fisse! Con max-width e max-height si fissano la larghezza ed altezza massime, min-width e min-height per i valori minimi. Di default larghezza e altezza si applicano alla content-box, escludendo padding e bordi. La proprietà box-sizing permette cambiare strategia:

- border-box → calcola le misure sull'intera box, compresa di padding e bordo. Molto più comodo per costruire layout
- content-box → default

Flussi inline, blocco e float

I flussi CSS indicano il modo in cui una box è posta rispetto alle altre. Esistono diversi flussi, tra cui quelli base sono:

- flusso blocco → le scatole sono poste l'una sopra l'altra in successione verticale (come paragrafi)
- flusso inline → le scatole sono poste l'una accanto all'altra in successione orizzontale (come parole della stessa riga)
- flusso float → le scatole sono poste all'interno del contenitore e poi spostate all'estrema sinistra o destra della scatola, lasciando ruotare le altre intorno

Ogni elemento HTML ha un flusso di default, che può essere sovrascritto tramite la proprietà display.

Posizionamento

La posizione dipende dalle altre scatole, dallo sfondo (canvas) o dalla finestra (viewport). Si usa la proprietà position:

- Posizionamento static (default) → la scatola viene posta nella posizione di flusso normale che avrebbe naturalmente.
- Posizionamento absolute → la scatola viene posta nella posizione specificata indipendentemente dal flusso (e nasconde ciò che sta sotto). La posizione è calcolata rispetto al primo elemento contenitore (padre o ancestor) che ha posizione assoluta e relativa
- Posizionamento fixed → la scatola viene posta in una posizione assoluta rispetto alla finestra (viewport), senza scrollare mai
- Posizionamento relative → la scatola viene spostata di un delta dalla sua posizione naturale. Gli elementi figlio possono essere posizionati in modo assoluto. Si usa quindi anche senza "spostare" l'elemento ma solo per aver un contenitore per posizionare gli elementi figlio. Utile anche per poter usare z-index (vedi prossime slide) che non si applica ad elementi static
- Posizionamento sticky → la scatola viene posta nella sua posizione naturale all'interno del suo contenitore, ma durante lo scrolling sta fissa rispetto al viewport fino a che il contenitore non esce dalla vista.

Definito il tipo di posizione, si possono controllare coordinate e dimensioni con altre proprietà (fino a 4):

- top, bottom, left, right → coordinate della scatola (rispetto al contenitore o elemento "ancestor" o viewport)
- width, height → dimensioni usabili invece di right e bottom

Se si indicano 4 proprietà, la scatola verrà creata delle dimensioni indicate, ma il contenuto in eccesso può uscirne (lo si gestisce con la proprietà overflow). con la proprietà overflow). Suggerimento: se non sapete in anticipo quanto contenuto avrete, specificate al massimo 3 proprietà dimensionali. Gestire correttamente l'overflow di una scatola troppo piena richiede molti sforzi.

Posizionamento: overflow

La proprietà overflow specifica come trattare il contenuto che non sta interamente nella scatola (forse con dimensioni troppo rigide). Valori possibili:

- ✓ visible → la scatola si espande per contenere tutto il contenuto
- ✓ hidden → il contenuto extra viene nascosto
- ✓ scroll → compare una scrollbar per l'accesso al contenuto extra.

Si possono specificare le proprietà overflow-x and overflow-y per permettere la comparsa di una sola delle scroll

Visibilità degli elementi

Due modi per nascondere o rendere visibili gli elementi:

- ☐ visibility: visible; (oppure hidden o collapse); l'elemento è aggiunto al DOM ma non visibile
- ☐ display: none (oppure altri valori per flussi diversi); l'elemento non è aggiunto al DOM e lo spazio che avrebbe occupato può essere occupato da altre box

Sono utili per creare, ad esempio, menù a scomparsa o tooltip e per questo sono spesso usati in combinazione con il selettore E: hover

Overlap e z-index

Le box possono sovrapporsi, ad esempio in caso di position: relative o absolute (e dropdown o tooltip). La proprietà z-index permette di indicare la posizione di una box nella pila di box sovrapposte. Più è alto il valore, più l'elemento è in primo piano. Per default il background delle box è trasparente. Per coprire i contenuti sottostanti deve essere quindi indicato un colore o un'immagine.

Proprietà float

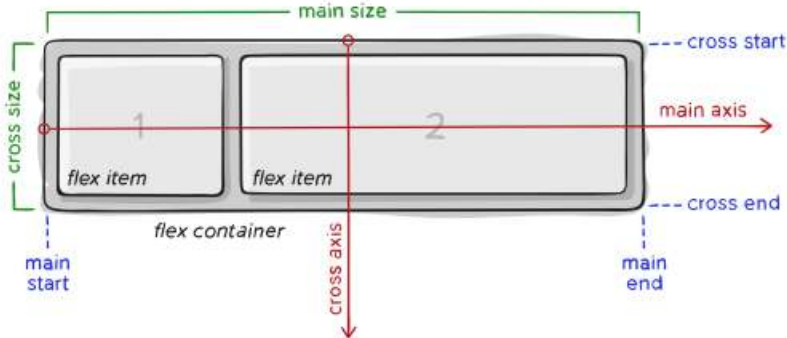
La proprietà float indica che una box deve essere spostata a sinistra o a destra all'interno del contenitore e le altre box ruotarle intorno (non display: float; ma una proprietà diversa, che può assumere valori left, right o none). Usata in combinazione con la proprietà clear (left | right | both | none) che indica il lato della box su cui non è possibile posizionare altri elementi di tipo float, che quindi seguiranno il normale flusso. E' stato lo strumento principale per costruire layout liquidi (in combinazione con la proprietà width per indicare larghezze percentuali delle box)

CSS Flexbox

Il modulo CSS Flexbox è uno strumento estremamente flessibile per costruire layout fluidi senza usare float. Introduce un nuovo valore per la proprietà display, appunto display: flexbox. Gli elementi si dispongono per usare al meglio lo spazio interno di un FlexBox (scatola flessibile). Due componenti: un FlexBox container all'interno del quale sono posizionati i FlexBox item. Le proprietà di container e item sono usate per decidere:

- direzione in cui disporre gli item
- possibilità di disporre gli item su più linee
- allineamento, spaziatura e ordine degli item

CSS Flexbox: terminologia



CSS Flexbox: alcune proprietà

Flex container:

- + flex-direction → definisce l'asse principale
- + flex-wrap → elementi su una a più linee
- + justify-content, align-items → allineamento sull'asse principale e secondario
- + gap → spazio tra i flex-items

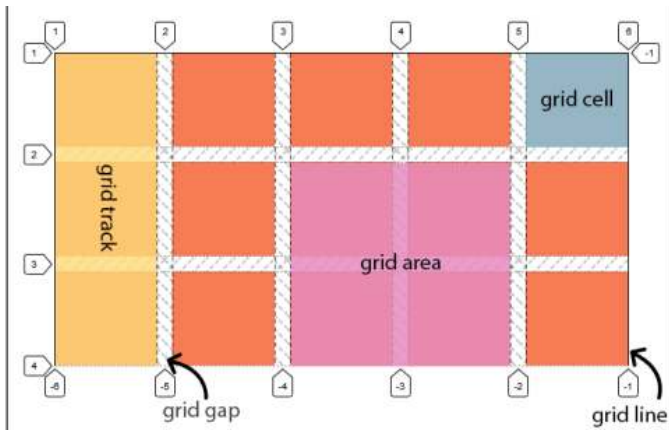
Flex item:

- + order → per modificare l'ordine degli elementi
- + flex → shorthand per flex-grow, flex-shrink and flex-basis che controllano come distribuire lo spazio rimanente dopo aver posizionato gli elementi. In particolare i tre valori controllano (per ogni elemento a cui si applicano) :
 - capacità di aumentare le dimensioni lungo l'asse principale
 - capacità di ridurre le dimensioni
 - dimensione di default prima di ridistribuire lo spazio
 - Nota: se si usa un solo valore es. flex: 3 corrisponde a flex: 3 1 0%

CSS Grid

Il modulo CSS Grid permette di costruire layout basati su una struttura a griglia, formata quindi da righe e colonne su cui sono disposti gli elementi. Anche in questo caso si usa la proprietà display:grid e il layout è formato da un contenitore (grid container) e un insieme di grid item. Le proprietà di contenitore e item permettono di controllare accuratamente il modo in cui la griglia è costruita e i contenuti sono disposti e visualizzati.

CSS Grid: terminologia



E su dispositivi diversi?

Le box che compongono i layout visti finora si ridimensionano correttamente grazie all'uso di unità di percentuali o relative al viewport. Questi layout sono detti fluidi proprio per la loro capacità di auto-adattare le dimensioni delle box. Questo approccio tuttavia non è completamente soddisfacente se si usano dispositivi con schermi più piccoli (o più grandi) per cui è utile disporre gli oggetti in modo diverso ed eventualmente nascondere qualcuno o cambiarne le proprietà. Le Media Query di CSS permettono questa ulteriore flessibilità

Media query

Le media query servono per specificare delle regole particolari che vengono attivate nel caso in cui il supporto usato per visualizzare la pagina Web soddisfa particolari vincoli. Tramite il loro uso, si può adattare automaticamente la presentazione di una pagina Web a dispositivi differenti senza cambiare il contenuto della pagina. Sintassi:

```
@media <query> {
  selettore , ... {statement; statement; ...}
  selettore , ... {statement; statement; ...}
  ...
}
```

Regole CSS. Si possono usare tutti i selettori e le proprietà, applicate in cascata e con ereditarietà

Una <query> è composta da condizioni su:

- tipo di dispositivo → print, screen, speech
- caratteristiche (più di 25 valori!) → aspect-ratio, width, height, hover, monochrome, ecc

Queste condizioni possono essere combinate con operatori logici: and, or, not






Layout responsive e breakpoint

Per creare layout responsive si individuano break-point, ossia "punti" in cui cambiare il layout, in corrispondenza di diverse larghezze dello schermo. Si usano le media-query per specificare il comportamento del layout prima e dopo il breakpoint (condizioni min-width e max-width). Vista l'ampia varietà di dispositivi è impossibile coprirli tutti. Un buon compromesso:

- fino 600px (smartphone)
- da 600px a 768px (large smartphones, portrait tablet)
- da 768px a 992px (landscape tablet)
- da 992px a 1200px (desktop)
- >= 1200 px (large devices, desktop)

Framework CSS e breakpoint

Non a caso i framework CSS usano alcuni break-point predefiniti per adattare il layout ai vari dispositivi

xs	Extra small <576px	 portrait mobile
sm	Small ≥576px	 landscape mobile
md	Medium ≥768px	 portrait tablets navbar collapse
lg	Large ≥992px	 landscape tablets
xl	Extra large ≥1200px	 laptops, desktops, TVs

At rules

Le Media Queries sono un particolare tipo di "At rules" (regole precedute da un simbolo "@"). Servono per specificare ambiti o meta-regole del foglio CSS. Altri esempi:

- **@charset**: serve per specificare l'encoding (es: UTF-8)
- **@font-face**: permette di specificare i font "customizzati" da utilizzare (vengono scaricati automaticamente)
- **@import**: permette di importare regole da altri stili
- **@namespace**: permette di definire namespace all'interno di un CSS ed usarli nei selettori
- **@page**: serve per definire caratteristiche di margine dell'intera pagina

Multicolonna

In CSS3 è stata introdotta la possibilità di gestire il layout multicolonna. In questo modo, il contenuto prosegue naturalmente (ovvero, senza l'uso di tabelle...) da una colonna all'altra. Il numero delle colonne può essere deciso (utile per documenti da stampare) o variare automaticamente a seconda della dimensione del viewport

Trasformazioni e animazioni

La proprietà transform permette di trasformare elementi HTML dentro uno spazio bi-dimensionale (2D) oppure tri-dimensionale (3D). inoltre è possibile cambiare lo stile degli elementi gradualmente e controllare lo stato di questo cambiamento. Si usa la keyword @keyframes per identificare un blocco di regole che controlla gli stati del cambiamento:

- + lo stato iniziale (from)
- + lo stato finale (to)
- + eventuali stati intermedi di una o più proprietà numeriche

Altre proprietà permettono di controllare durata, velocità e altri dettagli dell'animazione.

PREPROCESSORI E FRAMEWORK CSS

Alcuni limiti di CSS

Lo standard CSS ha subito una forte accelerazione negli ultimi anni. La scrittura di CSS complessi e portabili presenta tuttavia alcune difficoltà:

- ✓ Supporto variegato nei browser → ripetizione proprietà
- ✓ Struttura piatta delle regole → codice ripetuto → anche se lavora su un sistema gerarchico di HTML e XML.
- ✓ Uso esplicito di @media queries

Due soluzioni per semplificare la vita degli sviluppatori:

- Preprocessori → I preprocessori sono strumenti software che si occupano di ricevere in input uno pseudo CSS e fare alcune traduzioni che lo trasformano in un CSS che mi permette di visualizzare i contenuti.
- Framework (che usano internamente gli stessi preprocessori)

Un passo indietro: variabili CSS

CSS3 permette di condividere valori tra regole e calcolare dinamicamente i valori delle proprietà. Diventa quindi più semplice fare operazioni comuni e utili come:

- usare lo stesso colore o lo stesso font in molte regole
- definire regole sulla base di altre regole
- riutilizzare frammenti CSS

Due funzioni che ci vengono in aiuto:

- var() → inserisce il valore di una variabile, dichiarata in precedenza
- calc() → restituisce il risultato di un'espressione aritmetica. Sono ammessi gli operatori + - * /

Variabili CSS

Il nome delle variabili deve iniziare con i simboli '--'. Le variabili hanno uno scope, ossia una parte di script all'interno del quale sono visibili e utilizzabili. Si possono definire sia variabili con scope globale (selettore :root) che locale. Le variabili hanno accesso al DOM e possono essere lette e modificate in Javascript. I valori nelle variabili possono contenere anche unità di misura

Preprocessori CSS

Esistono alcuni linguaggi che estendono CSS per semplificare e potenziare queste operazioni (alcuni sviluppati prima di standardizzare variabili e funzioni CSS). Si scrive un CSS in una sintassi più ricca e potente, che sarà poi trasformato nel CSS finale:

- compilazione → il sorgente viene trasformato per generare un CSS "tradizionale" incluso con i meccanismi standard
- interpretazione → la pagina contiene uno script client-side che si occupa di processare le regole e applicare gli stili ottenuti

I preprocessori più noti:

- SASS
- LESS
- Stylus

Ogni preprocessore usa una propria sintassi ed offre alcune funzionalità specifiche ma esistono diversi elementi in comune:

- + Annidamento
- + Variabili, operatori
- + Mixins
- +

Annidamento

Più vicino alla struttura gerarchica di HTML, permette di non ripetere selettori se più elementi condividono uno stesso elemento padre o antenato. Alternativo alla sintassi compatta di alcune proprietà

Variabili

Permettono di riutilizzare valori nello stesso CSS. Simile a `var()` di CSS3 ma sintassi più semplice e flessibile. Anche qui scope globale o locale

Operatori

Valori e variabili possono essere calcolati tramite operatori:

- Matematici → +, -, *, /, %
- Relazionali → >, =, <=, >=
- Logici → and, not, or
- su stringhe → +, /

Mixins

I mixins permettono di riutilizzare frammenti di regole. Questi frammenti possono essere parametrizzati

@mixin: definisce il contenuto (può includere altri @mixin)

@include: include il contenuto, usa parametri posizionali o nominali

Framework CSS

Librerie con regole e classi predefinite che permettono di creare fogli di stile con facilità e velocità. Ce ne sono diversi e diversi tra loro ma hanno alcuni tratti in comune:

- ❖ classi e "griglie" per creare facilmente layout responsive
- ❖ componenti built-in facili da integrare nell'interfaccia
- ❖ regole e classi per ottenere un look omogeneo e professionale
- ❖ regole e stili che funzionano su tutti i browser (la maggior parte)

Qualche nome noto

Quale framework?

Come succede spesso in questi casi, non esiste una soluzione unica e indiscutibile. E' utile capire il funzionamento dei framework per poter cambiare e adattarsi in modo veloce ed efficace. Alcune aspetti da considerare:

- Stile omogeneo o meno vincolato
- Componenti pre-confezionate
- Capacità e facilità di personalizzazione
- Dipendenza da librerie Javascript e difficoltà di integrazione con altri framework
- Minimalità
- Curva di apprendimento
- Contesto e vincoli

Sorgenti e compilati: esempio in Bootstrap

Bootstrap, come gli altri framework, è disponibile in una versione pronta per essere inclusa nelle pagine (CSS ottenuto dal preprocessing di sorgenti SASS e minificato e JS minificato). I file possono essere inclusi da remoto (come nell'esempio precedente) o da copie locali.

Minificare: rimuovere tutti i caratteri non necessari dal sorgente, senza modificarne le funzionalità. Ma: difficile da leggere per il programmatore ma occupa meno spazio.

Inoltre è possibile ri-compilare i sorgenti e personalizzare i moduli da includere

Compilazione e personalizzazione Bootstrap

La compilazione si basa su una serie di script npm, il package manager di NodeJS. Non ci interessano i dettagli degli script ma i macro-passi che eseguono:

1. Preprocessing SASS (o LASS nelle versioni precedenti)
2. Aggiunta di prefissi per le proprietà specifiche dei browser
3. Aggiunta di codice CSS e/o JS per aggiustamenti e comportamenti specifici dei browser (browser hacks).

Esempio SASS in Bootstrap

- Bootstrap definisce un vasto set di variabili con valori di default per colori, dimensioni, effetti, etc.
- Ad esempio, una mappa di colori per il tema predefinito:

```
$theme-colors: (
  "primary": $primary,
  "danger": $danger
);
```

- Queste variabili si possono ridefinire in modo da usare i nuovi colori dopo aver ricompilato i CSS
- Le regole di precedenza di CSS permettono infatti di sovrascrivere i valori di default (indicati con la chiave `!default` nei sorgenti Bootstrap)

```
// valori personalizzati
$primary: #FFFFFF
$danger: #FF0000
// valori di default
@import "../node_modules/bootstrap/scss/bootstrap";
```

Esempio Autoprefixer

```
.example {
  transition: all .5s;
  background: linear-gradient(to bottom, white, black);
}
```



```
.example {
  -webkit-transition: all .5s;
  -o-transition: all .5s;
  transition: all .5s;

  background: -webkit-gradient(linear, left top, left
    bottom, from(white), to(black));
  background: -o-linear-gradient(top, white, black);
  background: linear-gradient(to bottom, white, black);
}
```

Stili di default

I framework CSS forniscono prima di tutto una formattazione di default per gli elementi HTML. Mettono inoltre a disposizione classi predefinite che possono essere aggiunte agli elementi per ottenere effetti tipografici (in fin dei conti sono tradotte in regole CSS). Solitamente includono anche set di colori predefiniti

Testo

Molti framework forniscono classi predefinite per ottenere effetti sul testo. Attenzione: velocizzano la produzione del codice ma specificano informazioni di stile direttamente sugli elementi, difficili quindi da generalizzare

Layout responsive

Una delle principali funzionalità dei framework CSS è fornire meccanismi semplici per produrre layout responsive. Questi meccanismi usano su break-point e media-query ma nascondono la complessità allo sviluppatore. Il framework:

- ✓ fornisce un sistema basato su griglia
- ✓ definisce un insieme di classi predefinite per indicare quanto spazio occupa ogni blocco su diversi dispositivi e viewport
- ✓ si occupa di tradurre queste istruzioni in opportuni blocchi `@media-query` per i diversi dispositivi e viewport

- ✓ si fa carico degli aggiustamenti necessari per far funzionare il layout su browser diversi

Griglia in Bootstrap

Bootstrap definisce una griglia virtuale:

- costruita sul layout flexBox di CSS
- e organizzata per righe e colonne

Ogni riga è divisa in dodicesimi. Ogni elemento può occupare una porzione di 1, 2, ... 12 dodicesimi della larghezza della riga. Se non si specifica lo spazio occupato da ogni elemento (si usa solo class="col", vedi prossima slide) questo viene distribuito equamente tra tutti gli elementi della riga.

Breakpoint e classi predefinite

Bootstrap mette a disposizione classi predefinite per indicare che un elemento è una riga (class="row") o una colonna (class="col"). Inoltre definisce gruppi di schermi predefiniti in corrispondenza di diversi breakpoints per controllare contenitori e colonne su diversi dispositivi.

Questi gruppi possono essere usati per differenziare le proprietà in base al dispositivo e, in particolare, il numero di colonne che un elemento occupa in una riga su quel dispositivo

Layout e classi predefinite

- Per usare una griglia quindi si identifica un elemento **contenitore** a cui si assegna una classe `container` o `container-fluid`
 - necessario per usare la griglia
- Si aggiunge la classe `row` agli elementi che corrispondono alle righe
- Si definiscono le colonne, usando le classi predefinite nella forma
`col-{screen}-{N}`
- per indicare che la colonna su uno schermo di tipo `screen` occupa `N` celle su 12. La parte `{screen}` si può omettere:
 - `col-xs-6` occupa metà della larghezza (6 su 12) del contenitore per schermi molto piccoli
 - `col-sm-4` occupa un terzo della larghezza (4 su 12) del contenitore per schermi piccoli
 - `col-3` occupa un quarto della larghezza indipendentemente dallo schermo
 - `col-2` occupa un sesto della larghezza indipendentemente dallo schermo

Breakpoints e griglie in Bootstrap

- Nella tabella i breakpoint di default in Bootstrap 4
- E' possibile personalizzare i valori (serve?) via SASS

```
$grid-breakpoints: (
  xs: 0,
  sm: 576px,
  md: 768px,
  lg: 992px,
  xl: 1200px
);
```

xs	Extra small <576px	portrait mobile
sm	Small ≥576px	landscape mobile
md	Medium ≥768px	portrait tablets navbar collapse
lg	Large ≥992px	landscape tablets
xl	Extra large ≥1200px	laptops, desktops, TVs

Griglia in Foundation

Molti altri framework usano sistemi basati su griglie. Ad esempio, anche Foundation usa una griglia divisa in 12 colonne e basata su FlexBox (ha anche una versione basata su CSS Float). Questo sistema, chiamato XYGrid, usa alcune classi principali:

- **grid-x**: identifica una riga
 - **cell**: identifica una cella
 - **<group>-***: indica il numero di colonne occupate dall'elemento su un dispositivo di un dato gruppo
- <group>** corrisponde a un insieme di dispositivi identificati da un breakpoint e anche in questo caso personalizzabili

Componenti

I framework forniscono diversi componenti grafici come barre di navigazione, tab, pannelli, finestre modali, ecc. Usano Javascript ma sono richiamabili interamente via markup. Le proprietà di queste componenti possono essere decise con opportune classi fornite dal framework. Capacità e flessibilità di ogni componente dipendono dal framework usato. Molti componenti richiedono plugin Javascript per funzionare (es. Collapse per elementi collassabili, come Accordion).

Una alternativa al call-back è usare le promesse.

e.preventDefault() blocca l'esecuzione del comportamento di default (in questo caso, la navigazione verso il contenuto originario del link).

La caraGeris?
peculiare ?pi
Javascript, ev
immediatam
la asincronici
filosofia di de

IntE

CEEE

L